CS262 Unit 4

4

rrandom

Contents

1 CS262 Unit 4

Contents

1.1 1. 00_l_introduction

Welcome back! We're about to start unit 4, and one of the things we're going to cover is the suprising power you can get from just writing down something you've already computed and referring back to it later. That actually came up once in my life. A number of years ago, I had the priviledge of working for Microsoft Research on a project to try to find and fix bugs in Windows. Now most of you are probably watching from a moon based in the not too distant future, but in the present and in the past, bugs in Windows were a big deal. In fact, it would often crash and lead to the dreaded blue screen of death. It turned out that as much as we like to pick on Microsoft, most of the bugs weren't in Microsoft code, but in third-party code written to drive various bits of hardware, like screens or printers or memory sticks. This software was called Device Drivers, and it might work something like this memory stick that I have right here. Microsoft wrote a tool to put this Device Driver software through torture tests. If you're memory stick, I might add data to you and then in the middle of reading it out, pull you out or turn off the power, or in general, apply these normal operations very fast or in a surprising order. And this really worked. They found a lot of bugs. So many bugs that's it's now a shipping product--the Microsoft Static Driver Verifier. The heart of this idea was a computer science notion known as model checking, figuring out how a program behaves by looking at its source code. A key to that was remembering things that had already been computed. If I already know how you behave when I turn off the power in the middle of an operation, I don't have to recompute it, which might be very expensive. So this relatively simple notion of writing down things so that we don't have to recompute them, is formerly called memoization, and it's one of the gems we'll get to in this unit.

1.2 2. 01_l_time-flies

Welcome back! This is unit 4 of programming languages, and if we turn the clock back to last time, we were posed the following question. Given a string S, like a webpage or some embedded JavaScript or any program, and a formal grammar describing our desired language, a former grammar for HTML or JavaScript, we want to know if that string S is in the language of G? To do this, we use 2 key techniques--lexical analysis, which broke the string down into a stream of tokens, and syntactic analysis or parsing, which takes a string of tokens and checks to see if they adhere to, if they conform to a context-free grammar. While we're on the subject of time and grammars--grammars that may possibly be ambiguous, let me introduce you to a phrase that you may not have run into yet. The phrase is, "Time flies like an arrow. Fruit flies like a banana." The ambiguity trick here is that time is a noun, flies is a verb--time flies-- and "like an arrow" is the modification. So here I've got time flying. You can tell because it's got labels, and those labels are always to be trusted. So you might think, based on parallel structure, that fruit is a noun and verb is flies. So here I've got a picture of an apple with wings, and their flying in the manner of a banana. But in fact, fruit flies is the noun. They are little insects that go after fruit, and this time, like is the verb. Fruit flies go after the banana. They enjoy the banana. This is the sort of ambiguity that we can run into in a natural language like English. We're going to have to deal with that same sort of issue in programming languages, like in JavaScript or Python.

1.3 3. 02_l_brute-force

In our last unit, we ended with a brute-force algorithm for enumerating all the strings in a grammar, step by step. Brute force is actually a technical term, which means to try all of the options exhaustively. Typically, the brute-force solution is easy to code, but relatively inefficient. In fact, our brute-force solution was so inefficient, it might go on forever. Consider this grammar for balanced parenthesis. We know how to enumerate strings in the language of this grammar. Suppose I give you the input open, open, close, and we want to know if it's in the language of this grammar. Well, in our brute-force approach, we would just enumerate things. We'd say, oh, well, 1 thing is the empty string. Is your input the empty string? No. Another string in the language of this grammar is open, close. Are you open, close? No. Another string in the language of this grammar is open, open, close, close. Is that you? Nope. How about open, open, open, close, close, close? Still no! How about 4 opens, followed by 4 closes? You are getting farther away. So cold! This is the wrong direction. So the algorithm that we described would enumerate all of these strings and many more--infinitely many more. Never noticing that we're never really going to match this. We're making strings that are too big. This is all just wasted work. I don't need to check 5--1, 2, 3, 4, 5--if 5 opens, followed by 5 closes corresponds to this input string. This has 10 characters. It is way too long. So that's a clear inefficiency in our previous brute-force algorithm. And that key insight that we can stop somewhere around here is what's going to lead us to a more efficient parsing strategy. Thus, our key parsing idea. We're going to win by being lazy and not duplicating work. We don't want to do any unnecessary work. We want to be super lazy. And in fact, here we've got a lazy person on a bed sleeping. You can tell it's a bed because I can't sketch. In fact, this notion that laziness is a virtue for programmers is widely celebrated. Larry Wall, the designer and inventor of Perl--P, e, r, l-- the pathologically eclectic rubbish lister-- a language that we won't be describing in this class, claims in his text books, we will encourage you to develop the 3 great virtues of a programmer-- laziness, impatience, and hubris. Sign me up! Those sound like great ways to lead one's life. But perhaps for computer programming, they actually are. The notion for laziness is it's this quality that make you go to great effort to reduce overall energy expenditures. In other words, I want to spend 5 minutes now to save 5 hours later.

1.4 4. 03_l_fibonacci

Most of you probably ran into the Fibonacci sequence of numbers, named after filius Bonacci in a previous computer science class. It's a great way to teach recursion. Here I've written out a Python definition for the Fibonacci sequence. To get the Nth Fibonacci number, well, if N is less than or equal to 2, we just return 1, otherwise, we return the sum of the 2 previous entries. So we're going to get 1, 1, 2 -- 1 + 1 = 2. No, I totally didn't do that correctly. Man, you can't take me anywhere. [Singing--hmm, hmm, hmm] Basic math. Don't mind me. There we go. This looks a lot better. You saw nothing. Alright, so there's our Fibonacci sequence. In an incredible surprise move, it actually shows up a lot in nature-- for example, in the patterns of seeds in a sunflower or in the whirls in a clamshell or in yellow chamomile plants or all that good stuff. However, Fibonacci involves a huge amount of work. Let's go see what goes on when we call Fibonacci. I'm going to abbreviate it with an f--Fibonacci of 5. Well, that's going to be based on Fibonacci of 4 and Fibonacci of 3. Now 4 is based on 3 and 2. 2 is a base case, so we're done. 3 is based on 2 and 1. Over here, 3 is based on Fibonacci of 2 and Fibonacci of 1. If you look carefully, a lot of these get repeated many times. We end of calling Fibonacci of 2--once, twice, 3 times I called it. Similarly, Fibonacci of 3 is called multiple times. We're redoing work. We're computing the value of Fibonacci of 2 and Fibonacci of 3 over and over again. That is wasted work. We want to be lazy and avoid that.

1.5 5. 04_q_recursive-definition

Just to make sure that we're all up on Fibonacci and its recursive definition, here's a quiz about it. I've written 4 statements. Check each one that's true. There might be multiple correct answers.

1.6 6. 04_s_recursive-definition

Let's find out what the right answers are. Fibonacci of 6 = 8. Well, if N is 1, 2, 3, 4, 5, 6. We said that Fibonacci of N was 1, 1, 2, 3, 5, 8. It does look like Fibonacci of 6 = 8. Great! Is Fibonacci of N always < or = N + 1? Well, 1 is < or = to 2. 2 is < or = 3. This is certainly true. The Fibonacci sequence is strictly nondecreasing. It either stays the same or gets bigger. This next one was a bit of a ringer--a bit of a trick. The vast majority of the time, almost always, albeit finitely often, Fibonacci of N is strictly > than Fibonacci of N + 1, except right here at the start when Fibonacci of 1 and Fibonacci of 2 are both 1, so they're = rather than <. So no dice there. Is Fibonacci of 20 > 1000? Yes. The sequence grows super fast. Let's just go check. I'll just write out the definition of Fibonacci right here. I'm declaring a procedure called fibo. It takes an argument n. Here's our base case: if n < or = to 2, return 1. Otherwise, we call ourselves recursively 2 times. And the answer is 6,765. Wow! That's immense. Notably, it's bigger than the 1000 we were asking about. True!

1.7 7. 05_l_memoization

You might have noticed that up here on the right, I made a very simple chart to try and explain how Fibonacci behaves to myself. We're going to use this same sort of chart to make Fibonacci much faster by voiding repeating a lot of work. Our official plan for this is going to be called Memoization. It's just like memorization, but missing an r. Here I've tried to draw 2 memos: a corporate memo and those yellow sticky notes you sometimes see, where you could write a little memo to yourself. A memo in English is just a document, a small document, that's written down-- memorandom. Why bother with this? Well, it's going to turn out that our current implementation of Fibonacci is super slow. Let me try to prove that to you. So let's see how long it takes to do 100 trials of the 20th Fibonacci number-- about .3 seconds. Let's up that a bit to the 24th Fibonacci number-- should take not that much longer, right? Oh! Significantly longer, from .3 seconds to 1.75 seconds. We went up a huge amount. Let's go up to the 25th Fibonacci number--oh! We almost doubled. We're now at about 2.8 seconds. In fact, we have reason to believe based on human studies that if a webpage takes longer than 6 seconds to get back to you, you go somewhere else and buy something different online. So we're already using up a huge fraction of that budjet just to compute the 25th Fibonacci number. And if you think about those trees I drew before, this is unsurprising. If we increase the number by 1, we almost double the work at each step. So this is untenable. We need something faster. Our solution we'll be to write it down in a chart, or a little memo, to ourselves. I'll just make a table mapping N to the value of Fibonacci of N. And when I'm going to figure these out, I don't have to do a huge amount of work. Let's say I'm trying to figure out this 6th Fibonacci number. I can just look back in the table, and reuse my old work. I don't need to recompute the 5th Fibonacci number. I already have it here. Just additional those 2 chart cells together and get the answer. This is going to be our trick for making Fibonacci so much faster. It's called memoization. So we can implement our chart as a Python dictionary, just filling in the numbers as we compute them. So I can make an empty dictionary, assign mappings to my dictionary, and then check to see if something like 6 is recorded in my chart, and if it is, print out the result. This is going to be super necessary now and maybe it wasn't before. One of the keys to memoization is looking to see if you've already done the work and written it down. If you have, great! You can just reuse it. But if you haven't, you're going to have to go and compute it manually the first time.

1.8 8. 06_p_memofibo

It's quiz time. Let's show off our knowledge of memoization. Submit via the interpreter a definition for a memofibo procedure that uses a chart just as we described. You're going to want the Nth value in the chart to hold the Nth Fibonacci number if you've already computed it and to be empty otherwise.

1.9 9. 06_s_memofibo

Let's go through a possible answer together. We initialize our chart to be the empty mapping, and I'm going to define a procedure named memofibo. If we're asked to compute the Nth Fibonacci number, and it's already in the chart, then we don't do any more work. We are super lazy. We just look it up in the chart and return that. Otherwise, we need to both set the chart and return the new value. So if n < or = to 2, the thing we want to write down in the chart is 1. Otherwise, we'll figure out the value of the chart by calling ourselves recursively on n - 1 and n - 2 and adding them together. In any event, since we set the chart here or here, we'll just return the value in the chart. Now I've asked us to print out the value of memofibo 24, and we get the answer that we're expecting. However, the real proof is in the timing. Using our timing code once again, I've now put in the code for memofibo instead, and we're trying to compute memofibo of 25. How long does it take to do this 100 times? Oh! Significantly less time! Remember before it was almost 3 seconds--almost half of our page budget. Now you can barely detect it--not a tenth of a second, not a hundredth of a second, but even smaller. This was a phenominal cosmic optimization. We are so much faster. It is not even funny.

1.10 10. 07_l_memoization-for-parsing

So this gives me a great idea! Let's use memoization to make parsing very fast. Let's cast our minds back to the glory days of regular expressions and finite state machines. In order to see if a string was accepted by a finite state machine, we'd essentially keep our finger on the state. So on input abb-a, b, b--I just sort of keep my finger on this middle state to see where things were going. If I stop here, then the string is not in the language. but if I add 1 more character, c--a, b, b, c--I just put my finger on it, and I can tell, we accept. We're going to use this same "put your finger on it" trick for parsing to keep track of where we are, to keep track of which state we're in. Now for finite state machine state, that was pretty easy. They were the circles. For our parser state, this is not so clear. We're also going to solve parsing by "putting our finger on it." But just like how how nondeterministic finite state machines might require 2 or 3 fingers, parsing might also require a number of fingers. It's going to be somewhat complicated. Consider this simple arithmetic expression grammar--has a starting nonterminal, but then quickly goes to E. E + E, E - E, and 1 and 2 instead of number. Let's just make it finite. Suppose the entire input is 1 + 2, which is in the language of the grammar. Currently, we've only seen the 1 and the +. Remember that to figure out if something was in the language of a finite state machine, we look at 1 character at a time. We're going to do pretty much the same thing for parsing. We're going to look at 1 token at a time. But the question is, where are we now? Well, we don't have states that look like circles, but we do have these rules. In fact, we've got 5 of them written over here, and if we've already seen the 1 and the +, we're about to see the 2. I claim that there are 1 or 2 of these rules that match more closely than others. For example, S goes to E--that doesn't seem particularly relevant. Now a minus sign--that doesn't seem particularly relevant. E goes to 1--if we've already seen the 1 and the +, we're kind of passed that. But these 2--E + E and 2--that's kind of where the action is. That's where we are now in some strong sense. In fact, I'm going to claim that we're right here. In the rules, E goes to E + E, we've already seen the E and the +. Here's my finger, and we're about to see the next E. Since I can't always leave my finger on the slide, we often formally draw a red dot in the middle of 1 of these rules to keep track of where we are. This is 1 example of a parsing state. The first E is going to correspond to or come from the 1 of the input. Ideally, the second E will match up with 2 in the input. We've already seen everything to the left of the red dot. We have not yet seen everything to the right of the red dot. The red dot is right where we are now. This is the past. This is the future. This is now.

1.11 11. 08_l_parsing-state

Let's trace what happens as we see a little more of the input. Let's say we actually walk over and we see the 2. Well, then our parcing state looks like this--E goes to 2. We've seen the 2, and there's nothing left. Everything important is in the past. There's nothing in the future. So conceptually, we can use this rule--this rewrite rule, E goes to 2 like we hinted at here with our purple text, we will also be here. E goes to E + E, but we've already seen the E, the +, and the E-- E + E. So we can go even further and go back to the first rule in our grammar where we were trying to parse a sentence, which could be an expression. We've already seen an entire expression--1 + 2--and now we're done. In fact, if you can get to the state corresponding to your starting nonterminal being completely finished, everything is in the past. Nothing to see in the future. Then you've parsed it! That string was in the language of the grammar. So formally a parsing state, a possible configuration of a parser, a world we could be in is a rewrite rule from the grammar augmented with exactly 1 red dot-- that's where I'm putting my finger. That's where we are now. The past comes before it. The future comes after it--on the right-hand side of the rule. You can never put the dot on the left. The dot is always to the right. Now for any given input as we've seen here, you could actually be in maybe 3 of these parsing states at once. One way of looking at the world is that I just finished seeing the 2. Another way of looking at the world is--actually I'm done with everything. Those can all be true at the same time.

1.12 12. 09_q_possible-states

Let us check our knowledge of parsing states with a quiz. Here I've written 6 inputs and 6 corresponding states, but I may be leading you astray. What I'd like you to do is for each input state pair, check the box if this state is really a possible state, a valid state, for our parser on this input, given the grammar above. You'll need to look at the grammar and the input to check and see if the state is correct. Go forth!

1.13 13. 09_s_possible-states

Alright, let's get started. We see just a 1, and we've seen all of it. Then we totally could be in this state. E goes to 1, and then there's my finger. That's where we are. We've already seen the 1. We're done with--we're ready to apply this rule, E goes to 1. Everything here is fine. Here the input hasn't changed, and we have another state, but remember that we said before, depending on your point of view, when you see a 1 you can either be reducing a 1 to expression or you could be completely done with parsing. We normally think of things like 1 + 2 - 3, but the lowly 1 alone is in this grammar. S goes to E. E goes to 1. And this says, I finished parcing the string. Great! Now we just see a 1, but this state says, oh, I've seen an E, and I've seen a +, and I'm looking for an E in the future. This can't work because it requires us to have seen the +. If I put my finger here, the + is in the past, and I haven't seen any +'s in the input, so we can't make that work out. The next one has the same input, but a slightly different state. I've seen an E, and I'm looking for a + and then another E. Yeah, I could totally imagine a + followed by another E filling this out. That could work, so this is a possible state that we could be in. We see 1 + as the input, and this is a little more complicated, and now the one that we had to reject before suddenly becomes valid. My finger is here, and E and a + are in the past. Here's the E. Here's the +, and I'm looking for some new expression in the future. This fits very well. The last one was a bit of a ringer. This was a bit of a trick question. It required you to have the definition well in hand. This looks very promising. I put my finger here and it says, oh, there's a 1 and a + in the past, and we're looking for an E--that all sounds good. But remember that the definition of a parsing state is that it's one of the rules from our grammar, augmented with a single red dot. E goes to 1 + E is not a rule in our grammar. The closest rule in our grammar is 5, which is E goes to E + E. Every symbol matters. This can't be a valid parsing state because E goes to 1 + E is not a rule in our grammar.

1.14 14. 10_l_charting-parse-states

So just as we applied memoization to Fibonacci, we're going to apply memoization to our attempts to parse a grammar. When we were trying to compute Fibonacci, the Nth position in the chart corresponded to the Nth Fibonacci number. Let's say I'm trying to parse a string made out of a big list of tokens-- token 1, token 2, token N, all the way up to token last. We're going to try to figure out the string 1 token at a time. So the Nth position in the chart is going to be all the parse states we could be in after seeing the first N tokens--the first N words--in the input. This means that instead of our chart returning a single number, our chart is going to return a set or list, an entire collection. Here's a much simpler grammar to try out this concept on. E goes to E + E or E goes to INT. We use INT like we use NUM or number. It represents integer. So here the language of our grammar includes things like INT + INI, INT + INT + INT. Suppose, in fact, that the input is INT + INT. I'm going to draw out a chart showing N and chart of N. Suppose we haven't seen any of the input yet. Where could we be? Well, conceptually my finger is going to have to be very far to the left. I could be looking for E + E, or I could be looking for INT, but regardless, I haven't seen any of it left. There's nothing in my past, and the whole world is in the future. The reason I can't be sure yet is that I've only seen 0 tokens of the input. So even though we, the viewers at home, the studio audience, know that eventually we're going to be using this top rule, our chart hasn't seen enough of the input yet to make that determination. Well, after seeing only 1 token, we've seen the INT. One possible state I could be in is, well, I'm trying to reduce an expression to an integer. You gave me an integer. I'm done. If I think the input is going to go on a little longer, I might expect to see a + and an E coming up later. So this is the second state that I could be in. If I've seen the INT and the +, then among other things, my world probably looks like this. I've got an E and a + in my past. I'm looking for an E in my future. There might be a few other elements in this state, and we might continue the chart a bit farther to the right, but this is the basic idea. It's just like our chart for Fibonacci, but instead of holding a single number, it holds a list or a sequence of parsing states, and each parsing state is a rule in our grammar augmented with a single red dot somewhere on the right-hand side.

1.15 15. 11_q_possible-tokens

Now let's dig into this notion of parcing states a little bit more. Let's say that our grammar is the same simple INT + INT grammar it was from before. If our current state is E goes to E + dot E, how many tokens could we have seen so far in the input? Remember a token is a terminal like INT or plus.

1.16 16. 11_s_possible-tokens

Well, let's take a look together. Could we have seen 0 tokens so far? No, when we've seen 0 tokens the red dot has to be really far to the left. Currently, the red dot suggests that we've already seen an E and a +. The + alone takes up 1 token, so we can't be here. Alright, but could we have just seen 1 token? Well, since we've seen an expression and a + and the smallest expression is itself, However, we could have seen just 2 if the input so far was INT +. We've seen 2 tokens and one of our states would be exactly this one. INT reduces to, or can be rewritten from, E using one of the rules in our grammar. The + is a terminal, so it always stays the same. Here's where we are. How about 3? Well, this is a little trickier. In our grammar, if we had seen 3 input tokens, our red dot wouldn't be right here after the +, it would be over one more. Something a little different would happen. It's very hard to have a string in the language of this grammar, where after 3 tokens you've just seen a +. But surprisingly, 4 tokens does. Let's make a little room and take a look and see why. E can be rewritten by INT. E can be rewritten by INT. This + stays the same. So the 3 of these together, E + E are themselves--one more E. So conceptually from the parsers point of view, what we've seen so far is an E and a +, assuming we've done all these rewrites over here on the left, and we're looking for a little more input. So in fact, we could have seen 4 input tokens INT + INT + and been in this parsing state. This might seem a little counterintuitive but remember the glory of parsing, or the glory of context-free grammars, is that a very concise grammar notation stands for an infinite number of strings. Even this very simple grammar has an infinite number of strings in its language, so it shouldn't be surprising that longer strings than 2 can have very concise parse states. The dot would have to be in another place. However, if we were to add a few more INTs, this trick that I've done here of reducing INT goes to INT goes to INT--if I had 1, 2, 3, 4, 5, 6 tokens, I could also be in the same state. So 2, 4, 6, 8--the pattern repeats.

1.17 17. 12_q_malformed-input

Let's get 1 more view into how this relationship between grammars and parsing states plays out. I've written a new grammar over here on the left, but actually if you think about it, it's simpler than our old grammars. Since this one isn't recursive, there are only a finite number of strings in the language-- INT + string and INT alone. Let's say that the full input is going to be INT + INT, which is not in the language of the grammar, but thus far, we've only seen a single token--INT. Our parser has to handle good input and also malformed input. Not everything out there on the web is super clean. We're going to want to write a web browser that can tell the difference between the good and the bad, the wheat and the chaff. So what I'd like you to do is figure out after just 1 token of the input, what are some parse states we could be in? I've listed 7 possible parse states in this multiple multiple-choice quiz. I would like you to check each one if we could be in it, after seeing only 1 token of this input.

1.18 18. 12_s_malformed-input

Alright, how about this? We've seen just INT. One of our grammar rules is A goes to INT + string-- INT + string--and I've put the dot right here, so we've already seen an INT, and we're expecting to see 2 more things. This is consistent with the world that we've been presented. We know that eventually this won't work because the full input is INT + INT, but we haven't seen that much yet, so we can't rule it out. Right now we think this state is okay. In the future, we'll give up on it. Similarly, one of the rules in our grammar is A goes to INT. We've only seen an INT, so we could be in this state. The INT is behind us. There's nothing in our future. We're really hoping the input ends now. The input doesn't end now. You and I know that there are 2 more tokens coming, but our parser doesn't know that yet. It's only seen 1 token. In the next step, the next iteration, the next recursive call, it will know and throw away this information. But for now, we're keeping it. Alright, how about this? A goes to INT +--this requires us to have 2 tokens in the past, and we've only seen 1 token of the input. That can't be true. Similarly over here, there's nothing in the past, and there's 1 token to the right. Looking at our grammar. There's really no way this could play out. We've seen 1 token, and this assumes we've seen 0. Over here, similarly, INT + string--this parsing state only works if we've seen 0 tokens, and we've already seen 1. So that doesn't match. And in fact, similarly here, S goes to A. S goes to A is a rule in our grammar. That's a good sign, but this version requires us to have seen nothing, and we've seen 1 token. S goes to A, and here we are. Actually this could totally work. If I've only seen just INT from the input, then I could be finishing off-- I could be accepting the string based on S goes to A and A goes to INT. So I've already seen a full A. A goes to INT--and now I'm done with S. Yeah! That's where I could be. Now again, we're going to rule this out as soon as we see the next token in the input-- that it's not the end of the string, but for now, it looks very promising. This is the big trick with parsing, I said earlier, we'd have to keep our fingers in many spots because until we see the whole input, we're not sure what the picture is.

1.19 19. 13_l_magical-power

So we just remembered that one of the great powers of grammars is that they can be recursive, just like procedures. You can have a grammar rule that expands to itself, allowing you to have an infinite number of utterances from a finite formal grammar. That gives us a huge amount of power, almost a magical amount of power, but it does mean that we'll need one more element of bookkeeping in order to correctly do parsing. We saw before in one of the quizzes that we could be in a particular state after seeing two tokens, four tokens, six tokens, eight tokens, so we'll need to keep track of one more number to know which one of those it was. We'll need to know where we came from or how many tokens we'd seen thus far. Here's a grammar we've seen before. E goes to E plus E or E goes to int. Our input string is int plus int. I'm going to start filling out that chart that shows us what parsing states we could be in if we've only seen a subset of the input. If we've only seen zero tokens, then we could either be looking for E plus E or we could be looking for int. Those are our two grammar rules. We haven't seen anything yet. There is nothing to the past. Everything is to the future. Once we've seen the single int, then we could either be in the middle of parsing E goes to int, and we're totally done with it. Or, if the input is longer, we could be expecting a plus and an E in the future. Here is where things start getting fun If we've seen the int and the plus, then we could definitely be in the middle of parsing E goes to E plus E with a dot right here. We've seen two things to the left. There is one thing in our future. But now we could also start looking for another int. We're expecting in int to be the third token. If we saw it, it would reduce or it would be rewritten from E goes to int. Our current parsing state is we have seen it yet, but we're really expecting it in the future. But here is where the potential ambiguity comes in. This states is exactly E goes to dot int. We saw that same state back here in chart position 0. However, they're not exactly the same. This one corresponds to the first int in our input. This one corresponds to the second int in our input. This is what we're thinking about when we haven't seen any tokens yet. This is what we're thinking about when we've seen two tokens by not the third. The parsing rule is similar, because the grammar is recursive. It has a small, finite structure. But we really need to remember one extra bit of information. When we're thinking about it this time, we've sort of seen zero tokens so far. Over here, we've seen two tokens so far. Or we decided to add this state based on reasoning about state 2. This fact that we could have two otherwise identical parse states means we'll need to augment our parse states with one more number of information. We're going to call this new information the "starting position" or the "from position" associated with a parse state. One last way to see way to see why we need it. Let's say one of our current states is E goes to dot int, and we actually see that int. It's part of the input. We need to know whether we should go on to sort of chart position one and start looking around here or whether we should go on to chart position 3, which I haven't filled in, and start looking there. We need to know where we came from in order to know where we're going. This is one of the reasons why context-free grammars are more powerful than finite state machines. Finite state machines did not really need to know where they were coming from. They were memory-less in some sense aside from the current state. We're doing all of this because we want to master parsing. We want to see which strings are in the language of a grammar, to see if HTML or JavaScript is valid before we try to through it to our web browser's rendering engine. Another way to think about this is that parsing is the inverse of producing strings. Rather than producing all the strings in the world, I want to see if this one string could have been produced by our grammar. Over here I've drawn a little diagram of parsing a simple sentence int plus int using our grammar. Well, one way to view this is to think about the parse tree, which I've kind of drawn here upside down. Conceptually, I could apply this E goes to int rule in reverse and rewrite this int with an E, changing the input string so that it has a nonterminal in it. Then I can do the same thing again over here, and now I have E plus E. I can rewrite that to be just E. It's as if I'm taking the rules and the grammar and changing the direction of the arrow. If I view this story this way, we're parsing. Magic trick of perspective--if I read from the bottom up we're generating or producing strings. Starting with E, I choose to apply E goes to E plus E. I choose to apply E goes to int. I choose to apply E goes to int. I end up with a string at the end of the day. This way, from the bottom to the top, is generating or producing a string. This way, from the top to the bottom, is parsing a string, applying the reductions in reverse order until you get back to the start symbol. If you could apply all the reductions in reverse order, then you know that the string is in the language of the grammar, because you have a script for generating it. Just do everything you did backwards.

1.20 20. 14_q_hidden-past

Now it's time once again for you to show your understanding of this forwards and backwards. I've got a relatively simple grammar up here in the upper left, but it's a grammar that couldn't be captured by any regular expression so it can't be that bad. P goes to open P close or P goes to nothing. You could imagine drawing the epsilon there. Our input is the four character string open open close close. I said before that we'd need to annotate each of our parsing states, and I have eight of them shown here with information about which state they came from, what the from position was, what the starting position was. What I'd like you to do is fill in each one of these blanks with a single number corresponding to the chart position that this state conceptually starts at. Another way to think about that is let's say that we're in a particular state like this one-- P goes to dot open P close. How many tokens must there have been beforehand for this to possibly make sense. We know we're in chart position one, but here it looks as if there's nothing in our past. How man hidden things would there have to be in our past for this to work out?

1.21 21. 14_s_hidden-past

Well, let's go try it out together. When we're in chart state 0, when we haven't seen any of the input yet, we started in position 0. there is no hidden input we're missing. There is no processing that we've already done. Even as we move initially into this first state in chart position 1, we've seen one token, and look, there's one token to the left of this dot. We got here by reading in the input, left parenthesis, from chart position 0. Oh, we were expecting a left parenthesis. Great. I just move my finger over to the right. Essentially, I copy this previous rule, P goes to dot open P close starting at 0, and I bring it down here into chart 1. But now I have the dot, I have my finger, before P. If I'm expecting to see a P in the future, I could see another open parentheses, because that's one of the things that P can rewrite to. I could also see nothing, because that's another thing that P could be rewritten to. I've included both of those--they're right here--one, two-- based on this rule and this P--one, two. Brought those two in. I brought them in based on thinking hard about chart one, and they assume that I've already seen this left parenthesis. These two rules start at position 1. Since one of the things I'm considering is that P is totally empty, up here I was thinking, oh, I could've seen a left parenthesis and been expecting a P and then a right parenthesis, but if P disappears then I'm already past it, and I'm expecting a right parenthesis. I got this from this rule up here, which came from chart position 0. Now, we just so happen to know what the actual next piece of input was. We saw an open parenthesis and then another open parenthesis. It must've been that we were looking at this very special rule right here. We saw the open parenthesis and this one was right, and all of the others were wrong. They were possibilities, but they didn't pan out. We took this rule here and move our finger over, shift it past this open parenthesis, so we get open parenthesis dot P close. That's right where we started here. It is bringing over this rule that had previously said starting at 1 it still says starting at 1, but now I do the trick and this is the part where the numbers change. If my dot is right in front of a P, then I could be expecting to see a left parenthesis, we're not going to in the input, but you never know, or nothing. Once again, based on this P, I'm going to start bringing in rules 1 and 2. P goes to dot open P close. P goes to nothing. But this time we started in position 2. This was a very tricky quiz. You should not feel bad if elements of this gave you grief. We're going to go over this in much more detail later on.

1.22 22. 15_l_building-the-chart

It turns out that if we could just build this chart correctly-- and that's not going to be easy, but it's going to within our power-- then we've solved parsing. Let's say that our grammar has some special start symbol S. So goes to E, and then E could be many things. The state we really want to be in is this one. I have seen everything. S goes to E, and there's nothing more. We are totally done. I mentioned before that we have to augment all of our parse states with this starting add information. Just to be a little more specific, I have seen S goes to E, and there was no additional previous information. Starting from zero tokens of input, I have seen enough to make the judgement S goes to E based on this input string. So if the input is T tokens long, we just look to see if S goes to E dot starting at zero is in chart T. If it is, our input is in the language of the grammar. If it's not, our input is not. Parsing totally solved assuming we can build the chart, but building the chart is going to be tricky--tricky but possible.

1.23 23. 16_l_closure

We know how our parsing chart starts on the left. Chart 0 starts with S goes to I haven't seen anything yet, but I want an E, and I'm starting from chart position 0. It'd be really nice if after all T tokens in the input we've got I have totally seen an E, and I'm done with it now, starting from position 0 in the input. I know the start of parsing, and I know the end if parsing, but there's a slight, huge, massive gulf in here-- the excluded middle of parsing that I just don't know how to construct. If only I had some intuition for it. Let's go see how that plays out. We need to know how to make additional entries in our chart. For example, we have S goes to dot E. What do we do? Can we bring some more stuff in? In the examples I've shown you we've added a few more things to chart position 0. I'm going to formally tell you how to do that. I'm going to formalize it using some abstract mathematics. Let's say that we're looking at chart position i. This means we've seen i tokens in the input. One of the things currently in that chart is the following parse state: S goes to E plus dot E coming from state j. This dot means we're expecting to see an E in the future. This is the future. This is the past. I'm going to need to look in our grammar for all the rules that start with E, because if E goes to elephant then I should be expecting to see an elephant in chart state i. If E goes to eggplant, then I should be expecting to see an eggplant in chart state i. We need to find all of the rules E goes to something in the grammar and somehow bring them in. Let me make this very generic to handle all possible situations. Let's say that we've got x goes to ab dot cd coming from position j in chart i. Normally for grammars I always draw nonterminals in blue and terminals in black, but I'm going to leave this a, b, c, and d. I don't know if they're terminals or nonterminals. I don't know what they are. In fact, a may even be empty. A may be nothing. B may be nothing. I'm going to be as generic as I can to handle all the possibilities. I'm not pinning these down to be either terminals or nonterminals. But I do note that our dot is right in front of c. I'm going to look in my grammar for all rules c goes to something. A could be empty. B could be empty. Pqr could be empty. Or they could be terminals. Or they could be nonterminals. Could be anything. For every such grammar rule, c goes to pqr. C goes to anything. Kumquat--oh, that doesn’t start with a c. C goes to chevalier. We believe ultimately that we're going to see a c in the future. If c goes to carbon then we should expect to see carbon in the immediate future. But we don't want to forget that we made this decision starting in chart state i that conceptually there were i tokens before us that we're sort of forgetting about or putting off to the side. Because even though it looks like this dot is right to the left, there are i tokens we've already seen in order for us to get to this point. We add c goes to dot pqr. We haven't see any part of c yet, but we think we might. It's a possibility. We're leaving our options open. We came to this idea from chart state i. That's how many sort of hidden pieces of input we're alighting before the dot. We add that to chart i. We do this for every grammar rule that starts with a c. If there are five grammar rules that start with a c, we're going to add five things to chart i. Formally, this operation of bringing in everything that c could become, because we're expecting to see a c, is known as predicting. I predict, because we want to see a c and c goes to cantaloupe, that we're going to see a cantaloupe. It's also called "computing the closure"--a more technical term from language theory where right before a c any rule that has c on the left-hand side should be brought in to close the state so that all possibilities are considered.

1.24 24. 17_q_computing-the-closure

Let's say we're in the middle of parsing--I've written new grammar for us, here on the left. This one has a new nonterminal F-- just to make things interesting. That is how we roll here, in programming languages. So here's our grammar; it has 4 rewrite rules. The input is: (int - int) but we've only seen 2 of those tokens so far: the (int) and the (minus). One of the elements of chart[2] is: (E --> E -), and Here We Are, and we're looking for an (E) in the future-- and we started this in chart state[0]. I have written out, down here at the bottom, five possible parse states. Why don't you tell me which of these parse states are going to be brought in by chart state[2], by computing the closure?

1.25 25. 17_s_computing-the-closure

Let's take a look at the answers together, It's going to turn out that we'll be able to compute this, fairly systematically, just by remembering the rule for how to compute the closure. We have a dot in front of the nonterminal E, so I go over to our grammar and find all of the rules that start with nonterminal E. And there are 3 of them, and I'm going to add all of them to chart[2] with a red dot right at the beginning and a from2-- because that's where we currently are--1, 2-- as they are little provenance information off here, to the right. So one of our rules is: (E --> int) so we will definitely add (E --> dot int from2). Again, the from2 is because we brought in or we computed the closure, starting in state[2]. Another possible rule is: E --> (F). So (E --> dot (F), coming from position 2) is a valid prediction we might make. We might see parentheses right after seeing a minus sign; that's valid for this language. Now our third rule is: (E --> E - E) so this option may look very tempting. It's got the (E - E). It's got the dot in the front, but it has the wrong (from) state information. It's included from0 instead of from2. We need to remember these 2 tokens we've seen previously--the (E) and the (-). We need to know which state we were in when we decided to take the closure. This one is not correct. Over here we see one that's very similar: (E --> E - E, with a dot in front). That's very good; it's a rule that starts with (E), and we need to start with (E) because the dot is before the (E). And this one correctly has from2. We computed the closure in chart[2]. And finally, this one's a bit of a ringer-- it has (F --> dot string from2). Well, the from2 looks pretty good; the dot at the beginning looks very good. But our rule is: since this dot was before an (E), we take all the grammar rules that start with (E) on the left-hand side, and that's it. So (F --> string)--that's out of place. I'm not going to predict seeing a string until I've seen an open parenthesis. If you think about this grammar, the only way to get to string is after an open parenthesis. I haven't seen one of those, so this is a bad prediction.

1.26 26. 18_l_consuming-the-input

All right. So we just saw Computing the Closure, which is one way to help us complete the parsing chart-- in fact, it's one of three ways. Now we're going to see a second that's sometimes called, Consuming the input or shifting over the input. Shifting over the input, consuming the input, or reading the input is one more way to help complete the parsing chart. We are going to need all 3 powers, combined, in order to make a totally complete parsing chart. But for now, let's worry about the input. So recall that, very generally, we could be in a parsing state that looks like this. This is a rule from our grammar, with a dot added and this (from) information or starting at information added. And I've drawn (a) and (b) and (c) and (d) in purple because we're not sure if they're terminals or nonterminals. We just saw what to do if (c)--the next thing we're expecting-- is a nonterminal. We compute the closure by looking at all the rules that start with (c) in the grammar. But what if (C) is a terminal-- a token, final part of the input? Well, then we'll just shift over it and consume it. If we were in this parsing state in chart[i], that means, after seeing (i) tokens, this is where we could be. I'm just going to take my finger and--whoomp--move it to the right one! So I can just move my finger over, so that instead of expecting a (c), I've seen the (c) if (c) was the ith input token. This is a prediction we're making: (c) may come in the future. I go look at what the user actually entered. If (c) was actually the next token in the input, the next letter they typed in, the next word in the program, then I can shift over it and say great--we have parsed that, that's just what we were expecting, that totally fits our plan--no problems at all. So if we were in chart[i], I'm going to put this new information in chart[i + 1] because, remember the number here in the chart corresponds to how many tokens we've seen. And we're only in this brave new world after we've seen the (c). The (c) was one token; previously we'd seen (i) tokens , so now we've seen [i + 1] tokens. This entire approach is called shifting.

1.27 27. 19_q_shifting

Let's test our knowledge of using shifting to fill out the parse table: the chart. Let's say that this is our grammar: P reduces to or can be rewritten as: open parenthesis P, closed parenthesis. or P can just disappear. Sometimes we write the epsilon and sometimes we don't--whatever we'd prefer. So chart state[0] includes the following parse states: (P goes to: here's right where I am, open P, close) or (P goes to: here's where I am and then there's nothing more), both coming from state[0]. What I'd like you to tell me is: What are we going to put in chart[1] if the input is ( ) because of shifting? What's shifting going to add to our parsing chart? In this multiple choice quiz--actually, there's only 1 right answer. Which one is it?

1.28 28. 19_s_shifting

Well let's go back and remember our rule for shifting. It only applies if we have a dot in front of a token, so we have sort of 2 possible worlds here: this has a dot in front of nothing, and this has a dot in front of a token. We are definitely going to use the dot in front of the token. And then we need to look at that token, and it better match the next part of the input. So we haven't seen anything yet--is the next token an open parenthesis? Oh, it is! We're so happy, this is going to work out perfectly. So then, conceptually, what I do is just shift my finger over one, just a step to the right. So let's go see which one of those that looks like. It should have this dot moved in, inside the parentheses. Well, this does not quite match. It doesn't have the parentheses at all--this represents using the wrong rule. Over here, we've got the right rule, but we didn't move the dot. This is just the parse state we started with previously. So that's not the result of shifting. This, however--this is looking very promising. We've moved the dot over one-- so now it's inside, and we haven't changed this starting offset. We're going to put this in chart[1], and this information assumes we started with zero tokens . Then we saw this left parenthesis and Here's where we are now. Finally, in this last one, We've mistakenly updated the starting position or the (from) information, and this would correspond to some other hypothetical input where we had already seen one token, and now we've seen one more open parenthesis and we're expecting to see a few more. That's not the input we're currently given. This doesn't match--there are no hidden tokens to the left that aren't shown in the rule.

1.29 29. 20_l_reduction

So now we really want to use the full power of our rewrite rules to help us complete the parsing chart. We've already seen 2 possible ways to do it. If the dot is right before a nonterminal, then we take the closure or predict what's going to happen, by bringing in and rewriting rules that start with a (c). If, on the other hand, the dot is right before a terminal, a token--a fixed part of the input-- we shift over it. We just move our finger to the right, assuming that this new token, (c), is exactly what we see in the input. But there is a third case--a "corner" case: actually, that's not hard to draw; something with 3 corners is a triangle-- in which there's nothing after the dot. What if there's no (c) and there's no (d), and we've reached the end? We've done a lot of shifting and our finger is already as far to the right as it can go. Well, now we're going to reduce-- by applying the rule: (x --> a b) in reverse. For example, let's say that the input was: <a b blah> and we were right here--we'd seen these 2 characters-- and one of the rules in our grammar was: (x --> a b). I match up this (a) with this one, this (a) with this one-- and I'm going to take my input and conceptually change it, removing the (a b) and replacing them with (x) as if I'm constructing the parse tree or applying the rewrite rules in reverse. We've seen before how one direction corresponds to string generation, and one direction corresponds to parsing. So once we've matched our predictions exactly, the input has corresponded to (a) and the input has corresponded to (b) and we have rewrite rule: (x --> a b), we're going to apply that rewrite rule in reverse to do parsing--removing (a) and (b) from the input--conceptually-- and replacing them with (x). This is called reduction.

1.30 30. 21_l_magic-reductions

So to get a better feel for this last way to fill out the chart, I'm going to walk through a bit of a parse to show you how it goes. I've got the grammar here in the upper right and here's my input string, and I'm going to abuse notation a bit, by using this red dot again to mean: Where We Are. So, conceptually, one of the first things we'll do is shift. We saw the integer token, and we were expecting that because one of our parse states here was: (E --> red dot int) from zero. But now we want to turn this (int) into an (E) by using this rule, in reverse. And that's a bit magical for now, but we'll see why we want to do it, real shortly. Then we'll want to shift over the (+). It fits with our grammar and it's the input token we saw, and next, we'll want to shift over this (int). And now we'll want to replace this (int) with an (E), just like we did above. So this was another instance of magic or this third new rule that we're going to be talking about. And then, finally, now we've got an (E + E) and if we look up here in our grammar, (E) can be rewritten as (E + E). So we use magic once again, and the process will continue. Each of these times, when we've taken a token or a terminal in the input and replaced it with a nonterminal, has been an instance of reduction. And if we were to view this in the opposite order, we'd see it as a parse tree. At the end of the day, I'm going to ally it a lot of details. We'd end up with this; and the shrinking input, as I replace these terminals or tokens with nonterminals, sort of corresponds to how my tree edges in here--same sort of pattern. I have more and more Whitespace on the left, more and more Whitespace on the left. So the relationship between this magic step and parsing should be pretty clear, but I still need to tell you how to do it-- how do we actually perform reductions.

1.31 31. 22_l_reduction-walkthrough

Suppose we have the parsing state: (E --> to E + E), and we've seen it all. There's nothing more in the future to see for this particular rewrite rule. Coming from chart state [B], we've previously seen (B) hidden tokens that aren't shown here on the left and this is all in chart state [A]. I'm leaving (A) and (B) abstract so that we know how to write a general program that does this. We decided we wanted to look for (E --> E + E) all the way back in chart state [B]-- all the way back, after we'd seen B inputs. So if we view this as our input or sort of the bottom edge of our parse tree, as we're working on constructing it, these are some previous input tokens-- maybe previous ints, previous pluses, maybe parentheses--if we extend our grammar. And at this point, we decided: I think I'm going to see an (E + E) in the future. And now we have. We've seen all three parts of it. So conceptually, it's as if we've seen the (E) right here. Let me firm this up with a concrete example. Let's go back in time and look at where we came from. Suppose I extend our grammar so that is has both Addition and Subtraction--whoa, the power--let's not get drunk! And one of our previous states in chart [B] had been: (E --> to E), minus--this is where we were, and we were expecting an expression in (E). So by reduction--by prediction-- we would have added in expectations to see things like (E + E). And in fact, that's where we got this state--that's why it says "from B"-- we brought it in in chart state[B], based on doing the closure here. Well, now we seen enough input to actually make a single (E). We've seen (E + E); that reduces to (E). So it's as if we have this (E) in our input and we're going to shift over it. In some sense, this third approach--doing reduction-- is like a combination of the previous two. If I've computed the closure in the past, and now we've seen enough of the input to actually use a reduction rule, it's as if I put an (E) in the input and we're going to shift right over it. So I'm just going to shift my finger over one, shift it over this (E)--where did this (E) come from? It came from here. We'd finished seeing all of its subcomponents. We now have a big (E)--whoop! We're done with it. Here's the real trick: this is one of the tricky parts of doing these reductions-- note which chart state I added it to. We added it back to chart[A] because I don't want to forget that we've already seen a lot of these tokens. Remember the particular index we're using into our chart corresponds to how many of the input tokens we have to have seen so far. (A) was the farthest to the right, (B) over here. We definitely want to remember that we've seen all of these tokens in order to get to this point. This is the trickiest rule, so we'll do a worked example together, and then I'll ask for your input. Let's say we have the following grammar--if you look carefully at it, there's actually only one string in the language of this grammar: (a, bb, c). But I've added some extra nonterminals so that we'll get a chance to see how (bb) reduces to a bigger (B), by applying this rule in reverse, and then we keep going and do all the parsing. Unsurprisingly, the input will be: (a b b c). The only string in the language of the grammar, it's "bee-tastic". And now I'm going to work on making the chart. So let's say we haven't seen any of the input yet. We pick our starting nonterminal and, by convention, that's just the first one I mention. And actually, this is all there is; there's only one rule for the starting nonterminal. We haven't seen any of the input yet, and this red dot is not before a nonterminal so there's no closure operations to do. So we're totally fine--that's our entire parsing chart state. Now let's say we've seen one part of the input. We've seen the first token, (a). Well, that matches up exactly with the token we were expecting, to the right of the red dot, so we get to shift over it. And now I want to go back and make sure that we're recording all the right information. Officially, we need (from)-or position information-- for each one of these parsing states. So here, we came from zero--we hadn't seen any tokens yet. This rule, we brought over from the previous state by shifting. It's still (from) position zero. There's nothing in the input that's not already visible here on the right-hand side. But now I'm going to bring in the closure, and we decided to perform the closure in state[1]. So we write a (from1) here. Another way for you to think about this (from) or starting position is: there's really one more token--the (a)-- that would be here on the left, but I'm not including it. So that's the one token we're missing. Here, the next input token is a (b). So can we shift on any of these rules? Well, this has a dot before a nonterminal, so we can't do anything here. But this one has a dot before a token-- and it's the token we were expecting--we are so happy: (B goes to bb from 1) Now let's make the chart for 3 characters in the input. We've seen another (b)--so we go and look back previously. Are there any shifts we could do--oh--we could totally shift over that (b). All right. So here is the moment of truth for performing reductions We have the red dot at the end of a rule. There's nothing to the right of it, no more input to consume. So we're going to apply this (B --> bb) rule in reverse. So we're going to look back to state[1]. We're going to use this (from) information-- Where'd it come from? It came from state[1]-- and see: I could turn this (bb) in the input into a big (B), using this rule. Is there anyone who wants to see a big (B)? Back in state[1]? No--yes, there totally is! This rule here: (T) goes to (a) dot (Bc)-- it's really looking to see a big (B). But we just made one-- by reduction, by applying the rule in reverse. So, conceptually, you could go back and say: Oh--what if we'd seen this big (B) in the input? We've seen it right over here: (abb). Instead of seeing these two lower case (b)'s, we'd see the upper case (B). We'd take this rule-- and transplant it over to this state, being careful to retain the original (from) information. Now let's just interpret this (from) information. Starting from zero tokens, we've seen (abb) in chart state[3], and those are all represented-- they're all encoded in everything before the dot. In some sense, this lower case (a) has length1 and this upper case (B) had length2. Those two, together, add up to three. So I don't need any more hidden tokens to be in the third chart state. So this last part, where I--from here, Step 1-- went back over to here and found this rule, and then brought it back over here-- is reduction.

1.32 32. 23_q_parsing-chart

So let's say we have a slightly more complicated grammar. T is our start symbol; it goes to (p Q r) or (p Q s), so there are two strings in this grammar. It will 100 percent increase, over the previous grammar-- (p Q r) and (p Q s) are both there-- and the input is: (P q r)--wow, that's really lucky! That's one of the strings in the grammar. It's almost as if we planned these things in advance. So now we'll give you a chance to try this out. I have written down, here, five possible facts about the parsing chart for this grammar, on this 3-token input. Chart[1] has this, chart[1] has that. This is an element of chart[1], this is an element of chart[2]. And what I'd like you to do in this multiple multiple choice quiz is check all of the boxes that are correct. So if we do see: big (Q) goes to little (q) dot, from 1 in chart[2], check this box. Check all the boxes that apply.

1.33 33. 23_s_parsing-chart

Well, if you'll permit me to doodle over our grammar, one way to get started with this is to think about what's in chart[0] These two are in chart[0]: (T --> dot p Q r) and (T --> dot p Q s), and we have seen anything yet. So if we start from that, in chart[0], the only operation we can do is to shift to get to chart position [1]. Shift over the input <p> if that's actually the first character of the input. Well, it totally is--that's super convenient. So we're going to move these dots over here and get: (T --> p dot Q r) and (T --> p dot Q s) from zero, in chart state[1]. Oh--so both of the first two are correct. Now we're going to have dots in front of the (Q). So we're going to bring in (Q --> dot q) from 1, in chart position 1. Now we're going to look at the input again and see that the next token is actually (q). So we're going to shift over this and put the result in chart[2]. So in fact, this one's correct as well--wow, we're on a roll! Now the last 2 involve doing reductions. We see: (Q --> q) and we've seen all the input we need by the time we're in chart state [2]. Zero, 1, 2. We've seen 2 characters for the input. So we're going to go back to states that had a dot in front of a big (Q) and say: we have found your big (Q) and we're done with it. So chart[1] had a dot in front of a big (Q). We'll conceptually shift over it and chart[2] will now have the dot after the big (Q). And another way to interpret this is: after seeing 2 characters in the input, we could be here. Well the first character is (p)--great. And the second character is little (q), which we reduce up to big (Q) so, yeah-- this is totally where we are; my finger is right there, starting from zero, with no more hidden tokens on the left. So, in fact, both of these are also right. We will do the reduction and do this sort of pretend shifting, for this first rule, and also for the second rule. In general, we do all the shifting, all the reductions, all the closures possible. Boy, it's a good thing we're going to have a computer program do this for us because doing it, by hand, is starting to get long. This was a particularly tricky quiz, so don't feel bad if you missed a few of these. In the end, though, all of them were right.

1.34 34. 24_p_addtochart

This particular chart-based approach is due to Jay Earley, a computer scientist and psychologist. And now that we've seen all the theory behind it, we're going to code it up in Python together. Now one of the tricks that's different between this and Memo_Fibo is that our chart may have many things in it. One of the differences between this chart-based algorithm and our memoized Fibonacci is that our charts hold groups--lists, sets--of items. After we've seen one token of input, there's an entire collection of parse states that we could be in. So we're going to represent that by having our chart be a dictionary that maps from numbers to lists. But we're going to use those lists-- we don't really want any duplicates, Just like we saw with Memo_Fibo, we want to keep adding things to the chart until we're done. We want to be lazy and not do any extra work. So I don't want to add any duplicate entries to the chart because that'll just be more work for me to do later, and it may not settle down. So I'm going to write a special Add procedure, treating the right-hand side of the chart as if it were a set. And you may have seen sets in mathematics, but if you haven't a list can have elements repeated. I can have this list: [1, 2, 2, 3]. But in a set, each element can occur, at most, once. So if I were to put: {1, 2, 2, and 3} into this set, I would just get: {1, 2, 3} as the final result. When you're about to put something in, you check and see if it's already there, and if it's already there, you don't do anything. I'm going to have you implement that set for me. What I'd like you to do is write a Python procedure, using the Interpreter called: addtochart that takes 3 arguments: the chart, a Python dictionary, the index, a number, and the state--more on that later, but it could be anything-- and it ensures that if I go look at chart[index] later, it will return a list--it will evaluate to a list-- that contains state exactly once. So if state was already in chart[index], don't do anything. If it wasn't, you want to add it. addtochart should return true if something was actually added to chart[index], and return False otherwise-- False if if was already there. And we're going to us this so that we can tell if we actually updated the chart and then if we didn't update the chart,maybe we're closer to being done. To simplify this problem bit, you can assume that chart[index] is valid--that index is already in this mapping and that it returns a list. Let's say it's the Empty list, if we haven't gotten started yet. But it's, at least, going to be a list. Go forth!

1.35 35. 24_s_addtochart

Let's write out one way to do it together. We're definitely going to need an if statement to tell the difference between whether state is already in chart sub index and whether it's not. Let me do the hard case first. I'll check to see if that state is in the list returned by chart bracket index. If it's not, we have to add it. One way to add an element to a list is to make a list out of it and use list concatenation or list append. Here, whatever chart[index] used to contain, we add that to the list containing just state, and we store the result. In this case we return true. Otherwise it's already there, so I should not do any updates. I'll just return false immediately. The key tricks--I have to check to see if state is in chart sub index, and if it's not, I have to make a bigger list that contains all the old stuff we used to have plus the new state we're being asked to add.

1.36 36. 25_l_revenge-of-list-comprehensions

We're going to use list comprehensions to help us write our parser. We were introduced to them earlier, and many computer scientists love list comprehensions because they allow you to state what must be true about a list and let the computer figure out how to get there. Just to remind you a bit of list comprehensions. We use the square bracket to say, "I'm making a list." but instead of listing all of the elements directly, I have some sort of formula. What I really want is to take all of the elements in 1, 2, 3, 4, 5-- let's call each of those x. I want my new list to be all of those squared, so x items x. So 1, 2, 3, 4, 5 should give us 1, 4, 9, 16, 25, and down here in the output it does. In list comprehensions we can also put little clauses or guards to only take some of the input list. Let's say that I only want to square those numbers that are odd. I write everything just the same as before, but at the end I put in this little guard that says "only yield this element if x modulo 2 is 1." That is, if x is odd, if the remainder when dividing x by 2 is 1. The second list only contains 1, 9, and 25 for the numbers 1, 3, and 5. We had our little refresher on list comprehensions. Now we want to parse a string according to a grammar. Let's say our grammar would look like this if we wrote it down on a piece of paper. We need to encode it in Python. Here is one way to do it. I'm going to take all of the left hand sides of all of the rules and write them out as the first part of a tuple. Then all of the right-hand sides, like open P close, becomes elements of a list-- open P close. Here my grammar had three rules, and here my grammar is a list of three elements. The second element corresponds to my second grammar rule-- P on the left-hand side, P in the 0th position, open P close on the right-hand side, open P close in a list on the first position. I'm going to need to do the same thing with parser states. Here is how I might draw on in color on a piece of paper, and here's how I'm going to encode it in Python. This is just one way to do it. We could pick a different way, but this is going to simplify our implementation. There are really four parts of our parsing state-- the left-hand side nonterminal, some list of terminals and nonterminals before the dot, some list after, and j. The right arrow, the dot, and the word "from"--we always have to write them, so I don't need to store them. I'm not going to write down "from" every time. We'll just remember it. I'm just going to make my state a 4-tuple, but instead of it being 1, 2, 3, 4, 1 will be the nonterminal on the left. This will be a list of a and b, and there might be more things here, or there might be nothing at which point it's the empty list. Three will be a list of things after the dot. There might be more things there, or again, there might be nothing, at which point probably we want to use reduction. Then j will just be some integer.

1.37 37. 26_p_writing-closure

Now you, via the interpreter, are going to write the first part of our parser. Our parser is just going to build up this big variable chart. We've already seen how to seed it with an initial value for chart state 0 and also how to see if a string is accepted by a grammar-- it's in the language of the grammar--by checking chart t if there are t tokens to see if it has the final state. Let's say we're deep in the middle of this. We're currently looking at chart sub i, and we see that x goes to ab dot cd from j. We're going to write the following code in our parser. We're going to call a special function called "closure" and pass it the grammar, just as we described before. I, which is going to be a number, that's the chart state we're looking at. X, that's going to be a single nonterminal. Ab, that's going to be this list here--could be empty, could have many things. And cd, that's this list here--could be empty, could have many thing. This closure function is going to return all the new parsing states that we want to add to chart position i. It's going to return a list of next states. For each one of those, we're going to add it to the chart. We've already written addtochart together, and we're going to figure out if there were any changes over all of the things you returned. For example, let's say you returned three things but two of them were already in the chart. Since at least one of those was a change, then we want any changes to be true. This blue code here, this is locked is stone. I'm definitely going to use this. But you can write any definition for closure that you like as long as it correctly implements how the closure is supposed to work. Go forth. This is relatively tricky, and here is my hint. If you're stuck, do a list comprehension over the grammar rules. Remember that you're trying to return states, and every state is pretty much like a grammar rule but with the addition of that red dot somewhere in the middle.

1.38 38. 26_s_writing-closure

Let's write out one way to do this together. I will assign the return value to this variable next_states, and then we'll just return next_states later on, but this will make it a little easier for me to think about it. The hint was definitely to do some sort of list comprehension over the rules in the grammar. That's going to look something like this. For every rule in the grammar put something in our output. Well, if there was something like E goes to xyz in our grammar, and we're bringing in the closure on E, the state we want is really just E goes to the red dot is all the way to the left, and everything that was the right-hand side of the rule comes to the right of the red dot. Remember that we're encoding parse states as simple tuples. The first part is this big nonterminal. Well, that's just the 0th part of the grammar rule. That's just E over here. Then the next part is what's before the red dot. When we're computing the closure there's nothing before the red dot. That's this white space right up here. Then there's what comes after the red dot. Well, that's xyz. That was just the second part of our grammar rule. Then finally we need to know what the current state is. For us, based on our definition of closure, and you can go back and take a look if it has slipped your mind, that's just i--the state we're currently looking at. When we're computing the closure, we add more information to the current chart state. This is pretty much two-thirds of the answer. The trick is there might be other rules in our grammar like T goes to abc, and we don't want to bring them in. We only wanted to compute this closure on E. We're going to need a little guard here in our list comprehension. I don't want to take every rule in the grammar and bring it in. I only want to bring in some of them. Well, what's the thing I'm supposed to be bringing in the closure for? It's based on cd. Cd is whatever we saw to the right of the dot. Remember that our current state is something like x goes to ab dot cd. First I have to check if cd is not empty. If it's not, then c is E, is the thing that we should be looking for. I only want to do this if cd is not empty and if this E, which was rule 0, matches the first part of cd. That is if this E is the same as c. If it is, we bring in the closure, and that's it. This is one of those examples that really shows off the power of list comprehensions. We want to take a bunch of grammar rules, slightly modify them into parsing states, and we only want to do that based on the rules of how the closure is supposed to work.

1.39 39. 27_p_writing-shift

We've seen before that there are three ways to build up the chart. One is by calling the closure or predicting. The next is by shifting, and the third is by performing reductions. I'm going to have you do all three, and this is the second one--shifting. Let's say we're currently looking at chart sub i, and there is a state in there--x goes to ab dot cd from j. This time we're going to look at the input tokens, and they're in a list called just "tokens." I'm going to have the following code in our parser framework. We're going to figure out if there is a candidate next state by calling a special procedure "shift." Shift gets to see the tokens--the entire input, which token we're on, also which chart state we're on, x, ab, cd, and j, the current state we're considering. Based on that there may either be a possible shift or there might not be. Shift will either return None, at which point there is nothing to do, or it will return a single new parsing state that presumably involved shifting over the c if c matched up with the ith token. Then we'd add that to the chart in position i + 1, the correct place, and we'll keep track of whether there have been any changes. You should write shift.

1.40 40. 27_s_writing-shift

Now let's walk through how shift might work together. We can only shift if the next input token, the one we're currently looking at, exactly matches c, the next thing we expect to see. Now, you might have been tempted to have an i + 1 in here, but remember that in Python lists and strings are indexed from zero, so tokens bracket 0 is actually the first element of the input. One of the first thing we have to do is check and see is cd empty or is it something? Well, if cd is not empty, then we can take a look at its first element c, and we'll just check to see how that compares to tokens i. If they match exactly, then we can shift over that token. We're going to return a new parsing state that still has x at the front, but now instead of ab it should have abc, because we're shifting the red dot one. Remember that c was the 0th element of cd. We've shifted the red dot over one, and now instead of cd on the right it's just going to have d on the right. We want to peel off the first element of this list. We can use Python's range selection to peel off all but the 0th element. Let me just make my writing a bit more clear there. This was really from j, and we are also from j. If the stars did not align--either if cd was empty or it didn't match up the next token-- then we were supposed to return None.

1.41 41. 28_p_writing-reductions

Now we are ready to finish off our parser. There were three ways to add to the chart, and you've already done one and two, computing the closure or predicting what's happening next and shifting. The last way is reductions. Let's say once again that we're looking at chart i, and one element of it is the state x goes to ab dot cd coming from j. I'm going to write this code. I'll lock it into our parser. Next_states equals reductions, a function that gets to look at the chart, i, x, ab, cd, and j. It's going to return a list of possible next states. For each one, we will add it to the chart and notice if anything changed. Reductions are relatively tricky, so my first hint for you is you only want to do reductions if cd is empty. Remember reductions only apply if the red dot is as far right as possible. My other hint for you is that you'll have to look back previously at the chart. Remember when we worked through examples together, we'd start over here, go back over the chart, and then go back to the right. You'll have to do the same sort of thing, so we're passing in the chart. Try it out. This one's a bit tricky.

1.42 42. 28_s_writing-reductions

Let's go through one way to do this. Hopefully what we're currently looking at is x goes to ab dot nothing come from j. Hopefully then, if we look back to chart j where we originally came from, it will have some rule something go to blah, blah, blah dot x. This is the important part. We're reducing goes to ab, so I really hope somebody was looking for an x. If they were, then that can be one of our reductions. Once again, we're going to use the phenomenal cosmic power of Python list comprehensions. In general, we're going to take all of these states that were already in chart j and just modify them a bit. Let's call each one of those states in chart j jstate. Conceptually, what we're going to do is move the red dot over one. Our return value, the new state we're returning, is going to have this same y that we saw from jstate. Whatever that is that's still going to be our left-hand side. Then we want to take whatever jstate had before the dot, and that corresponds to all of this stuff that I've sort of left out here, but then add on x, because we're shifting over x, conceptually, as we do the reduction. Now we want to take everything jstate had after the dot, except we want to remove the x, because we shifted the red dot over it. Everything jstate had after the dot was j-state bracket 2. and we're going to do range selection on that to get rid of the first element. Then it looks like I can't preplan. Whatever this k value was, we're just going to leave it alone. Jstate 3 corresponds to k. However, we only want to do this if certain conditions hold. First, cd has to be the empty list which corresponds to this red dot being as far to the right as possible. The second thing we have to check for is that this x and that one match exactly. This x was the first element of jstate 0 1 2, so I'll check to make sure that jstate 2 is not empty. If this red dot were all the way to the right, there would be nothing there to check for. If it's not empty, I'm going to check its first character and make sure that matches up with our x. Those are all of the states we bring in as part of doing reductions.

1.43 43. 29_l_putting-it-together

All right. Now that you've gone to the hard work of defining all of those procedures, let's see the big payoff as we use it to parse arbitrary strings in arbitrary context-free grammars. Here I've got the procedure addtochart that you wrote. We have the procedure closure that you wrote, once again defined using list comprehension. We've got shift, which either returns something or nothing. We've got reductions, which has a complicated return based on jstate in the chart. Way up at the top of this file, I've defined a particular grammar. It's that grammar of balanced parentheses. This is just our encoding of start symbol goes to P, P goes to open parentheses P, P goes to nothing, and then down here I've got a candidate input open open close close. That's in the language of the grammar, so I desperately hope that our procedure is going to find that out. Down here I have the parsing procedure skeleton that I wrote around your code that does the heavy lifting. One of the first things I do is take all the input tokens and add in a special end_of_input marker. That's because sometimes we need to look ahead, for example, for shiting to see if the input token matches what's there, and I don't want us to walk off the end of a list. I'm just sort of padding out data structure by one. Here is the chart. It initially starts out totally empty. It's a Python dictionary with nothing in it. Our starting rule is just the first rule in the grammar. That's by total convention. We're going to start with S goes to P. I pre-initialize all of the elements of the chart with the empty list. Remember in that quiz I let you assume that it would always a well-defined list. I'm making that true here. Then our start state just works on this start rule. It uses this same symbol S. There is nothing before the red dot. Then we've got the red dot, then we've got the P, and that's started in state 0. Initially, the only thing in our chart is that at chart position zero we have this starting parsing state. What we're going to do is be super lazy and write ourselves a bunch of memos in this chart. Over and over again we're going to consider additional characters in the input and keep using your three procedures of closure, shifting, and reduction until there aren't any more changes. I is going to range over all of the possible tokens. Then until there are no more changes, we consider every state in the chart, and the state is something like x goes to ab dot cd from j. I just extract those into conveniently-named variables by pulling out the 0, 1, 2, 3rd element of this tuple. Now we're going to go through 3 options that correspond exactly to the work that you did. If the current state is a ab dot cd, we could compute the closure. If c is a nonterminal we look for each grammar rule c goes to pqr, we make a next state, blah, blah, blah. Here we're about to start parsing c, but c may be something like expression with its own production rules. We want to bring those in. Here is the code that I promised you in the quiz that I would write. Next_states is a called to closure. You implemented closure. Then we checked to see if there are any changes. In addition to the closure, there is also the possibility that we're going to do shifting. Ab dot cd, and if the tokens are c, if the next token is c, then we're totally going to shift. We're looking for parse token c, and the current token is exactly c. If that happens we are super lucky. We can parse over it and move on to j plus 1. Here is the code that I promised you in the quiz I would add, and there it is. Finally there is our third option for computing reductions. This one is the most complicated. If cd is empty we then we go back in time to chart j and bring something from it forward to this current location. You just finished implementing that. Down here we have the code that I promised that I would include in the parser that calls your function reductions. Then we're just going to keep repeating this until nothing changes. Remember that this was in a while true loop, so we're going to loop over and over and over again until there are no changes, and then we break out of the loop. Down here I have some purely debugging information. This is all just to print out the chart at the end so that we can take a look at it. We wouldn't actually need this if we were doing a parser. This is for explanatory purposes only. Then down here I've defined the accepting state. We reasoned to this earlier, which is basically the starting state, but with everything to the left of the red and nothing to the right of it, coming from state zero. If the accepting state is in the chart in position t when there were t tokens, then we parse the string successfully. Otherwise we do not. Down here, I'm checking to see what this value is. Is this string in the language of the grammar or not? We just print that out. In this particular example, the string is in the language of the grammar. Hopefully, that's what we'll see. Well, our output is quite voluminous. We see chart position 0, chart position 1, chart position 2. In fact, this was our starting state, S goes to dot P from 0. Then we brought in these other two from the closure. A good quiz question to ask yourself is why do we have this one--S goes to P dot from 0? My hint is P can go to nothing, so actually the empty string is accepted by this grammar. We end up filling in 0, 1, 2, 3, 4. This one actually corresponds to look ahead, that sort of end of input symbol that we saw there. Eventually we discover that wow, our string is in the language of this grammar. We're so happy. This is exactly what we wanted. If I were to change this a bit. I've been very minorly devious. Now, instead of having balanced parentheses, I have three open followed by one close. Now I've changed it so that the strings shouldn't be in the language of the grammar. We have three opens followed by one close. I click run. The chart is actually going to be very similar at the beginning, changing only near the end--possibly a little bit. But now we report that the string is not in the language of the grammar as expected. Here is the code for the unit:

```
def addtochart(chart, index, state):
    if not state in chart[index]:
        chart[index] = [state] + chart[index]
        return True
    else:
        return False
```

```
¶
def shift (tokens, i, x, ab, cd, j):¶
···· # x->ab.cd from j tokens[i]==c?¶
···· if cd != [] and tokens[i] == cd[0]:¶
········ return (x, ab + [cd[0]], cd[1:], j)¶
···· else:¶
········ return None¶
¶
def reductions(chart, i, x, ab, cd, j):¶
···· # ab. from j¶
···· # chart[j] has y->... .x ....from k¶
···· return [¶
········ (jstate[0], jstate[1]+[x], (jstate[2])[1:], jstate[3])¶
························ for jstate in chart[j]¶
································ if cd == [] and jstate[2] != [] and (jstate[2])[0] == x ]¶
¶
¶
def closure (grammar, i, x, ab, cd):¶
···· #x->ab.cd¶
···· next_states = [¶
········ (rule[0], [], rule[1], i)¶
········ for rule in grammar¶
············ if cd != [] and rule[0] == cd[0]]¶
···· return next_states¶
¶
def parse(tokens,grammar):¶
·· tokens = tokens + [ "end_of_input_marker" ]¶
·· chart = {}¶
·· start_rule = grammar[0] # S -> P¶
·· for i in range(len(tokens)+1):¶
···· chart[i] = [ ]¶
·· start_state = (start_rule[0], [], start_rule[1], 0)¶
·· chart[0] = [ start_state ]¶
·· for i in range(len(tokens)):¶
···· while True:¶
······ changes = False¶
······ for state in chart[i]:¶
········ # State ===·· x -> a b . c d , j¶
········ x = state[0]¶
········ ab = state[1]¶
········ cd = state[2]¶
········ j = state[3]¶
¶
········ # Current State ==·· x -> a b . c d , j¶
········ # Option 1: For each grammar rule c -> p q r¶
········ # (where the c's match)¶
········ # make a next state··············· c -> . p q r , i¶
········ # English: We're about to start parsing a "c", but¶
········ #· "c" may be something like "exp" with its own¶
········ #· production rules. We'll bring those production rules in.¶
········ next_states = closure(grammar, i, x, ab, cd)¶
········ for next_state in next_states:¶
·········· changes = addtochart(chart,i,next_state) or changes¶
¶
········ # Current State ==·· x -> a b . c d , j¶
········ # Option 2: If tokens[i] == c,¶
········ # make a next state··············· x -> a b c . d , j¶
········ # in chart[i+1]¶
········ # English: We're looking for to parse token c next¶
········ #· and the current token is exactly c! Aren't we lucky!¶
········ #· So we can parse over it and move to j+1.¶
········ next_state = shift(tokens, i, x, ab, cd, j)¶
········ if next_state != None:¶
············ any_changes = addtochart(chart,i+1,next_state) or any_changes¶
```

```
¶
········ # Current State ==·· x -> a b . c d , j¶
········ # Option 3: If cd is [], the state is just x -> a b . , j¶
········ # for each p -> q . x r , l in chart[j]¶
········ # make a new state··············· p -> q x . r , l¶
········ # in chart[i]¶
········ # English: We just finished parsing an "x" with this token,¶
········ #· but that may have been a sub-step (like matching "exp -> 2"¶
········ #· in "2+3"). We should update the higher-level rules as well.¶
········ next_states = reductions(chart, i, x, ab, cd, j)¶
········ for next_state in next_states:¶
·········· changes = addtochart(chart,i,next_state) or changes¶
¶
······ # We're done if nothing changed!¶
······ if not changes:¶
········ break¶
¶
## Comment this block back in if you'd like to see the chart printed.¶
¶
·· for i in range(len(tokens)): #print the chart¶
··· print "== chart " + str(i)¶
··· for state in chart[i]:¶
····· x = state[0]¶
····· ab = state[1]¶
····· cd = state[2]¶
····· j = state[3]¶
····· print "··· " + x + " ->",¶
····· for sym in ab:¶
······· print " " + sym,¶
····· print " .",¶
····· for sym in cd:¶
······· print " " + sym,¶
····· print "· from " + str(j)¶
¶
·· accepting_state = (start_rule[0], start_rule[1], [], 0)¶
·· return accepting_state in chart[len(tokens)-1]¶
¶
grammar = [¶
·· ("S", ["P" ]) ,¶
·· ("P", ["(" , "P", ")" ]),¶
·· ("P", [ ]) ,¶
]¶
tokens = [ "(", "(", "(", ")", ")"]¶
result=parse(tokens, grammar)¶
print result¶
¶
```

1.44 44. 30_l_prisoner-example

Now, one of the big draws of having a universal parser like this was that I could fill in any context-free grammar and check any string of tokens against it. For example, here I've defined a new grammar that accepts the word "prisoner" followed by a list of numbers. N is a list of numbers. It's at least 1, but you can have more. This is a recursive rule so we can have as many as we want, and I've gotten lazy. We only put in 0, 1, 2, 3, 4, 5, 6, but I could go all the way 7, 8, 9, 10. One of my favorite prisoners is number 6. Let's go see if this string, prisoner 6, is accepted by this grammar. Here the chart is a bit bigger, because we have sort of a separate state for each one of these. This makes us glad that the computer is doing the memorization instead of us doing it by hand. But down here at the end we accept. By contrast if I just have the word "prisoner," this shouldn't work, because this list requires 1 or more integers. And in fact down here we can see that it is not accepted. Let's do just one more of these. If there were another prisoner vying for the affection of my heart, I'd ask you to bring me prisoner 24601. Perhaps his time is up and his parole has begun.You know what that means. Let's check and see if the string is accepted by the language of the grammar. Here, all the way down at the end of the day, we see that prisoner 24601, famously Jean Valjean from Victor Hugo's Les Miserables, a nice piece of French literature, is accepted by the language of this grammar. But we have a large number of chart states--5, 4, 3, 2, 1, 0--to accept this string.

Let's do one more of these just to show off our very arbitrary power. Now I've put in the B grammar from before. We know how this one is supposed to work because we did it out together on paper. The input string I've put in is abbc, and that string is in the language of the grammar. If I forget one of the b's, we expect it not to be. When I forget one of the b's it is not in the language of the grammar. The real trick is basically that you have done it. This is enough of a parser to be given a formal grammar for JavaScript or HTML and determine if a string, a webpage, a program is in that language. This is very exciting.

1.45 45. 31_l_parse-trees

So now we have all the machinery we need to tell if a string is valid. However, it's going to turn out that's not enough. Remember those upside down parse trees we talked about earlier? We really wanted those, as well, for our HTML and JavaScript programs in order to interpret them--to get at their meaning correctly. So here, I've written a pretty standard arithmetic expression grammar. An expression can be a number or an expression plus an expression or an expression minus an expression or maybe a negated expression, like negative 3. And we'll want to build up parse trees for this. Now this time, I've written the tokens as plus and minus and not instead of the symbols, + or -. That's our choice; we can do it either way we want. And the particular format I'm going to pick for our abstract syntax tree is nested tuples--or nested lists, in Python. So if we end up using this rule: expression goes to number, we're just going to return the tuple: ("number", 5)--if the input was 5. Similarly, if the input is something like: not 5, We'll end up returning: ("not", of the "number", 5). Note the nesting. So let's say I call this number: number 1. We really want to return this tuple: "number"--in quotes, just as a string, so we know what it is-- followed by the value of the token. If this was Thing Number 2 in our reduction rule-- not expression--I'd really want this to be filled with a 2. If over here, this was a 3, I would want to return "binop". That stands for Binary Operator, binary just meaning "two things". So things like: Plus, Minus, Times, and Divide-- those are arithmetic operations that take two arguments-- one on the left, and one on the right. We call those Binary Operators, as a class, just to save space. But whatever this third expression was, that's what I'd want this subtree-- this subpart of my tuple--to be. So just as we've seen before how we can encode token rules in Python and do some processing, like chopping off the quotes after we've specified how the token works, using regular expressions, it's going to turn out that there's a similar way for us to do that for grammar rules in Python. Now let me explain a little bit about what's going on. This format is totally arbitrary, but it's going to be very easy for us to use for assignments and to test your knowledge. For tokens, we used a "t_" to mean I'm defining a rule for a token. For parsing rules, we're going to use a "p_" to define the name of a parsing rule. And then here--just to help us out-- we're going to write down what the left-hand side of the rule is. This is how you parse an expression when that expression is a number. And just as out token rules were, in some sense, under-the-hood functions of this object, (t), our parsing rules are under-the-hood functions of this object, (p). And this is the parse tree-- or, more accurately, a number of parse trees. Here's our rule, written out, and this is very similar to: (exp --> number)--except that there's no great way to write the arrow, so instead, we'll just write a colon, by convention. But you should view this as the arrow. So this is: expression can be rewritten as NUMBER, and we just put it in quotes, like a string, and then down here we have to tell Python--or tell our parsing library-- how to build up the abstract syntax tree. p[0] is our returned parse tree. The numbering here is every one of these elements of our grammar rule-- except the colon gets a number. So the expression on the left is zero, This NUMBER over here is 1. So the parse tree I want associated with this expression, when we're all done, is a tuple that I make, by combining the word "number" with the value of this actual token. Let me show you another one of these, and then it'll be a little clearer. So here, once again, I start with the (p_). We're going to do that for all of our parsing rules. Here's what I'm telling you how to parse; I'm telling you how to parse an expression. There might be multiple different ways to parse an expression. It could be a number, it could be a (not) expression. So we use another underscore, in being a little more specific. And then down here I've written out my grammar rule in almost English--and again, this colon is like the arrow that we would normally draw. And then below that, I have written out how to construct the final abstract syntax tree. This expression is number zero, this (not) is number 1, This expression is number 2, so we want our parse tree for number zero to be the tuple I make, by putting the word "not"-- so that I know what it is--in front of the parse tree for number 2. If we were to see the input: NOT 5 executing these two rules, in the right order-- this one first, and then that one-- would give us this tree: "not", ("number", 5). Note the nesting. I could alternatively draw it as-- This is just a Python was of encoding this visual representation.

1.46 46. 32_p_ramas-journey

So let's say that we've put in these 3 parsing rules into Python. We are trying to get back to parsing HTML. We want to understand how to parse Web pages and turn them into abstract syntax trees. And a Web page is just a list of elements. So it's either an element, followed by More or We're Done Now. And elements could be a number of things, like tags or maybe-- more simply, just words. Suppose our input is two words: Rama's Journey. What I'd like you to do, as a quiz, is submit, via the Interpreter: Define a variable that holds the final parse tree for this input. And again, the input is 2 words: Rama's Journey.

1.47 47. 32_s_ramas-journey

Well, let's think it through together, and then write out the details. At a high level, this is going to be: an element , an element, and then Empty. So we're going to end up making a list that has element 1 in it and element 2 in it, and then nothing else-- based on this rule up here, where we just make a bigger and bigger list, out of the list containing the first element and everything else we've gathered up. So this will be the final value of our parse tree, corresponding to this more graphical parse tree-- really more of a list, but a lot of things in computer science are. Rama's Journey is more commonly known as the Ramayana. It's a Sanskrit epic that's a very important part of the Hindu canon and it explores human values and Dharma. If you haven't already had a chance to read it, I strongly encourage you to make a rendezvous with Rama. It's time well spent.

1.48 48. 33_q_parsing-tags

Let's say we want to continue formalizing our HTML grammar in Python. One of the other types of elements in HTML, aside from bare words, is tag-decorated words. You might put <bold> or an <anchor> or even something more complicated, like this, that changes the color. So just to remind you of what these HTML tags look like, they start with this Left Angle, there's some name; they might have some arguments, there's a Right Angle; there can be any HTML in the middle; then there's this Left Angle Slash, another word, and a Right Angle. And here, I've just written out that grammar rule: this LANGLE corresponds to this part. this word goes here, tag arguments--color = "Red", Right Angle is that one, HTML is here, LANGLESLASH is these two, and so on. And, here, I'll build up my parse tree by using part[2]: zero, 1, 2, the word, like span or bold or underline; the tag arguments--if there are any, the body--the words that are being modified by bold or underline, and then, finally, the final word-- just to make sure, later on, that you've opened <span> and closed </span> or opened <bold> and closed </bold>. Remember, we want those parentheses to match. And our input text is: hello <baba> yaga, and we've got <baba> bolded. I'm going to ask you to take apart this concept and do it backwards. I have written out the parse tree, down here at the bottom, but I've left 3 blanks. I would like you to fill in the blanks with a single word that will make this parse tree correspond to what our parser will produce on this input.

1.49 49. 33_s_parsing-tags

Well, let's go through it together--our parse tree is just going to be a list of elements, and here there are three: hello, the tag element, and yaga. And hello is just a simple Word_element so it fills in our first blank. Then we've got this tag_element and the trick to getting this question right is looking at the order in which we store them up here-- more or less in order of appearance. So since this is a <bold> tag, when this next part here is a (b), this empty list means there were no particular arguments to our <bold> tag. Here, I'm seeing arguments: color = "red". There's nothing like that down here. And then inside, we've got the Word_element, baba--and then we're done. Baba Yaga was a crone or a witch in Slavic folklore who was known for--among other things-- riding around in a house supported on chickens' legs--fun stuff!

1.50 50. 34_l_parsing-javascript

Take a bow--we are done parsing HTML! That was it; we've seen all of the relevant rules. Well, I haven't actually shown you the detail for handling tag arguments, but we'll get a chance to look into that later. For now, let's go on to JavaScript, which is actually going to have very similar rules to HTML. You may have already guessed that a lot of the options in JavaScript are very similar. We have a large number of binary operators: Plus, Minus, Times, Divide, Less Than, Greater Than-- and it turns out that there is a very convenient notation when we're programming grammars, for getting those all in. Rather than having to make a separate little parsing function-- or parsing rule--for each one, I can just write out multiple related parsing rules in the same string, and give one piece of code that applies uniformly to all of them. So we said before that this colon kind of meant the arrow. This vertical bar--it's as if we had written this same nonterminal (exp) one more time, and then another arrow: expression goes to expression times expression. This is just a concise notation for your benefit, so that we don't have to type out as much. And here, I'm showing how to make a abstract syntax tree, which, again, for us is just nested tuples for a JavaScript binary operator. And, in reality, we'd want to add in another rule for Divide, Less Than, Greater Than-- we might have ten of these at the end of the day. But it turns out that our old friend, Ambiguity, is going to rear its head. If my input is: (1 - 3 - 5) there are actually two ways we might interpret that-- or two parse trees we might end up with-- and, depending on which one we pick, we get a slightly different answer. This could mean: (1 - 3) - 5, at which point, we'll get (-7). Or it could mean: 1- (3 - 5), at which point, we'll get (3). We say that this first option is what is known as Left Associative because it puts the parentheses on the Left or the tree ends up being sort of unbalanced towards the Left. Similarly, this second option is Right Associative.

1.51 51. 35_q_resolving-ambiguity

Well, just to make sure that we're following along with this, a brief quiz: If subtraction is Left Associative, what is: 1 - 2 - 3 - 4? Fill in the blank--single numeric answer.

1.52 52. 35_s_resolving-ambiguity

Hmmm--it turns out the answer we're looking for is -8. And here's how to see it: If Subtraction is Left Associative, then we want to put as many of these parentheses as far to the left as possible. So we're going to do (1 - 2) first, and then subtract 3--and then subtract 4. So (1 - 2) is -1. (-1 - 3) is -4. (-4 - 4) is -8. And that's the answer we got.

1.53 53. 36_l_precedence

So by this point, we've totally conquered Ambiguity, right? Ah--not so right. Even if I know whether an operation is Left or Right Associative, I'm still not sure, when there are multiple operations, which one to do first. I could do the Multiplication first, and get: 8 + 6 is 14 or I could do the Addition first, and get: 2 * 10 is 20. This isn't the same problem as associativity because it's not about whether we're associating to the Left or the Right, it's about the--sort of the precedence of these Operators. which one is more important, which one should I deal with first--which one binds more tightly. In Standard mathematics, we'd want to do the multiplication first. Multiplication has higher precedence than Addition. It gets serviced first.

1.54 54. 37_q_higher-precedence

Just to make sure that we're all on the same page about Precedence and the difference between Precedence and Associativity, let's say that Multiplication and Division have higher Precedence-- that is, you should do them first-- compared to Addition and Subtraction--and this is how we normally do things. Then what is: 3 * 4 - 8/2? Fill in the blank for this quiz.

1.55 55. 37_s_higher-precedence

Well, the answer we're looking for is: 8. We just multiply the (3 * 4) first--that's a 12. And the 8 over 2--that's a 4; we do that first. We have to do both of those things before we can do the Subtraction because they have higher precedence. So we end up with: (12 - 4) is 8.

1.56 56. 38_l_setting-precedence

Precedence and associativity were not so tough, which is convenient because they're also super easy in Python. For our parser, in Python, we can just write out a table: a single variable, called Precedence, that lists lower precedence operators at the top and higher precedence operators at the bottom. And I know what you're thinking: this is totally reversed-- and you're exactly right, but it's not the first thing we've grown upside down in computer science programming languages. So lower precedence operators that bind very weakly--up here at the top; higher precedence operators that you have to do first--down here at the bottom. And we can also indicate their associativity at the same time. So we're going to have our precedence and associativity. This says that Times and Divide are both left-associative and they're very high precedence. Plus and Minus are both left-associative and they're lower precedence. And then our parser will automatically get rid of the ambiguity for us by using these rules.

1.57 57. 39_p_optional-arguments

So we're going to test out that knowledge by having you submit via the interpreter, as a quiz-- some parsing rules that are going to handle a real part of JavaScript. Now, a lot like Python, JavaScript allows function calls-- you write out the name of your function and then you just pass in some number of arguments, possibly none. For that particular function, we would want the parse tree to be a tuple. That's how we're representing parse trees. The first part is "call"--telling us that it's a call expression. The next part is the name of the function, and then there's a list of all of the arguments. And this list may be empty if there are no arguments or it may contain expression parse trees. And, in fact, I'll do the first and second parts for you. Here's a rule for making expressions that are function calls: That's an identifier, like "myfun", followed by a Left parenthesis, followed by some optional arguments, followed by a Right parenthesis. And we just build up our parse tree out of a tuple--the word "call", the identifier--that's p[1]--zero, 1, 2, 3, 4-- and the optional arguments are position[3]-- I definitely need the (p) there--there we go: p[3]. Similarly, our rule for expressions that could be numbers, expression can become a number, at which point, I just make up this tuple, "number", followed by the actual value. and that's how we got things like this for ("number", 11). So here's the quiz: I'd like you to fill in the value for parsing optional arguments, and you may find that you have to define a few more of these parsing rules-- maybe some for there being no argument, some for there being at least one, and that kind of thing--try it out.

1.58 58. 39_s_optional-arguments

All right. Let's go through it together--for optional arguments, there are two possibilities. The arguments could be: at least one--some real arguments-- or they could be Empty. The first two lines are going to look a bit like this. We have to call our nonterminal (optargs) because that's what I called it up here, and it has to match. I'm just going to make up this new nonterminal, (args), meaning "one or more". If we don't see any of them, we can return the Empty list. This means there's one or more arguments so I'll just pass the buck and assume that (args) is magically going to make, for me, the answer I want. And I'll just copy it from p[1] into p[0]. One of the real tricky parts of handling arguments is that they're separated by commas, rather than terminated by commas. So once you have your first argument-- if you're going to have a second, you need a comma but otherwise, you don't. This seems a little weird when we say it verbally, but if you look at it, it's more or less just what we expect. If there's only one argument, then it's just an expression--some number. But if there are more, then we put commas in between them. So here, for multiple arguments, we have an expression, a comma, and then any more arguments we like. And then, finally, we get to the last one, which is just an expression. For this one, we'll just make a Singleton list out of the only argument you gave me. So this would be the list, containing (1), for this example up here. And in this other case--myfun(2,3)-- we take this first part--just the (2)-- and we put it in a list by itself, and then we use (+) to get list concatenation-- list.append--to put this single element list together with all of the rest of the arguments we've gathered up. That's it for defining parsing functions for function calls and arguments.

1.59 59. 40_l_interpreting-languages

So now we've seen all the gory details in how to implement parsing for languages like HTML and JavaScript. Recall that parsing takes some tokens as input and produces a parse tree-- something like this, perhaps. In our next exciting episode we're going to learn how to interpret languages by walking over their parse trees. For example, maybe this is equal to 7. Let's find out.

1.60 60. 41_l_conclusion

You've just learned how to encode a grammar for HTML and JavaScript, and that is no mean feat. In fact, a number of years ago, for my research, I had to do something similar-- but for perhaps an evil or more production, more popular language--Java. We wanted to analyze Java programs, to look for particular errors and there weren't really any convenient parsers available at the time. So I was faced with a decision: Should I use a tool that didn't really fit the bill? Or should I try and write my own? And I thought, boy, Java's a Real World production language; it's got to be really hideous to write down a parser for it-- I'm sure its grammar is really hard to follow. So I figured I'd give myself a day to look at the official Java grammar and try to write a parser for it, using the same sorts of techniques we've covered in this class. Imagine my surprise, when it turns out that the official Java language specification actually uses the same sort of format-- the same sort of context-free grammar that we've been going over here. In fact, if you'll take a look, their handling of if-then-else statements or argument lists should look very familiar to you. It's, more or less, exactly what we covered for JavaScript. And I ended up writing a parser for Java 1.1 at the time-- this was many years ago--that worked for our research. I was able to make a tool that fit me, even though there was none available, using exactly the sort of techniques that you have just mastered through.

1.61 61. 42_l_memoization-at-mozilla

So one of the topics we're covering in this course is Memoization or caching, writing down values that we've already computed so that we can be lazy and not have to recompute them later. We've used this as one implementation of parsing but in the Real World, it might come up in many other places. Brendan, have you had a chance to use it at Mozilla or in your other projects? Not so much in parsing because we are in a tight competitive regime with other browsers, and you have to parse very efficiently and lex very efficiently. But in building a browser, you end up memoizing or caching a lot. You end up trying to remember decisions you made that were expensive, that can be preserved under some rules and reused. We also have a memory management system that's quite complex. We have a garbage collector for JavaScript, we have a reference counting system for our C++ code. The two have to meet, and it's possible to form cycles, which then have to be collected. We use an old David Bacon data programming trick. It's Bacon and Rajan PLDI 2001, I think. It's a cycle collector for reference code objects, and it buffers pointers that it suspects of forming cycles. And a pointer, in a reference kind of graph, can form a cycle. Every reference count is going down, from above 2 to 1 or above so that you may still have a stuck reference there, due to a circle. And so we use these techniques. We constrain a little bit in using C++ that has to run on Windows, Mac, and Linux. But in Python and JavaScript, people memoize all the time. And there's a craze now for PEGs-- Parser Expression Grammars? I think that's the acronym; I always mix it up with something else. But it's a parsing technique that can use memoization when it's backtracking and accord it Choice, so we're familiar with this, and we try to use memoization to a good effect. At Mozilla we have less use in parsing--more runtime, I'd say.

1.62 62. 43_l_abstract-thinking

Excellent work! You've just completed your first tough question on filling out the charts for parsing. Because parsing is everywhere in the computing world-- from HTML to E-mail, to languages like JavaScript or Python or C and C++-- this notion of memoizing-- writing down intermediate results so that we can refer to them-- is critical to performance; and because parsing happens all the time, this performance really matters. Now these are some of the first questions to really stretch our ability to do abstract reasoning, so don't worry if it's taken you a few more trials on these exercises than it may have in the past. On the other hand, we should definitely be excited about the goal that we're working towards. Very soon, we're going to have a complete parser for HTML and JavaScript.