[MUSIC-- "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

PROFESSOR: Well, there's one bit of mystery left, which I'd like to get rid of right now. And that's that we've been blithely doing things like cons assuming there's always another one. That we've been doing these things like car-ing and cdr-ing and assuming that we had some idea how this can be done. Now indeed we said that that's equivalent to having procedures. But that doesn't really solve the problem, because the procedure need all sorts of complicated mechanisms like environment structures and things like that to work. And those were ultimately made out of conses in the model that we had, so that really doesn't solve the problem.

Now the problem here is the glue the data structure's made out of. What kind of possible thing could it be? We've been showing you things like a machine, a computer that has a controller, and some registers, and maybe a stack. And we haven't said anything about, for example, larger memory. And I think that's what we have to worry about right now.

But just to make it perfectly clear that this is an inessential, purely implementational thing, I'd like to show you, for example, how you can do it all with the numbers. That's an easy one. Famous fellow by the name of Godel, a logician at the end of the 1930s, invented a very clever way of encoding the complicated expressions as numbers. For example-- I'm not saying exactly what Godel's scheme is, because he didn't use words like cons. He had other kinds of ways of combining to make expressions. But he said, I'm going to assign a number to every algebraic expression. And the way I'm going to manufacture these numbers is by combining the numbers of the parts.

So for example, what we were doing our world, we could say that if objects are represented by numbers, then cons of x and y could be represented by 2 to the x times 2 to the y. Because then we could extract the parts. We could say, for example, that then car of, say, x is the number of factors of 2 in x. And of course cdr is the same thing. It's the number of factors of 3 in x.

Now this is a perfectly reasonable scheme, except for the fact that the numbers rapidly get to be much larger in number of digits than the number of protons in the universe. So there's no easy way to use this scheme other than the theoretical one. On the other hand, there are other ways of representing these things. We have been thinking in terms of little boxes. We've been thinking about our cons structures as looking sort of like this. They're little pigeon holes with things in them. And of course we arrange them in little trees. I wish that the semiconductor manufacturers would supply me with something appropriate for this, but actually what they do supply me with is a linear memory.

Memory is sort of a big pile of pigeonholes, pigeonholes like this. Each of which can hold a certain sized object, a fixed size object. So, for example, a complicated list with 25 elements won't fit in one of these. However, each of these is indexed by an address. So the address might be zero here, one here, two here, three here, and so on. That we write these down as numbers is unimportant. What matters is that they're distinct as a way to get to the next one. And inside of each of these, we can stuff something into these pigeonholes. That's what memory is like, for those of you who haven't built a computer.

Now the problem is how are we going to impose on this type of structure, this nice tree structure. Well it's not very hard, and there have been numerous schemes involved in this. The most important one is to say, well assuming that the semiconductor manufacturer allows me to arrange my memory so that one of these pigeonholes is big enough to hold the address of another I haven't made. Now it actually has to be a little bit bigger because I have to also install or store some information as to a tag which describes the kind of thing that's there. And we'll see that in a second. And of course if the semiconductor manufacturer doesn't arrange it so I can do that, then of course I can, with some cleverness, arrange combinations of these to fit together in that way.

So we're going to have to imagine imposing this complicated tree structure on our nice linear memory. If we look at the first still store, we see a classic scheme for doing that. It's a standard way of representing Lisp structures in a linear memory. What we do is we divide this memory into two parts. An array called the cars, and an array called the cdrs. Now whether those happen to be sequential addresses or whatever, it's not important. That's somebody's implementation details. But there are two arrays here. Linear arrays indexed by sequential indices like this. What is stored in each of these pigeonholes is a typed object.

And what we have here are types which begin with letters like p, standing for a pair. Or n, standing for a number. Or e, standing for an empty list. The end of the list. And so if we wish to represent an object like this, the list beginning with 1, 2 and then having a 3 and a 4 as its second and third elements. A list containing a list as its first part and then two numbers as a second and third parts. Then of course we draw it sort of like this these days, in box-and-pointer notation. And you see, these are the three cells that have as their car pointer the object which is either 1, 2 or 3 or 4.

And then of course the 1, 2, the car of this entire structure, is itself a substructure which contains a sublist like that. What I'm about to do is put down places which are-- I'm going to assign indices. Like this 1, over here, represents the index of this cell. But that pointer that we see here is a reference to the pair of pigeonholes in the cars and the cdrs that are labeled by 1 in my linear memory down here.

So if I wish to impose this structure on my linear memory, what I do is I say, oh yes, why don't we drop this into cell 1? I pick one. There's 1. And that says that its car, I'm going to assign it to be a pair. It's a pair, which is in index 5. And the cdr, which is this one over here, is a pair which I'm going to stick into place 2. p2. And take a look at p2. Oh yes, well p2 is a thing whose car is the number 3, so as you see, an n3. And whose cdr, over here, is a pair, which lives in place 4. So that's what this p4 is. p4 is a number whose value is 4 in its car and whose cdr is an empty list right there. And that ends it.

So this is the traditional way of representing this kind of binary tree in a linear memory. Now the next question, of course, that we might want to worry about is just a little bit of implementation. That means that when I write procedures of the form assigned a, [UNINTELLIGIBLE] procedures-- lines of register machine code of the form assigned a, the car of [UNINTELLIGIBLE] b, what I really mean is addressing these elements. And so we're going to think of that as a abbreviation for it.

Now of course in order to write that down I'm going to introduce some sort of a structure called a vector. And we're going to have something which will reference a vector, just so we can write it down. Which takes the name of the vector, or the-- I don't think that name is the right word. Which takes the vector and the index, and I have to have a way of setting one of those with something called a vector set, I don't really care. But let's look, for example, at then that kind of implementation of car and cdr.

So for example if I happen to have a register b, which contains the type index of a pair, and therefore it is the pointer to a pair, then I could take the car of that and if I-- write this down-- I might put that in register a. What that really is is a representation of the assign to a, the value of vector reffing-- or array indexing, if you will-- or something, the cars object-- whatever that is-- with the index, b. And similarly for cdr. And we can do the same thing for assignment to data structures, if we need to do that sort of thing at all. It's not too hard to build that.

Well now the next question is how are we going to do allocation. And every so often I say I want a cons. Now conses don't grow on trees. Or maybe they should. But I have to have some way of getting the next one. I have to have some idea of if their memory is unused that I might want to allocate from. And there are many schemes for doing this. And the particular thing I'm showing you right now is not essential. However it's convenient and has been done many times. One scheme's was called the free list allocation scheme. What that means is that all of the free memory that there is in the world is linked together in a linked list, just like all the other stuff. And whenever you need a free cell to make a new cons, you grab the first, one make the free list be the cdr of it, and then allocate that.

And so what that looks like is something like this. Here we have the free list starting in 6. And what that is is a pointer-off to say 8. So what it says is, this one is free and the next one is an 8. This one is free and the next one is in 3, the next one that's free. That one's free and the next one is in 0. That one's free and the next one's in 15. Something like that. We can imagine having such a structure.

Given that we have something like that, then it's possible to just get one when you need it. And so a program for doing cons, this is what cons might turn into. To assign to a register A the result of cons-ing, a B onto C, the value in this containing B and the value containing C, what we have to do is get the current [? type ?] ahead of the freelist, make the free list be its cdr. Then we have to change the cars to be the thing we're making up to be in A to be the B, the thing in B. And we have to make change the cdrs of the thing that's in A to be C. And then what we have in A is the right new frob, whatever it is. The object that we want.

Now there's a little bit of a cheat here that I haven't told you about, which is somewhere around here I haven't set that I've the type of the thing that I'm cons-ing up to be a pair, and I ought to. So there should be some sort of bits here are being set, and I just haven't written that down. We could have arranged it, of course, for the free lift to be made out of pairs. And so then there's no problem with that. But that sort of-- again, an inessential detail in a way some particular programmer or architect or whatever might manufacture his machine or Lisp system.

So for example, just looking at this, to allocate given that I had already the structure that you saw before, supposing I wanted to allocate a new cell, which is going to be representation of list one, one, two, where already one two was the car of the list we were playing with before. Well that's not so hard. I stored that one and one, so p1 one is the representation of this. This is p5. That's going to be the cdr of this. Now we're going to pull something off the free list, but remember the free list started at six. The new free list after this allocation is eight, a free list beginning at eight. And of course in six now we have a number one, which is what we wanted, with its cdr being the pair starting in location five. And that's no big deal.

So the only problem really remaining here is, well, I don't have an infinitely large memory. If I do this for a little while, say, for example, supposing it takes me a microsecond to do a cons, and I have a million cons memory then I'm only going to run out in a second, and that's pretty bad. So what we do to prevent that disaster, that ecological disaster, talk about right after questions. Are there any questions? Yes.

AUDIENCE: In the environment diagrams that we were drawing we would use the body of procedures, and you would eventually wind up with things that were no longer useful in that structure. How is that represented?

PROFESSOR: There's two problems here. One you were asking is that material becomes useless. We'll talk about that in a second. That has to do with how to prevent ecological disasters. If I make a lot of garbage I have to somehow be able to clean up after myself. And we'll talk about that in a second. The other question you're asking is how you represent the environments, I think.

AUDIENCE: Yes.

PROFESSOR: OK. And the environment structures can be represented in arbitrary ways. There are lots of them. I mean, here I'm just telling you about list cells. Of course every real system has vectors of arbitrary length as well as the vectors of length, too, which represent list cells. And the environment structures that one uses in a professionally written Lisp system tend to be vectors which contain a number of elements approximately equal to the number of arguments-- a little bit more because you need certain glue. So remember, the environment [UNINTELLIGIBLE] frames. The frames are constructed by applying a procedure. In doing so, an allocation is made of a place which is the number of arguments long plus [? unglue ?] that gets linked into a chain. It's just like algol at that level. There any other questions? OK. Thank you, and let's take a short break.

[MUSIC-- "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

PROFESSOR: Well, as I just said, computer memories supplied by the semiconductor manufacturers are finite. And that's quite a pity. It might not always be that way. Just for a quick calculation, you can see that it's possible that if [? memory ?] prices keep going at the rate they're going that if you still took a microsecond second to do a cons, then-- first of all, everybody should know that there's about pi times ten to the seventh seconds in a year. And so that would be ten to the seventh plus ten to the sixth is ten to the thirteenth. So there's maybe ten to the fourteenth conses in the life of a machine. If there was ten to the fourteenth words of memory on your machine, you'd never run out.

And that's not completely unreasonable. Ten to the fourteenth is not a very large number. I don't think it is. But then again I like to play with astronomy. It's at least ten to the eighteenth centimeters between us and the nearest star. But the thing I'm about to worry about is, at least in the current economic state of affairs, ten to the fourteenth pieces of memory is expensive. And so I suppose what we have to do is make do with much smaller. Memories

Now in general we want to have an illusion of infinity. All we need to do is arrange it so that whenever you look, the thing is there. That's really an important idea. A person or a computer lives only a finite amount of time and can only take a finite number of looks at something. And so you really only need a finite amount of stuff. But you have to arrange it so no matter how much there is, how much you really claim there is, there's always enough stuff so that when you take a look, it's there. And so you only need a finite amount.

But let's see. One problem is, as was brought up, that there are possible ways that there is lots of stuff that we make that we don't need. And we could recycle the material out of which its made. An example is the fact that we're building environment structures, and we do so every time we call a procedure. We have built in it a environment frame. That environment frame doesn't necessarily have a very long lifetime. Its lifetime, meaning its usefulness, may exist only over the invocation of the procedure. Or if the procedure exports another procedure by returning it as a value and that procedure is defined inside of it, well then the lifetime of the frame of the outer procedure still is only the lifetime of the procedure which was exported.

And so ultimately, a lot of that is garbage. There are other ways of producing garbage as well. Users produce garbage. An example of user garbage is something like this. If we write a program to, for example, append two lists together, well one way to do it is to reverse the first list onto the empty list and reverse that onto the second list. Now that's not terribly bad way of doing it. And however, the intermediate result, which is the reversal of the first list as done by this program, is never going to be accessed ever again after it's copied back on to the second. It's an intermediate result. It's going to be hard to ever see how anybody would ever be able to access it. In fact, it will go away.

Now if we make a lot of garbage like that, and we should be allowed to, then there's got to be some way to reclaim that garbage. Well, what I'd like to tell you about now is a very clever technique whereby a Lisp system can prove a small theorem every so often on the [? forum, ?] the following piece of junk will never be accessed again. It can have no affect on the future of the computation. It's actually based on a very simple idea.

We've designed our computers to look sort of like this. There's some data path, which contains the registers. There are things like x, and env, and val, and so on. And there's one here called stack, some sort which points off to a structure somewhere, which is the stack. And we'll worry about that in a second. There's some finite controller, finite state machine controller. And there's some control signals that go this way and predicate results that come this way, not the interesting part.

There's some sort of structured memory, which I just told you how to make, which may contain a stack. I didn't tell you how to make things of arbitrary shape, only pairs. But in fact with what I've told you can simulate a stack by a big list. I don't plan to do that, it's not a nice way to do it. But we could have something like that. We have all sorts of little data structures in here that are hooked together in funny ways. They connect to other things. And so on. And ultimately things up there are pointers to these. The things that are in the registers are pointers off to the data structures that live in this Lisp structure memory.

Now the truth of the matter is that the entire consciousness of this machine is in these registers. There is no possible way that the machine, if done correctly, if built correctly, can access anything in this Lisp structure memory unless the thing in that Lisp structure memory is connected by a sequence of data structures to the registers. If it's accessible by legitimate data structure selectors from the pointers that are stored in these registers. Things like array references, perhaps. Or cons cell references, cars and cdrs.

But I can't just talk about a random place in this memory, because I can't get to it. These are being arbitrary names I'm not allowed to count, at least as I'm evaluating expressions. If that's the case then there's a very simple theorem to be proved. Which is, if I start with all lead pointers that are in all these registers and recursively chase out, marking all the places I can get to by selectors, then eventually I mark everything they can be gotten to. Anything which is not so marked is garbage and can be recycled. Very simple. Cannot affect the future of the computation.

So let me show you that in a particular example. Now that means I'm going to have to append to my description of the list structure a mark. And so here, for example, is a Lisp structured memory. And in this Lisp structured memory is a Lisp structure beginning in a place I'm going to call-- this is the root. Now it doesn't really have to have a root. It could be a bunch of them, like all the registers. But I could cleverly arrange it so all the registers, all the things that are in old registers are also at the right moment put into this root structure, and then we've got one pointer to it. I don't really care.

So the idea is we're going to cons up stuff until our free list is empty. We've run out of things. Now we're going to do this process of proving the theorem that a certain percentage of the memory has got crap in it. And then we're going to recycle that to grow new trees, a standard use of such garbage.

So in any case, what do we have here? Well we have some data structure which starts out over here one. And in fact it has a car in five, and its cdr is in two. And all the marks start out at zero. Well let's start marking, just to play this game. OK. So for example, since I can access one from the root I will mark that. Let me mark it. Bang. That's marked. Now since I have a five here I can go to five and see, well I'll mark that. Bang. That's useful stuff.

But five references as a number in its car, I'm not interested in marking numbers but its cdr is seven. So I can mark that. Bang. Seven is the empty list, the only thing that references, and it's got a number in its car. Not interesting. Well now let's go back here. I forgot about something. Two. See in other words, if I'm looking at cell one, cell one contains a two right over here. A reference to two. That means I should go mark two. Bang. Two contains a reference to four. It's got a number in its car, I'm not interested in that, so I'm going to go mark that. Four refers to seven through its car, and is empty in its cdr, but I've already marked that one so I don't have to mark it again. This is all the accessible structure from that place. Simple recursive mark algorithm.

Now there are some unhappinesses about that algorithm, and we can worry about that a second. But basically you'll see that all the things that have not been marked are places that are free, and I could recycle. So the next stage after that is going to be to scan through all of my memory, looking for things that are not marked. Every time I come across a marked thing I unmark it, and every time I come across an unmarked thing I'm going to link it together in my free list. Classic, very simple algorithm.

So let's see. Is that very simple? Yes it is. I'm not going to go through the code in any detail, but I just want to show you about how long it is. Let's look at the mark phase. Here's the first part of the mark phase. We pick up the root. We're going to use that as a recursive procedure call. We're going to sweep from there, after when we're done with marking. And then we're going to do a little couple of instructions that do this checking out on the marks and changing the marks and things like that, according to the algorithm I've just shown you. It comes out here. You have to mark the cars of things and you also have to be able to mark the cdrs of things. That's the entire mark phase.

I'll just tell you a little story about this. The old DEC PDP-6 computer, this was the way that the mark-sweep garbage collection, as it was, was written. The program was so small that with the data that it needed, with the registers that it needed to manipulate the memory, it fit into the fast registers of the machine, which were 16. The whole program. And you could execute instructions in the fast registers. So it's an extremely small program, and it could run very fast.

Now unfortunately, of course, this program, because the fact that it's recursive in the way that you do something first and then you do something after that, you have to work on the cars and then the cdrs, it requires auxiliary memory. So Lisp systems-- those requires a stack for marking. Lisp systems that are built this way have a limit to the depth of recursion you can have in data structures in either the car or the cdr, and that doesn't work very nicely.

On the other hand, you never notice it if it's big enough. And that's certainly been the case for most Maclisp, for example, which ran Macsyma where you could deal with expressions of thousands of elements long. These are algebraic expressions with thousand of terms. And there's no problem with that. Such, the garbage collector does work.

On the other hand, there's a very clever modification to this algorithm, which I will not describe, by Peter Deutsch and Schorr and Waite-- Herb Schorr from IBM and Waite, who I don't know. That algorithm allows you to build-- you do can do this without auxiliary memory, by remembering as you walk the data structures where you came from by reversing the pointers as you go down and crawling up the reverse pointers as you go up. It's a rather tricky algorithm. The first time you write it-- or in fact, the first three times you write it it has a terrible bug in it. And it's also rather slow, because it's complicated. It takes about six times as many memory references to do the sorts of things that we're talking about.

Well now once I've done this marking phase, and I get into a position where things look like this, let's look-- yes. Here we have the mark done, just as I did it. Now we have to perform the sweep phase. And I described to you what this sweep is like. I'm going to walk down from one end of memory or the other, I don't care where, scanning every cell that's in the memory. And as I scan these cells, I'm going to link them together, if they are free, into the free list. And if they're not free, I'm going to unmark them so the marks become zero.

And in fact what I get-- well the program is not very complicated. It looks sort of like this-- it's a little longer. Here's the first piece of it. This one's coming down from the top of memory. I don't want you to try to understand this at this point. It's rather simple. It's a very simple algorithm, but there's pieces of it that just sort of look like this. They're all sort of obvious. And after we've done the sweep, we get an answer that looks like that.

Now there are some disadvantages with mark-sweep algorithms of this sort. Serious ones. One important disadvantage is that your memories get larger and larger. As you say, address spaces get larger and larger, you're willing to represent more and more stuff, then it gets very costly to scan all of memory. What you'd really like to do is only scan useful stuff. It would even be better if you realized that some stuff was known to be good and useful, and you don't have to look at it more than once or twice. Or very rarely. Whereas other stuff that you're not so sure about, you can look at more detail every time you want to do this, want to garbage collect.

Well there are algorithms that are organized in this way. Let me tell you about a famous old algorithm which allows you only look at the part of memory which is known to be useful. And which happens to be the fastest known garbage collector algorithm. This is the Minsky-Feinchel-Yochelson garbage collector algorithm. It was invented by Minsky in 1961 or '60 or something, for the RLE PDP-1 Lisp, which had 4,096 words of list memory, and a drum. And the whole idea was to garbage collect this terrible memory.

What Minsky realized was the easiest way to do this is to scan the memory in the same sense, walking the good structure, copying it out into the drum, compacted. And then when we were done copying it all out, then you swap that back into your memory. Now whether or you not use a drum, or another piece of memory, or something like that isn't important. In fact, I don't think people use drums anymore for anything.

But this algorithm basically depends upon having about twice as much address space as you're actually using. And so what you have is some, initially, some mixture of useful data and garbage. So this is called fromspace. And this is a mixture of crud. Some of it's important and some of it isn't.

Now there's another place which is hopefully big enough, if we recall, tospace, which is where we're copying to. And what happens is-- and I'm not going to go through this detail. It's in our book quite explicitly. There's a root point where you start from. And the idea is that you start with the root. You copy the first thing you see, the first thing that the root points at, to the beginning of tospace. The first thing is a pair or something like, a data structure.

You then also leave behind a broken heart saying, I moved this object from here to here, giving the place where it moved to. This is called a broken heart because a friend of mine who implemented one of these in 1966 was a very romantic character and called it a broken heart.

But in any case, the next thing you do is now you have a new free pointer which is here, and you start scanning. You scan this data structure you just copied. And every time you encounter a pointer in it, you treat it as if it was the root pointer here. Oh, I'm sorry. The other thing you do is you now move the root pointer to there.

So now you scan this, and everything you see you treat as it were the root pointer. So if you see something, well it points up into there somewhere. Is it pointing at a thing which you've not copied yet? Is there a broken heart there? If there's a broken heart there and it's something you have copied, you've just replaced this pointer with the thing a broken heart points at. If this thing has not been copied, you copy it to the next place over here. Move your free pointer over here, and then leave a broken heart behind and scan.

And eventually when the scant pointer hits the free pointer, everything in memory has been copied. And then there's a whole bunch of empty space up here, which you could either make into a free list, if that's what you want to do. But generally you don't in this kind of system. In this system you sequentially allocate your memory. That is a very, very nice algorithm, and sort of the one we use in the scheme that you've been using. And it's expected-- I believe no one has found a faster algorithm than that.

There are very simple modifications to this algorithm invented by Henry Baker which allow one to run this algorithm in real time, meaning you don't have to stop to garbage collect. But you could interleave the consing that the machine does when its running with steps of the garbage collection process, so that the garbage collector's distributed, and the machine doesn't have to stop, and garbage collecting can start.

Of course in the case of machines with virtual memory where a lot of it is in inaccessible places, this becomes a very expensive process. And there have been numerous attempts to make this much better. There is a nice paper, for those of you who are interested, by Moon and other people which describes a modification to the incremental Minsky-Feinchel-Yochelson algorithm, and modification the Baker algorithm which is more efficient for virtual memory systems.

Well I think now the mystery to this is sort of gone. And I'd like to see if there are any questions. Yes.

AUDIENCE: I saw one of you run the garbage collector on the systems upstairs, and it seemed to me to run extremely fast. Did the whole thing take-- does it sweep through all of memory?

PROFESSOR: No. It swept through exactly what was needed to copy the useful structure. It's a copying collector. And it is very fast. On the whole, I suppose to copy-- in a Bobcat-- to copy, I think, a three megabyte thing or something is less than a second, real time. Really, these are very small programs. One thing you should realise is that garbage collectors have to be small. Not because they have to be fast, but because no one can debug a complicated garbage collector. A garbage collector, if it doesn't work, will trash your memory in such a way that you cannot figure out what the hell happened. You need an audit trail. Because it rearranges everything, and how do you know what happened there?

So this is the only kind of program that it really, seriously matters if you stare at it long enough so you believe that it works. And sort of prove it to yourself. So there's no way to debug it. And that takes it being small enough so you can hold it in your head. Garbage collectors are special in this way. So every reasonable garbage collector has gotten small, and generally small programs are fast. Yes.

AUDIENCE: Can you repeat the name of this technique once again?

PROFESSOR: That's the Minsky-Feinchel-Yochelson garbage collector.

AUDIENCE: You got that?

PROFESSOR: Minsky invented it in '61 for the RLE PDP-1. A version of it was developed and elaborated to be used in Multics Maclisp by Feinchel and Yochelson in somewhere around 1968 or '69.

OK. Let's take a break.

[MUSIC: "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]

PROFESSOR: Well we've come to the end of this subject, and we've already shown you a universal machine which is down to evaluator. It's down to the level of detail you could imagine you could make one. This is a particular implementation of Lisp, built on one of those scheme chips that was talked about yesterday, sitting over here. This is mostly interface to somebody's memory with a little bit of timing and other such stuff. But this fellow actually ran Lisp at a fairly reasonable rate, as interpretive. It ran Lisp as fast as a DEC PDP-10 back in 1979. And so it's gotten pretty hardware. Pretty concrete.

We've also downed you a bit with the things you can compute. But is it the case that there are things we can't compute? And so I'd like to end this with showing you some things that you'd like be able to compute that you can't. The answer is yes, there are things you can't compute.

For example, something you'd really like is-- if you're writing [UNINTELLIGIBLE], you'd like a program that would check that the thing you're going to do will work. Wouldn't that be nice? You'd like something that would catch infinite loops, for example, in programs that were written by users. But in general you can't write such a program that will read any program and determine whether or not it's an infinite loop.

Let me show you that. It's a little bit of a minor mathematics. Let's imagine that we just had a mathematical function before we start. And there is one, called s, which takes a procedure and its argument, a. And what s does is it determines whether or not it's safe to run p on a. And what I mean by that is this: it's true if p applied to a will converge to a value without an error. And it's false if p of a loops forever or makes an error.

Now that's surely a function. There is some for every procedure and for every argument you could give it that is either true or false that it converges without making an error. And you could make a giant table of them. But the question is, can you write a procedure that compute the values of this function? Well let's assume that we can.

Suppose that we have a procedure called "safe" that computes the value of s. Now I'm going to show you by several methods that you can't do this. The easiest one, or the first one, let's define a procedure called diag1. Given that we have safe, we can define diag1 to be the procedure of one argument, p, which has the following properties. If if it's safe to apply p to itself, then I wish to have an infinite loop. Otherwise I'm going to return 3. Remember it was 42. What's the answer to the big question? Where of course we know what an infinite loop is. Infinite loop, to be a procedure of no arguments, which is that nice lambda calculus loop. Lambda of x, x of x, applied to lambda of x, x of x. So there's nothing left to the imagination here.

Well let's see what the story is. I'm supposing it's the case that we worry about the procedure called diag1 applied to diag1. Well what could it possibly be? Well I don't know. We're going to substitute diag1 for p in the body here. Well is it safe to compute diag1 of diag1? I don't know. There are two possibilities. If it's safe to compute diag1 of diag1 that means it shouldn't loop. That means I go to here, but then I produce an infinite loop. So it can't be safe. But if it's not safe to compute diag1 of diag1 then the answer to this is 3. But that's diag1 of diag1, so it had to be safe.

So therefore by contradiction you cannot produce safe. For those of you who were boggled by that one I'm going to say it again, in a different way. Listen to one more alternative. Let's define diag2. These are named diag because of Cantor's diagonal argument. These are instances of a famous argument which was originally used by Cantor in the late part of the last century to prove that the real numbers were not countable, that there are too many real numbers to be counted by integers. That there are more points on a line, for example, than there are counting numbers. It may or may not be obvious, and I don't want to get into that now.

But diag2 is again a procedure of one argument p. It's almost the same as the previous one, which is, if it's safe to compute p on p, then I'm going to produce-- then I want to compute some other things other than p of p. Otherwise I'm going to put out false. Where other then it says, whatever p of p, I'm going to put out something else.

I can give you an example of a definition of other than which I think works. Let's see. Yes. Where other than be a procedure of one argument x which says, if its eq x to, say, quote a, then the answer is quote b. Otherwise it's quote a. That always produces something which is not what its argument is. That's all it is. That's all I wanted.

Well now let's consider this one, diag2 of diag2. Well look. This only does something dangerous, like calling p of p, if it's safe to do so. So if safe defined at all, if you can define such a procedure, safe, then this procedure is always defined and therefore safe on any inputs. So diag2 of diag2 must reduce to other than diag2 of diag2. And that doesn't make sense, so we have a contradiction, and therefore we can't define safe.

I just waned to do that twice, slightly differently, so you wouldn't feel that the first one was a trick. They may be both tricks, but they're at least slightly different.

So I suppose that pretty much wraps it up. I've just proved what we call the halting theorem, and I suppose with that we're going to halt. I hope you have a good time. Are there any questions? Yes.

AUDIENCE: What is the value of s of diag1?

PROFESSOR: Of what?

AUDIENCE: S of diag1. If you said s is a function and we can [INTERPOSING VOICES]

PROFESSOR: Oh, I don't know. I don't know. It's a function, but I don't know how to compute it. I can't do it. I'm just a machine, too. Right? There's no machine that in principle-- it might be that in that particular case you just asked, with some thinking I could figure it out. But in general I can't compute the value of s any better than any other machine can. There is such a function, it's just that no machine can be built to compute it.

Now there's a way of saying that that should not be surprising. Going through this-- I mean, I don't have time to do this here, but the number of functions is very large. If there's a certain number of answers possible and a certain number of inputs possible, then it's the number of answers raised to the number inputs is the number of possible functions. On one variable. Now that's always bigger than the thing you're raising to, the exponent. The number of functions is larger than the number of programs that one can write, by an infinity counting argument. And it's much larger. So there must be a lot of functions that can't be computed by programs.

AUDIENCE: A few moments ago you were talking about specifications and automatic generation of solutions. Do you see any steps between specifications and solutions?

PROFESSOR: Steps between. You mean, you're saying, how you go about constructing devices given that have specifications for the device? Sure.

AUDIENCE: There's a lot of software engineering that goes through specifications through many layers of design and then implementation.

PROFESSOR: Yes?

AUDIENCE: I was curious if you think that's realistic.

PROFESSOR: Well I think that some of it's realistic and some of it isn't. I mean, surely if I want to build an electrical filter and I have a rather interesting possibility. Supposing I want to build a thing that matches some power output to the radio transmitter, to some antenna. And I'm really out of this power-- it's output tube out here. And the problem is that they have different impedances. I want them to match the impedances. I also want to make a filter in there which is going to get rid of some harmonic radiation.

Well one old-fashioned technique for doing this is called image impedances, or something like that. And what you do is you say you have a basic module called an L-section. Looks like this. If I happen to connect this to some resistance, r, and if I make this impedance x, xl, and if it happens to be q times r, then this produces a low pass filter with a q square plus one impedance match. Just what I need. Because now I can take two of these, hook them together like this. OK, and I take another one and I'll hook them together like that. And I have two L-sections hooked together. And this will step the impedance down to one that I know, and this will step it up to one I know. Each of these is a low pass filter getting rid of some harmonics. It's good filter, it's called a pie-section filter. Great.

Except for the fact that in doing what I just did, I've made a terrible inefficiency in this system. I've made two coils where I should have made one. And the problem with most software engineering art is that there's no mechanism, other than peephole optimization and compilers, for getting rid of the redundant parts that are constructed when doing top down design. It's even worse, there are lots of very important structures that you can't construct at all this way.

So I think that the standard top down design is a rather shallow business. Doesn't really capture what people want to do in design. I'll give you another electrical example. Electrical examples are so much clearer than computational examples, because computation examples require a certain degree of complexity to explain them. But one of my favorite examples in the electrical world is how would I ever come up with the output stage of this inter-stage connection in an IF amplifier. It's a little transistor here, and let's see. Well I'm going to have a tank, and I'm going to hook this up to, say, I'm going to link-couple that to the input of the next stage.

Here's a perfectly plausible plan-- well except for the fact that since I put that going up I should make that going that way. Here's a perfectly plausible plan for a-- no I shouldn't. I'm dumb. Excuse me. Doesn't matter. The point is [UNINTELLIGIBLE] plan for a couple [UNINTELLIGIBLE] stages together. Now what the problem is is what's this hierarchically? It's not one thing. Hierarchically it doesn't make any sense at all. It's the inductance of a tuned circuit, it's the primary of a transformer, and it's also the DC path by which bias conditions get to the collector of that transistor. And there's no simple top-down design that's going to produce a structure like that with so many overlapping uses for a particular thing.

Playing Scrabble, where you have to do triple word scores, or whatever, is not so easy in top-down design strategy. Yet most of real engineering is based on getting the most oomph for effort. And that's what you're seeing here. Yeah?

AUDIENCE: Is this the last question?

[LAUGHTER]

PROFESSOR: Apparently so. Thank you.

[APPLAUSE]

[MUSIC-- "JESU, JOY OF MAN'S DESIRING" BY JOHANN SEBASTIAN BACH]