CS262 Unit 3

4

Gundega

Contents

1 CS262 Unit 3

These are raw transcripts from video subtitles. Please feel free to improve them!

Contents

1.1 1. 00_l_introduction

Last time we finished up breaking up HTML and JavaScript into tokens. We'd specified them by regular expressions, which are implemented by finite state machines. And this has always been one of my favorite parts of programming languages, and it actually sticks in my mind when I was a student taking this in a more traditional classroom, the instructor asked after talking about regular expressions in lexing "Is there anyone interested in doing research?" And this was actually my first step towards becoming a professor, but actually, I was doing really poorly in the class at the time. I was not a very good student, so I went to the professor's office. He said, "Show up if you want to do undergraduate research." I was there, and when I came in the door, there was a second where I could see in his eyes this "You? Not you." But then he quickly sort of changed to a more positive attitude, and I continued to work with him, and that began my career pushing forth the boundaries of human knowledge by studying programming languages. And to some degree, what I want you to take away from a story like this is it's not necessary to have mastered these concepts right at the beginning or the first time you see them. What I really want is for you to master these concepts by the end of the course, so don't feel too bad if the original units have been giving you a bit of difficulty. They gave me a lot of difficulty when I learned them the first time. We finished up with regular expressions, and this time we're going to move on to context-free grammars, the sentences of programming languages. Let's get started.

1.2 2. 01_l_bags-of-words

All right, everyone, welcome back. In our last exciting episode, we learned how to take a string or a sentence and use lexical analysis to break it down into a list of tokens or words. And lexical analysis was based on our old best friends regular expressions. And remember that we need to break down HTML or JavaScript source code into tokens and then into valid utterances in order to understand them and build our web browser. However, it turns out that just having a list of tokens or a list of words is not enough. We can still be confused. For example, I've written here 2 collections of words. The first, "Simone de Beauvoir wrote 'The Mandarins,'" and the second, "wrote wrote Simone de de de." Even though this second collection of words uses only words that occur in the first, we like this one, and this one makes us very confused, so just a list of words isn't enough. They have to adhere to a valid structure. There's a subject and a verb and an object, and down here in bag of words 2, it's not really clear what's going on. In particular, we're tempted to say that bag of words #1, "Simone de Beauvoir wrote 'The Mandarins,'" follows English grammar, follows the rules of how we construct sentences or thoughts in English, and the second does not. The grammar for any modern natural language, be it Mandarin Chinese, English, French, admits an infinite number of utterances. But not--and this is super critical--not all utterances, all interesting grammars, rule something out. Provide structure by saying that you can't say gibberish. You have to say something meaningful. Despite the fact that we're going to rule out quite a few bags of words, we're still going to have plenty of room for creativity, and you'll see how in just a minute.

1.3 3. 02_l_syntactic-structures

Noam Chomsky is a philosopher and a linguist, and in his seminal 1955 work "Syntactic Structures," he suggests that utterances have rules, syntactic rules, and they're governed by formal grammars. The problem with the wrote, wrote, wrote, de, de, de bag of words above is that they don't form a grammatical sentence. We much prefer grammatical sentences. It's easier to interpret them and figure out what they mean. We can write down these formal grammars using a special notation. Here these 5 lines together are my formal grammar, and each one is what is known as a rewrite rule. The words that I've written in blue are called non-terminals. If you have one of these things written in blue, you can rewrite it with whatever is to the right of the arrow. These words that I've written in black never occur on the left of any one of our rewrite rules, so they can never be replaced. Once you get there, you're stuck, and the process terminates. We call them terminals. Using these rules, if I start with sentence, I can rewrite sentence to be subject verb, and then I could rewrite that by picking any one of the rules that has subject on the left. Let's pick students. I've only replaced subject, leaving this verb non-terminal alone. But then I could replace verb with any one of these rules, and at this point I think we're done. The process terminates because I can't replace students or think with anything, so here I've used 1, 2, 3, rewrite rules to start from sentence and end up with a valid utterance. This sort of maneuver with all these arrows is sometimes called a derivation because I was able to derive "students think" starting from sentence using these rewrite rules. Even using this relatively simple grammar, however, I have a few options. Here I've shown another derivation starting with sentence. Sentence goes to subject verb, verb goes to write, subject goes to teachers, and I've produced another string in the language of the grammar. Both "students think" and also "teachers write" can be produced by that grammar.

1.4 4. 03_q_deriving-sentences

Here I've recopied the grammar. I've abbreviated the non-terminals a bit. Subject became subj, and probably whether or not these are upper or lower case has been a little iffy. Let's allow all of those details for now. I don't care about case. We just care about the concepts in grammar. And now down here I've written 6 possible utterances: students think, teachers write, plus 4 more. It's quiz time. In this multiple, multiple choice quiz, I'd like

you to identify which of these 6 utterances could be derived using this grammar starting from sentence. And again, don't worry about exact spelling or upper or lower case.

1.5 5. 03_s_deriving-sentences

Let's go through it together. Starting from sentence, I can get to subject verb. One possible subject is students. One possible verb is think. We saw this one before. This is definitely in the language of the grammar. However, students teachers is not. That's 2 subjects in a row, and there's just no way to get there starting from sentence. We can turn sentence. We can rewrite it into subject followed by verb, so we can match students at the beginning, but teachers is not a verb. There's just no way to make this happen. Students write, this is one of our subjects, this is one of our verbs. That looks good. Teachers think. Teachers think, that's a subject followed by a verb. That looks good. Teachers write. That also looks good. Think write. These are 2 verbs, and our rules up here say a sentence has to be a subject followed by a verb. This just doesn't match the pattern. And in fact, in this particular grammar, these 4 are all of the strings in the grammar. There are only 4 possibilities. This grammar is perhaps not super exciting, but we're going to get to something, many things, much more exciting than this in just a bit.

1.6 6. 04_l_infinity-and-beyond

So in our original version of the grammar, only 4 strings were derivable. Here's our grammar from before. I'm going to add just 1 rule, and that one itty bitty rule is going to give me phenomenal cosmic power. Here I've added a rule to allow us to make compound subjects. This rule is super special awesome. Let me show you a new derivation that uses this new rule. Starting from sentence, we go to subject verb, subject and subject verb, students and subject verb, students and subject think, students and teachers think. Here, right here when we turned this subject into subject and subject, that's when we were using this starred rule. And notice that now, rather than a 2-word sentence, we've produced a 4-word sentence. Amazing. Do students and teachers think? That beggars belief. Both of us think? Teachers certainly don't think. Let's see how this plays out. This new power that we've discussed is called recursion in grammar. You're already familiar with recursion in programming where a function calls itself. Here we can replace a non-terminal subject with that same non-terminal and some other stuff, so just as we might define factorial in terms of factorial of x - 1, we can define subject in terms of subject and subject. This is a recursive rewrite rule, and thus the whole thing is now recursive grammar. Recursive grammars, that's where all the real power is because in essence, it allows a bit of a loop, and this gives us some room for creativity. This is how we can have a finite structure that admits infinite utterances but not all utterances.

1.7 7. 05_q_counting-utterances

So now it's time to make sure that you're following along with a quiz. How many utterances, how many sentences, how many strings can our new grammar produce? Recall we've got this new exciting rule, subject goes to subject and subject. Is it 0, 4, 6, 8, 16, or infinite? Choose the single best answer.

1.8 8. 05_s_counting-utterances

In fact, it's now infinite. I can reapply this subject goes to subject rule as many times as I want. Students and students and students and teachers think. And if someone were to challenge me and say, "Oh, there are only 5 strings in your grammar," I could always make a new one by using this rule one more time, so for any number you might think are produced, I can always get 1 more, so the number of potential strings I can produce is infinite. That's a huge amount of scope for creativity.

1.9 9. 06_q_an-arithmetic-grammar

And in fact, this general notion is one of the greater glories of context-free grammars. The grammar is finite. We can write it down. But for a non-trivial grammar, the number of possible utterances is infinite. Just by following simple replacement rules, I can make bigger and bigger utterances until I get tired and eventually pick terminals. And in fact, a number of academics, including Chomsky, argued that this is one of the ways or one of the reasons that our finite human brains--I myself am a bear of very little brain-- can produce an infinite number of potential creative ideas. A finite amount of matter can produce an infinite number of utterances. That's a relatively heady notion, so I'm going to skip past it for now. And it turns out that we're going to be able to see this same sort of infinity of possible utterances in computer languages like Python or JavaScript. Wow, my infinity symbol needs work. This gets a label because otherwise you might not be able to tell. In both of these languages, you can write down arithmetic expressions. You can imagine them being generated from an arithmetic expression grammar like this. Exp is just an abbreviation for expression. And in fact, were your student senses tingling? It's time for a quiz on this. Given this arithmetic grammar, check all of these utterances that are valid exp's, that are valid expressions. Starting from expression, which of these 5 options can I generate? Check all that apply.

1.10 10. 06_s_an-arithmetic-grammar

When I'm starting from exp, I can use any of these 3 rules, so in particular I could use the third one and just replace exp with number, so number is definitely in the language of the grammar. It's one of the strings that I can derive. Unfortunately, - number is not. There's no way to get a minus sign without an exp before it. If we take a look at all 3 of our rules, the minus sign only appears once, and there's something before it, so this isn't going to happen. Over here, number + number + number, it looks pretty big, but I could get it by applying this first rewrite rule twice and then turning each one of these expressions into numbers. I've drawn it in kind of this tree format. We will see more on that in a minute, but for now, just imagine I did this because I ran out of space in the corner. Poor planning on my part. Similarly, we could get number - number + number just by getting a minus sign here instead, using rule 2 and rule 1 in combination. Over here, there's no way to get number next to number. Whenever we get multiple expressions on the right-hand side, they're always separated by a terminal. This can't happen, so just to review, this uses rule 1, rule 2, and rule 3 three times. This is rule 1. That's rule 2. That's rule 3 three times in order to derive this string.

1.11 11. 07_l_syntactical-analysis

Recall from our last few units that lexical analysis is the process of breaking up a string into a list of tokens. We are now getting into syntactical analysis, which takes a list of words and tells you if that list of words is a valid derivation in the grammar, follows the rules of the formal grammar, is in the language of the grammar. All 3 of those mean the same thing. Just as the process of doing lexical analysis is sometimes called lexing, the process of doing syntactical analysis is called parsing. Parsing or syntactical analysis involves breaking down a list of tokens to see if it's valid in the grammar, or breaking down a list of words to see if they follow the rules of a language. To make our web browser our interpreter for HTML and JavaScript, we're going to combine the lexing we've already learned about with the parsing that we'll be learning in this unit and the next, and that's going to give us a huge amount of power. Or to put this another way, lexing word rules plus sentence rules from parsing is going to give us a scope and a structure in which to express creative utterances. It's time to put those 2 great tastes together. Here on the left I have a simple expression grammar that mentions numbers as a terminal, and over here on the right I have a rule in our lexer for figuring out when a string is a number, if it matches this particular regular expression. If I put the 2 of these together plus a similar rule for what it means to write a plus sign and what it means to write a minus sign, then I can suddenly tell that more realistic strings like 1 + 2 are valid in our language. The plus matches our rule for plus. number plus number, is in the language of our grammar for expressions, so this is good. Similarly, 33 is a number. 44 is a number. Minus is the minus sign. This is great. And over here I've got 7 + 2 - 2, or if you like, 7 + or - 2, which I hear is a magic number. Down here at the bottom I have a slightly longer derivation that shows off both the grammar part and also the lexical analysis part. Grammar turns expression into expression plus expression. Lexical analysis turns number into 5 or 6 or some such.

1.12 12. 08_q_statements

Here I've written a slightly more complicated grammar that actually describes part of Python. We can make assignment statements-- stmt is a common abbreviation for statements-- by assigning to an identifier an expression value. Using the rules for identifier and number and +, - and = that we've established previously, I've written down 4 possible utterances. The question for you, multiple, multiple choice, is which of these are statements according to this grammar? Check all that apply.

1.13 13. 08_s_statements

All right, let's go over the possible answers together. We say that a statement has to be an identifier followed by an equals sign followed by an expression. Lata, l-a-t-a, is definitely an identifier using our previous rules. That's a collection of upper and lower case letters that may include an underscore, but the underscore can't come first. This is an identifier, and expression can go directly to number, and 1 is a number, so this is totally valid. Here we have Mangeshkar = 19 + 29. Well, Mangeshkar is longer, but it's still an identifier. It's a collection of letters, and it may have underscores, but it doesn't. And then we have the equals sign. That matches. And now we're going to use this rule here, expression goes to expression + expression to get this + sign, and then expression will go to number for 19, and expression will go to number for 29. I like this. Here we have Lata = Lata + 1. This one is tricky because it's totally valid in Python, but it's not valid in this grammar. I haven't said that expression can go to identifier, so Lata is an identifier. The equals sign is an equals sign. But over here I need something like 1 + 1. I don't yet have a rule that would allow me to have Lata + 1. If I had this mysterious fifth rule, then this would be in the language of the grammar. But I don't yet, so it isn't. And then over here, Mangeshkar = 25,000 - 1, that's an identifier =, and now we're going to use this rule here, expression - expression. This totally works out. Lata Mangeshkar is a famous Indian playback singer. She's recorded over 25,000 songs, a world record for quite some time, and she's also received India's highest civilian honor.

1.14 14. 09_q_optional-parts

Often both natural languages and programming languages have optional parts that don't have to appear in every utterance. For example, "I think" is a totally valid sentence, but so is "I think correctly," where we've added the adverb "correctly" that modified the verb "think." You don't need to do this, though. You can leave it out. The sentence on the left is perfectly fine. We're going to want to represent this optional construction, this optional adverb, in our formal grammar. Here on the bottom half of the screen I've drawn a slightly more complicated grammar. Sentence goes to optional adjective, subject, verb. Subject and verb work a lot like they did before. This time our subjects are either William or Tell, and our verb is either shoots or bows. But our optional adjective can either be the adjective "accurate" or it can be nothing. It disappears. We can either leave this blank, or if you like, we could write that same epsilon we used to have there when we were talking about finite state machines that means the empty string or no input. Looking at this grammar, I have a quiz for you. Fill in the blank. Starting from sentence, how many valid utterances are there? How many strings can we make in this language? How many different things can this formal grammar produce?

1.15 15. 09_s_optional-parts

The answer is 8. Let's go through it together to see why. Sentence only has one re-write rule. These are sometimes called "production rules" as well. Only one production rule. Only one thing to do with it. We're always going to get started with optional adjective followed by subject followed by verb. This could go to accurate or nothing. The subject could go to William or Tell. The verbs could go to shoots or bows. We have two choices here, two choices in the middle, and two choices at the end. When we multiply that out, we get eight possibilities-- accurate William shoots, accurate William bows, accurate Tell shoots, accurate Tell bows, William shoots, William bows, Tell shoots, Tell bows--eight possibilities in total. William Tell was a 15th century Swiss folk her. Famously, he did not bow. He was requested to bow to the hat of a city leader or some such. In exchange, he was asked to shoot--or offered to shoot--the apple off of his son's head, using a crossbow. This is not a crossbow. At least two of the sentences above--William Bows, Tell bows, accurate William bows, accurate Tell bows--four of them didn't happen as far as the story goes. Even if you're not familiar with this story, you may have heard the overture. It's actually quite a bit of fun.

1.16 16. 10_l_more-digits

In the past few units we've used regular expressions to classify or layout sets of strings. It turns out that grammars can encode all of these regular languages or regular expressions that we've been working with previously. Here I've written a grammar for number that's going to recognize exactly the same language, exactly the same set of strings as the regular expression above. We can rewrite number to be a digit followed by more digits. This construction is meant to mimic or get the same idea as this plus. We need at least one, but we could have some more. Down here I've just listed out all of the digits longhand. Well, I haven't on this particular page but you could imagine I could write out More digits is where a lot of the action happens. One possibility is that we have one digit followed by potentially even more, and another is that we give up. I could write epsilon. This means more_digits goes to nothing. For example, down here at the bottom, I have a derivation starting with number getting to the number 42. Number goes to digit more-digits, using our first rule--rule number one. Then we're going to turn more digits into digit more_digit, using rule number 2. Then--oh my gosh, classic professor mistake--you cannot take me anywhere. Shwoop--oh look. It wraps. That that amazing. The from digit digit more_digits, we're going to turn more_digits into the empty string, using rule three. Now we're left with digit digit nothing. I'm going to turn that second digit into a 2--this empty string isn't really there. Then I'll turn that first digit into a four. Huzzah.

1.17 17. 11_q_grammars-and-regexps

That was not a proof, but it just so happens that grammars are strictly greater, have more expressive power than regular expressions. Anything we could specify with a regular expression, we can also specify with a grammar. Let's try it out. Here's another regular expression: p plus followed by i question mark. I'm going to start writing out the grammar for it. Uh oh. Here I've almost written the grammar but I'm going to need you to help me out. I've left two blanks. What I want you to do is fill in the blanks so that the regular expression and the grammar have the same language, accept exactly the same set of strings.

1.18 18. 11_s_grammars-and-regexps

Now let's take a look. This was a potentially tricky quiz. Don't feel bad if you had a little trouble with it. We want this pplus nonterminal to correspond to one or more Ps. No matter which branch you take here, you're definitely getting at least one P, but we need you to have the chance to get more. This option, pplus goes to P, corresponds to picking just one. Pplus goes to P followed by pplus--this recursive rewrite rule Gives you the possibility to have as many as you want--possibly even infinite. We've seen that before with expression goes to expression plus expression. Over here, this i-question mark is supposed to relate to i-option. One possibility is that you don’t want

to write an i at all, but another possibility is that you do. Here this question mark means have either 1 or 0, this the rule for having 1, and this is the rule for having 0. Some strings in this language or p, pi, ppi--that sort of thing. In fact, if you have a bit of a musical theater bent, you could imagine context-free grammars singing across to regular expressions the classic line from Annie Get Your Gun, "Anything you can do, I can do better. I can do anything better than you." To which, classically, the opponent replies, "Can you bake a pie?" "No." "Neither can I." But in fact, any regular expression relating to pi or something similar, as we've seen above, can be handled both by our regular expression protagonist and our grammar protagonist. Just a fun little interlude.

1.19 19. 12_l_context-free-languages

Using formal terminology, regular expressions describe regular languages. Remember that officially a language is a set of strings. These grammars that we've been introducing describe something new-- context-free languages. For now, context free just means that if you have a rule A can be replaces by B in your grammar and you have an A, you can replace it with a B regardless of what this nearby context is as long as you hold the context the same. If I can use the rule now, I can also use it if there's no x on one side, no z on the other, if there's nothing on the left, if there's nothing on the right, if there are flying monkeys on the left and flying monkeys on the right regardless of the context I can apply this rule. The set of languages that can be described by re-write rules or grammers like this are called "context-free languages." All the formal grammars we are going over describe context-free languages-- a more powerful set of string than our old friends the regular languages. Here I've drawn a little chart showing three different regular expression forms and the corresponding context-free grammar forms on the right. If you had a regular expression concatenation-- regular expression a followed by regular expression b-- we could build a grammar than did the same thing just by putting a and b next to each other on the right-hand side. If you had a star--regular expression a*--we could do the same thing-- the same effect of having zero or more a's--with two rules. Either we have nothing or we have an a followed by zero or more a's. Finally, here we have regular expression disjunction or choice--either a or b-- that's easy to write with two separate grammar rules. G goes to a or g goes to b. For all three of these cases and regular expressions there is something corresponding we can do in the world of context-free grammars. Now, I haven't shown you how to do something like a+, but remember that a+ is just a a*. I've already shown you how to do concatenation and the star. In theory, you could compose it over here as well. At this point it might seem like regular expressions and context-free grammars are equal, but they are not.

1.20 20. 13_q_parentheses

Consider this grammar. It's time for you to test your knowledge by thinking about which of these five utterances are in the language of the grammar. Check all the apply. Which of these give strings could be derived from P in this grammar. Here are the two rules in this grammar, P goes to open parenthesis, P closed parenthesis, or P goes to nothing and can be erased. Here I have various combinations of open and closed parentheses. Stare at it for a while and tell me which of these five sequences of strings could be derived from P in this grammar. Check all that apply.

1.21 21. 13_s_parentheses

All right. The question was which of these five strings are in the language of this grammar. The first one looks pretty good. I just apply rule 1. P goes to open P close and then rule two to get rid of the P, and I'm in like Flynn. How about this? Well, I'm going to apply rule one twice. I'll end up with open open P close close, and then I'll use rule two to get rid of the P. Similarly, over here I'm going to apply rule one three times. Open open open P close close close. Then I'll apply rule two to get rid of the P. Totally in the language of our grammar. But this--tsk, tsk, tsk--this is going to be problematic. Just an open parenthesis. If you look over here, there's actually no way to make just a lonely open parenthesis that does not have a matching close parenthesis. There's only one rule that introduces the open parenthesis, and it makes just as many close parentheses at the same time. This cannot happen. Here this looks good. These parentheses match, open close open close, but if you take a look at this grammar, there is never a way to have an open after a close. I can apply this rule as many times as I want in this pattern, but I'm never going to end up making something like this--not as written. We could write a grammar for this, but not currently shown. This sort of system is known a balanced parentheses. The parentheses are balanced, because the number on the left equals the number on the right. One to one, two to two, three to three. In fact, you could imagine some sort of scale that'd have to be exactly equal-- two openings, two closings. Balanced parentheses. This is a super-important problem in computer science.

1.22 22. 15_l_intuition

We've proved that one particular regular expression didn't do the job. It was too permissive, but that doesn't mean that there isn't a smarter, super tricky regular expression that would capture balanced parentheses. However, it just so happens that there isn't. It turns out that it's impossible to capture balanced parentheses with a regular expression. A formal proof may be presented in the supplemental material, but for now I'm just going to give the intuition. Here's what we want: an open parenthesis followed by a close parenthesis, each repeated the same

number of times. This intuition or this notation is meant to remind you of mathematics. X-squared is just x times x, so open parenthesis raised to the power of 2 would just be two open parentheses next to each other in a string. What we really want is open parenthesis to the power of N, closed parenthesis to the power of N, but this has to be the same N in order for the parentheses to be balanced. Unfortunately, all we can write with regular expressions is open parenthesis star close parenthesis star. In regular expressions, these two stars need not be the same. In fact, if you think about the finite state machine interpretation of regular expressions, remember when we were simulating finite state machines the only thing we really had to remember was where we currently were and what the input was. We didn't really remember where we'd come from. In order to matchup the same number of opens and closes, we'd have to where we came from. Regular expressions just don't have that kind of memory. With regular expressions, I can say zero or more open parentheses followed by zero or more close parentheses, but those two numbers don't have to be the same. Regular expressions can't always remember to different numbers and force them to be equal. This notion of balanced parentheses is worth paying a lot of attention to, because balanced parentheses are everywhere in HTML and JavaScript. These tags for beginning bold and beginning italics, ending italics and ending bold, have to be perfectly nested for valid HTML. It's as if the beginning bold b were some sort of parenthesis and the ending bold b were a closed parenthesis. Then the italic tags maybe were angle brackets or some other type of parentheses that really had to match up. These have to be perfectly nested, and they have to open and close each other in tandem. Open the bold, open the italics, close the italics, close the bold. We can see the same sort of thing in a Python or Javascript assignment. These two parentheses match up. So do these two, and they're perfectly nested. Here we see an example of malformed HTML. The tags, the parentheses of HTML, don't match up properly. Let's see what happens when I try to draw that same sort of diagram. Oh, there's no way to connect these two i's without crossing the lines and crashing over each other. This tells us that the parentheses do not match. We could also look at it like this. If we view it as a mathematical formula, the normal parentheses and the square parentheses don't open and close in the right order. In order for something to be valid HTML, tag openings and tag closings must be perfectly nested. This is not valid HTML. For our web browser to work correctly, we're going to need to tell the difference.

1.23 23. 16_q_well-balanced

Let's check out knowledge of this with a quiz. Here I've written five different utterances or five different sentences variously in JavaScript or Python or in HTML. In this multiple multiple choice quiz, I'd like you to indicate to me which of these have well-balanced parentheses and/or HTML tags.

1.24 24. 16_s_well-balanced

Let's start here--palace minus walk. This is a lot like 5 minus 3 if "palace" and "walk" are identifiers. Here the parentheses are balanced. We like this utterance. Here we start a bold tag and right "naguib," but we never close off the bold tag. This is not balanced. Here we start an italics tag and write "mahfouz," and then we close off the italics tag. This is well-balanced. Here we write open "pulitzer" plus open "prize" close close-- open parenthesis, open parenthesis, close, close. This one matches there. This one matches there. These are well-balanced. You might think there's no reason to put "prize" in parentheses, and you might not if you were writing, but the grammar--the language-- allows it as long as the parentheses are balanced. Over here this one is particularly tricky. We have a close parenthesis followed by "1956" followed by an open parenthesis. Now, the number of open and close parentheses is the same, so you might think it's balanced, but they're in the wrong order. We need to do the opening before the closing. This does not match. "Naguib Mahfouz" is a relatively famous Egyptian writer. He wrote Palace Walk. It's set in about 1917 around the end of World War I in Egypt in Cairo. It details what goes on with Al-Sayyid Ahmad Abd al-Jawad and his family. My best attempt at doodling Egypt or Cairo is someone next to a pyramid. You can tell it's a pyramid because it's clearly labeled.

1.25 25. 17_l_extracting-information

Now that we have an understanding of formal grammars, we're going to need to use them to understand or describe HTML and JavaScript. We're going to design together formal grammars that exactly capture HTML and JavaScript or at least interesting subsets of them. If we were trying to understand English sentences, there is a special kind of diagram we could make. Here I have a very high level partial grammar for English. A sentence has a noun phrase, a verb phrase, and an object. A noun phrase is just a decorated noun, like maybe a noun with an adjective in front of it. A verb phrase is a decorated verb, maybe a verb with an adverb in front of it. Then the object could be a noun phrase or a noun or something more complicated. For example, here is a sentence: "Wes reads Romance of the Three Kingdoms." Now, there are various different ways to diagram English sentences. We'll do something relatively simple. We just want to indicate which part matches which grammatical form. Our noun phrase is going to be "Wes." Our verb phrase is "reads." The object is "Romance of the Three Kingdoms." We could divide it up so that the object falls under the phrase or that we draw these slightly differently, but mostly we want to know which part goes where. Who is doing the reading? Is it Wes or the book. This sort or diagram will help us figure that out. In fact, in formal languages or with a grammar like this, there is a more common way, a more pictoral way of representing these sentences. We can draw something that looks a bit like an inverted tree. I'm going to take each one of these productive rules and puts its left hand-side at the top, and then give it children, or branches, based on which rule is chosen and what's available on the right. Here we have a sort

of a pyramid-like shape, or what's officially called a parse tree. Remember that syntactic analysis is sometimes called parsing. This might not look like a tree, but what you want to do is imagine the tree growing upside down. Here is our tree, and at various places--although it's starting with the trunk or the root-- it branches out, and all of these intermediate parts in the tree represent none terminals in the grammar. All of the leaves right at the end represent terminals or tokens in the final sentence or utterance. This is called a parse tree by longstanding tradition. We draw them upside down, starting at the root with the start nonterminal. While we're here, you too should consider reading Romance of the Three Kingdoms. Written in the 14th century, it's a Chinese historical novel about the end of the Han Dynasty. Good stuff. Here's one of our favorite grammars from before--a simple arithmetic expression grammar. Let's say that our input is 1 plus 2 plus 3. Here is a possible parse tree. We start at the top, at the root, with our nonterminal expression. Each time we have a number of children equal to the number of symbols to the right of the arrow. If I'm using expression goes to expression plus expression, I have one node with three children. I need to get 1 plus 2 plus 3, so I'm going to need to use this first rule at least twice--once to get this plus sign and once to get that one. Here I'm using it once--one instance of rule one. Here I'm using it twice--another instance of rule one. You'll note that they overlap. The first time we use this rule we started drawing this part of the tree. This expression node was the result. The second time, it was the thing we were choosing to expand. This is normal. This represents the recursive structure of a grammar. In a parse tree like this, these parts at the that have no children are called "leaves." The parts in the middle are called "nodes." Sometimes the leaves are called "leaf nodes," which is a bit ambiguous. These are always going to be terminals. The intermediate nodes are always going to be nonterminals.

1.26 26. 18_q_fill-in-the-tree

Here I have a grammar that we've seen before, but I've abbreviated it still further. Instead of identifier, I've just written "id," and instead of number, I've just written num. This sort of abbreviation is all too common in computer science. Programmers can't help it. Unfortunately, it's a bit of a vice, because while it's easy for me to see what's going on it can be hard for someone reading it later--myself included--to tell what my variable names mean. Whenever possible, you should take the time to make your variable names as descriptive as possible. Add lots of comments. It turns out that 90% of software is maintenance. Just like a popular book is read many more times than it's written, a program is read or maintained or executed many more times than it's written. It's worth taking the time to do it right the first time. That said, clearly I'm not, so this must be one of those do what I say and not what I do back professor sorts of maneuvers. Regardless, back on track. Here's the grammar. Statement goes to identifier equals expression, and expression is the same sort of arithmetic we've seen before. We're not going to consider this particular sentence "yggdrasil equals 6 minus 5. I've written out a parse tree--a bit squashed at the bottom but still recognizable-- for this phrase "yggdrasil equals 6 minus 5," but I've forgotten two labels. It's quiz time. Fill in the blanks. What would have to go in each of these blanks in order to this parse tree to match this input utterance according to this grammar?

1.27 27. 18_s_fill-in-the-tree

Well, our rule for statement is statement goes to id equals expression, so this box has expression in it. Then expression goes to expression minus expression. Since this number was 6, this number must be 5. By the way, Yggdrasil is worth knowing about. It's the World Tree in Norse mythology. My drawing of it is clearly not to scale.

1.28 28. 19_q_ambiguity

One trait shared by programming languages and natural languages is ambiguity. Consider the sentence "I saw Jane Austen using binoculars." It's relatively clear what this means, right? Here's me. I have binoculars or perhaps pants, but let's image that they're binoculars. I'm looking through them. Over here in the distance I see Jane Austen. We can tell it's Jane Austen, because she's in a park. Mansfield Park, let's say. I am using the binoculars to see Jane Austen. However, if you think about it there's an alternative implementation or an alternative interpretation of this sentence-- another way to look at it that's also perfectly valid. Here in this alternative interpretation, I see with my naked eye Jane Austen. She is using binoculars to look at something else. Maybe she's spying on an Abbey over here. Let's call it Northanger Abbey. Actually it looks a bit more like a barn that was badly painted, but let's imagine. Both of the interpretations are correct or the sentence is ambiguous. It's not clear whether using binoculars modifies Jane Austen. Is she the person using binoculars? Or whether using binoculars modifies "I saw." The picture on the left that's how I'm seeing. It's quiz time--a quiz on ambiguity. Consider the expression 1 minus 2 plus 3. Keeping ambiguity in mind, if this were a Python or JavaScript or even just on the whiteboard or on a piece of paper in mathematics, this is a mathematical expression. What might it evaluate to? I've got four choices in this multiple multiple choice quiz. Check all that apply.

1.29 29. 19_s_ambiguity

Let's take a look. Here is one possible reading. I'm going to take 1 and subtract 2 so I have negative 1 plus 3, and that's going to give me 2, so 2 is definitely an answer. If I do 1 - 2 first, I'll have -1 plus 3. However, what if I

want to do this 2 + 3 first? I'll have 1 minus 2 plus 3 or 1 minus 5. That's going to give me -4. Zero and 4 aren't really reachable using this expression, but depending on whether you do the subtraction first or the addition first, you could either get 2 or -4. This sentence is ambiguous. We're not sure what it means or it has multiple meanings just like the Jane Austen binoculars example from before.

1.30 30. 20_l_to-the-rescue

As the previous example suggests, we can use parentheses to control ambiguity. Here I've taken our expression grammar from before and updated it with a new rewrite rule. Now expression can be replaced by open parentheses, expression, close parentheses. This should totally remind us up the P goes to open P close rule from before for balanced parentheses. It's going to behave mostly the same way. With this new grammar, both of these utterances-- open 1 minus 2 plus 3 and 1 minus open 2 plus 3 close-- both of them are in the language of this grammar. However, while both of these are okay, so is this one. It doesn't forbid the ambiguous one. It just allows us to have more precise renditions. We've solved some of the problem. If we're thinking ahead, we can use parentheses, but we're still allowing ambiguous phrasing. Here I've drawn our favorite utterance from before--1 minus 2 plus 3-- and I've written out two different parse trees for it. This one corresponds to 1 minus 2 plus 3. At the top level we're subtracting, and then the 2 plus 3 are grouped together. This one corresponds to 1 minus 2 plus 3. At the top level we're adding, and the 1 minus 2 is grouped together. Formally, we say that a grammar is ambiguous if there is at least one string in it-- Here I've drawn very stylized versions of the parse trees. If you can find even one string for which this is true, officially the whole grammar is ambiguous.

1.31 31. 21_q_finding-ambiguity

Here I've written a number of strings and a number of utterances. We're going to use it to check our knowledge of ambiguity. We've got seven strings here, and what I'd like you to do in this multiple-choice quiz, is check each one that has more than one parse tree in our expression grammar.

1.32 32. 21_s_finding-ambiguity

Let's go through these and find an example I'm going to focus my attention here. This could either be 1 minus 2 minus 3 or 1 minus 2 minus 3. This ends up being 2--1 minus -1. This ends up being -4. Very different answers. The parse tree for this interpretation looks a bit like this. It's left heavy with this other subtraction--1 minus 2--down and to the left. Over here the parse tree would be right heavy. Now, I'm not drawing all the nodes in the parse tree just to save a bit of time. Mostly, I'm highlighting its shape. This is an example of a string that has more than one parse tree in our grammar. Similarly, this bigger string has more than one parse tree in our grammar. It might even have four different parse trees, possibly even more. You'd have to try that and find out. Notably, it contains 1 minus 2 minus 3 as a subpart. Even if this -4 at the end is perfectly unambiguous, it still has exactly the same problems we saw here. It still has at least two parse trees. It's even worse than its friend two lines above. If you look carefully, all of the other examples don't demonstrate ambiguity. Each one of these has exactly one parse tree. Even this big complicated thing at the bottom only has one parse tree. It's perfectly balanced--1 minus 2 on the left, 3 minus for on the right.

1.33 33. 22_l_grammars-for-html-and-js

Since both HTML and JavaScript have some essential ambiguity, we needed to handle that before we could move farther with our web browser. But now that we know a bit more about grammars and ambiguity, we can actually move onto making grammars for HTML and JavaScript. Just to remind you, here's an example HTML utterance, "Welcome to my webpage!" Here I've drawn a relatively simple grammar for a surprisingly large subset of HTML. One of the first challenges that we'll have to deal with is that a webpage can have a list of words, like "welcome to my webpage." Our starting nonterminal, HTML, has a recursive structure. Using this rule, HTML goes to element HTML, we can apply rule one over and over again to get as many elements as we need. Maybe one, two, three, four for welcome to my webpage. Then eventually when we're done replace with epsilon or the empty set. Element can either be a word, like "welcome to my webpage," or the beginning of a nested tag. Here tag-open and tag-close are a lot like open parenthesis and close parenthesis. Whenever we make one, we're going to make the other. Then in here tag_open is <word> and tag_close is </word>. For example, this part matches a tag_open, and this text matches a tax_closed. Actually, as it stands, I'm only allowing a single word to be inside any tag. Inside these bold tags, we've got quite a lot of stuff going on. I'll show you the power of a recursive grammar. Watch this trick. Now at the top level, an HTML document is a list of elements, as many as you like. Each one of those elements may themselves be a tag, and inside tags we have another list of elements--another, in essence, entire webpage. Here at the top level, we just have one--tag_open for bold. But inside it there are four elements--the word welcome, the word to, the tag_open for my, and the word for webpage. I'm going to draw the parse tree for this utterance using our grammar. I'm going to use "elt" to abbreviate for element. I'm going to use "to" for tag_open and "tc" for tag_close. Now, I've only drawn or sketched out a small portion, maybe about half, of this parse tree, but we're still going to be able to see things match up by comparing the leaves-- remember a leaf is an node without any children-- to what we saw in the original string. Here my first leaf, working my way down

and to the left, is this open left angle bracket, and that matches up here. My next is this b, which matches up here--match up there. Then after that tag_open, there is an HTML, which is an element list, and the first one is the word "welcome." The next one is the word "to." Now, although this tree structure is cumbersome for us, it's very convenient for computers, because it tells us exactly which part of the tree to draw or to apply in a certain manner. It might not be obvious which word should be bolded or which word should be italicized, but if I have a tag_open and a tag_close, this entire subtree is influenced by this tag. How much of the webpage is bolded? Exactly this part over here. The bold's great, great uncle or something like that. Go up a few, over one to the right, and then back down. We're going to use this special structuring to help write our web browser.

1.34 34. 23_q_making-valid-html

Here I've written two partial HTML utterances, but I've left some blanks. This makes it entirely reasonable for you to come in and put things right by adding a small number of characters--the smallest you can-- to fill in each blank and make it valid HTML according to previous grammar. Remember that mostly means the tags have to match up correctly.

1.35 35. 23_s_making-valid-html

Let's go through the answers together. For sentence 1, this open tag for bold--bold Baroque music!-- must be paired, must be balanced with a closed tag for bold. That looks like this. It's the left-angle slash token. Then the bold closes off. Down here Johann Sebastian Bach was-- this looks like the beginning of an italics tag, and you see that we're closing it off later. For this to be valid HTML, we really need to make this a well-formed tag. Johann Sebastian Bach was German. In fact, he was born around 1685 and is known for his fugues. They repeat in a structure that's actually surprisingly similar to regular expressions or context-free grammars. More on that later.

1.36 36. 24_l_revenge-of-javascript

Now that we know how to specify grammars for well-balanced expressions and arithmetic and well-balanced webpage tags in HTML, we're going to return our attention to JavaScript. JavaScript is actually very similar to Python. Just like I showed you a formal grammar for HTML, we're going to work our way up to seeing a formal grammar for JavaScript. But before we get there, I just want to make sure that we really understand how JavaScript programs are interpreted. I'm going to show you a few more in Python and in JavaScript for comparison. Over here on the left, I have a Python function that computes the absolute value of its integer argument. If you give me a negative number, like -5, I will return positive 5. If you give me a positive number, like +9 million, I will return 9 million. The return value of this function is always either a zero a positive number. Now I'm going to write the same thing in JavaScript to provide for a comparison. Everything I've drawn in blue is a special keyword or punctuation mark used by the language. For example, to define a function in Python we use "def." In JavaScript, we write out the word function, but then it's still our choice what to call it-- I'm called it "absval" in both cases-- and how many parameters it should receive and what they're names are. In both cases, we have one parameter named x. In Python we use colons and tabbing to tell what the body of a function is, what the then branch for an if is, what the else branch for an if is. In JavaScript, we use curly braces and closing curly braces to denote this sort of lexical or syntactic scope. This is sort of curly brace 1, and it matches up with closed curly brace 1 over here, But in general the logical structure, the flow or the meaning, is the same. In both cases we check to see if x is less than 0, and we return 0 minus x in that case, or just x alone in the other case. One of the most important operations in any language is printing out information, displaying it to the screen so that we can see the result of computation or just to help us debug. In Python we use the print procedure. We pass it a bunch of strings. Here I'm adding together the strings "hello" and exclamation mart to make a very enthusiastic greet--"hello!" Over here on the right, I'm showing the same thing in JavaScript. The equivalent of "print" is "document.write" or perhaps just "write." In this class, we'll almost always abbreviate it down to just "write" to save space. If you're familiar with object-oriented programming, which is not required for this class, you might guess what the dot is about. We might talk more about that later. One of the key differences, though, is that all of our JavaScript functions have to have these open and close parentheses like a mathematical function has parentheses around its argument.

1.37 37. 25_p_up-to-ten

Here I've written a Python procedure called uptoten. If the argument you pass in in less than 10, it returns that argument, so if you pass in 5, it's going to return 5. But it clamps values at 10. If you pass in 15, I'm just going to return 10. I just showed you the similarity between Python and JavaScript before. I would like you to write a JavaScript program. Because we're going to be write a web browser that deals with JavaScript as strings, I would like you to write your JavaScript program as one big string constant. Call it "javascript" and submit via the interpreter. You should declare a JavaScript function called uptoten that takes and argument x and does exactly the same thing as this Python one. Translate this Python code into JavaScript and then put it in this big string.

1.38 38. 25_s_up-to-ten

Let's go through it together. The JavaScript equivalent of def is function. We write out the keyword function. Then we pick a name for our function. I asked you to call it uptoten. We mention all of its arguments, and then we use an opening curly brace instead of colon. The word if remains unchanged. Less than is unchanged. But again, instead of a colon we're going to use an opening curly brace. Here is our answer. If x is less than 10, we return x. Otherwise, we return 10. Again, the big differences are these curly braces, which must be balanced. They have to match up, but we already know how to do that instead of colons and indenting.

1.39 39. 26_l_universal-meaning

It's worth noting that although we changed some keywords-- they had a different spelling where it's almost as if we had to translate them into a different language-- the underlying meaning was the same in both languages. We can translate programs, functions between Python and JavaScript provided that we know both of them. In linguistics, people will sometimes talk about a universal grammar as there may be some grammar that would sit behind and describe Python, JavaScript, English, and French. I don't want to get into that for natural languages, but we will see either in this class or in subsequent ones, that for computer languages like Python or JavaScript, C and C++, Visual Basic, C Sharp, OCaml, Free Basic, they're all the same in a very strong sense. They're all Turing-complete. I'm not going to much detail now, but it suffices to say that Python and JavaScript are equally powerful. Any thought that I can think in one I can also think in the others. Now I'm going to bite the bullet and actually write out a partial grammar for JavaScript. This is only a partial grammar because it only handles expressions, and it only handles some of the expressions. An expression can either be an identifier--like a variable name x or a function like sine or absval, a number--1, 3.5, -2, a string--like "hello" in quotes, literal constants--like True or even False. These are the equivalent of numbers, but for the Booleans there are only two numbers--True and False-- and then a large number of binary operators. If you have two expressions, you can add them together, subtract them, multiply them, divide them--watch out for division by zero. You can compare them to see if one is less than the other. You can compare them to see if they're exactly equal. If they're Booleans, you can check to see if the first is true and the second is true. Just a little note--if you haven't run into this before-- two ampersands means "and" and two vertical bars means "or." In Python where you might write and or or, in JavaScript you use these binary operator symbols instead. Just as we have less than here, there'd also be greater than, greater than or equal to, less than or equal to, and many more. This JavaScript expression grammar should be familiar to you, because aside from some minor details of spelling like the equal signs here or the capital word "TRUE" to mean true. It should be very similar to Python.

1.40 40. 27_q_fill-in-the-javascript

Now it's quiz time just to show that a lot of your knowledge about Python's grammar and the grammar fragment I showed you before, will translate into you being able to write JavaScript expression. I've written here four JavaScript expressions--a, b, c, and d-- and each one contains a blank. I'd like you to fill in each blank with a single token, a single terminal, that causes the entire expressions to evaluate to True. Try it out. This is a bit of a puzzle, and you'll sort of have to work backwards. What would this blank have to be for 1 plus a blank to be equal to 2?

1.41 41. 27_s_fill-in-the-javascript

Let's go through it together. Just like in Python, JavaScript supports string concatenation using addition. I didn't tell you about this explicitly, but I did tell you to use your Python intuition. We're going to fill in this blank with one as well, but this one is a string O-N-E. The string P-H plus the string O-N-E will equal the string P-H-O-N-E. Over here this is perhaps the trickiest on the screen. In order for this whole expression to be True, both of these sub-expressions, both of these conjuncts if you prefer mathematics have to be true. X must be equal to 8 and this complicated math over here must also be true. Well, if x is equal to 8 then x divided by 2 is 4. X divided by 2 equals 4 should return True. This'll be true on the left and true on the right. Finally, over here, what's anything that we could put in this blank that would possibly make this expression turn true? J\ust write out the Boolean constant True.

1.42 42. 28_l_javascript-grammar

If we make analogy with natural languages, if expressions like "tmp" are noun phrases, then operators like "assignment" are verbs and entire statements like tmp gets 5 are sentences. This whole thing together is a statement. It involves two expressions--tmp an identifier, 5 a number-- and an operator--the assignment operator. Just as I might write the English sentence, j becomes 3--perhaps it's her birthday today, I can write the JavaScript assignment statement, j becomes 3. J is assigned the value 3. Similarity between the syntactic elements--the subject, verb, object, and punctuation. Subject- verb, object, and punctuation. Identifier, operator, expression, semicolon is relatively direct. Now that we have an intuition for statements, lets add them to our JavaScript formal grammar. One of the most common kinds of statements is the assignment statement where we have an

identifier on a left-hand side of an equal sign and then an arbitrary expression on the right--x becomes equal to 5. Another kind of statement is the return statement. At the end of absolute value or Fibonacci or factorial or almost any function or procedure we want to return with the final value, return an arbitrary expression that becomes the value of the function. There are also statements that influence what gets executed and under which conditions. Formally, these statements are said to refer to control flow because they guard how execution flows through your procedure. The if statement checks to see if a certain expression is true, and if it's true, then we execute the then branch. There is also an if then else. If the expression is true the then branch else the else branch. I haven't said what a compound statement is, although since we've seen JavaScript before we're going to guess that it involves those curly braces. Let's write it out right now. A compound statement is an opening curly brace, a closing curly brace, and some statements in the middle. This is a list of statements terminated by semicolons. I'm going to show you how that works. Here I've added a recursive grammar rule. Statements can be one statement followed by a semicolon followed by as many more statements as you like, or you can decide that you're done and replace it with nothing. Or if you like, we'll draw the epsilon there to mean the empty string. For the subset of JavaScript that we'll be handling in this class, this grammar gives more details about what's possible. In addition to expressions, we have statements that build upon expressions.

1.43 43. 29_q_valid-statements

Here I have written six possible JavaScript statements. I ask you which are valid stmt according to our grammar. That is, if you start from the statement nonterminal, which of these six can you derive using all the rules that we've seen in our grammar so far. This is the happy-fun quiz.

1.44 44. 29_s_valid-statements

Well, x equals three is certainly a statement. We've seen examples like this before. You might have thought that a statement has to end in a semicolon, but if we scroll back to our grammar a bit, we see, no, a statement can be identifier equals expression, and those semicolons come in when we have multiple statements, like in a compound statement block. Similarly, x equals 3 divided by 2. Any expression can go on the right-hand side of the equal sign, and we saw that our grammar for expression earlier included binary operators like division. This totally works. This series of tokens, if x is less than 5 then open curly brace x equals x plus 1 curly brace looks very convincing. It's almost right, but it's missing a semicolon. Inside these compound blocks we really need those. This one doesn't work. The one right below it has that semicolon, so it can be derived from "stmt" in our grammar. Over here inside our compound statement we have two smaller statements-- x is x plus 1 and y is 3. Each one of which is an individual statement, and each one of which is terminated by a semicolon. That's the rule we established earlier. Finally over here we have an if-then-else statement. This was a little tricky. In Python conceptually this might work out, but in JavaScript we explicitly need the else token. This one is not derivable from stmt, even though the rest of it is well-formed.

1.45 45. 30_l_javascript-functions

So now we've seen how to do expressions in our formal grammar for JavaScript as well as statements like the assignment statement or the return statement or if-then-else, but we are not done. There is 1 last key element of a computer program. We need functions, and this splits into 2 parts. We need to know how to declare functions, to list how many arguments they take and what their bodies are, and we also need to know how to call functions in expressions. We know from experience that a Python program is basically just a list of statements and function definitions. Luckily for us, a JavaScript program is exactly the same thing, also a list of statements and function definitions. We sometimes call statements and function definitions elements because they're the key elements of a program at the top level. Let's get started on the highest level of our JavaScript grammar. A JavaScript program is either a single element, possibly followed by another JavaScript program, which is another element, so this ends up being a list of elements. And at some point, we get tired and stop writing in elements. An element can either be a function definition or an arbitrary statement, just like in Python. A function definition in JavaScript starts with the word function instead of the word def. There's an identifier. That's the name of the function. You must have these parentheses. You can declare some optional parameters in there. Maybe it's a function of one variable x. I'll have to show you how to do that in a minute. And then the body of a function is just a compound statement, which we've already seen. That's a list of statements terminated by semicolons. You can also just have statements at the top level, like print hello if you're testing things out. Any statement you want followed by a semi-colon. When we're giving the definition for a function, it can have any number of formal parameters that we decide. It could be a function of x, y and z. It could just be a function of a, or it could be a function of no parameters. That's why this is optional, opt. An optional parameter declaration could either be some actual parameters-- this is the common choice--or nothing. If we're going to have 1 or more parameters, here's how it goes. You can declare the names of those variables. It's a function of x, y, z. And after each one, you have to have a comma until you get to the last one, which is not followed by a comma. We haven't hinted on this previously, but this is kind of a cute gem in context-free grammars. Note that our statements are terminated by semi-colons. In a compound statement, every statement is followed by a semi-colon. By contrast over here with parameters, they're separated by commas, so the last one doesn't have a comma after it. Pretty cute. And we have all the rules for expressions that we used to have before, but now we're going to add function calls. When you're calling a function, you have to give the

function name, sign, print, abs val, and then you pass in some number of arguments. There could be none, which is why this says optional, or there could be 1 or more, and you're going to see this play out in a very similar pattern to opt params. The parameters in a function declaration are always just names like x, y, or z, and then you get to refer to them in the function body. But when you're calling a function, you can put in actual values. You can call sign of 3, so we can't just reuse params. We need to make a new set of rewrite rules for actual arguments. The arguments could be nothing, or it could be 1 or more arguments. If it's 1 or more arguments, they're going to be expressions separated by commas. Down here we had function calls, which are expressions. You could take sign of x and divide the result by 2. And up here at the top, function definitions. These happen more rarely. These are elements. This is actually pretty much it. This is the majority of our JavaScript grammar. The real devil is in the details. For example, you may have noticed that nothing in this grammar prevents you from declaring sign as a function that takes 1 parameter and then later on calling it with 2 parameters. Uh-oh. More on that later.

1.46 46. 31_p_translating-javascript

All right, so here I have written out Number 1, a function definition of a function mymin, given 2 arguments a and b, if a is less than b we return a. Otherwise we return b. We're returning whichever one is smaller. Over here, another function definition for a function called "square." It takes in a variable x and returns x times x. And then over here, number 3, a statement. We're calling write, or if you prefer, document.write, to display the result of calling mymin on the square of -2 and the square of 3. In essence what we're trying to figure out is which is smaller, -2 squared or 3 squared? And then we're going to print out that result. Your mission, should you choose to accept it, translate this to Python. Submit via the interpreter. Write a Python program that does exactly the same as this JavaScript.

1.47 47. 31_s_translating-javascript

Well, to see how this plays out, first I'm going to show you what the JavaScript actually does. Here I've typed it out. Here's our webpage with embedded JavaScript in it. We declare the function mymin of a and b. If a is less than b, return a. Otherwise return b. We declare the function square to take x and return x squared. And then we called document.write of mymin of the square of -2 and the square of 3. Well, -2 times -2 is 4. and 4 is returned. How happy are we? So happy. All right, now we're going to get this same thing, a 4, in Python. All right, and now we're back at our Python interpreter, and I'm going to type out the same code starting here. Note again that in Python, instead of the opening and closing curly braces, we have tabbing and colons. We also don't necessarily need to terminate our statements with semicolons. The Python equivalent of document.write is print, and we print out the 4. Exactly what we wanted to see. Let's actually go try something out. Let's go put in a bunch of extra gratuitous semicolons in Python to make it look a little bit more like JavaScript. What do we think will happen? In fact, nothing special happens. Python is totally fine with you terminating statements with semicolons if you would like. You just don't have to. If you want to get in the habit of it in order to make it easier to transition between Python and JavaScript, no problem.

1.48 48. 32_l_lambda

Now we have our formal grammar for JavaScript, and previously we had a formal grammar for HTML, but there's a slight difference between the creative work of making up the grammar, thinking of the right one, designing your own language, making the new Esperanto, and checking utterances to see if they're valid. For example, we really want to accept expressions like like this 1 + + +) 3. This makes us super sad. It's not immediately clear how you check to see if an utterance is in a grammar. We can do it in our heads, but how would we write a computer program to do it? Remember, we're making a web browser, and we want to render as much valid HTML in JavaScript as we can, but web designers sometimes make errors, and we're going to need to know how to detect that, so this decision, is it in my formal language, or is it not? Am I super happy, or am I super sad is something that we're going to need our web browser, a computer program, to do. We're going to need some sort of automated technique. One approach--and this approach is super slow. We're talking snail speed here. Actually, I think I may claim this doodle is so good that I do not even need to label it. We'll just know it's a snail. One approach would be to enumerate all strings in the language of the grammar and then just check to see if yours is in there. If it is, you win. If not, your string must have been invalid. Unfortunately, we mentioned earlier that often a finite grammar has an infinite number of strings, so spending an infinite amount of time to enumerate them all is probably not feasible, but let's try it anyway. We're going to try this approach first, even though we have some intuition that it's not going to work out, and the reasons are, 1, it's instructive--to see why we need more complicated parsing techniques later-- but also because it gives us a chance to learn a little bit more about Python and some cool programming techniques that will help you in this class and later on. Here are my power tools. This is a hammer, which I guess is technically not a power tool, but it's possible for me to draw it, and this is my--let's say--circular saw. It could also be an unhappy squid. This one definitely needs a label. Let's imagine circular saw. I'm going to introduce you to a new type of expression in Python, lambda, which is another way of defining functions. Here I've written Python code to make a function addtwo(x), which returns x + 2. You can put a Python function like this all on 1 line as long as it fits, and if I write addtwo(2), I expect to get 4 out. Here I've written something apparently completely different. I'm assigning to the variable mystery the result of a lambda expression. Here the word lambda is fixed. It's a terminal. And then you list some number of arguments, and then

you have a colon, and then here you can put any expression you like. If I call mystery(3), I'm going to get out a 5 because the interpretation of this is that lambda has made a function that takes x as its argument and returns x + 2. We just take this 3, and we substitute it in for x. Mystery and addtwo are in fact the same, but now I can do cute things like saying pele gets mystery, and then if I write pele(4), I'm going to get 6. I can make up functions with this lambda expression and then assign them around. Pele was a Brazilian footballer, one of the best of all times. Lambda means make me a function, and it's very popular in a paradigm of programming known as functional programming. Functional programming sometimes contrasts with object oriented programming and imperative programming. You'll get a chance to learn about those in other classes. Lambda means make me a function. It's a lot like def.

1.49 49. 33_q_list-power

That's not the only new power we're going to introduce. Suppose that you wanted to square all of the numbers in the list 1, 2, 3, 4, 5. Here's one way to do it. The function map takes a function as its first argument and then a list, and it applies that function to each element of the list in turn, creating a new list. I'm going to make a new list down here that's the output of map, and we're going to take 1 and square it, so we get 1 x 1 is 1. Now we're going to take 2 and square it. Each one of these led directly to part of my output. This function map is a big part of functional programming, and it's also a big part of how Google is able to make very scalable search engines and related services. Map reduced is an easily parallelizable functional programming paradigm, lots of power here. I said that map takes a function as its first argument. One way to make a function is to define it earlier and refer to it by name, but we can also make a function right now right in this expression when we need it and pass that in. This map is going to produce the same result as the previous one. But I didn't need to have mysquare defined in advance. This use of lambda is sometimes called an anonymous function because it was an important function, but we never actually gave it a name. But now here comes the real convenience. I'm going to show you a third way to make that list that's even more natural than the previous two. This approach is called a list comprehension. You ask for a list, but instead of actually putting elements in it, you write out a little formula here and then say that you want that formula applied for every element in some other list or collection. This is saying for every x in 1, 2, 3, 4, 5, so x is going to be 1, and then it will be 2, and then it will be 3, then it will be 4, then it will be 5, for each one of those, put x times x in the output list. These formulas can be arbitrarily complicated, any expression you like. Here x is going to be hello and then my and then friends, and we're going to make a list out of the length of each of those, so that's going to be 5 and then 2 and then--actually, you tell me. What's going to show up here?

1.50 50. 33_s_list-power

We're just applying the length function to all these strings and friends--1, 2, 3, 4, 5, 6, 7--has 7 characters in it, so we're just going to get a 7. This approach to defining a list where you use square brackets-- but instead of explicitly listing the elements list a formula that's applied to every element in another list--is called a list comprehension. We comprehend the new list to be made out of transformations of the elements in the old one.

1.51 51. 34_l_interview

Now that we've seen this one way, let's see this from a different perspective. [Brendan Eich, CTO, Mozilla] [Wes Weimer, Professor, University of Virginia] Let me ask you a question that's actually of personal interest and curiosity to me. I'm teaching in this class a number of elements of functional programming, list comprehensions in Python, filtering a list to retain only some required elements, and I noticed that in JavaScript you included the ability to create an anonymous function with the function keyword. And for students initially, that might be a little bit confusing because declaring a function at the top level also uses the function keyword, but we can get past that with the grammar. I'm curious, why did you think it was important to include the ability to have anonymous functions? JavaScript had to live in the shadow of Java, but it was its own language, and the original pitch-- the sort of fraudulent pretense for getting me to Netscape--was to do Scheme. Now, I didn't have time to do Scheme, and I didn't do anything as pretty as Scheme, but JavaScript is a language with first class functions, and functions are the main building block, so it's important not simply to have function declarations and require you to name them and put them at a certain level in nesting but to allow you to express them freely as anonymous functions or even named function expressions, and this is incredibly popular today. This is used in lieu of modules and to make sort of super constructors and class systems in JavaScript today using functions and function expressions.

1.52 52. 35_q_working-backwards

So let's make sure that we really understand list comprehensions by doing them backwards. Suppose what I really want is for the output to be the list 1, 2, 3. I have 5 expressions here, each one of which is a list comprehension. In this multiple, multiple choice quiz, mark each one that's going to produce 1, 2, 3 as its output.

1.53 53. 35_s_working-backwards

Well, the first one is in some sense the identity transformation. We're not actually doing any work here. For everything in 1, 2, 3, just put that in the output. That will totally work. Here we're going to take 0, 1, 2, but we're going to add 1 to each, so 0 + 1 is 1, 1 + 1 is 2, 2 + 1 is 3. That will give us the output we expect. Here we're going to do the reverse and subtract. This will be -1, 0, 1. That's not what we were hoping for. Here we're going to divide a bit. 3 divided by 2 is 1.5. That's kind of close, but 6 divided by 2 is 3. This is not getting us the answer we want in the right order. And finally, over here, length of a is 1. My is 2 and the is 3, so we get the answers we expect. Here I've written those answers out explicitly in Python in the interpreter, and we're just going to check and see that we got the answers that we expected. This was much too low. Here the division didn't work out properly, but in the last one it worked out fine. We could totally fix this last one. If we divide by 3 instead, we will get 1, 2, 3, 1, 2, 3, but it was off by 1.

1.54 54. 36_l_generators

These list comprehensions are amazing. They are one of my favorite parts of Python. This is really a great way to program. It's very declarative. You just say what you want, and the system makes it for you. Unfortunately, they have a slight downside, which is that thus far we've had to write out the starter list, and that's almost as much work as just writing down what we need. If only there were some way to generate this list, especially if it's really big, without us having to write it down explicitly. For example, suppose I started with a big list like this, 1, 2, 3, 4, 5, 6, 7. I assert it's big. It's big for me. This is heavy lifting for your professor. And what I want to do is filter it down so that we only have the odd numbers. I want to get this part out. I'm going to show you a new way to do that in Python. Here I've written a procedure called odds_only in Python that takes a list of numbers, and it's going to iterate over them. For every n in that list of numbers, if that number is odd, we divide it by 2 and check the remainder. We yield that part into our results. Note that I did not write the word "return." Yield is a new special keyword that means we can use this sort of procedure to gather up multiple results. Let's imagine that this big list here was numbers. We'll yield 1, not do anything with 2, yield 3, not do anything with 4, yield 5, not do anything with 6, and yield 7, and that's exactly the output that we wanted. You can view this as sort of a convenient way of filtering. Here I've written out our odds_only procedure. I'm just going to show you in the interpreter how this plays out. I'm using a list comprehension. I want to print out every value of x that's in odds_only of 1, 2, 3, 4, 5, and we get 1, 3, and 5, as we expected. I'm also going to show you an even easier way to do this. Snap, it'll be so cool. Here I've written a list comprehension. I want [x for x in [1, 2, 3, 4, 5]. But over here on the right I've put this sort of if conditional, a guard or a predicate we might call it in mathematics, and this is saying I only want you to yield those numbers for which the predicate is true. Only include x in the answer if x was an odd number, and look, we get the answer we wanted. I've written x a few times, but I can make these formulas arbitrarily more complicated. Here I've said take all of the odd numbers, and multiply them by 2. Since 1, 3, and 5 were the odd numbers, 2, 6, and 10 are the multiplication by 2. I love list comprehensions, and you will too soon. A function like odds_only that uses yield to potentially return multiple answers is called a generator because you can use it to generate another list or another enumeration.

1.55 55. 37_p_small-words

So now let's have you try it out on your own. Using the interpreter, write a Python function called small_words that accepts a list of strings as input and yields only those that are at most 3 letters long. Since you'll be using yield, this is a generator function.

1.56 56. 37_s_small-words

Well, let's go write out the answer together in the interpreter. I'm going to start by defining my function small_words, and it takes a list of words as input. I definitely want to iterate over each one. The problem definition said that we want those involving at most 3 letters, so those are the small words that we yield. Now I'm going to add some debugging to see if I've gotten the right answer. I'm going to use one of these list comprehensions because I love them so much. I'm going to print out each word that's in small_words applied to "The quick brown fox." I'm just sort of eyeballing it. "The" and "fox" are at most 3 letters, so I hope to see these 2 in the answer, but "quick" and "brown" are too big to make the final cut, and that's exactly what we got out. Recall that our goal was to enumerate all the strings in a grammar. That was our super slow approach to check and see if HTML or JavaScript was valid. Well, unsurprisingly, these generators are the trick we're going to use to enumerate a lot of strings in a concise manner.

1.57 57. 38_l_checking-valid-strings

We're going to write a Python program to check and see or to enumerate strings in this language so that we can check strings for validity. Here I've written a grammar, but it's on kind of a pencil and paper format with these arrows and what-not. I'm going to need to encode it so that Python can understand it just like we had to encode the edges in our finite state machine so that Python could understand them. Here on the right I've written a potential

Python encoding for this grammar. Expression goes to expression plus expression. The way I'm going to encode this is if we had A goes to B, C, I'm going to make that the rule A goes to B, C. We've got the left-hand side and the right-hand side, left-hand side and the right-hand side. Expression goes to expression plus expression. Expression can also go to expression minus expression or expression in parentheses, or just number, and here I'm writing all of my terminals and non-terminals as strings just to make equality checking easier. Then given a seed utterance like print exp followed by semi-colon, we would represent that as a list of 3 strings, and what I want to do is take our grammar rules like this one and our utterances like this one and combine them together to get--I've replaced the expression that was already in here. Using this rule, I've replaced it with expression minus expression. Well, if I've stored this in a Python variable called "utterance," and the position I want to change is position 1--that's where I'm going to apply this rule-- I can get out this cool result with this Python that uses list selection. Utterance from 0 up to but not including pos, this part here corresponds to this part in our example, the word "print," something that came before the part we were interested in. The right-hand side of the rule was expression minus expression, and then the rest of the utterance, this semi-colon, something else we weren't concerned with this time.

1.58 58. 39_p_expanding-exp

So let's start coding this up. There's our first grammar rule, expression goes to expression + expression. Our second grammar rule, very similar but for the minus. All right, so here I actually have quite a bit of Python code, and we're going to walk through it together, and then you're going to help me finish it out, so here's our definition of our grammar. It has 4 rewrite rules, and where I'm going to need your help is given a list of tokens and a grammar, I want to find all possible ways to expand it using those rules. Let me just show you what I mean by that. Here's our grammar again, and let's say that we started with "a exp." I would want us to come out with "a exp + exp," "a exp - exp," "a (exp)," and "a num." For each of these possible token positions and for each grammar rule, we removed the starting token and replaced it with the right-hand side of the grammar rule. This is the way we're going to enumerate all of the valid strings in the grammar. Now, you'll notice that I've made more expressions in many of these cases, so I could go from here and start and expand it again and get a few more strings until eventually I'm full entirely of terminals. I'm going to call the number of times I do this the depth. If we start with a exp and we expand it to depth 1, we get 4 new utterances, 4 new sentences. You're going to have to help me write that expansion procedure. Down here I've got some code to help us print it out. For now, we're only going to enumerate things up to a depth of 1, and we're going to start with just expression, and then we're going to use your expand procedure to make many more utterances, many more sentences starting from expression, probably 4 more, so then we'll have a total of 5, and then at the end of the day we print them all out. If you do it correctly, this is the output you expect to see, our original sentence, but then it's been expanded with 4 more, expression + expression, expression - expression, open expression close and num. And in fact, if we go back up here and change this to a exp, the example we worked through in the comments, we get the expected output. A is unchanged because there's no rewrite rule in our grammar for dealing with it, but a exp becomes a expression + expression, a expression - expression, a open expression close, and a num. Your job, submit via the interpreter the correct definition for the expand procedure so that it does this. Here's a hint. You're going to need yield with high probability, and you may also want to end up using list comprehensions. By the way, this is a very tricky quiz. This is not easy to get right the first time, so you should not feel bad if something goes wrong. Give this your all, but it is very difficult compared to what we've been up to so far.

1.59 59. 39_s_expanding-exp

So it's going to turn out that I can complete the definition of expand with just 2 lines if I use yielding generators and list comprehensions later on. Let's go take a look and see how this goes. Well, we're passed in a set of tokens like "a exp," and now we're going to consider each possible position, the zeroth position which is a, and the first position which is exp, and then for each rule in the grammar, we want to see if it matches. Now remember, none of the rules in our grammar match a, so we're going to need our program to deal with that correctly. If tokens[pos] might be a the first time and exp the second time. We have to check and see if that matches the left-hand side of the grammar rule and remember--let's go back up to our encoding of the grammar-- we always had the left-hand side as the zeroth element of this tuple, and the right-hand side was the first. We're going to check and see if these strings equal exp. Only if they do do we want to apply the transformation, so if we're in the right position, like we're in position 1, and we see an exp, great. Now what I need to do is for each of the rules in our grammar, yield or return one of those results. Conveniently, we've got this other for loop up here. For each rule in our grammar, I'm going to use this expression that I showed you before, and we take tokens 0 through the position. For us that's that a that's being unchanged. We're just carrying that along. I'm going to yield that. We're going to add to it the right-hand side of the rule, expression plus expression, expression minus expression, and then also any tokens that came afterward. We could have done a expression b, and we would have seen a b on the right-hand side of all of these. And actually, that's pretty much it, but here's the real power of this. Let's expand the depth up to 2 and start seeing what kind of answers we'll get, so now we're going to re-expand expression, expression, expression. There are 1, 2, 3, 4, 5, 6 places we could do this expansion. It's immense. It goes down quite a ways. But you can start seeing much more complicated expressions, like for example, here we have something that might be Here we've got some uses of parentheses to help remove ambiguity, and you could imagine that I could go up to depth 3 and depth 4 and start enumerating lots of strings in this grammar. In fact, wow, there's a huge amount of output, and

that's why I was suggesting that this is not a very efficient way of doing things because I could enumerate a lot of strings of length 3 or 4 or 5 or a lot of strings that have involved 3 or 4 or 5 rule rewrites. So we just saw a slow way to encode formal grammars in Python and enumerate strings in them. We will definitely need this power, but in our next exciting episode I'm going to teach you a more efficient way to encode grammar rules, like the rules for HTML in JavaScript that we will need for our web browser, and check to see if a string is valid without having to enumerate all possible strings first, because that might take, let's say, one more second than I have, so we're going to see very efficient ways to solve this problem. Super cool. Stick around, and I'll see you next time.

1.60 60. 40_l_tetris

So years ago, while I was working at a company that shall not be named, I was bored out of my mind, and one of the things that people at this company did to pass the time was playing Tetris. I ended up liking this Tetris game so much that I decided to try and write my own version of it, write a computer program to play Tetris, and if you haven't seen it before, Tetris is a Soviet falling blocks puzzle game with a number of different pieces, all of which are made up of 4 smaller blocks. I decided that I wanted to support not just the normal Tetris pieces, but because I'm a bit of a math person, the ability to have pieces that were 5 long as well. In order to do that, I ended up making a piece definition language, a way of writing out what a Tetris piece should look like that this Tetris program would then read in. Now, I didn't use all of the lexing and parsing techniques that you're going to learn in this class, but I did use the same sort of concepts, and the point I'm trying to get across here is that in some sense, learning how to create your own language is like learning how to read. It really opens up a world of possibilities. It's a new tool in your belt that will come up and help you when you least expect it.