All right, everyone, welcome back to the CIS, 120 Deep Dove videos.
0:09
In this video, we're going to cover a couple of things related to defining data types and working with pattern matching in camel.
0:14
And as we'll be seeing in the videos that go along with this deep dove, a lot of what we're doing is working with three structured data.
0:23
So I thought we'd pick an example that should be fairly familiar from you, with you.
0:33
So basically what we're going to see along the way is features for naming, naming existing data types,
0:37
defining new tree shaped data types and then working with some functions to let us do useful things with that.
0:47
And this will also give us an opportunity to do more practice programing with lists and recursion.
0:55
And I'll also show you a little bit about how to use Euterpe, which is Okemos interactive interface.
1:01
OK. So let's set up the problem.
1:09
So I thought I would motivate the the problem by a problem you've encountered probably daily, which is a file system, data structure.
1:12
So if you're using the finder and on Mac or browsing through the files on Windows or Linux,
1:23
you're used to having nested folders that you can put files into.
1:32
And I thought I would use that as an example to work with some tree structured data, since it should be pretty familiar with you.
1:36
So let's think about, you know, as usual, following our design process.
1:46
What's the first step is sort of understanding the problem? Well, there are lots of different things we could do with file systems.
1:52
But for the purposes of this example,
1:58
I thought it would be useful to write a utility that searches a file system and gives you all of the paths that lead to a particular file.
2:02
And so that raises a lot of questions, including how are we going to represent files and the file system and so on.
2:12
And as we'll see, a natural way to do that is with camels, data types and writings and programs that work with trees.
2:19
So let's just begin. So first of all, let's just think about what what is a reasonable way of modeling a file system?
2:28
Well, as we saw in camel, user defined data types have a couple of different pieces.
2:42
We typically have a different variance.
2:53

So if we just think naively about at a simple level, the kinds of things that you find in a file system are files and folders.
2:56
And of course, there are lots of different types of files and there's lots of different metadata and things associated with the folders.
3:06
But for the sake of simple example, we can start off by saying, well, we have a file or we all set up.
3:12
We have a folder that has itself a collection of data inside that.
3:20
And in Carmel, we define data types like so.
3:28
So this would be a a data type that comes in two flavors.
3:36
So either a file by itself is considered to be a file system or a folder consisting of a list of file systems is itself a file system.
3:43
Now, we haven't yet added any interesting data to this, so maybe files should have some useful information,
3:56
like a file name and maybe some data that's associated with it.
4:02
And maybe a folder itself should also have a file name, the name of the folder and the error that Campbell is giving us.
4:08
Here is what we haven't said. What is a file name?
4:18
And for the purposes of this this demo, we can just say that, well, we're just going to use a string to represent a file name.
4:21
And similarly, I haven't said what is the data that we're going to store here.
4:31
And so we could also use it again for simplicity.
4:36
We could use strings. Now, a couple of remarks.
4:40
So here was we're seeing that the usual way we define the user defined data types specifies different variance.
4:46
So here file is a variant and folder.
4:54
It's a variant each variant has with comes with a tag or sometimes I'll call these the constructors of that variant.
4:58
In this case, capital file. Or capital folder are two ways of constructing a file system value.
5:06
And each of those can can can sort of carry along some data with it.
5:14
And we use typically two to represent components of the data.
5:19
So in this case, the file constructor, we expect to be accompanied by a file name and some data paired with the tuple constructor.
5:25
This pair and a folder will have a name and a list of file systems.
5:38
And recall that these types can refer to themselves.

5:44
So intuitively, this gives us a kind of tree structured system.
5:48
So let's just write down some examples.
5:54
So as always, with Cam all the way to think about the values associated with a data type is we build them up using the constructors.
5:58
So let's make a simple example of a file system, which is just a file.
6:09
And I'm getting an error here because we need to have two arguments, the file name and the data that go on with that.
6:18
So let's just make this an example, dot text and some uninteresting text.
6:24
So that's a perfectly good file system, but it's not a particularly big file system.
6:34
In general, we want to have you know, typically the top level of your file system is a folder called The Root Folder that has lots of other things.
6:41
So we could say, you know, let an empty root beer.
6:49
That would be the file system, which is equal to the folder.
6:54
And again, we have to give it a file name. And here where we're asked to provide a list of file systems so we could provide it the empty list.
7:01
And that would be just fine. And we did have another example of another another file system called this one example, too.
7:13
It's another file system that has a folder.
7:22
They will also call this one route. But maybe this one has the example, one file as part of its one of its components.
7:28
OK. And we can go on to use these constructors to represent the shape of a file system.
7:39
And just to give you a few few other examples and to save myself from having to do a lot of typing,
7:48
I've already created a bigger example here called my drive, and it looks like this.
7:55
So my drive is a file system as as with the examples above, it has a kind of root folder and inside of the root folder inside of this list.
8:07
And we have actually three sub folders, each of which themselves have several things in it.
8:21
So maybe one of your folders is called the desktop. And you have three files to do, notes, dot text and insurance on AML.
8:29
And in many, many of these cases, I've just left the data as these strings of dot dot dots because we don't really care so much about that.
8:39
The data there could be another folder called Documents that maybe has your CV, PDAF,

8:47
and maybe you have a folder called Classes that has some folders for Sisse 120 insists 160 and made maybe inside of this is 120 folder.
8:54
You have homework, one folder and one homework to folder and and you get the idea.
9:03
So the point is that with this simple kind of data structure like this,
9:09
we can quite easily begin building up rather complicated data of values like this file system that captures the structure.
9:14
And, you know, these sort of things show up everywhere. In fact, there's probably if you look over encoding of this little widget on the side here
9:28
that provides a kind of nested file view of the files available on my machine,
9:35
probably internally uses some kind of data structure like this to to say how to render this this part of the interface.
9:40
OK, so that's a setup for what the file system is and how Ruiner represent the file system.
9:49
So let's think about how to do a bit of programing with that. So first of all.
9:57
So the task that I have in mind is. A little utility that will search through a file system and look for files of a particular name and
10:04
then give you back the paths starting from the root to all of the occurrences of files with that name.
10:18
So think of this as a really simplified finder or or something like that.
10:24
OK. And so, as usual, with our design process, we want to think about what is the the shape of the interface that we're building.
10:30
So we want to define a function, let's call it. Find that takes a file of a file name and it takes a file system.
10:43
And what I'd like to do is have it return basically all of the paths through this through the file system that that lead to this file name.
10:57
And for simplicity, we're only going to look for files of that file name.
11:12
A good exercise after you finish this video would be to try to adjust the code so that we can also search for folders.
11:16
But let's just for simplicity, assume that. And let's documented.
11:22
F is the name of a file search for.
11:27
S is a file system in which to search and the result here.
11:35
We would like it to be a list of paths.
11:42
We haven't really said what a path is yet. In fact, this is a kind of good exercise in the design.
11:47

So what is a path? If you think about it, you know, you're used to seeing it.
11:54
For example, up here in my browser,
11:58
this kind of slash separated thing is a path through the tree that starts with Codie dot com goes to Steeves and it's like much like straight emblems,
12:01
etc. So if you think about it, a path through one of these file system is is just the name of the folders that you pass through,
12:11
typically separated by a slash. And then maybe to get to say this intro dot airmail file, we would go to route slash desktop,
12:21
slash each word on N.L. and that would be one path and maybe there's another path to the same file.
12:31
So we have route slash classes, slash six 120, slash homework, one intro e-mail.
12:38
So it turned out Armelle appears twice in this file system. OK,
12:48
so that means that we can just save it at a path is really a list of file names
12:53
and a path to a particular file will be a path that ends with that file name.
12:58
So. So that gives us a way of kind of defining this type like this.
13:03
We're defining the type path to be just a file name list.
13:10
And notice that already we're getting a lot of useful information just from the type.
13:18
So a path itself is a list, but we've named that structure and we actually want to get a list of paths.
13:22
Why? Because this file name might show up in several different places like intro dot M.L.
13:29
showed up in two places in this file system and we want to get back all of them.
13:34
OK. So that's that's sort of the first step is identifying this, identifying this.
13:39
And at this point, we would want to say fail with unimplemented.
13:47
And we could also go on to write some test cases already.
13:54
So, for example, we said that in in the file system, the empty file system there.
13:59
There aren't any files. So if we try to do something like look up the look up the file name,
14:09
insert AML in the if we try to find this intricate ide AML in the empty root file system,
14:17
then we're going to expect to get the empty set or the empty list of paths.
14:32
All right. Right.
14:37
And I have to remember to import the assert library, so let me go up here and add that to the top.
14:45

And as usual, with with this,
14:55
I have to actually build the project and then refresh my browser to get Cody to recognize that the insert actually has been built.
14:58
But at this point, the test cases should work and we should be fine,
15:10
although we haven't yet implemented find and we could imagine that if we test and test the case where we look for intrude AML in my drive,
15:17
well, we're gonna expect to get back to pads and we'll have to say, you know, what are those paths?
15:34
Well, we said it was route, desktop, intradermal.
15:41
So that's that path. Is route desktop insured on AML would be one of the paths.
15:46
And the other path would be it's going to do a little bit of formatting.
15:59
Another path would be grouped as classes.
16:05
So, yes, 120 one work, one insured now and hopefully that test will eventually pass.
16:12
OK. So let me.
16:32
So let me summarize what we're trying to do.
16:36
We're trying to implement this fine function that takes the name of a file name and returns the list of all the paths, not just the first one.
16:38
So this is, as you can see, by a non-trivial problem.
16:47
And we're programing basically with this list data structure.
16:51
So the way I like to think about how to structure these problems is to, well, let the type information guide me.
16:56
So we're gonna be searching for this file name inside the file system.
17:03
And the way in general that we examine the structure of data types is by pattern matching.
17:08
So kind of at the top level, we know that we want to match the file system.
17:17
And again, remember, the defining characteristic of OK, well, data types is we knew that file systems come in only two flavors.
17:27
It's either a file or it's a folder. And so those are the only two cases we really have to worry about.
17:36
So if it's a file with some name and some data that we're going to want to do one thing.
17:43
And if it's a folder with a name and some other file system, let's call that the contents of the folder,
17:53
then we're gonna want to do something else and work on each of these parts separately.
18:04
Pattern. OK, so we can get a start by just failing in each of these two cases.

18:16
But at least we've made a bit of progress. So in the case that the file is the one that we're looking for, we we want to actually keep that that path.
18:23
So if the name of the file in the file system is equal to the file that we're searching for,
18:37
then, well, we have to think of what is the list of paths to this to this file?
18:46
Well, the entire file system from this point of view is just this file.
18:51
So there's only one path, namely the path that has the the file name itself and it's equal to the path and otherwise.
18:57
Well, if the file and the name are different, then were we haven't.
19:08
And the entire file system is just this file. Then there aren't any paths.
19:13
And so we would want the empty, empty list of paths. OK.
19:17
So I claim that this is actually the correct implementation for this sort of base case where we have the entire file system.
19:21
It consists of just a file. Now, I want to go over one common pitfall with pattern matching.
19:31
So here I'm checking to see whether the name that we're saying here is the same as the file name there.
19:37
And it's pretty tempting to say, oh, I really want to match this pattern where the file name here is the same as the file name there.
19:44
But in fact, this is not correct. And the reason is because this F out here and this F in here are both identifiers.
19:55
And the pattern doesn't recognize that this file name should be the same as that one.
20:07
In fact, the systems to this this use of F is shadowing that use of F upshot is basically
20:13
you should think of the patterns as just ways of naming subcomponents of,
20:21
in this case, a file system. And you're not allowed to use the computation from an outer scope when defining the pattern for an Interscope.
20:25
OK. So if that just means that we actually have to write the code,
20:37
there's a non-trivial comparison that has to go on between these two files, between F and the name.
20:40
And you might think that. Okay, well, we'll be smart enough to think to imagine that you mean equality, like some kind of equality test.
20:47
If you write that pattern. But in general, there could be lots of complicated relationships between them.
20:54
And Campbell sticks with a simple you can just name this component.
21:01
Okay. So that's that's not such a hard case.
21:07

And we could go on in and see if we could make a test for that.
21:12
But I'd like to press on and think about how do we handle folders?
21:18
Well, let's think about what it would mean to search for a file.
21:23
And again, here, remember, we're only going to be looking for files, not folders.
21:30
But we want to look through all of the contents to see whether the file we're searching for appears somewhere in one of those.
21:34
So as a as a first step, I would say let's let the pads for the basically we want to find you might be tempted to
21:43
write something like find F in the contents and use that to somehow build up the answer.
21:54
And that's on the right track. But we'll get an error here.
22:06
Well, a couple of different errors. So, first of all, we need to make sure that our function is recursive.
22:11
So that would allow us to try to try to do something like this to call find recursively.
22:16
But the type of find expects a file system here and contents, remember, is a list of file systems.
22:21
And so this type error here is actually guiding us towards the correct solution.
22:30
And with a bit more experience, we could sort of see this just from the data type itself.
22:35
So the file system here isn't just recursively defined in terms of a file system like the binary trees we'll see elsewhere in class.
22:39
But it's defined in terms of a list of file systems.
22:49
And what that means is that when we hit this case to do recursion, we need to also take into account that there's a list at this point.
22:53
And so what would it mean? Well, we we sort of want to use find at each point in this list.
23:01
And that's the usual example of why you would.
23:08
Want to help or function, and we could define a helper function outside of the scope here, outside of find.
23:11
But in this case, the fact that file system is defined, file system is defined in terms of a list of file systems.
23:22
That's a strong clue that we might want to have sort of a helper function that is local to find.
23:31
So let's let's call this find in list.
23:39
And what is the structure of this? Well, we're still going to be searching for the same file name, but instead we're gonna have a L,
23:44
which is a or we can even call it contents, which is a file system list.
23:52
And what are we going to get out of this? Well, we we're gonna get back a list of paths which might be in all

of those.
24:02
So. So here where we're defining a helper function,
24:11
that's going to help us use the find operation in all of these sub lists and all of the file systems that are contained in this list.
24:20
And now, instead of just saying paths here, we could just say find the list of this contents.
24:30
And in fact, since we will use the same F from up here, we don't even need to give it F as the argument.
24:39
And now we're making some progress and we have we have paths here.
24:48
We could we paths will not turn out to be the correct answer.
24:54
We'll think about why in a second. But basically,
24:59
what this code is doing is it's kind of the skeleton for using the structure of the data type to calculate information about the data type as a whole.
25:04
Right. So we have this pattern match for the outer file system structure.
25:16
And because the folder case has a list inside of it, we want sort of a helper function that we can define here that will help us process that list.
25:21
And it might itself need to use find because of the recursion, the structure involved.
25:32
OK. So let's think about what happens in this case.
25:39
So let's imagine that paths here.
25:45
If this finding list works correctly, it's going to give us the list of all the paths that have the file name that appear in contents.
25:50
And to get to those paths, we need to go into this folder.
25:59
So each of each of the paths in contents are those belong to this folder.
26:04
So what we need to do actually is to prepend the name to all of those paths.
26:11
Right. So we want to add this name to the start of each of the paths in there.
26:18
So maybe an example would be useful. So if we go back up to our concrete example at the root, we have all of these sub folders.
26:23
So our paths here will be the list of all the pads and those content.
26:31
So maybe if we're looking for intradermal, one of those paths will be desktop interest on Emmental.
26:36
And one of those paths will be classes, SIST 120 homework one intrude on N.L.
26:43
But we're searching for all this under the route. So we need to add a route to the beginning so that, you know, the final answer will be route slash,

26:49
desktop slash Internet email and route slash classes, et cetera.
26:58
OK. So that's what this prepend operation will do. And that's just a simple exercise.
27:04
Homework, one style. We'll see other ways of doing this.
27:10
But if I want to prepend a file name, which is a F, which is a file name to P, which is a list of paths and get back a new list of paths.
27:15
No, this is just the usual kind of structural recursion.
27:31
So if the list of paths is nil. Well, we don't have to pretend to anything.
27:40
And if it has some P path, P with some remainder list.
27:45
Well, we want to add the file name to P and then prepend the file name two P's.
27:50
Right. So that's just a little helper function.
28:02
We could write test cases for it. But basically intuitively it sticks this file name.
28:06
As the first thing on all of these paths to produce a new list of paths.
28:11
And so we could now we could experiment with that and I'll show you a little bit more about how to experiment that using you top in just a second.
28:19
OK. So that takes care of this.
28:28
And again, I want to emphasize that the common way of thinking about recursion is assume that this gives you this correct solution.
28:30
So this is going to be the list of all the paths through the contents.
28:37
And now we just have to add this folder name to the front of each of them.
28:42
And that's what Prepend is doing. So now it just remains to find all the contents in the list.
28:46
But that's actually just another one of these simple kind of list recursion.
28:51
So now we have a list of file systems and we're certainly want to find in each one of those.
28:55
Well, again, if the if the file system is if the list of file systems is empty,
29:07
well, then there aren't going to be any paths that have this file name. So we're done.
29:14
And otherwise, if it's some efforts, Constanta, the rest of the file system.
29:19
Well, in that case, what do we want to do?
29:27
Well, basically, we want to take the list of paths we want to find in this file system, all of the paths that lead to this file name.
29:29

And then we want to collect that up together with all of the paths in the rest of the file systems that are in this list.
29:44
And here I'm gonna use the built in append operation.
29:51
You actually implemented that in homework one, so I'm not going to worry about that here.
29:56
We're using append for appending two lists of paths.
30:00
And what do we want to spend it to? Well, find in the list the rest.
30:05
And again, this is telling me that we need to use the rec keyword because as usual, this is a recursive type.
30:16
OK. So at that point, we could now run our program and hope that that passes the test cases.
30:23
High test error interred on AML reported stack overflow.
30:33
This is the kind of error that you might get if you make a mistake in your code.
30:37
And if you get a stack overflow.
30:43
Yes, that's actually the name of a kind of error or not the name of a Web site that it's telling you to go go look up.
30:46
But we might want to look up for where this program could be going into an infinite loop.
30:52
And I happened to see the problem right away, which is actually I made a mistake in this prepend operation here.
30:59
You can see that I the the parse list here.
31:05
PE's is what I called prepend on here.
31:10
So I didn't actually use the structural recursion rest.
31:13
And so what happens in that case is this Prepend keeps trying to build a list from the same list over and over again,
31:18
and it will just grow infinitely and actually exhausts the computers, the region of the memory called the stack, which is why we get a stack overflow.
31:27
OK. So. So that's good. Let's see what happens if we try to run the project now.
31:36
And in fact, we can now pass both tests. And that's that's useful.
31:40
Maybe a little bit unsatisfactory. So in the sense that, well.
31:46
OK, we could write some tests. But now if I wanted to run this on another example, we would have to write some more code.
31:51
So I thought I would segway to showing you how to use you top to also experiment.
31:57
So experiment with running programs. So you top is available from the menu up here.

32:03
If I click on you top, it will bring me over here. And when you start it, it'll basically bring you into the oh, camel top level interpreter.
32:10
And by default it will have loaded the cert package.
32:20
But if we want to use a MLV file like our files.
32:29
This is called File System Amell. So where did you go?
32:34
I have to use this command.
32:40
So hash you use. And then the name of an e-mail file is as though you've typed it into the top level.
32:45
And you can see that what happens is it tells us that we've defined the file name type.
32:51
And then the data type. And we've defined this file system.
32:56
And it'll even print out. No, the my drive. Big example.
33:01
We have all of the code that we have here. It ran the test cases for us.
33:06
We can see that they're passing here as well. And now we could try to call our find function.
33:10
We have to give it the name of the. So we want to call intro's, find intro on email.
33:19
And I want to run that on the file system that I named my drive.
33:24
And when we run that, you see that we actually get these two paths. Like, so.
33:32
And so that's that's kind of nice. But we can do a little bit better.
33:37
This is not the nicest user interface. So I thought I would show you a little bit more at the bottom of this file I've already printed out.
33:42
I've created some code for printing out a file system.
33:51
The actually turning it into a list.
33:56
And I thought we'd do one piece of it. That's kind of missing as another example. But.
34:00
This code, which I'm calling string of file system, takes in a file system and and produces a string.
34:05
And it also takes a kind of indentation level to help with kind of making the format a bit nicer.
34:13
And the thing I want to point out here is that the structure of this code,
34:19
which processes a file system and produces it into something nicely formatted string,
34:23
has exactly the same shape as the find operation that we did up here.
34:28
So you can see that the main body just checks to see whether the file system is a file or a folder and does, you know, some kind of indentation.

34:33
And Pretty Prince, the file names, introduces some new lines and things, and it rehearses on using this string of list,
34:43
which is the helper function that processes a list of file systems and builds up strings from those.
34:52
So the effect of doing this, if I go over to the boobs, go over to the top.
35:00
Well, now we have a string of file system and we can.
35:07
We can say what? S.T. the string of the file system.
35:14
My drive. And and made a typo and.
35:21
Oh, right. We need to give it an indentation level. Luckily it's telling me that.
35:29
And if I print that string, we'll get a somewhat nicer formatting.
35:34
It's a little bit like the kind of thing you see over here where it says, here's the root file.
35:40
There's a desktop folder that has to do notes and intro documents, etc.
35:45
OK. So that's one one thing we can do. The other thing that might be handy is to make these pads print a little bit nicer.
35:52
So at the top level, we saw that a path is a just a list of file names.
36:01
But the way we print them conventionally is by using something called string.
36:06
By using slashes to separate. So if we have a path, we want to turn it into a string.
36:15
This is just, again, a very simple standard kind of list processing program.
36:23
So we match VSP. If it's the empty list, then we want the empty path.
36:33
And if it's some file name with the rest, well, we just want the file name concatenated with a slash,
36:40
concatenated with the rest of the file in that path.
36:48
And and we actually have to call string of path on the rest.
36:52
So that this hyper was telling me that I need to remember to do that.
37:00
And so now if we go back over to you top, you can use the up error to produce the file system.
37:04
And now we could say something like blit p PS equals find intro data Amelle in my drive that will name this list of paths.
37:11
And now I'm going to use a function called List Dot Map that will get two very soon in class.
37:25
But basically what it does is it applies some function to every element of the list.
37:33
And in this case, I want to do a string of path to peace.

37:38
And but what it what it does is it turns each of those paths in the list.
37:45
Using this string of path. And so we'll get this nicer, nice, more nicely formatted example.
37:52
And if I were to do the same sort of thing, I could just say, let's stop map string of path.
37:58
Let's find the, let's say DNA.
38:08
And now in my drive and you'll it will be able to find that path for me.
38:12
So that's gonna be coming up. And more to add in homework in subsequent homework.
38:21
We'll see how this dot map works. OK.
38:25
So that's all I wanted to show you.
38:29
This is a been a I hope, a good deep dove into your programing with data types and pattern matching and reminding you of list recursion.

(end of excerpt)