With the instructions. And this is probably the most important part of the document. This is what I want you to do now. Of course we are. Remember we are implementing a recursive descent parser. So in general, with recursive sent parsers, there's going to be you need to define one subprogram. For each grammar rule, that's not always exactly true, and will cover some cases here where that's not true. But when I want us to do, I want us to write a parser that's just able to get syntax errors and print up our strays. We're not going to generate a tree this week, so. Here's some examples of what I want you to do every time you enter a rule or leave a rule. We looked at an example of recursion sent parts. If I call rule a, I want you to print entering a row whatever rule is. If I enter rule be entering B rule, etc. And every time you leave the rule, print out your exit. There are two exceptions. I know we said that typically we're going to ride. One method. For each rule in the grammar, that method is responsible for implementing that rule, however. Two of the rules in our grammar. Also happened to be tokens, like. They have their own token name, token type. So for these we don't really need to go into a rule to try to figure out what these are. We can literally just look at the token and see what it is. So for these, we don't really need rules for that. That's going to be ID and integer. So for ID an integer, you don't need rules for that. Just look and see what it is. If it's an idea, here's what I'm going to do. If it's an integer, here's what I'm going to do. So these are the two exceptions to reentering the exiting rule. I have. Next thing I'd like you to do, every time you find a code, every time you identify Lexie, I want you to print what you saw, what you see. I found an ID token. Here it is. You know Dom. For rules that have epsilon definitions. You also need to tell me if that's the definition you chose. So some of our rules, remember they have some rules that have a definition that just needs nothing. Then how do we indicate that? Well, we tried to identify the something from the first part of the definition. If that's not it, we try to identify something for the second definition. If we can't do that, we just leave. Well, if you just leave, I want you to tell me I'm choosing epsilon. I'm choosing nothing. I think should specify. Spell it out that did not find. You know, I was supposed to see a plush, doping or minus token. I'm choosing the epsilon production. Also every time. You detect a syntax error. I want you to tell me, hey, I was supposed to see for example, ID. I found a question mark. I don't know what it is. And if it's possible, the error could have been multiple things, you know, tell me all of the things you were supposed to have seen. It could have been one of these things. I don't know which one, but here's what I saw. So, for example, if you think about the the recursive descent parser that the book, the code that is in the book that we went over in class, remember the very last item? The that rule had three definitions, and each of those had something concrete. We could have looked for parenthese and ID, or a or a number, or something like that. So if you enter a rule like that, you don't have a choice. You have to identify one of those definitions. So if you don't, that's in terror. So if you have something like that, if you're faced with something like that, then you simply tell me all the things that should have been. So I expected an ID or an integer or a (but I found, you know? To show me the left sink that you found. It's not one of those. It might not even have a categorize. Just print Alexi. In the finally, when you're done, I want you to tell me how many parse errors you found. Found ten hairs, found five errors, found zero errors, and it should look exactly like this now when I am grading. Your program. What I'm going to do, I'm going to run your code. And I'm going to capture your health, but these are all print statements. You're going to be printing all this stuff. I'm going to capture all your output, and I'm going to save it into text file. And then I'm going to run my program. On exactly the same input file and I'm going to capture all my output and save text file and then I'm gonna do a dip. On those two text files, and they're not identical, you will fail that test. So we'll talk about some, look, I'll give you an example later or some things you can do to. Spot. Text based differences and text, because sometimes they'll actually be So we'll go over some techniques. 2. Help identify that easily later. But that's pretty important. It does have to match exactly what my output is, and to help you with that. I tell you what the output is, so when you run the code I will tell you hear exactly what you should have output. Here's exactly what you output, and then you can just compare the two things and see what was different, what was wrong. So what am I getting this time? I give you a couple of files. I gave you a parser class that extends your lexer. So we can just call all the vendors that are in it. We don't have to redefine them, we don't have to do anything fancy with it. We don't even forwarded. I've defined and just like last time. I have. I've given you most of the code, you have like 75% of the code and you just have to finish the last little piece. The only class. You need to modify. Yeah, hey, it's the partial class you don't have to touch. Who should not touch any of the other classes? Let me say that. Let me not say you don't have to touch it. You should definitely not touch any of the other classes, because. When I raise your cold. The only the only file I'm going to use as your parser file. From all the rest of the files, I'm going to use my file. So I'm only going to do partial so you make any changes to any other files it's not going to carry over to the test. Please only modify the powers that don't modify anything else. I find a couple of methods for you. There's consume method. That's very similar to the next character method from the previous assignments. So remember the previous assignment? That was the next character method that was scanned the file. Take one character and put it in your character buffer. Well, now we're writing a parser. We're dealing with tokens. So instead of a character buffer, we have a token buffer now. And what I'm going to do, instead of scanning the character, I'm going to call your leg, Sir, and I'm gonna say give me the next token, and I'm going to put that token in our token buffer. So Sue method calls next token from the lexer and puts it inside. The variable name is at look ahead, so that's our. Token buffers and look at that bumper. Further model parts. And we just throw away whitespace tokens. Every time I see something, it's a white face I just throw away and stand next to keep scanning until I see something that's not align space. I also gave you a match method. That tries to help you detect errors. So. If you if you choose to use the method. The way it works, you pass it a token name ID int. (you pass it on the name. And it will compare that token name you gave it with the token that's inside the token buffer, and if they match, great. If they don't match, it'll print an error message for you. And then it'll come soon to put the next token in the buffer. The program method. I've written a couple of the methods for you, so I went ahead and wrote program and statement for you. So those two rules, I wrote those two methods so you can kind of use them as an example to help you write the rest of it. So again, just like last time, I've given you examples of everything that you need to do, every type of thing you need to do, you you should be able to look and see an example from what I've done already. And from member if we look at the grammar in a minute, but if you look at our grammar, the very first rule is programmed. And what is a program? It's just one or more statements. So let's program. Do they just call statement over and over again until there are no statements left? Which that's what a program is that. So that's all it does. Just call state and repeatedly until there are no more state. Mr process. And then we'll take a look at statement in a moment. We'll actually open the code, the file and then take a look at it. And you just have to write the rest of the rules. That's all you have to do. Right methods or all of the rest of the rules. I've given you a working. A lexer and working token class, so all the other classes work. Again, we only want to modify the partial. That's the only file you really want to modify. The other thing I've done, again because of the nature of how I'm testing this. I've actually given you exactly the same input files I'm going to use to test it. So the input file I give you 5 input files. And when I test your code, I'm going to use those five input files. So if you're able to write your code and you get it to work successfully on those files, you should be able to submit it and get a hunting. It is actually the same files I'm going to use. Um. One through three, the first three text files should complete with zero errors. So if you're partial detects syntax errors then your codes wrong. They should have no errors in the first three finals. The last two files will have parse errors, so you should definitely some parts of errors there. And here are examples of what the output should look like. It should look something like this. Right, so here is well, should look exactly like this. Here's input three. Here's the output for the third text file. Here's the output frame before you see here at four parts errors. Here's the output for five three parts errors. So it should look exactly like this. So you have all this kind of help, you create the right thing, get your program to work correctly. According to what I'm asking you to do, that would correct means in this case what I'm asking to do. Now, with with that being said, before we take a look at the code, do you all have any questions for me from a high level, as it kind of do, it doesn't make sense what we're supposed to do with this assignment. Yes, OK. Alright then. Let's take a look at the code.

So. I guess let's start with Alexa. So here's Alexa. I gave you working lexer. This is similar to. This is the the lecture you had to write for the last assignment. You can see how I am going to there's no one exactly the right way to do it. You all could have done it differently. You can see what I did. I had a case statement for all of the one character. Typed opens and for all the multi character lexemes I just used exact. I used a while. So I did all of whom the same way while living in a statement, and I just modeled after whitespace. The same thing after whitespace I did for every single Multicare control time. Appreciate the change. Anything else I guess right here so we don't crash, we just check to see if it's open. If that file names there, great, if not. I just read a message and close program token. Your token may look different. You can see here that I've defined some token names. Now, unlike the first assignment. Where you have the freedom to choose your own token names. You don't have a choice in this one. I'm giving you the code in class and you have to use these names. You don't have a choice. These are the names you have to use because these are ones I've defined and you have to use my. My lecture that. Aside from that. It's identical to the other. I just have some token names identified here, and these are names that I chose to use. You may have chosen different things, it's fine, but these are the names that I chose to use to identify the different lessons in the grammar. Let's go to the parser. So here is what you have to finish. You can see here. Remember we are extending scanner, so here's our my lecture. That's how end Ruby. That's how you indicate that you're extending the class. I'm extending it. I have to start by calling the constructor. For the scanner, I'm gonna pass in the file name so that it can. Create the first open right and I'll call him. Soon I'll consume to get the first open and put it in my token buffer. If you look at consume, what do we do? There's that there. There's an aerial that I'm going to use. At what am I going to do? I'm going to call next token and whatever token it returns, I'm gonna put it inside my look ahead buffer. Then I'm going to examine that token, the type of token it is, and if it's a white space token, I'm going to throw it away. How do I throw it away? I throw it away by calling next token again. Keep doing that until it's not a white space open anymore. Here's the match method I was telling you about. The match method you can see what we do if you give me a token type ID INT. (you know, plus whatever the token names are. I will look. I did look ahead, buffer. Now compare that type to what you passed in, and if they're different, you know, offering a message for you. Hey, I expected this, but actually, here's what I found. And what should we do? Here's the token type I expect you to give me here at Alexian that I saw. And then I'll call him soon, so either way. If there's an error or not, I'll consume them. Put the next token at above for you. If you use that method, you don't have to use it, you can modify it. If you want it, you can change. You can change whatever you want to change in here as long as it works correct. What the program so if we look at our grammar. Program is just one or more. A program is just one or more statements. So if you look at our program method, what do I do? Wild. Whatever television, the buffer is not intended. I'm going to call state over and over again. That's all I do. Now, remember I said I want you to print something every time you're entering a rule or exiting a rule? I chose to print it here. I know about to call it. Anyway, you don't have to print it here. You can print it as soon as you enter the rule. If you want to, you can print it in here. It'll be like one of the first thing you do. It doesn't really matter. Ultimately, this is going to get printed and then I'm going to get into that room. So it's kind of the same thing as far as what you see on paper. So entering the rule, then I'm gonna call statement and that's all we do. As long as we're not at the end of file, I'm gonna keep calling statement over and over again and processing statement. Well, if we take a look at this rule. We can see. That statement. Has two definitions. Right, it's either. Whatever that is. I don't know what it is. I don't care what it is. Assignment knows what it is. Or. It's literally the word print. Followed by whatever that is. Again, I don't know what it is. I don't care what it is. So how do we write that as code well? The first thing we probably want to do, we could just call assignment first if we wanted to. But since we actually have something concrete we could look for, why don't we do that first? We could call assignment and wait for it to go all the way down and come back. Or I literally have Lexie here that I could look for. So I'm just gonna look for the word print and if I see the word print. If I see a print token, what do I know? I know I must be looking at the second definition. So that's what we say here. GIF. The token time is print. Oh, I must be looking at the net direct at the print the second definition. I'm gonna print what I found. Hey, I found the print token. Here's the left scene. I'll call the match here. I use Max which calls from soon for me. Then I'm going to call expression 'cause. That's what print. That's what the second definition says. It says it's the word print. Word print. Followed by whatever expression. So that's what I do if I see the print token. Great. I'll call Max or column two and then I'll just call the expression method and I'm done with this. We're finished. As far as if it's not, if I don't see the word print, then what must it be? Even if it's not, it has to be an assignment. Say that it can't be anything else. If I don't see print, it has to be a design statement. So I'm just gonna call the assignment method because I don't have to know what that is. Assignment has to know what it is. And then you have finished the recipes and of course we're done. You know, I'll print. I'm exiting the statement rule. So. Just quickly. I've done these two. You don't have to do these. Are the wisest place we're going to throw away. You don't really need to write a rule for that if you're gonna throw it away. Riley and in what we have tokens for that so I don't need to call the rule if I can literally just look at the token name and then for alpha and digit you know what is an ID. Well an idea is one of these things. I'm really using this to help us drive way ideas and I'm using this to help us drive end is. I don't need to know those things, my lectures able to identify that and I can just look at opening. Because the only thing you all have to do. You need to define. Methods for these right here. OK. Any questions so far now that we look at the code? Does that make sense? Alright. Remember? If you want to work on it, invoke the area. We do have a shell. This is a Linux shell, so if you want to all this stuff I demonstrated in class, for example. Specifically, I'm referring to the interactive Ruby header. You can do it on here. Hello. I can load my files, load. Parser. Great, now I can run individual methods if I want to write X equals. Archer. Marsar. New file name will do input. Text or type. Great, I want to call one method at a time X dot like maybe I want to test date. I remember we want to test it one at a time. They want testing. Statement. Of course I have a problem. It's not. You can test, you can advance tokens. It is a. Remember we did extend. Um. We did extend the lexer so you can call the methods in the lecture. I think I needed to assume first. That's what I miss. Then say X com statement. OK, 'cause, I haven't actually defined that method that right so? So you see, you can you can test it from with it here or you can do it locally so if you have. If you decide where on it locally, you have the same thing. But I just wanted to let you know you got the same tools available online. You don't have it on your local computer. That's fine, you can do it right there. My terminal doesn't work. Usually there's a button down there that says press enter or something, and if you press it, it launches your terminal. It doesn't always launch immediately. So if I let's go, I'm gonna switch views, real teacher views back to student view. This is exactly what you should see. You should see exactly the same thing I do. So studentvue. And this is what you should see. Oh well, my initialize right away. Usually there's a button down here that says launch terminal, and you click that and that starts the terminal up for you. If that doesn't work, I would suggest trying a different browser. You might have some browser issues where it's not letting that dislike and I frame, so maybe it's not letting the content come through that I frame. So maybe try a different browser if you're not comfortable modifying the security settings that browser that you're using. And see if that helps Taylor. Alright. Any questions about this? One other thing I want to show you, so let's go. We're going to talk about troubleshooting now. When there's walls and walls of text, so I'm going to click submit. I didn't change anything, so obviously I'm going to fail. It's gonna be wrong. But I just want you to see what it's gonna look like. So I'll run one test case for each file. And of course you can scroll through here. Now the terminal does have a limited amount of characters it's allowed to display, so it's possible a little clip the output. So if you want to see the full output you want to click view grading report, so I'll click this. And this is the full output. And you can see test one fail. Here's the correct output. Here's exactly what your output needs look like. For now I'm just going to copy this. You see, that's quite hard. Copy this. Paste it. I'll put it in Page 1. So where's my apple? Right? And then we'll go back and here's what you output that course. This is obviously. This is gonna be pretty obvious here. I only have two lines. I didn't do anything, but you would have more. Do the same thing with the, copy it and paste it. Into another file. Right. And then what do we see? Well. Let's do something like. Something like this? So very soft, right? So if you look at the two files. If you look the two files side-by-side. They're going to look identical now. You saw a lot changed. You saw what I changed so right here. ID from oops. Capital letters. If you're looking at these two side by side, it's very easy to overlook something like that. I promise you, people have done it. It's very easy to overlook that especially. If you try to do it in here. It's not an easy thing to spot. So what can we do to try Spotify? Glad you asked. We can use a special tool called DIFF. So one thing I could do. I could do something like this. As you can see here. It tells me it shows me this is the. Line number where its box the difference one file two

Number file one file 2 and you can see right here difference. There are other flags you can pass it to make this a little bit easier to read. If you're in windows. You can use something called notepad++. If you're on Mac you can use Microsoft Word or there are other dev tools I wasn't able to find really nice easy use, but you could use Microsoft Word that works too. I had. If this were a Windows machine, I would show you, but it's not. I'm going to have to point you to my video now. Going to manako 'cause I need to log in. That'll put my phenomenal credentials in my cash. And then if you go to the announcements. You can see there's announcement that says how to use diff. Click this and there's a short. There's a. 5 minute videos not even five or six minutes very short.