CURRAN KELLEHER: So my name is Curran Kelleher. I'll be lecturing today about recursion and fractals. Justin Curry's not here today. So I'm going to fill in.

So today I'm going to just do a bunch of example programs, computer programs that are recursive. Some of them don't make pictures, and some of them do. And when they make pictures, they're fractals. Fractals are things that are self-similar at different scales. So you can zoom in on them. We're going to take a break at 4 o'clock for 10 minutes. And then towards the end, hopefully I'll show you a bunch of examples of fractals and play some Bach music.

So first of all, let's consider a recursive mathematical function, factorial. Something factorial-- like 3 factorial is 3 times 2 times 1. 4 factorial is 4 times 3 times 2 times 1. So this is factorial, the exclamation point.

So the way this is defined is actually recursive. So if you take anything factorial-- let's say n factorial is n minus 1 factorial. Let's say n is 4. n minus 1 would be 3. So 4 factorial is-- wait, n times n. So it's going to equal n times n minus 1 factorial. So 4 is n times this whole thing is 3 factorial. It's n minus 1 factorial.

So this is a recursive definition. And if you look in the handout, I wrote a little computer program on page 3, I think, that does the factorial. So it goes like this. So just for those of you who don't know much about programming, def means define. So we're defining a function called Factorial. And it takes as an argument n. So you can call this function, pass it a number. And inside the function, it's referred to as n.

So this factorial function says if n is greater than 1, return n times factorial of n minus 1. Else, return 1. So what makes this function recursive is the fact that it calls itself. Factorial is defined as n times factorial of something else. So a recursive function is a function that calls itself.

So I'll give you an example. So if n is 5, say, the way we call this is, we say factorial of 5. So when we say this, it calls this function and gives n the number 5. So what it's going to do is n is

going to be 5. So n is greater than 1. So we return n times factorial n minus 1.

So say we're doing this algorithm. 5 is n. So we're going to return 5 times factorial of n minus 1. So we call factorial with the value 4. And that's also greater than 1. So we return 4 times factorial 3. So 5 times 4 times factorial 3. So we loop-- we call itself a bunch of times until we get down to 1.

So that's what actually ends up happening. This is recursion. So another simple example, which is sort of like the factorial, is the Fibonacci numbers, Fibonacci sequence. So the Fibonacci numbers, it goes 1, 1, and then the next one you add the first two together. So it goes 1, 1, 2. 1 plus 2 is 3. 2 plus 3 is 5, and so on, 8, blah, blah, blah. These are the Fibonacci numbers.

So this is a recursive definition. Let's say this is-- this is like the number of the element. Like they're just numbers-- indices, if you will. So Fibonacci of 2 is Fibonacci of 0 plus Fibonacci of 1. And so Fibonacci of 5 is going to be Fibonacci of 3 plus Fibonacci of 4.

So generally, Fibonacci of n is going to be Fibonacci of a n minus 1 plus Fibonacci of n minus 2. So we could say that here. Instead of factorial, we call it Fibonacci. And we'll notice that they're almost the same thing. I'll say fib. If n is greater than 1, we return Fibonacci of n minus 1. And this gives us the Fibonacci numbers actually. Does that make sense to you guys? So in the handout, there are some example outputs of both of these. You could see that's what happens.

So who has their copy of *Godel, Escher, Bach* today? So if you will, look on page 132 of *Godel, Escher, Bach*. Oh no, now I can't look at it. Oh well. 132 of *Godel, Escher, Bach* has these two diagrams that are recursive transition networks. They define a grammar, sort of like English. It's not English. It's not complete. It's a simplified version of English. But it communicates the essence of the notion of a grammar, a recursive grammar.

So you'll notice that-- it's hard to do it in my head. Fancy noun, one of the notes calls fancy noun again. It loops back out on itself. So this is where the recursion is. So what I did is I took this diagram and wrote a little computer program that whenever there's a choice of the transitions, it chooses one of those transitions at random. And this is the program I wrote. I think it's on page 5 of my handout.

So if you look at that-- yeah, I wish I had the projector. It's unfortunate. Actually, can I look at

the diagram? So if you look at fancy noun, we begin, and it calls ornate noun. And if you look at the program in the handout, you find fancy noun. It's sort of halfway down. It says the [? RTN ?] for fancy noun.

Fancy noun equals-- and this is a function call. Well, first of all, fancy noun equals-- when you put curly braces around something, it makes it a function, pretty much. So fancy noun equals-- and it copies pretty much directly from the diagram. Ornate noun, which is also a function call, which is defined above, plus-- I'm back in fancy noun. So ornate noun plus pick from preposition or a relative pronoun or nothing.

And if you look at the diagram in *Godel, Escher, Bach,* the arrows coming out of ornate noun point to relative pronoun, nothing-- the end-- and preposition. So you can make this into a computer program, which is what I did. So if you look at my program for a little while, you'll notice that all the arrows in the diagrams correspond to function calls in the program. And they're recursive because they eventually loop back on themselves.

So here, Hofstadter is trying to communicate the fact that languages themselves are defined by recursive grammars. This is why we can nest sentences inside of each other. It's recursive. So recursion leads to nesting, and sometimes incident nesting. And that's where fractals come from.

So right after this program in the handout, there's a sample output. So just read through some of those sample outputs. They're pretty funny. I have an old version. Can I look at someone's handout, just to read? So small, small bagel inside the strange cow. It makes sense as an English sentence. And it was generated by this computer program. So I think that's just fascinating. But of course, some of them don't make sense, like large small bagel that runs large. Small large horn. It just doesn't make any sense.

So next example. So the next example on page 5, I think, is a tree. We're going to make a tree picture using recursive functions. So I'm going to write some pseudo code. It's not real code. It's sort of pseudo code to communicate the idea of what this program is doing.

So we have a function that grows a tree. It starts from a single branch. I'll do it [? at that one ?] here. This is like the starting point. This is the entry point. So it says, like class, all this stuff, tree. Tree parentheses, that gets called when the program starts. So this function call, growtree, [? 0.50 ?] trunk height, all this stuff, is this first one. This is what initiates the process.

And then what the function does is pretty much if the depth is greater than 0-- so this tree function calls itself twice, once for each branch. So the first time the program calls this function, it makes this. It draws it on the screen. So notice here, it adds the actual line. If you look at the code, it says add new JV line, x1, x2, that stuff. That actually adds a line to the screen.

I wish I could show you the actual code running, but I can't. So this is the first time. And then depth is the number of times it's going to branch out. And so depth is 11. It's going to make a big tree. But what it's going to do is make two subbranches. This is growtree. This one correlates to this one here. This one corresponds to this one here.

And if you look at the code in the handout, it says growtree x2 y2. x2, y2 is to the endpoint of the previous branch. Root length time size factor. So size factor is a factor by which it's going to scale. So this would be like maybe half the size, or 0.7. Size factor is 0.58. So it's about half the size. And root angle plus angle factor, root angle minus angle factor. These are in the two growtree function calls.

Angle factor, which is pi over 4. It's exactly 45 degrees. So each time it branches, the two branches are going to branch out at that angle. And so here, say we do depth of 4 when we call it the first time. The depth here is going to be 4. And then you pass into the function depth minus 1. So here, inside the function, it's generating this depth. It's going to be 3.

And so it's going to keep going down until depth is 0. So here depth is-- well, depth is 4, 3, 2. That's 1. And it does it on this side too. So it just keeps going like this. This is a fractal. And you can imagine, if you were to continue this infinitely-- instead of saying depth of 10 or 11, just say depth of infinity. Say theoretically, if we could do this, this shape could be zoomed in on infinitely forever. And this is the notion of a fractal. And it would look the same as it does on the large scale. So any questions about the tree?

So next we're going to do the Koch curve. The Koch curve is sort of similar to the tree, except that the rules are different. So first, conceptually, this is what a Koch curve is. You start with a line, or in some cases, a triangle to make the whole thing. And you divide it into three parts. And then you make an equilateral triangle, meaning the sides are all the same, out of the thing in the middle. And get rid of this line.

So this is the rule. Go from a line to this thing. And then this rule is applied to each one of these segments. So you go like this, and so on, like to each of these smaller segments. So the

fact that the simple rule is being applied to something over and over again, and the thing it's being applied to is the result of the previous execution of the rule, makes it recursive. So we could keep drawing it.

Let's look at the actual code, the program. It says Koch. I think it's on page 7. So let's see, createCurve. What createCurve does essentially is call itself four times. And each of those four times corresponds to this one, this one, this one, and this one.

So yeah, let's look at the actual code. You see these four function calls to createCurve in the middle there? x1, y1, ax, ay, cx, cy, by, all the stuff. If you look at the little red diagram, this point is x1, y1. And it's in the diagram what the points are. So the first function call pretty much says apply the rule to this segment. The second function call starts from a, which is this point here, which says apply the rule to this segment. The third one starts at c, which is the top point, and it says apply the rule to this segment. And the fourth one, apply it through to this one. So it's another beautiful recursive fractal.

So what happens-- so this is the Koch curve. If you generalize this and add randomness to it-- say these points are a-- it doesn't matter what they're called. If you move this one up and down a little bit randomly-- so you start from this-- instead of making these points exact, make them like a little bit off. And then connect the lines and make a triangle there. And this is also a little bit off. And then if you keep doing this, adding a little bit of randomization each time, you actually get lines that look just like coastlines around pieces of land. And if you generalize this into three dimensions and do this randomness, it actually generates 3D mountains, like virtual 3D surfaces that look exactly like real mountains. So it's sort of strange, these recursive structures are definitely in nature.

AUDIENCE: What does it mean for the front page, page 6 that Koch simply has finite error but infinite [INAUDIBLE]?

CURRAN KELLEHER: Oh yeah. Yeah, I forgot to mention that. That's a really cool thing. So the Koch snowflake is when you start with a triangle, and you apply the rule to each size of the triangle. And you get this curve on all these sides. So it's been mathematically proven, or extrapolated, that if you were to do this rule an infinite number of times-- which you can do in math, because it's all theoretical-- the volume inside of this object will be finite. It's a definite amount.

But the surface area is infinite. Oh, not surface area. The perimeter. The perimeter is infinite. That's because every time you apply the rule, you actually increase the perimeter. So this has

a certain length. And then once you apply the rule to it, if you do this, then this new curve, the total length is longer than this one. So you can imagine if you do it infinitely, it's just going to be infinitely long. So it's sort of mind boggling. And this goes--

AUDIENCE: [INAUDIBLE] smaller and smaller?

CURRAN KELLEHER: Yeah, it gets smaller and smaller, definitely. But if you do it theoretically an infinite number of times, it'll still exist. If you look at the picture on page 6, right next to the title, the Koch snowflake. That curve there, you see it? That's what it would look like, even if you did it an infinite number of times.

AUDIENCE: So [INAUDIBLE] go on infinitely?

CURRAN KELLEHER: Say again.

AUDIENCE: [INAUDIBLE] maybe have a kind of snowflake [INAUDIBLE]?

CURRAN KELLEHER: So you just cut it in half?

AUDIENCE: Yeah, not in half, but just kind of [INAUDIBLE] the segment to go from one part of [INAUDIBLE].

CURRAN KELLEHER: I don't really understand. You can draw it on the board if you want.

AUDIENCE: On the board?

CURRAN KELLEHER: Yeah.

AUDIENCE: Have like a circle-- I just had this part over here. And then even if this is finite area, you're saying that if I cut it like that, [INAUDIBLE].

CURRAN KELLEHER: Oh, I see.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: I see what you mean. So say you had this Koch curve, this shape, and it were a string. And you would cut it. So the string would now be loose. And if you pulled it, it would go on forever. Yeah. It would.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: Yeah. That's what it means to have infinite perimeter, which is why it's just so fascinating. It's crazy. People tried to measure the coast of Britain. How long is the coast of Britain? Have you heard about this problem?

If you look at it on a large scale, like from a satellite image of the whole country, you could just draw a line around it and say oh yeah, it's this length. This is the length of the coast of Britain. But if you zoom in on it, you'll find that those big lines that you drew are actually really wrong. They don't actually line up. So if you make it more precise to its new zooming angle, the zooming-- it gets longer. And so actually the more that you zoom in on the coast of Britain, the longer the perimeter gets.

AUDIENCE: [INAUDIBLE] just adding [INAUDIBLE].

CURRAN KELLEHER: Yeah, just adding little pieces. So say the actual coast of Britain is like that. But you look at it at this huge distance away. You say, like oh yeah, this is approximately that line. But if you look at it in more detail and refine it, you say, oh, it's not actually that line. It's maybe like these lines. But the new lines are actually-- the whole thing is longer. And if you do it even more and more precisely, you just get longer and longer and longer. So nobody's been able to really figure out how long the coast of Britain is. It's a fractal.

AUDIENCE: [INAUDIBLE] is it even possible in this [INAUDIBLE] to have [INAUDIBLE]?

CURRAN KELLEHER: Well, not really. I mean, so he asked, is it possible in this world to have something that's really infinite like that? No, it's not. Because I mean, there's only finite space on the Earth.

AUDIENCE: Yeah, but even that, I mean, you just said that given [INAUDIBLE].

CURRAN KELLEHER: Yeah. So the Koch curve theoretically, in math language, if you do it mathematically, it has infinite perimeter but finite area. But keep in mind, this is a theoretical creation. It's just in the world of math. And it can't really exist in the universe. But things come pretty close to it in nature. It's not actually infinite. The coast line of Britain is not actually infinite. But it really resembles this sort of shape. So let's see. Yeah. So any questions about the Koch curve? It's

really interesting.

So the next example is the Sierpinski triangle, page 8. So the Sierpinski triangle is very interesting. Whoa. You take a big triangle, and you add a smaller triangle inside of it like this. So this is the rule for the Sierpinski triangle. And this is colored in or something. So this is the rule. And what you get is three new triangles. And then you apply the same rule to these new triangles.

So that's recursion, when you apply a rule to something that you already applied a rule to. So you apply it again, and you get these even smaller things, these smaller triangles. And it goes down infinitely, if you do it infinitely. So imagine this.

So this is one way of computing it, one way of doing it, one way of looking at the rule. But what I did, if you look in the lecture notes, I talk about iterated function system. So that means-- well, I'll just do it, and you'll see what it means.

So let's say we start with a point. Say this point here. It could be any point. And we have three possible choices. This is called the chaos game. We have three possible choices of something to do with this point. We either bring it halfway to this point, halfway to this point, or halfway to this point.

So let's say we bring it halfway to this point. We go right here, right in the middle. And what we do, in the iterated function system, we do this over and over and over again, picking randomly which one of the three points we go halfway towards. So say we go to this one next. We go here, halfway. Then we go halfway to this one. Halfway to this one again. Halfway to this one again.

And then halfway to this one. Then halfway to this one. Halfway to this one. And halfway to this one, halfway to this one, and halfway to this one. So we just plot these points. And the program that I wrote, which is in the handout, does this. It executes this. It just keeps going, and it keeps plotting the points. And eventually what you get is the Sierpinski triangle. It's crazy.

So page 8, it's-- the picture of this is on page 8. Yeah.

AUDIENCE: So when you do it randomly, it doesn't matter which one you choose? No matter what you choose, whenever you do [INAUDIBLE], it's going to go to the same? [INAUDIBLE] started the

program again, but now this time, since it's going to have random [INAUDIBLE] same triangle?

CURRAN KELLEHER: Yeah, exactly. So he's getting at say you run the program again, it's going to choose differently. Maybe it will choose, instead of going to this one first, it'll go to this one first. Say it does it 10 times or 100 times to this one. But the program chooses randomly which one to go to. [INAUDIBLE] it just chooses randomly. So he asked, even though it's random, will it still generate the same picture? And the answer is yes, it will. It's crazy. It's just fascinating.

And we'll understand this better after we do the next example, which is making a fern, a fractal fern, which is really cool. So let's just look at the code quickly. And think about this. Is the code for the Sierpinski triangle recursive?

So what it does is, see def drawSierpinski. That's the thing that draws the triangle. Def a, b, and c are these three points. So this is like a, b, c. Def points equals a list of a, b, and c. Current point is just start at point a. This statement while true means, execute this code repeatedly. Just keep doing it an infinite number of times, until you stop the program.

So it says def next point equals pick from points. And pick from is a function defined above, which pretty much picks at random out of the list that you give it. So that pick from is the thing that picks randomly which one of these to go to. So it says, next point pick from points. So that gets you a point. And then current point x equals current point x plus next point x divided by 2. So what you're doing is averaging the x-coordinates of the current point and the point that you're going to go to next.

And if you do that for x and y, what you do is that's going halfway between the current point and the next point. That's what that does. And [? image.fillpixel.point. ?] So that's what plots it on the screen. And the picture here was actually generated by this very program. So it keeps going, keeps plotting it.

So here's the question. Is this recursive? Is this thing actually recursive? Because we said a recursive function is a function that calls itself. Let's hold off on that answer until after we do the next one. Any questions so far? So the next thing we're going to do is the fern. Yeah.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: So recursion is-- generally, recursion is something which is defined in terms of itself. Something that loops back on itself.

AUDIENCE: [INAUDIBLE]?

CURRAN KELLEHER: Say that again?

AUDIENCE: If you apply a rule-- it will become the same thing you started with.

CURRAN KELLEHER: If you apply which rule?

AUDIENCE: Any rule.

CURRAN KELLEHER: The recursive rule?

AUDIENCE: Yeah.

CURRAN KELLEHER: It will become the same thing that it started with? No, not necessarily. Because each time it applies the rule, there is a slight change. With this factorial, it's not just saying n factorial equals n factorial. It's saying n factorial equals n minus 1 factorial. The factorial part is recursive, but it doesn't loop back on itself exactly. There's some change. And with recursive functions, at least-- so the function is defined, it calls itself.

I don't know if this will really answer your question, but say if we didn't have these parts-- if the function just was this, def Fibonacci, return Fibonacci of n minus 1 n minus 2, what would happen is, it would call itself and just keep going infinitely, which is a problem.

So all recursive functions, in order to do anything, have to bottom out at some point. They have to stop. And that's what this is. Only do this if n is greater than 1. If n is less than or equal to 1, return. So this stops it. So does that answer your question? You can ask it again, if you want.

AUDIENCE: So for the triangle [INAUDIBLE] might be a recursion?

CURRAN KELLEHER: It is.

AUDIENCE: Because it has to return to some place [INAUDIBLE].

CURRAN Only functions, computer functions, that are recursive need to bottom out. There are other

KELLEHER: kinds of recursion.

AUDIENCE: Natural recursions [INAUDIBLE]?

CURRAN KELLEHER: Well, they do like. Natural recursions don't bottom out. Well, they do have to bottom out eventually. Take, for example, a tree in nature. It branches and branches and branches. But eventually it just gets to a leaf. And it stops branching.

There's some sort of program that's executing inside of this tree that's recursive. And there's a certain signal that this program gets when the branch gets to a certain size, I think, or something. That signals it to generate a leaf. So I mean, those kind of recursive processes do bottom out. But this one is actually recursive. We'll see why in a minute. But it doesn't bottom out because it just keeps going forever.

AUDIENCE: Do you mean that when you have [INAUDIBLE] they don't [INAUDIBLE] practical purposes like [INAUDIBLE]? But if you have to get a result from the program, then you have to say [INAUDIBLE].

CURRAN KELLEHER: Yeah. Exactly. Exactly. Yeah, that was very well put. So what he basically said was, if you want to get any real manifestation of recursion, it has to bottom out in order to return you the result. But yeah, you can imagine theoretical worlds where it doesn't bottom out, goes on forever. Actually, Douglas Hofstadter in the dialogue before this chapter does that. A genie has to ask a meta genie has to ask a meta genie. Did anybody read that?

AUDIENCE: Yeah, but every time it does it, it gets smaller and smaller [INAUDIBLE].

CURRAN KELLEHER: Yeah. The time is smaller and smaller. So eventually, it would just repeat an infinite number of intensely small--

AUDIENCE: Yeah, but still only lasts a finite amount of time.

CURRAN KELLEHER: Yeah. Yeah. Isn't that crazy? Yeah, it's really something to think about. Yeah. But it's theoretical. But it is very interesting to think about. So we'll see how this is recursive after we do the fern. So the fractal fern is next, page 9. Yeah, this is really, really cool.

So the fractal fern. It looks beautiful, doesn't it? So we have this triangle. This isn't the picture in the handout. There's an outside triangle that's black. And then there's an inside triangle that's blue and some other triangles that [INAUDIBLE] leaves.

So first of all, I want to talk about this notion of a coordinate transformation. Say we had a coordinate transformation between this Rectangle And this rectangle. What would happen-- it's a function on a point, a two-dimensional point. So say you had a point right here, this point. You apply this transformation to the point. And what you get is this point here.

So you had this point here, and you apply the transformation to that point, you'll get this point here. It's just like a little copy of this. OK. So this is a function. This is the function part of iterated function systems. Iteration is the fact that you just keep doing it. And a system is the fact that there's more than one of them. It's like a bunch. In this case, it's three. I'll talk about it in a minute.

So with a fractal fern, what we have is a rectangle like this. So the fractal fern is an iterative function system that has four possible functions. This one has three. Each one of those points is a function. Going halfway to one of those points is a function.

But in the fern one, there are four functions. This is one of them. And each function is a coordinate transformation. This function is a coordinate transformation from this outer one, this outer rectangle to this inner rectangle. So if we had this point in this rectangle, and we applied this transformation, we would get this point here.

So say we have the middle point of this outer rectangle, and we apply this transformation once. We would get the middle point of this inner rectangle, which is about right here. But here what we do is we say, OK, now this new point is actually in the outer rectangle. And we'll apply the transformation again on that.

So this point in the outer rectangle maps to about this point in the smaller rectangle. And you say, OK, this point is now a point in this one, and you apply it again. And you get this point here and this point here and this point here and all these little points. And eventually, they just get smaller and smaller because the corner points of these two rectangles are at the same point I think [INAUDIBLE].

So let's look at another one. Let's see. So this is the thing that's going to map to. It's a line, but think of it as a rectangle. It's a coordinate transformation. So we go from any point in this outer rectangle to a point on here. So say if we have this point here in the outer rectangle, it's going to go to the top point of this line. If we have this point here, it's going to go to the bottom point of this line. If we have the center point, it'll go to the center of this line.

So let's imagine an iterated function system with only these two functions. What's going to happen is-- and let's say each one is chosen-- well, let's say each one is chosen about half the time randomly. It picks between each one. Or no, let's say that it applies this mapping about 90% of the time.

So say we start with this point here. It'll map to this point here and this point here, and it'll just go up and up into here, as long as this transformation is being applied. But say it does that like 100 times, and then one time it goes down to here.

So say it's all the way up here, and we map to this one. It's going to start here. And it's going to map here. It's going to go up. And say we [? stop ?] at this point the computer decides to map to here. It's about [? 3/4 ?] up. It's going to go to [? 3/4 ?] up here.

So because it's random, if you do this just forever, it's going to eventually fill in pretty much every point on this line, which is very interesting to think about. Any questions so far? So let's have this rectangle. All four of the transformations are from the outer rectangle, this big one, to one of the ones inside, one of the smaller ones inside.

So say we have this point up here, and we apply this transformation to this one. It's going to go to this point here. Make sense? And if we have this one, it's going to go to this point here. So the transformation is pretty much going like this. So say our IFS, Iterated Function System, has these three. Say we go about halfway up here, and we choose to apply this mapping. It's going to go about here, the middle of this one.

And then we apply this mapping a bunch of times, just keep going up and up and up. We apply this mapping again. But it's closer to the top this time. So it's going to be out here, closer to the top of this rectangle. And then we apply it again and again and again. And maybe it only goes up 1.

So it will be down here. So eventually, it's going to fill in this rectangle with this pattern. It's going to look like this. And this-- since every point in here is probably going to get applied to this mapping a bunch of times. So what we're going to get is this fern structure. OK. Isn't that really cool?

AUDIENCE: [INAUDIBLE] small triangle the points that you have in there apply to the biggest point [INAUDIBLE].

CURRAN KELLEHER: Are you talking about the Sierpinski triangle or the fern?

AUDIENCE: The fern. If you have a bunch of points [INAUDIBLE].

CURRAN KELLEHER: Yeah, exactly.

AUDIENCE: [INAUDIBLE] smaller and smaller [INAUDIBLE].

CURRAN KELLEHER: Right. You can't, actually. And this is-- OK, I'll go through an example. Say we have-- instead of just applying it to one point, we apply it to a bunch of points. Say all the points on this line, or almost all points. Say we apply this mapping a bunch of times. So we get this one, this one, this one, this one.

And say we apply this mapping to all of these points. It's going to map to here. And then we do that again and again and again. And then what we have, say all those points, eventually are going to get mapped to this one. So you have a whole little copy of this thing in here. So you have these little branches. And then that is going to be mapped up here. And [INAUDIBLE] branches, branches, branches.

And then since you have the smaller branches here, you have two levels down. And then this whole thing gets mapped back on to here, including the new branches. And it gets filled in as it goes on. But it doesn't just keep going up to here and just-- say we were to apply this mapping 100% of the time. It would just go up here and go nowhere else.

But say we apply this mapping 7% of the time. It'll go up different lengths. And then it'll map back down there and go up and back down. And if we add this transformation, going from this one to this one, we have this. And it's recursive. The reason why it's recursive is because the mappings map onto themselves eventually.

If we look at the code, it's really similar to the Sierpinski triangle code. The code itself is not recursive, but the mappings are. And that's what makes this whole thing recursive. So yeah, let's just look at the code a little bit. [? Class ?] fern. So [? drawFern, ?] while true-- so that means execute this. Just keep doing it over and over and over-- pick from these list of transformations.

So the first one says maps to the stem. That means it goes from here to here, to the stem.

The second one maps to the left branch, meaning it goes from here to this one. The third one maps to the right branch. It goes down here. And the fourth one maps from here to here. It just goes up like that.

And following that, it says pick from a list of functions. And then a list of probabilities. It says 0.01, [? 0.07, ?] [? 0.07, ?] 0.85. So these are the probabilities at which these mapping functions are going to be chosen. If you choose them all equally, then the chances are very low that this mapping is going to occur more than like five times or something. So you just go up here and get mapped.

So it would be a very sparse fractal. Most of the points would be down here. It just didn't look very good. I tried it. I wish I could code it and show you. So the 0.01 means that the mapping to the stem happens 1% of the time. The mapping to each of the branches happens 7% of the time. And the mapping up and spiraling happens 85% of the time.

So I'm going to take a break now for 10 minutes. Any questions? So I guess we'll start up again. Before we leave these iterated function systems, I want to point out how the Sierpinski triangle is pretty much the same thing as the ferns in terms of mappings. With a Sierpinski triangle, we have these three mappings.

One of them maps from this triangle to this triangle here. One of them maps from the big triangle to this triangle. And the other one maps from this triangle to this triangle. And each one of them are applied with equal probability. So if you think about it, going halfway from any of these points to one of the other points is essentially mapping it down.

So say you have this one, this line. You apply the function to all these points. What it does is maps it down to that triangle. So you could see these triangles map onto themselves recursively. And that's why it actually-- there is a recursion going on here, but it's not in the code itself. The code itself is just repetitive. But what it's repeating is these recursive mappings onto themselves. Yeah.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: There's always more than one way to represent an algorithm. Yeah. It's really fascinating, especially with these fractals and things. It seems like there's always another way to do it to achieve the same result. And that's one thing, especially Sierpinski's triangle. It's really amazing.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: In my what?

AUDIENCE: [INAUDIBLE] you don't have a recursive function. So [INAUDIBLE]?

CURRAN KELLEHER: Ah. So how can you figure out if the output is going to be recursive without running the code?

AUDIENCE: [INAUDIBLE] you have the code, but it has no recursive function. Yeah. You have [INAUDIBLE] recursive function [INAUDIBLE] how can you find out that the output is going to be recursive without actually [INAUDIBLE]?

CURRAN KELLEHER: So he said since the code itself is not recursive, how do you know that the output is going to be recursive?

AUDIENCE: [INAUDIBLE] you don't have any recursive functions.

CURRAN KELLEHER: Even without any recursive functions. Right. I mean, the thing is, when you write the code, if you realize that what the code is doing is mapping these two-dimensional mappings, you can sort of think in your head, OK, these mappings are going to be applied with equal probability. So just by mentally extrapolating in your head-- so you're sort of stepping out of the system. You're saying, OK, what's the thing actually doing?

AUDIENCE: But are you even allowed to do that? I mean, can you just [INAUDIBLE] in the system?

CURRAN KELLEHER: While you're in the system? Well, being in the system in this case is actually running the code and being a computer program running it. Stepping out of the system, but what I mean by that is taking a higher level view of what's going on.

AUDIENCE: So when you abstract [INAUDIBLE] which actually does show recursion.

CURRAN KELLEHER: Yeah. Exactly. Exactly. Yes, that's it. So he said when you abstract away from all the details of what's going on with the actual functions, you begin to perceive this higher level thing, which is itself recursion. Yeah. And has recursion in it. So if you abstract your thought process significantly enough, you'll be able to logically tell that it's going to create this recursive shape. Yeah.

AUDIENCE: But it's not like a simple algorithm [INAUDIBLE].

CURRAN KELLEHER: Yeah. So you mean like a test?

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: No. You have to think about it. That's what humans can do. Like there's no strict test that could be applied to some computer program that would tell you whether or not it's going to create a recursive result. Because in this case, there's no recursive function. So the computer can't know, OK, can I abstract into-- only humans can do that yet, so far. And it's not coded up as a system of mappings. It's just these simple functions. I don't know. So yeah. It's really cool.

So yeah, if you just think about mapping, mapping, mapping. Then you map it over here, all the stuff that was in here that goes infinitely down is now in this little subsection of this little triangle. And you just-- yeah. It's a recursive structure.

So it's going to the next example, which is very interesting, the Mandelbrot set. First, now that we have this up and running, I just want to run the programs that are in the handout to give you a sense of what's going on. So the recursive transition network, let's run it and see what happens. It's going to generate 100 sentences. Because if you look at the code, there's a print statement that goes 100 times.

Yeah, page 5. It says, [? 0to100.each, ?] print line, call the function fancy noun. So this is just fancy notation in the Groovy programming language to-- oh, crap. It's not working. Oh well. I'll just try to get this to work as I'm talking. I can't do the examples. Yeah, I've been having problems with this thing all day.

So the next example and the last example is the Mandelbrot set. You guys probably have heard about the Mandelbrot set. It's very famous, probably one of the most famous fractals of all. It's called Mandelbrot because this guy Mandelbrot really loved fractals and tried to communicate the notion of fractals to the world. And this is really a great fractal. So it'll get a little bit mathy, but that's OK.

So we'll have-- it's in the complex plane. So that means that we have this plane where these are the real numbers, and these are the imaginary numbers. I think this is [? the way it goes. ?] So the Mandelbrot set is defined by the function z equals z squared plus c.

And when you square a complex number-- well, first of all, a complex number is a point in this plane. So say this point is-- this is b, say, and this is a. This point is represented by a plus bi. And i is the square root of negative 1, which is not really possible. That's why it's called imaginary.

So if you multiply two complex numbers together, it's not really that simple. You have to do the actual multiplication out. What you get is ac plus-- I don't know if I should do it all out. It'll take a little bit of time. But you got this sort of a mapping function essentially that's not really linear. It's not really that predictable, what it's going to do.

So you have this point here. And so the way the algorithm works, you take a point on the complex plane. c gets the value of that point. And then z starts off at 0, just 0, 0. And you apply this function a bunch of times. So say we apply this function once. z equals z squared plus c. So 0 squared is nothing plus c. So the first iteration, z gets the value of c. So z is going to be there. And you apply the function again. And this z is now the new z that we just got.

So z equals z squared plus c. So z squared, you apply the squaring function. And you add c. You add the real and imaginary components of c. So it just maps this point to some other point over here, say. Then you apply this again and again and again and again. So apply it here, here. So I don't know if I'm going too fast, but we get this point after applying the function to this. So now this is the new z.

And then the new z loops back around. And so we apply now the z squared plus c. c is always going to be the initial point, but z will be changing. And it maps to here, for example. And now the new point is z. [? And you see ?] z squared plus c again that maps right here.

So I wrote a little applet that demonstrates this mapping and what it actually looks like. Yeah.

AUDIENCE: [INAUDIBLE] z can be anywhere in the plane?

CURRAN KELLEHER: z can be anywhere?

AUDIENCE: [INAUDIBLE] starting point.

CURRAN KELLEHER: The starting point could be anywhere. Yeah. But when you actually run the algorithm, you only go from negative 2 to 2. Negative 2, 2. Because that's the only region in which interesting things happen with this fractal. So I didn't finish explaining the algorithm.

So say a point in here. A point that's very close to the origin-- 0, 0-- what it tends to do is spiral inward as you iterate it. But a point out here, what it tends to do-- definitely a point outside of 2, what it tends to do when you apply this function is to get mapped out here, and then to get mapped over there, and just go really far away.

So what we do to render the Mandelbrot set is apply the function a number of times. And if the point eventually goes outside this circle, at the origin of radius 2-- sorry, it's a bad circle. If the point eventually goes outside of this circle, we stop performing the function. Because we know that after it gets outside, it's just going to keep going forever out. It has escaped. It's called the escape iterations, the number of iterations it takes to escape outside of the circle. It's the escape iterations.

So what we do is we color the point depending on its escape iterations. So when we actually-- and the other part of it is, if it never escapes, we color it black. To actually test that, we just iterate a certain number of times, say like 500 times or 70 times. And if it hasn't escaped after that many iterations, we sort of assume that it's never going to. It's not valid. So it's an approximation to the actual set. But we have to do it to render the actual thing. Then we color it black.

So when we actually render it, what we do is go through each pixel on the screen and apply this algorithm. So for this point, this point becomes c. And we iterate, test, iterate, test, iterate, test. And if the test-- what we're testing is if it's outside of the circle. If it's outside of the circle, we assign it a color based on the number of iterations it took. And we do that for each one.

So we keep going. And say this one, for example, is going to spiral in. It's not a continuous line. It just applies the function and spirals in. So that's never going to escape. So we're going to color it black. And this applet is excellent at showing this.

So I implemented this algorithm in Java, and it's going to output the point, and it's going to draw the lines between each point. So say it starts here. It's going to draw a line here, here, here, here. So what we're going to see is really cool.

So this is what we get in the end. And as I move my mouse around, it does these tests. So here we can see that there's only one iteration. The one iteration gets the color light blue. And then here, there are two iterations. And they're listed up there, one, two. So you can see all the way, wherever it's this color, there are only two iterations before the point gets outside of

this circle to [INAUDIBLE] I think.

So as we go deeper in, it takes three iterations, four, five, six, a lot of iterations. So all the points that are colored, they do eventually escape. But all the points that are black don't escape. And so the patterns that are inside are really interesting. See these in the middle, they just sort of spiral in and keep going. The ones here, there's some looping pattern. See, the function loops back on itself. So it's sort of recursive. Like I don't really understand how it's recursive, but it sort of is.

So if you go out to this other nub, the second nub out, it gets this really cool shape. So what we can do is zoom in actually on this. And it redraws it. So it's calculating this algorithm for every single point as we speak. It's doing it really fast. So we can see all these really cool shapes that generate this fractal. And it's just crazy.

So we can sort of see recursion here. It looks like they're definitely some recursive thing going on. And the recursive part of this is the fact that z gets applied to itself. The n-th z is equal to the z that came before it squared plus c. And then this z is reframed as the old z. And this reframing and re-applying of the function is what makes it recursive.

So yeah, it sort of makes sense. Let's just go through the code quickly. And then I want to show some really cool things. Oh, where did it go?

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: So yeah, in the black part, it just recurses infinitely. And the code itself is not recursive, but the mapping function, again, is what's recursive. So yeah, the black parts are in the set. So the real actual Mandelbrot set is just the black points. We just color the other points so it looks pretty.

So let's look at the code. Def [? drawMandelbrot. ?] See that there? window.set negative 2, 2. So it just sets the window, the plotting range to be in this range. For every x pixel in the height of the window, or the width of the window, and every y pixel in the height, c.real and c.imaginary get the x and y values on the screen.

So what we're doing there is saying, basically the pixel gets this actual point in the space that we're working-- the complex plane. Def iterations equals calculate iterations c. And we pass it c. Keep in mind, this is c. The starting point is c. So let's just stay inside this function for now.

Def color equals calculate color iterations. So we're calculating the color based on the number of iterations it took to escape. And then fill the pixel with the color. That plots it on the screen. So let's look at the calculate iterations function. Int calculate iterations. This notation means that it will return an integer number.

It takes a complex number c. So z real, z imaginary, start at 0. Start there. And then iteration starts off at 0. So this is-- we're going to count how many iterations it takes. So while the circle contains z, that's another function that just tests if it's inside. And the number of iterations is less than max iterations. Max iterations is the number of iterations after which we assume it's just going to spiral inward and just stay inside.

So while that stuff is true, z equals z squared plus c. z equals z times z plus c, which [? is ?] [? squared. ?] So we can do that because I overloaded the operators, the multiplication and addition for a complex number. So it behaves correctly. It does this strange multiplication thing. And so we do that, iterations plus plus. That means we increment the number of iterations by one. And calculate color just calculates a color based on number of iterations.

So we have 20 minutes left. I want to blow you away with music and fractals. So I'm going to play three pieces of Bach. And yeah, they're just great. So the first one, it's actually not written by Bach. Oh no, what's going on here?

It's not written by Bach. But it's the "Little Harmonic Labyrinth," which is the song that the dialogue is based on in *Godel, Escher, Bach.* English language has a grammar. We could nest sentences inside, which we can nest more sentences, [? in which ?] we can [? mention ?] more sentences. And that itself was a nested sentence.

So music, sort of like English, has sort of a grammar to it. These chord progressions and melodies and harmonies particularly have a sort of a language in which they can move between keys and whatnot. So that's one way in which Bach embeds recursion inside of music. He has these nested harmonic structures of chord progressions. So the chord-- yeah, in chapter 5 of *Godel, Escher, Bach,* he talks about it. And the dialogue is modeled after that first song that we heard, "Little Harmonic Labyrinth."

And melodies also can sort of have recursion, in that they can have patterns. Like this theme, for example. This theme-- Bach does this a lot. What he does is he takes themes, and he restates them with maybe different harmonies and embeds them inside smaller and larger scales. So for example, I don't know if he did this, maybe on a larger scale, the actual

harmony might actually follow this sort of theme. And inside those-- for a long period of time, be this key, and a long period of time, be this key. And then when you're in that key, he plays this theme. So that's an example of nesting.

So recursion per se is not there, but the result of recursion. This is the distinction between recursive processes and recursive structures. Recursive structures are basically any structure that has nesting to it. So English language is a recursive structure because it comes from a recursive grammar. It's nesting. And music also is a recursive structure, not a recursive process.

AUDIENCE: So recursive process is almost like a function that defines how you can get the recursive structure.

CURRAN KELLEHER: Exactly. That's it. He said a recursive process is sort of like a function or something that defines how you end up with a recursive structure. That's exactly it. Yeah.

AUDIENCE: So [INAUDIBLE] just going like that, [INAUDIBLE].

CURRAN KELLEHER: Yeah, it's hard to hear. It takes a really fine musical ear to hear this. So we'll see-- how much time-- three minutes. So we talk about pushing and popping of stacks. So a function, a computer function, say F-- Foo, actually. Foo is a function. Or better yet-- oh, no, forget it. And inside this Foo, it calls a function bar.

And here's a function bar. It does some stuff, and it calls maybe g and does some other stuff. And here's g. g maybe does just one thing, and I don't know, ends. So what you have is a stack, a call stack. It happens in any grammar actually that's sort of recursive. So in computer programs and also in English, you have this notion of a stack.

So here, you do some stuff. You do some stuff. And then when you call bar, your focus actually changes to over here, bar. But you retain sort of in your mind or in the memory of a computer where you were, where are you left from. So you-- yeah.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: Calls?

AUDIENCE: [INAUDIBLE] goals.

CURRAN KELLEHER: Goals. Yeah, exactly. So that's why computer programming is tough, because you have a high level goal. But then to get to that goal, just like you said, you have other goals that you need to meet. And each of those goals requires some other goals. So it's actually recursive. It's crazy.

So when this program executes, it does this stuff. It puts this position, a, let's say, on the stack. So this is the stack, a. And it goes into here, bar, does all this stuff. And it calls g. But to remember where this is, let's call this position b. It puts b on the stack.

And it calls g, and then does all this stuff. And then after this, it returns. And when it returns, it asks, where was I the last time? And this is what the stack is for. So where was I the last time? I was at b. So we pop. Putting something on the stack is pushing. Taking it off is called popping.

We pop it and say, OK, here's b. b is where we were. We go back to there. And we finish this. Once we finish this, we say, oh, where we were the last time. So that's a. And then go back to a and finish.

AUDIENCE: [INAUDIBLE] make it so that you have those things, right? Those are things like [INAUDIBLE].

CURRAN KELLEHER: Yeah, sure. Yeah, exactly.

AUDIENCE: [INAUDIBLE].

CURRAN KELLEHER: Yeah. So the dialogue, which reflects "Little Harmonic Labyrinth," has this sort of structure. It goes inside something, inside yet another thing, and then back out, and then [? back out. ?]