CS262 Unit 2

7

Gundega

Contents

1 CS262 Unit 2

Contents

1.1 1. Introduction

Last unit, we learned about regular expression and finite state machines. In this unit, we're going to put those concepts together to make a lexical analyzer--a program that reads in a web page or a bit of JavaScript and breaks it down into words, just like I might break and English sentence down into words. This is going to be a really important tool in our arsenal, and it's one of the first steps towards making a web browser,

1.2 2. Welcome back

Welcome back. In our last exciting episode we learned about regular expressions, a concise notation as a way to write down or denote or match a number of strings. This 4 through 7 in brackets corresponds to 4 different strings 4, 5, 6, and 7. We learned to write more complicated regular expressions like this one-- "b a +."--this plus means one or more copies of a's, yielding words like ba, baa, baaa, and eventually yielding my sheep. I assert that it's a sheep. You can tell because of the label. Those labels never lie. We also learned how you can use regular expressions in Python by importing, bring in, the functions and data types from the regular expression library. An example of such a function was findall, which, given a sort of needle regular expression, would return all of the places in the haystack that it matched. We also learned that you could turn regular expressions into finite state machines. This finite state machine accepts the same language as our ba, baa, baaa regular expression from above. Starting in a start state, on a b we transition to the middle state, on an a we end up in the third state, which is an accepting state. You can tell by the double circle. Then there's a self loop back. That was last time.

1.3 3. Specification

Now, we're going to learn how to specify important parts of HTML and JavaScript, and, in an incredible surprise move, we're going to do this specification using regular expressions. Just a quick reminder, an outline of the overall project, we want to start with a web page and then break it down into important words. Maybe the less than and the greater than sign used for the tag are important words, the 1, the plus, and the 2, but we're largely ignoring this sort of white space or these new line characters. Then we want to take those words and diagram them into this cool tree-like structure. Of course, this tree is growing upside down, but that won't be a problem at all. Finally, we're going to interpret that tree to figure out what it means to get the result.

1.4 4. HTML

HTML stands for the "hypertext markup language." Many of you may have some previous experience with HTML, but that's not necessary. HTML as w know it was invented to Tim Berners-Lee, a British computer scientist working in Switzerland around 1990. For our purposes, HTML just tells a web browser how to display a webpage. In fact, HTML is not all that different from using symbols like stars or underscores to emphasis text that you're writing to someone else. In HTML this emphasized plain text becomes "I," and then this special punctuation that means let's do some bold now, "really," this special punctuation that means I'm done with bold,

let's go back to normal, and then "like you." The "b" stands for "bold." Let's go see how that pans out. Here in this particular window, I'm showing the raw HTML source on the left and how it might look in a web browser on the right. Here when I've added the bold tags around really, we see "I really like you." The "really" is rendered in bold. Other comment approaches in HTML for emphasis are the use of underlines, the "u," and italics, "i." Each one of these is called a "tag." This special syntax with the "b" in angle brackets and the "b" sort of similar angle brackets, the "u," the closing "u," the "i," the closing "i," is a tag that's associated with that word, or that span of text, and tells the web browser how to render it. This part here, the left angle bracket and the right angle bracket--that's a starting tag, and this other part--the left angle bracket followed by a slash, the slash is super important, begin an end tag. That's a little complicated to say. Mark the start of an ending tag and tell you that the current tag is about to stop. Here's a beginning bold tag. Here's an ending bold tag. You can see that play out on the right. Only the word "really" is bolded.

1.5 5. Really

Let's check your knowledge of that with a multiple choice quiz. Here a number of times I've written the sentence "George Orwell was really Eric Blair." You might think I've written it, say, 1984 times, but really, just four. I hear if you repeat something like this enough, it becomes true. What I'd like you to do is mark in this multiple multiple choice quiz which of these will end up showing the world "really" in bold. They can show other words in bold, but I want to know that they'll show "really" in bold.

1.5.1 5.1. Really answer

Let's go through these together. This is well-formed HTML, we're beginning the bold tag. It ends after the word "really." This looks great. Unfortunately, in this next sentence we end the bold before we start it, and then we start it over here with Eric Blair. That's not going to work out well. I will show you in just a minute what that looks like. Here, we begin the bold tag and then we have lots of space and then the word "really." It turns out that is totally fine. Web browsers use the same sort of techniques we talked about in the last unit to break up sentences like these into words based on white space. All this extra space doesn't matter. Finally, down here we start bold at the beginning of the sentence, so all the of these words--George-Orwell-was-really-Eric-Blair"-- they're all bolded. Notably "really" was bolded as well, so this works out. Let's go see how this plays out. Here we have the first option--"George Orwell was really Eric Blair"-- and really is definitely bolded. If I reverse these, it's harder to interpret. This bold tag closes nothing. It's ill-balanced. This makes me super unhappy. But this next one applies to "Eric Blair," and then falls off into the end of the universe. This isn't very good. I can put huge numbers of spaces here, and as we see, this does not influence the rendered web page at all. Then in this version I have the tags at the start and the end of the sentence, and the whole sentence is bolded. George Orwell is perhaps best known for writing 1984, and I hear that HTML has always been at war with JavaScript.

1.6 6. Tags

As we hinted before, this special syntax in HTML is called a tag. It's kind of like a price tag you might attach to a shirt or another item you're buying. This modifies nearby text and tells you how to interpret it, whether you can wash it or not in a machine, how much it costs, whether or not it should be bolded or underlined--that sort of thing. Another super common kind of tag is the anchor tag, which is used to add hyperlinks to webpages. In some sense this is the defining characteristic of what it means to be a webpage. Here I've written a fragment of HTML that includes such an anchor tag. It begins here, but unlike the relatively simple bold and underline tags, it has an argument. This means pretty much the same thing it did when we were talking about functions in Python or math. Here the argument or my sine function is pi. Here the argument or modifier for my anchor tag is href equals. This stands for hypertext reference--the target of this link. Here I've given a string that is a URL, a web address. Hypertext transfer protocol google.com. This text in the middle is often rendered in blue with an underline, although it doesn't have to be. Then over here we're ending the anchoring tag. Let's see how this plays out. Here I've the old "Eric Blair was really George Orwell" text, but I've added a new sentence--"Click here for a link to a webpage." Right after the anchor starts, the text is rendered in a slightly different color. If we were to click on it, you can potentially see down in the lower left that it goes to google.com. Just to break this down, if this is a fragment of HTML, then the words "Click here and now" will all be drawn on the screen. This syntax marks the beginning of the anchor tag. This syntax, left angle bracket slash a right angle bracket, marks the end of the anchor tag. This part in here is the argument of the tag. It contains extra information for things that are more complicated than simple bold or underline.

1.7 7. Interpreting html

Here I've written a significantly more complicated fragment of HTML, and I would like you, gentle student, to help me interpret it. In this multiple-multiple choice quiz, check each box that corresponds to a word that will be displayed on the screen by the web browser.

1.7.1 7.1. Interpreting-html

Let's go through it together. Href is actually one of the arguments to this anchor tag. It's not displayed. If the user clicks on this link, they'll go to Wikipedia.org, but they'll never see the href. This is not shown. Mary, on the other hand, is not the name of a tag or the argument to a tag. It will be shown. Similarly, Vindication will be shown. It'll be shown as part of a link and italicized, but it'll be there. Wikipedia is part of the argument to this anchor tag,so it will not be shown. Wrote, however, will be shown, and then this i, the italic tag, isn't shown. Instead, the next is actually slanted. Here we're taking a look at how it would render in a web browser. We can see Mary Wollstonecraft wrote A Vindication of the Rights of Women. Href, Wikipedia, and i are not printed on the screen.

1.8 8. Taking-html-apart

Now that we understand how HTML works, we want to separate out these tags from the words that will be displaced on the screen. Breaking up words like this is actually a surprisingly common task in real life. For example, ancient Latin was often written or inscribed without spaces. This particular set of letters "SENTATUSPOPULUSQUEROMANUS" is inscribed on the arch of Titus, which I've doodled over here as a column, but what can you do? Arches are apparently beyond my power. I know. It has just become an arch. Those labels never lie. Roman inscriptions like this were written without spaces, and it requires a bit of domain knowledge to know how to break this up. "Senate and the People of Rome." That inscription was made quite some time ago. Similarly, in many written Asian languages, they don't explicitly include spaces or punctuations between the various characters or glyphs. In this particular Japanese example, and both my handwriting and my stroke order are very, very poor--have pity--some amount of domain knowledge is required to break up "ano" from "yama"--"that mountain." Finally,even if you're not familiar with Asian languages or ancient Latin, you might have seen the same sort of thing in a much more modern guise, in text messaging. Some amount of domain knowledge is required to break this up into "I love you" even though no particular spaces are given. We will want to do the same thing for HTML to break it up into words like "Wollstonecraft" and "wrote" that will appear on the screen or this special left angle bracket slash maneuver that tells us that we're starting end tag, this special word in the middle that tells us which tag it was, and then this closing right angle bracket. Once again, for this HTML fragment we want to break it up into this first word, the start of the closing tag, another word, the end of the closing tag, and then another word. We're going to need to do this to write our web browser. In order to interpret HTML and JavaScript, we're going to have to break sentences down into their component words to figure out what's going on. This process is called--dun, dun, dun, dun-- lexical analysis. Lexical here has the same roots and "lexicon" like a dictionary. This means "to break something down into words." You'll be pleased to know that we're going to use regular expressions to solve this problem. Here I've written another one of those decompositions. We might have broken an HTML fragment down into these word-like objects, but this time you're going to help me out by doing the problem in reverse. So in this multiple multiple choice quiz, I'd like you to mark each one of these HTML fragments that would decompose into this sequence of five elements.

1.8.1 8.1. Taking-html-apart

Let's go through it together, and this first one starts with a left angle bracket, which looks super promising, but then it has this slash which we don't see reflected up here, so this one doesn't match, could not have produced this sequence of 5. Over here we have a left angle bracket, a b, a right angle bracket. Looking great. Salvador, looking good. Dali, looking great. Oh, yeah, this totally matches. Down here we have almost the same sentence, but there's no space between salvador and dali. This is very close, but instead of getting 2 separate words at the end, it would break down into just one word at the end, salvadordali, so this one doesn't match. Over here we start with a bold tag, have salvador and then dali, but then we have a few more characters that aren't shown in this list of 5, so this doesn't match exactly. Here we have salvador followed by the bold tag. That's getting the order wrong, and the order of this breakdown is really going to matter. We really need to know the order of words in a sentence. Super important, it is. Finally, over here we start with bold, and we have salvador dali again. This looks great. No problems there. Notice the spacing was a little different. Here we had a space between the bold tag and salvador. Here we had kind of a space over here. These spaces don't matter very much. Salvador Dali was a Spanish artist famous for his surrealist paintings, probably most famous for painting The Persistence of-- I can't remember. Let's just go on.

1.9 9. Html-structure

Since HTML is structured, we're going to want to break it up into words and punctuation and word-like elements, and we use the special term token to mean all of those. In general, a token can refer to a word, a string, numbers, punctuation. It's the smallest unit of the output of a lexical analysis. Remember, that's what we're currently working on. Mostly tokens do not refer to white space, which is just a formal way of referring to the spaces between words. We're going to be focusing on lexical analysis, a process whereby we break down a string, like a sentence or an utterance or a webpage, into a list of tokens. One string might contain many tokens in the same way that one sentence might contain many words. Here I've written 6 HTML tokens, given them names on the left and examples on the right. Now, the naming of tokens is a bit arbitrary. In general, though, tokens are given uppercase names to help us tell them apart from other words or variables. Here this left angle corresponds to an

angle bracket facing left presumably--not quite sure how to draw that. The smaller end is to face. Left angle slash is a < followed by a /, division sign. The right angle bracket, > facing to the right. Here's the angle. Here's the face. The equal sign is just =. A string is going to have double quotes around it, and a word is anything else, welcome to my webpage, punctuation like that. Now, it turns out that the naming of tokens is not quite an arbitrary matter. You may think I'm as mad as a hatter. No, that's a different story. We're just going to go with these token names for now, but if you were designing a system from the ground up, you can rename them to be anything you like.

1.10 10. Specifying-tokens

We're going to use regular expressions, which are very good at specifying sets of strings to specify tokens. Later on we'll want to match a bunch of different tokens from webpages or JavaScript, and this is how we write out token definitions in Python. The t_ tells us--and it tells the Python system-- that we're declaring a token. The next letters are the name of the token. You either get to make this up yourself, or in the homework I'll tell you what I want it to be. Tokens are in some sense going to be functions of the text they match. More on this a bit later. Skip me for now. Next, we have a regular expression corresponding to this token, which in this case, for the right angle token, there's really only 1 string it can correspond to, so we've written out the regular expression that corresponds to a single string. And then here on the last line we're returning the text of the token unchanged. We could transform it, and you'll see us do that for more complicated tokens like numbers where maybe we'll want to change the string 1.2 into the number 1.2. Now it's your chance to define your first token. What I would like you to do is write code in the style of the procedure I just showed you before for the LANGLESLASH token. The LANGLESLASH token is surprisingly important. We really need it to know when all of our tags end. Use the interpreter to define a procedure t_LANGLESLASH that matches it.

1.10.1 10.1. Specifying-tokens

Let's go through a possible answer together. I have to name my procedure with t_. That tells the interpreter that I'm defining a token. Now I give the name of the token, LANGLESLASH. That was given as part of the problem. All of our tokens are actually functions. We've been eliding that bit. We're still going to skip over it. Next I have to have a regular expression for the string that matches the token, and here again there's only 1 string that matches, and I'm going to return the token unchanged.

1.11 11. Token-values

It's not enough to know that a string contains a token, just like it's not enough to know that a sentence contains a verb. We need to know which one it is, and formally we refer to that as the value of the token. By default, the value of a token is the value of the string it matched. We can rebuild it, however. We have the technology. Let's see how that plays out. Here I've written a definition for a slightly more complicated token, a number, one or more copies of the digit 0-9, and now I would like you, our last, best hope for victory, to help me understand it. If the input text is 1368, what will the value of the token be? Check all that apply.

1.11.1 11.1. Token-values

All right, let's go through the possible answers together. We're definitely going to match 1368 because it's in the language of this regular expression. It's 4 copies of 0-9 together. By default, the value of the token, that is, when it comes into us, it's just the string 1368. But we're going to convert it to an integer using this cast or conversion here, so at the end of the day, it's going to be 1368. It's not going to be the string 1368 because we have this special conversion. It's not going to be the string 1 because the maximal munch rule means that we're not going to match just one. We match all 4 digits. Similarly, it's not going to be the integer 1 because we match 1368. It's also not going to be the empty string. We definitely match 1368, and 1368 is a good number to match. That's the approximate year when the Ming Dynasty started. They featured a strong central government in China, but perhaps they're best known because most of the Great Wall that we can see was either built or repaired during the Ming Dynasty. This is my Great Wall sketch. You can tell it's the Great Wall because it's got a label that says "Great Wall."

1.12 12. Whitespace

So we're using these token definitions in regular expressions to break down HTML and JavaScript into important words. As we've seen before, there can be lots of extra space between various tokens. We really want to skip or pass over spaces and possibly newline characters and tabs. More on that later. We do that using the same sort of token definition as before, so here I've made a regular expression that just matches a single space, but instead of returning the token, we pass it by. This is the power. Let's test out our knowledge with a quiz. We've already seen how to do left and right angle bracket sorts of tokens. We've taken a look at strings before. And now let's do words, which are almost everything else on a web page. Let's say that we want a word to be any number of characters except a left angle bracket, a right angle bracket, or a space, and here I really mean the single character

pressing the space bar, not this 5-character word, but it's hard to write out. And when you're writing your function to match word tokens, you should leave the value unchanged. Submit a definition for t_word using the interpreter.

1.12.1 12.1. Whitespace solutio

So let's go through a possible answer together. Once again, the real trick is just coming up with a good regular expression, so we can't have a space, a left angle, or a right angle, but we can have 1 or more copies of anything else, and we'll just return the token unchanged because that's what we are asked to do.

1.12.2 12.2. Quoted-strings

When reasoning about HTML, it's critical that we understand quoted strings. They come up in almost every anchor tag, and anchor tags are the essence of hypertext. They're the interlinks between documents, so we really need those. They rely on quoted strings. That means we're going to need to understand quoted strings. It is convenient then that we are amazing at quoted strings. You have plenty of practice with them from the previous unit. Once again, it's time for a quiz. Suppose that the strings we care about start with a double quote, end with a double quote, and in the middle they can contain any number of characters except double quotes. I'd like you to write a definition for the Python function t_string and submit it via the interpreter. And just to make this a bit clearer, we should definitely accept quoted strings like cuneiform or sumerian writing. But we should not accept 30th century BCE because it doesn't start and end with quotes, and we should not accept this string which has an escaped double quote. We may want to get to that later, but for now, don't worry about escaped double quotes. Go to it. Submit via the interpreter.

1.12.3 12.3. Quoted-strings solution

Let's go through the answer together. We definitely want to name our procedure t_string, and now the real trick to this is just writing a good regular expression. Here I've written a regular expression that starts with a double quote, ends with a double quote, and can have anything that's not a double quote-- remember that super useful carat character--0 or more times.

1.13 13. Lexical-analyzer

You've now seen a bunch of these token definitions, one for words, one for strings. A lexical analyzer or lexer--this is a term of art-- is just a collection of such tokens. You tell me what makes a word, what makes a string, what makes a number, what makes white space. You put it all together, and the result is a lexer, something that splits a string into exactly the token definitions you've given it. For example, once I put these 3 rules together, they become a lexer, or lexical analyzer. And in fact, suppose we passed to this lexical analyzer the input string 33 is less than 55. Oh, gentle student, tell to me which one of these token output sequences could result. In this multiple choice quiz, indicate which of these 3 possible output lists could correspond to the values of the tokens extracted from this input string using these rules. Let's put it all together.

1.13.1 13.1. Lexical-analyzer solution

Let's take a look at the answers. This 33 is definitely going to match, we hope-- we're going to get to this in a minute--our numbering rule. And then we're going to end up converting it into an integer. Is will match the word is. Less will match the word less. Than will match the word than. And all of this white space in the middle will be dropped. This is a possible answer. Here we have pretty much the same thing except that we appear to be returning a space as a word. That's not going to happen because our words can't include spaces, and this white space rule would skip over it beforehand. That's not it. Here looks a lot like the beginning, but instead of matching 33 and 55 as numbers, we appear to have matched them as words because we've returned this string unchanged. This isn't really what we wanted, but actually, nothing prevents that from happening. or our number definition, and I haven't told you how to resolve this sort of ambiguity. Oh, that word is back from the dead. This is also a possible answer.

1.14 14. Ambiguity

Now let's take a look at the same concept but do it backwards. Let's assume that we've fixed the problem from the last quiz. We're going to assume that number is preferred to word. Whenever we could match something as a number, we'll match it as a number instead of a word. What I'm going to do is write out a few phrases, a few sentences, and I want you to notice which of them could produce, could be decomposed into a word followed by a word followed by a number. Multiple, multiple choice. Check all that apply. Which of these 4 inputs could break down into word, word, number using the rules that we've been going over so far?

1.15 15. Crafting-input

All right. Let's go take a look at the big reveal, for how it actually started. Up here is our string; we start at characters zero, on line 1, with ' This '. and we're starting the Left Angle, then the ' b ', then the Right Angle, then the ' webpage '. And to reverse engineer this, you might note--for example-- that you know the Left Angle is 1 character. If it starts on character 11, and the ' b ' comes right after it on character 12, there must be no spaces between them. Similarly, since the ' b ' is 1 character and it starts on character 12, the RANGLE, there must be no spaces between the ' b ' and the RANGLE. So the real trick here is figuring out what happens after the "is" and before the ' b ', and knowing that you need these 3 extra spaces ao all the action's happening here.

1.16 16. Commented-html

Just as we have to separate words in HTML and JavaScript based on white space, we also have to take into account comments, and comments in HTML serve the same purpose that they do in Python, documentation or removing functionality. You can add a comment containing English text to explain what a program should be doing. You can do the same with a webpage or with a JavaScript program. Or you could comment out a function or a line to see how things behave without it. In HTML, comments start with this interesting 4-character sequence and end with this 3-character sequence. Left angle, bang, hyphen, hyphen begins a comment. Bang, bang, right angle ends a comment. They look a bit like tags. Here I have an HTML fragment, "I think therefore," and then there's a comment, "I am." Je pense, donc je suis. We're going to see how to implement this in our lexical analyzer, but recall that our lexical analyzer was just based on regular expressions. I could recognize these comments with another finite state machine, and all I have to do is just merge these 2 finite state machines together conceptually. If I could have one set of rules describing comments and another set of rules describing all of my other tokens, I'll just put them together into one big machine. It might have too many states for us to be comfortable with, but it is entirely fine for a computer. When we're processing a comment, normal rules don't apply. Let's consider a super tricky comment example. Here we have "Welcome to <b> my," comment that closes the bold tag, "webpage," close the bold tag again. My question for you is how will this render? Which of these words will be bolded? Well, it turns out that when something is in a comment, we ignore it entirely. It's as if it weren't there, so even though this looks like it's closing the bold tag, it does not, and the words "my" and "webpage" will both be bolded. In fact, it's almost as if everything in the comments were entirely erased and had no impact on the final rendering of the webpage at all. And now without the distracting text, it's relatively clear that "my" and "webpage" should both be bolded. Here I've written another HTML fragment that includes a comment, and the quiz for you is--in multiple, multiple choice fashion-- to tell me which of the following HTML tokens--and I'll draw them now-- could be found by our lexer. Check all that apply. Based on this string, assuming that we've added the right rules for comments to our lexer, which of the following would be found?

1.16.1 16.1. Commented-html answer

Well, let's go over it together. It looks temptingly like there might be a few left angle brackets, like this one or that one, but if you look carefully, you'll see that those are both part of the comment, so it's as if they were never there, so we don't see any left angles. We do see a left angle slash over here at the end of the string. Any right angles? Yes, right here at the absolute tail end of the string, but note that this one in here did not count. Any strings? "World" looks super tempting, but again, it's inside the comment, so it does not count. Word--and I've intentionally chosen these so that they conflict a little-- word, both "hello" and "confusing" totally apply.

1.17 17. Html-comments

So now I'm going to show you how to add HTML style comments to our lexer so that it gets the behavior that we want. Just as we had to list all of the tokens, we're going to have to list all of these possibilities for things we could be doing. Either we're breaking the input down into tokens, or we're handling comments. These 2 possible choices are called lexer states, a word I really don't like because it's super ambiguous. We have states in finite state machines. Let's not have states elsewhere. This particular syntax for specifying them is arbitrary. It's just a function of the library we're using, so I'm going to declare a new state called "htmlcomment," and that's exclusive. If I'm in the middle of processing an HTML comment, I can't be doing anything else. I'm not going to be finding strings or words. Here's my rule for entering this special mode for HTML comments. If I see the beginning marker, <!--, then I want to enter this special HTML comment state and not find words or strings or numbers or tags but instead ignore everything. When I reach the end marker for an HTML comment, I want to jump back to the initial mode, whatever I was doing before, that time when I could find l angle slash and l angle and r angle. By default, that mode is called "INITIAL," all in capital letters. I'm actually going to do one other thing all at the same time. When we're in our special HTML comment mode, we won't use any of our other rules, even our super happy newline rule that's counting the line number, so what I'm going to do is call this string.count function to count the number of newline characters that occur in the token in the entire comment and add those to the current line number. This is a little tricky, so don't worry if this doesn't make sense the first time. Now, we've said what to do when an HTML comment begins, and we've said what to do when it ends, but any other character we see in this special HTML comment mode isn't going to match one of those two rules. Anything that doesn't match one of our rules counts as an error, and what I'm going to say is just skip over that single character in the input. It's part

of the comment. I don't even want to see it. This is a lot like writing "pass" except that it gathers up all of the text into one big value so that I can count the newlines later. Again, this is a pretty tricky maneuver. Now I've changed my webpage so that it says "hello." There's an HTML comment with the word comment in it, and then there's the word all, and if I've written all of this code correctly the first time, which really that does not happen in Lake Wobegon, then we'll get a chance to see how this plays out. There was an error on line 22, this line number increment, and it's yelling at me. Let's go back up to line 22. I had split these onto 2 lines so that it would be easier for you to see. But let's not do that because Python hates it. And now with that one simple syntax error fixed, our webpage produces exactly the 2 tokens we expect, hello, starting at position 0, and all, Excellent.

1.18 18. Token-counting

So given that we've just seen how to code up HTML comments, let's test out that knowledge. Suppose we have our definition of HTML comments plus a rule for word tokens that are anything that's not a space, a left angle, a right angle, one or more of the above. We return that token, and now I give you the following HTML input fragment, "ob fuscation tuse tangle." For this blue string, for our lexer, we've got word tokens. We've got HTML comments. How many word tokens are we going to find?

1.18.1 18.1. Token-counting answer

It turns out that the answer that we're looking for is 3. Ob is one and tuse is another, and it's really tempting to put obtuse together because they're separated by a comment. You think "Boy, if I just remove that comment, they'd be one word, obtuse." But remember, you really want to treat comments as if you were totally erasing them so it leaves a gap. The lexer is entering the HTML comment. Mode, it's coming back. Ob and tuse are really quite separate, and then we've got tangle as the third at the end.

1.19 19. Five-factorial

All right, so now that we've mastered lexing or breaking up into words HTML, let's turn our attention to JavaScript, the other language we'll be considering in this class. And I'm going to introduce you to JavaScript by jumping right into an example by way of a parable. Over here on the left we see some webpage code, apparently a webpage owned by Steven. Welcome to Steven's webpage. And Steven would really like to compute five factorial, 5 times 4 times 3 times 2 times 1. Over here on the left, we see the HTML source code, and on the right we see the result as it would render. This p tag I haven't introduced you to yet, but it means begin a new paragraph. We'll write out the words "Welcome to." We'll show "Steven" in bold. We've got this five factorial, this bang. The exclamation mark often means factorial in mathematics. Printed in italics you can see it slanted. But unfortunately, Steven is super sad. He can't remember the value of five factorial. Well, this is exactly the sort of thing that a programming language like JavaScript could help us out with. It can carry out computations just like Python, so we can do work in the middle of a webpage. Let's write our first JavaScript script together. Here, this line starting with script type= "text/javascript" and then this document write line, and then ending here with this closing script tag, all of this, these 3 lines together are a JavaScript program embedded inside an HTML webpage. JavaScript programs always begin with this special script tag, and this tag has an argument because there might be multiple types of tags out there in the universe. We've seen tag arguments before with the anchor tag. Here I have an anchor tag where the argument is a hypertext reference. Here I have a script where we're telling the web browser you should treat this as a JavaScript program. This JavaScript program is very simple. It's the equivalent of print "Hello World" in Python. JavaScript's name for the print function is document.write, which we'll sometimes just abbreviate as write. But the semantics, the meaning is largely the same. It's also worth noting that we've put parentheses around the argument to document.write almost as if it were a mathematical function. We can do that in Python. It's allowed, but often we don't. And we've ended the line with a semi-colon, whereas at the end of a Python line we often don't have a semi-colon, but again, you can put semi-colons at the end of Python lines. We just typically don't. Now we're going to try to use the full phenomenal cosmic power of JavaScript to compute five factorial. To do so, I'm going to make a recursive function called--surprise, surprise-- factorial that's going to compute the value. Let's walk through every part of this JavaScript code together. The word function means I'm declaring a function. This is very similar to def in Python. Then I give the name of the function, and then I write the arguments just like I would in Python. In Python I'd have a colon here, but JavaScript requires slightly different punctuation, this open and curly brace, and in this regard it's more like languages such as C or C++ or Java or C#, curly brace languages. Our factorial function is going to be recursive, and every recursive function needs a base case, a time when it stops. Our stopping condition is when n is 0. We could have picked n is less than or equal to 0, n is less than or equal to 1, so I have an if statement that's checking that. Again, in Python this would probably look very, very similar except that we'd use a colon instead of an open and curly brace. If n is 0, we return 1, and I have a semi-colon at the end of all my statements. Then in Python, I would know that I'm done with the if statement because of the tabbing. JavaScript doesn't use that sort of readable tabbing rule to figure out the control flow when an if statement ends, so instead you have to explicitly close off this open and curly brace just like you'd have to close off a tag in HTML or close off parentheses once you start them. We're going to study this a lot more as time goes by. I close off the then branch of my if. I have a semi-colon, and now I have a new return statement, and this is basically just the formula for factorial. It's n times the factorial of n - 1. This part here is a function call, in fact, a recursive function call, just like you'd expect to see in Python. I'm

ending the whole thing with a semi-colon. This is the end of my function definition, and at the end of the day I print out-- and the JavaScript version of print is document.write-- factorial of 5, and over here on the right you can actually see it. We've got the 120, which is the correct value for factorial, so what that tells me is that if prettiness matters, I should delete those question marks because now we are super happy because we can compute five factorial using embedded JavaScript in the middle of a webpage, or to put it another way, a way that's a bit more puntastic, [singing] it's a good thing JavaScript can run on the page of Steven. And basically my voice is telling me not to quit my day job.

1.20 20. Honor-and-honour

Many of the differences between JavaScript and Python are similar to the differences between American English and British English. Sometimes the spelling changes a bit, or the pronunciation changes a bit, but in general, they are mutually intelligible. It should look very similar, but with 1 or 2 extra letters. JavaScript is big on curly braces. Python is big on tabbing. JavaScript is big on semi-colons. Python not so much. JavaScript really loves parentheses. Python could take them or leave them. Python could have these, could have these, but doesn't have to. They're somewhat optional. And in some sense, preferring one over the other is a bit like preferring Brutus to Caesar. I've heard that Brutus is an honorable man, or at least so Mark Antony told me. They're both entirely reasonable from our perspective. Is this a dagger I see before me? I guess this one must be Brutus, and this one must be Caesar, and you know which one is which because of these convenient labels. Here I've written a new JavaScript function called "ironcrutchli." This one is not recursive, but it takes a formal argument x and then does some reasoning based on x and returns values as a result. I want to make sure that you're following along with JavaScript, so I'm going to have you tell me what this function evaluates to based on a few different inputs. In this fill-in-the-blank quiz, for each one of these inputs, imagine we're calling ironcrutchli on 0, so this 0 is bound to the value of x. What would the output be, similarly for 1, 2, and 9?

1.20.1 20.1. Honor-and-honour solution

Let's go through the answers together. If we call ironcrutchli with x as 0, then 0 is less than 2, so we're going to return 0. If we call ironcrutchli when x is 1, This function not so exciting thus far. But now we pass in 2 for x, and 2 is not less than 2. It is equal to 2, however, so we're going to return 2 + 1, or 3. And then over here if we pass in 9, 9 is not less than 2, so we're going to return 10. Iron Crutch Li was one of the 8 immortals of traditional Chinese mythology. Not a particularly nice guy, but he did help out the sick.

1.21 21. Identifier

In the previous example, I had highlighted ironcrutchli and x in red. They were identifiers, which is a formal way of saying variable name or function name. Identifiers are textual string descriptions that refer to program elements. Things like factorial, s and tmp are all identifiers. We often call them identifiers because they identify a particular value or storage location. We're going to allow our identifiers to start with lowercase letters or also uppercase letters, and they can finish out with any number of upper or lowercase letters. Here we have another uppercase letter as time goes by, and sometimes people like to use underscores to add readability. That's totally allowed in the identifier, but we'll say not at the beginning. You have to start with a letter, but after that you can have underscores. Here I've shown 6 examples of identifiers, and I would like you to use the interpreter to write an identifier token rule for these sorts of JavaScript identifiers. Try it out using the interpreter.