Gregory C. Chaitin

Professor Gregory Chaitin is at the IBM Watson Research Center in New York. In the mid 1960s, when he was a teenager, he created algorithmic information theory, which combines, among other elements, Shannon's information theory and Turing's theory of computability. In the three decades since then he has been the principal architect of the theory. Among his contributions are the definition of a random sequence via algorithmic incompressibility, and his information-theoretic approach to Gödel's incompleteness theorem. His work on Hilbert's 10th problem has shown that in a sense there is randomness in arithmetic, in other words, that God not only plays dice in quantum mechanics and nonlinear dynamics, but even in elementary number theory. His latest book is *The Limits of Mathematics* (Springer-Verlag).

Elegant Lisp Programs[1]

Call a program "elegant" if no smaller program has the same output. I.e., a LISP S-expression is defined to be elegant if no smaller S-expression has the same value. For any computational task there is at least one elegant program, perhaps more. Nevertheless, we present a Berry paradox proof that it is impossible to prove that any particular large program is elegant. The proof is carried out using a version of LISP designed especially for this purpose. This establishes an extremely concrete and fundamental limitation on the power of formal mathematical reasoning.

[1] Lecture given at DMTCS'96 at 9 am, Thursday 12 December 1996 in Auckland, New Zealand. The lecture was videotaped; this is an edited transcript.

Introduction

Good morning, everyone! I'd like to talk about an old subject and give it a new twist. The subject I want to talk about is from the 1930's. It's Gödel and Turing's incompleteness results in their two famous papers from 1931 and 1936. I want to throw two new things into the stew. I'm going to use an approach more like Turing's than like Gödel's. So algorithm is very important the way I'll do it. But I'm going to throw in a new thing, which is program size—I'm going to look at the size of computer programs. And the other thing is that I'm not going to use Turing machines or lambda calculus or recursive function theory or fixed point theorems. I want to actually write out programs and run them on computers using current techniques that are used in the industry. You know, good software, 1996 vintage. So the idea is to look at some very old ideas from the 1930's revisiting them using the best software technology that we have available now, at this moment. So it's a mixture of extremely philosophical stuff that probably no mathematicians are interested in, because it deals with the limits of mathematics—and on the other hand I want to make it as practical as possible because I want to tell you about actually getting your hands dirty programming this, and getting it to run efficiently and fast on our current computers with current software.

To give you another hint of the difference in viewpoint, Gödel's approach to incompleteness is "This statement is false!" And instead I use an approach based on the Berry paradox, which is "the first positive integer that you can't name in a billion words". Or even better, "the first positive integer that this statement is too small to name". So there is no self-reference. Actually there is, but it's a much weaker kind of self-reference than Gödel needs to use. Also, I'll be using LISP as my programming language, versions of LISP that I have to invent. And a nice thing about LISP is that I can invent a LISP and program it on a new computer in a new programming language in about a week. So it's easy, it's small enough, it's less than a thousand lines of code to do a LISP. So I think that's a nice approach. So you won't see any fixed point theorems, you won't see any recursive function theory or lambda calculus—I want to actually run programs efficiently.

And I won't need to have a statement say of itself that it's false. For a statement to refer to itself you need to use some cleverness, right? I don't need to use cleverness. I only need to have a statement know how big it is. That's the only self-reference I need, in order for it to achieve something that it's too small to achieve. For a statement to know its own size is easy. To put a statement within itself is impossible, right? It doesn't fit! But just to put the size of a program or statement within itself is easy, because that's going to be about $\log N$, it's going to be very small compared to the object. So it's very easy to have an object know its own size and it takes much more cleverness to have an object know itself completely. So this is a rather different viewpoint, it's a much easier self-reference than Gödel's.

Why I Love (Pure) LISP

But let me start telling you why I think LISP should be loved by mathematicians. I think it's the only computer programming language that is mathematically respectable, because it's the only one that I can prove theorems about!

LISP

So why do I love LISP?! Well, the answer is, because it's really set theory, and all mathematicians love set theory!

Set Theory

LISP is just a set theory for computable mathematics rather than for abstract mathematics. Of course in set theory the basic object would be a list, say, of three objects

$$\{A, B, C\}$$

And as a joke one way to explain LISP is to say, well, take all the curly braces and make them into parentheses and take the commas out and make them blanks!

$$(A\ B\ C)$$

Syntactically that will show you what LISP is like. LISP objects are these parenthesized expressions which can be nested to arbitrary depth.

$$(A\ (B\ C)\ 123)$$

And objects are just separated by blanks. So this

$$(A\ (B\ C)\ 123)$$

is a list with three elements, first A, second $(B\ C)$, and third 123. The second element $(B\ C)$ is in turn a list with two elements. So these are just sets of sets. The only difference between a list and a set is that there is a first, a second, a third element, and elements can be repeated. But otherwise this is just a computerized version of set theory. Just like in set theory where you create everything out of sets—in fact, if you're an extremist, you create everything out of the empty set—in LISP this

$$(A\ (B\ C)\ 123)$$

is everything. This is your universal substance. This is the wood out of which you build the world! And it's simultaneously data and programs, they are these objects, which are called symbolic expressions (or S-expressions). And the things you can put inside S-expressions are words or numbers.

Also LISP is very mathematical in that you don't think about time, and you don't think about executing a program and that it does things that change the state of the world. What you think of in LISP is you think a program is

an expression and you evaluate the expression giving you a value. But nothing happens! You don't think of time and you don't think of values being assigned to variables, you don't think of goto's. Instead in a LISP expression you define functions, you apply the functions to values, and the final thing you get is a final value. So it's very much a mathematical notion, of expressions giving values.

Now let me give you an example of a LISP program. I can't give you a complete course on LISP—if I had an hour I could! Of a reduced LISP that I've invented. So let me give you an example. Let's take factorial, which is a typical LISP function. By the way, for experts, my LISP is Scheme-ish, it's a LISP that looks a lot like Scheme, but I had to make some changes. So let's define factorial of N.

```
define (fact N)
```

And we're going to say that if N is equal to 1, then it's going to be 1.

```
if = N 1  1
```

Otherwise it's going to be N times factorial of N minus 1.

```
* N (fact - N 1)
```

So the final result, if we put in all the parentheses, is this.

```
(define (fact N)
(if (= N 1)  1
            (* N (fact (- N 1)))))
```

So we're just using Polish prefix notation. For example, this

```
(- N 1)
```

is N minus 1. Okay, so this program defines factorial. We're going to call it `fact` of variable N. If N is equal to 1, then factorial of N is 1. Otherwise it's the product of N times factorial of N minus 1. And then to use it to get 3 factorial, you write

```
(fact 3)
```

and this gives 6 after the definition of factorial has been processed.

```
(fact 3) ---> 6
```

Now actually I don't like to write all these parentheses. LISP programmers haven't heard about parenthesis-free Polish notation! So I just write

```
define (fact N)
if = N 1  1
          * N (fact - N 1)
```

The other parentheses are understood.

Also in theory what we just did was bad, because we defined a function in one S-expression and then we use it in another S-expression. These are two separate S-expressions. That means that the first S-expression is having an effect, it's leaving a definition. You don't want to do that in LISP. In theoretical LISP, an expression has to define within itself all the functions that it needs, and then use them. Because there is no lasting effect of evaluating a LISP expression. But you can define a function locally and then use it. In fact, the only way to get a value to be assigned to a variable is to have it be the argument of a function which binds the value to the variable within the appropriate scope. Anyway, I don't want to get into all the details. Let me just show you the correct one-expression version of our factorial example. It uses let-be-in, which is a three-argument function.

```
let (fact N) if = N 1 1 * N (fact - N 1)
(fact 3)
```

This expands to

```
('lambda (fact) (fact 3)
 'lambda (N) if = N 1 1 * N (fact - N 1)
)
```

or

```
((' (lambda (fact) (fact 3)) )
 (' (lambda (N) (if (= N 1) 1 (* N (fact (- N 1))))) )
)
```

whose value is 6. Here ' is the one-argument quote function, meaning no evaluation occurs, and triples of the form

```
(lambda (arguments) body)
```

are function definitions.

And if I add that `car` gives you the first element of a list, that `cdr` gives you the rest of the list, and that `cons` puts them back together, then you know essentially all of LISP! Oh, I forgot to say that `nil` is another name for the empty list `()`. And there's a way to test if something has elements or not, that's `atom`.

Okay, that's all the time I can devote to LISP. But I think that you can see from these examples that LISP is very pretty, it's very elegant, it's very mathematical, and it's not at all like a normal programming language.

Proving LISP Programs Are Elegant

Now let me give you an incompleteness result. I think it's a very dramatic incompleteness result, if you like LISP, that you can do with LISP. I emphasized that LISP programs are expressions. Now let's define an elegant LISP expression

Elegant LISP Expression

to be a LISP expression with the property that no smaller expression has the same value. So now we're looking at the size of LISP expressions. LISP expressions are written out in characters, and you just take some standard format for writing them out with blanks in the right places, and you ask, "How big is it?" You measure the size in characters. You count the blanks too, you have some standard format, and this gives a natural way to define the size of a LISP expression. And I'll say that a LISP expression is elegant if no smaller expression gives the same value that it does. Okay? By the way, the value of a LISP expression is also a LISP expression. Everything is an S-expression in this world.

So clearly for any LISP object there is a most elegant expression that gives it as its value, and there may even be several. But what if you want to **prove** that the LISP expression that you've got is elegant, that no smaller expression has the same value? Well, the surprising answer is that **you can't prove that!**

Now I'm going to prove this incompleteness result. I'll start with a hand-waving proof, and then I'll tell you the trouble you get into if you try to program out the proof.

The hand-waving proof goes like this. You start like the incompleteness result in Turing's original paper in 1936. You say, let's assume you have a set of axioms and a set of rules of inference which are so formal, so well specified, that there's an algorithm to check if a proof is valid. Then you run through all possible proofs in size order, check which ones are correct, and you get one by one all the theorems—they're in order of the size of the proofs. So you're given this formal axiomatic system

$$\text{FAS}$$

and you start running through all possible proofs and getting all the theorems.

$$\text{FAS} \longrightarrow \text{Theorems}$$

And I'll simplify the formal axiomatic system because I'm only interested in theorems which give elegant LISP expressions, where you prove that a particular S-expression is elegant. So I'll think of a formal axiomatic system as a computation which starts running and every now and then it throws out a LISP expression that it claims it's demonstrated is elegant. So it's just a black box that every now and then outputs an expression that's elegant, that it's demonstrated is elegant. Okay?

So you start doing this and you just keep going until you find a LISP expression that's elegant, that you've proved is elegant, but it's much more complicated than the formal axiomatic system. And then I'll show you that you get into trouble, you get a contradiction.

Oh, I forgot to say that at this point the formal axiomatic system is in the form of a LISP expression.

$$\text{FAS (S-expression)} \longrightarrow \text{Theorems}$$

I hadn't told you this before. LISP is a nice language for doing things like formal axiomatic systems, because it's a symbolic language, but I'll have to explain in more detail later how this works.

Now let me start over and be more precise. The way this proof goes, is you're going to have a large LISP expression which somewhere in it is going to have the formal axiomatic system, contained within it.

((formal axiomatic system))

The proof of this incompleteness result consists of exhibiting this large LISP expression which in fact is going to be exactly 410 characters of LISP bigger than the formal axiomatic system that it contains. So you put the formal axiomatic system that you want to show has limitations in the right place in this big expression. And there are 410 additional characters of LISP programming that I wrap around the formal axiomatic system. What are these 410 additional characters for?

(410 characters (formal axiomatic system))

What this large LISP expression does, is it starts running the formal axiomatic system, getting the theorems that it produces, which are elegant LISP expressions, until it finds an elegant LISP expression that is larger than it is. How does this large LISP expression know its own size? It gets its own size by adding 410 to the size of the formal axiomatic system that it was given; 410 is just a constant embedded in the large expression. So the large LISP expression

(410 characters (formal axiomatic system))

takes the formal axiomatic system and determines its size (we provide a built-in function for doing that), adds 410 characters to that, which happens to be the number of characters in the wrapping for the formal axiomatic system, and at that point this expression

(410 characters (formal axiomatic system))

knows its own size exactly. Then it starts running the formal axiomatic system looking for the first elegant LISP expression that is larger than it is. Once it finds this elegant LISP expression, it runs it to get the value of the elegant expression, and then it returns this value as its own final value. So the value of this big LISP expression

(410 characters (formal axiomatic system))

is the same as the value of an elegant LISP expression which is larger than it is. But that's impossible! This contradicts the definition of elegance, because this

(410 characters (formal axiomatic system))

large LISP expression is at least one character too small to produce that value.

In other words, we have a LISP expression which is 410 characters larger than the formal axiomatic system that it contains. It's given a formal axiomatic system, it measures its size and adds 410 to that, which happens to be the

right way to calculate the exact size of the entire LISP expression. Then it starts running the formal axiomatic system searching for a proof that some LISP expression is elegant that's larger than this

(410 characters (formal axiomatic system))

whole thing is. And once it finds this elegant LISP expression, it runs it, and produces as value of this

(410 characters (formal axiomatic system))

expression the value of that elegant LISP expression. But this

(410 characters (formal axiomatic system))

is too small an expression to produce that value! That's the whole point! So either the formal axiomatic system was lying, and produced a false theorem, or in fact this

(410 characters (formal axiomatic system))

won't work, because it will never find the elegant LISP expression that it's searching for, it will never find an elegant LISP expression larger than it is, it will never find an elegant LISP expression that's more than 410 characters bigger than the formal axiomatic system that it's using.

So we've gotten an upper bound on the size of provably elegant LISP expressions. The upper bound is this: A formal axiomatic system whose LISP complexity is N cannot prove that a LISP expression is elegant if the expression's size is greater than $N + 410$. So at most finitely many LISP expressions can be shown to be elegant.

That's great, but I should emphasize that this overview of the proof sweeps a lot of programming problems under the rug! To get the 410 characters of LISP that I need to make the above proof work, I have to add some things to normal LISP. So now I'm going to tell you about these programming problems.

What We Have to Add to LISP

As I said, there are some problems programming all this. Normal LISP really isn't good enough. But any other programming language would be even worse! I had a version of this proof in 1970, and in words you can explain the idea; it's very simple. But let's say you want to actually program this out and run it on a computer, on an example, and check that it works. Well the answer is, no existing programming language is really adequate for the task. And I really want to run this on a computer. I'm a computer programmer, I earned a living as a computer programmer for many years! I think LISP is almost the right language. But it's still not quite right. So I had to take the heart of a normal LISP, pure LISP, LISP with no side-effects, and add a few things to it, to make things work.

The main thing that I added to LISP is this. Normal LISP is based on a function called `eval`.

```
eval
```

`Eval` is the LISP universal Turing machine, it's the LISP interpreter. LISP is not a compiled language, it's an interpreted language. So the LISP interpreter is always present while a LISP program is running. And since you have the interpreter there all the time, a LISP program can create a LISP program and then immediately run it. In a normal programming language, you have to compile a program before you can run it. But in LISP it works seamlessly, you just use `eval`.

So the LISP universal Turing machine is called `eval`, and it's built in, it's a primitive function that's provided for free. You could program it out in LISP, just like Turing programmed out his universal Turing machine. But in fact you're just given this

```
eval
```

as a built-in function. Unfortunately, this is not the right built-in function for my incompleteness proof. I need a time-limited `eval` that I call `try`.

```
try
```

Recall that in LISP notation

```
(f x y)
```

means just what

$$f(x, y)$$

means in normal mathematical notation, it's the function f applied to the arguments x and y. Now let me explain what `try` does. Here's how you use `try`. You give it a time limit and a LISP expression.

```
(try time-limit lisp-expression)
```

It's a way to try to evaluate the given expression for a limited amount of time, just in case the LISP expression goes on forever and never returns a final value.

Why do I need this? Well, in my proof I've got a formal axiomatic system, and it goes on forever producing theorems, it never stops. So `eval` would be no good. If somebody gives you a formal axiomatic system and you run it using `eval`, you never get anything back, it just goes on forever. So what I need is a time-limited `eval`, a way to run the formal axiomatic system for a certain amount of time and see which theorems show up before the time runs out. And then I'm going to loop and run the formal axiomatic system for more and more time, until I find the theorem that I'm looking for.

If you read the source code for a LISP interpreter, in a low-level language like C or, God forbid, machine language, well, it's just `eval`, that's all the interpreter is. `Eval` is constantly calling itself recursively. My interpreter isn't based on `eval`, it's based on `try` instead. `Try` plays the same role that `eval` does in a normal LISP interpreter.

So we have a formal axiomatic system, and we try running it like this

```
(try time-limit formal-axiomatic-system)
```

and then we gradually increase the time limit while we examine the theorems produced by the formal axiomatic system. How does `try` give us the information that we need to do this? More generally, what is the value of the following try?

```
(try time-limit lisp-expression)
```

`Try` always returns a value, it never gets stuck in an infinite loop. In fact, `try` always returns a triple of the following form

```
(success/failure value/out-of-time captured-intermediate-results)
```

If we're trying a formal axiomatic system, this triple will be

```
(failure out-of-time theorems)
```

`Success` means that the try was a success because the evaluation completed. `Failure` means that the evaluation did not complete. If the try was a success, then the second element will be the value of the LISP expression that was being evaluated. If not, it will indicate here that the evaluation ran out of time. And the third element will always be a list containing all the output, all the intermediate results, produced during the evaluation. In the case of a formal axiomatic system this will be a list of theorems. In fact, with the formal axiomatic systems that we considered before, it will be a list of elegant LISP expressions.

So `try` provides a way of handling infinite computations that output intermediate results instead of having a final value. It's the way that I deal with formal axiomatic systems in LISP. `Try` captures all the intermediate results, it gives us all the theorems. And the 410-character wrapping in my proof that you can't prove that large S-expressions are elegant uses `try` to run the formal axiomatic system for longer and longer amounts of time, until it finds an elegant LISP expression that's bigger than the formal axiomatic system and its wrapping. If such an elegant LISP expression is found, then it uses `eval`, which is just a `try` with no time bound, to get the value of the elegant LISP expression. That's the final value that

(410 characters (formal axiomatic system))

returns, and that's how we get the contradiction that proves my incompleteness theorem!

Okay, this is straightforward. It's a simple proof. The idea is simple, but it took me a quarter of a century to do the programming! It wasn't easy to come up with the LISP expression that proves this incompleteness result. But now that I've done the work, we're rewarded with a very sharp incompleteness result. To prove that an N-character LISP expression is elegant you need a formal axiomatic system whose LISP complexity is at least $N - 410$. Before, all we had here was $N - c$, not $N - 410$, and we had no idea how big c might be.

Discussion

So this is fairly straightforward, it's a very simple proof! And I have two claims about this piece of work that I want to discuss with you. First of all, I claim that **this is a very fundamental incompleteness result!** Secondly, I'm going to try to convince you that **LISP is beautiful!**

Why is this a very fundamental incompleteness result? The game in incompleteness results is to try to state the most natural problem, and then show that you can't do it, to shock people! You want to shock people as much as possible! Now there are many ways to shock people—this is the best that I can do. What's so shocking here? Well, the notion of an elegant LISP expression is very straightforward. There are lots of them out there! An infinity of them! But you can only prove that finitely many LISP expressions are elegant, unless you change the rules of the game by changing the formal axiomatic system. And you can't prove that a LISP expression is elegant if it's more than 410 characters bigger than the LISP implementation of the axioms and rules of inference that you're using to prove that LISP expressions are elegant.

I hope that computer scientists will find this shocking. After all, LISP expressions are very natural objects. Although the notion of an elegant LISP expression has no practical significance, it's not too farfetched, and it has a straightforward mathematical definition.

Of course, computer programmers don't usually want an elegant program, they want a program that works, that they can get running as fast as possible. Their boss wants the programs to be understandable in case a programmer quits and gets a better job elsewhere. And sometimes elegant programs are cryptic and hard to understand. Nevertheless, as a sport programmers sometimes try to outdo each other in the compact cleverness of their programs. The notion of elegance is not entirely foreign to the spirit, to the ethos of computer programming, to computer programming as a sport or as an art. Even though this may not be the way that a company that pays programmers wants them to do things!

And another good thing about this incompleteness result is that it's easy enough to understand what it is that you can't do, that it may be interesting that you can't do it!

Now another thing that's interesting about this is that in 1970 I had essentially this proof. In words you can explain it very easily. The novelty here is that I've taken the trouble to actually program this out on a real computer in a real computer programming language. And in spite of the fact that the ideas are simple, one could never really do this before. So the other message that I have for you theoreticians is that LISP, or some version of it, because I had to invent one, LISP really is beautiful from a mathematical point of view. I view LISP as the set theory of computational mathematics. And if I were in a university, which I'm not, and I wanted to give a first course in theoretical computer science, this theorem about elegant LISP expressions would be the very first thing that I would give the students. In fact, I would assume that the students knew no computer programming at all. I would give them LISP, a toy version, an elegant version, the heart of LISP, as their first programming language. And

then I would hit them over the head with this incompleteness result! That would be my approach. Yes?

Question. Do you think you'd get tenure?!

Answer. No! You see why I'm not at a university!

Okay, unfortunately normal LISP wasn't quite good enough. I admit though that I made some changes just for the fun of it, because it's so easy to do a LISP, that every time anyone does a LISP, they always "roll their own" version. That's the problem with creativity! You can't stop it! That's why I've made many, many different LISP dialects. But in addition to the changes that I made mostly for the fun of it, I did have to invent a way for LISP expressions to produce an infinite amount of output. And I had to change `eval` into `try` in order to be able to carry out my incompleteness proof elegantly.

Now you may object that there was no reason to add `try`, and that I could have defined `try` in LISP, without adding any new primitive functions to LISP. Well, that's true, but it would be a gruesome piece of work. Though LISP programmers do love to show that you can do LISP in LISP, that you can program `eval` as a LISP function, that LISP is powerful enough to easily express its own semantics. But why should one program `eval` in LISP, when the interpreter itself is `eval`?! You're doing the same work twice! So I don't think that it is cheating to provide `eval/try` as a primitive function. It's not substantially more work to do my LISP built around `try` than to do a normal LISP built around `eval`, and this makes my proof run much, much faster than if `try` were programmed in LISP. So partly I do this for programming convenience, partly for execution speed, and partly because I'm trying to understand what are the right primitive functions, the right fundamental notions, for doing metamathematics.

By the way, let me mention that I originally wrote my LISP in Mathematica, because it's the most powerful programming language I know. The LISP interpreter is about three-hundred lines of Mathematica code. Then I redid it in C, and it's a thousand lines of C, and the program is incomprehensible, which means that I'm a good C programmer! The C version of the interpreter runs a hundred times faster than the Mathematica version, but the program is completely incomprehensible. You can find this software and my course on the limits of mathematics, from which my result on elegant LISP expressions is taken, you can find all this in my web site at

`http://www.cs.auckland.ac.nz/CDMTCS/chaitin`

In fact, it's a course that I gave at Rovaniemi, Finland, and the precise URL is

`http://www.cs.auckland.ac.nz/CDMTCS/chaitin/rov.html`

You can also find some of this material in my article in *J.UCS,* Vol. 2, No. 5.

Okay, so this is my fantasy for a first course on theoretical computer science, and I think that this could even work with bright high school students. There's very little in it that's technical, and the toy LISP is easier to learn than a real LISP. My goal has always been to teach quantum mechanics, general relativity and Gödel's incompleteness theorem to bright high school students! So obviously,

the way I would teach such extremely bright youngsters programming would be with LISP—that's for theory. To actually get work done on the computer, I would teach them Mathematica, which is what high school students are learning in Rovaniemi!

Algorithmic Information Theory

Okay, I've used up half my time, but this discussion of elegant LISP expressions was actually just a warm-up exercise! This is not really the incompleteness result that I'm most proud of. Let me tell you what else you have to do starting in this spirit to get to my best incompleteness result. I'll outline the rest of my Rovaniemi course.

First I should tell you that I'm not going to use LISP expression size as my program-size complexity measure. That was simple and easy to understand, but it's not the right complexity measure for doing algorithmic information theory! So the first thing I want to do is define a new universal Turing machine.

UTM

Instead of taking in LISP expressions and putting out LISP expressions, which we did before, now the input program will be a bit string, and the output will be a LISP S-expression.

UTM: Bit String $\longrightarrow$ LISP Expression

So this is the computer that we're going to use to measure the size of programs. How does this computer work? Well, the program will be a long bit string, which the universal Turing machine will read from left to right.

Bit String—

The beginning of the program is going to be a LISP expression in binary, and I use eight bits for each character.

Bit String—LISP Expression (8 bits/char)

So every program for my universal Turing machine starts with a LISP expression telling you the other Turing machine to simulate, that's the idea. And LISP is a good language for expressing algorithms.

So to make a program, you take a LISP expression, and then you convert it into a list of 0's and 1's, and I have a primitive function in my LISP for doing that, and you end up with a very long bit string. It will have eight bits for each character. And I also provide a function to measure the size of a LISP expression, and another for determining the length of a list. In fact, I provide primitive functions for all the right things so that my proofs will be easy to do!

Okay, so the universal Turing machine starts off by reading in a LISP expression in binary, eight bits per character, and how does it know where the

LISP expression ends? Well, I'll just put a special character next to serve as endmarker, that'll be the next eight bits of the program. In UNIX there are a number of characters that people use to indicate the ends of things, and I've picked one of them, the newline character `\n`. In fact, my primitive function for converting a LISP expression into a bit string automatically supplies this special eight-bit pattern at the end.

Bit String—LISP Expression, NL

So there's this special character and when you get to it you know that you've finished reading the LISP expression.

After it finishes reading the LISP expression at the beginning of the bit string, the universal Turing machine starts running it, it starts evaluating the LISP expression. Think of this as the program, and as data we're going to give it the binary program for the Turing machine that's being simulated.

Bit String—LISP Expression, NL, Data

How does the LISP expression that's being evaluated get access to its binary data? Access is tightly controlled. Basically, the only way to access the data is by using a primitive function (with no arguments) that returns the next bit of the binary data. And it's very, very important that this primitive function can only return a 0 or a 1, but it cannot return an end of file indication. If you run out of data, the program aborts! This forces the program to be self-delimiting, which means that it has to indicate within itself how far out it goes. In other words, the initial LISP expression has to decide by itself how much data to read. For example, one simple convention is to double each bit of the binary data and then use a pair of unequal bits as an endmarker. But there are much more clever schemes for packaging binary data.

And if the LISP expression doesn't abort because it requested data that wasn't there, then the final value that it returns will be the final output produced by my new universal Turing machine. There may also be additional, intermediate output, which will also be LISP S-expressions. (The intermediate output is produced by a special primitive function that's an identity function, but which has the side-effect of outputting its argument.)

Why do we have to give binary data to the LISP expression? It's because the bits in a LISP S-expression are redundant due to LISP syntax restrictions. So you have to add to the LISP expression, which is a powerful way to express algorithms, raw binary data "on the side." There you have maximum flexibility, there you can really take advantage of each bit. And the initial LISP expression is going to determine the scheme that's used for reading in the raw binary data.

To convince you of the power of this self-delimiting scheme, let me show you how easy it is to use subroutines—you just concatenate them! The fancy way of saying this is that because programs are self-delimiting, algorithmic information content is subadditive. What does this mean? Well, let's say that you're given two programs to calculate two separate S-expressions. Then it's easy to combine

these programs and get a program to calculate the pair of S-expressions. In fact, to do this you only need to prepend a particular 432-bit prefix.

Let me restate this. I like to use $H(.)$ for the size in bits of the smallest program to calculate something. $H(X)$ is the algorithmic information content or complexity of the S-expression X. Then the following basic inequality states that the complexity of a pair of S-expressions is bounded by a constant plus the sum of the individual complexities:

$$H((X\ Y)) \leq H(X) + H(Y) + 432.$$

This inequality states that algorithmic information is (sub)additive, that we can combine subroutines. How does it work? Well, there's a 432-bit prefix, which consists of a 53-character LISP S-expression and a `\n`. This 53-character expression reads in a LISP S-expression (there's a primitive function for doing that) and runs it (another primitive) to get X. Then it reads in a second S-expression and runs it to get Y. And then it returns the pair $(X\ Y)$. That's 53 characters of code in my LISP. That's the general idea; I don't want to go into the details.

By the way, this inequality

$$H((X\ Y)) \leq H(X) + H(Y) + c$$

has been around for a long time, in fact ever since I redid algorithmic information theory using self-delimiting programs in the mid 1970's. This inequality appears in a lot of papers (with $H(X,Y)$ instead of $H((X\ Y))$), but we never knew how big c could be. It depended on the choice of universal Turing machine. Well, now I've picked a particular universal Turing machine, and c is equal to 432! I think it's very interesting to get a specific value for c.

Algorithmic information theory is a theory of the size of computer programs, but up to now you've never been able to actually run these computer programs. I don't like that! So my new version of algorithmic information theory is very concrete and down to earth. You have to learn LISP programming, but you get a theory about the size of **real computer programs,** programs that you can actually run and use to get results. And after you put together a program and test it, you just look at its size and that gives you an upper bound on the program-size complexity of something! It's that easy! I think that this makes my theory much more concrete and much more understandable. And as a result of this, **I** certainly understand program-size complexity better. So hopefully this approach will also help other people!

Okay, in summary, we've got this universal machine. Its programs are like this:

Bit String—LISP Expression, NL, Data

There's a LISP expression in binary followed by a delimiter character followed by raw binary data. And you just measure the size of the whole program in bits, and that's our program-size complexity measure. The next thing you do with this theory, that I do in my course, is to show that information is additive.

$$H((X\ Y)) \leq H(X) + H(Y) + 432$$

In the course I construct the 432-bit prefix and you see it working. You actually run the programs.

The last time I gave this course was in Rovaniemi, Finland, this May, end of May, when it never got dark, and I had the pleasure of seeing a room full of people working on their computers, running programs on my universal Turing machine. They were able to do it and get results! That was really a thrill for me. Here's how you do it: First you write out a LISP expression—that's easy to do—then you use a primitive function to convert it to binary, and you append the binary data. Then you feed the result to my universal machine, which is just one line of code in my LISP. To define my universal Turing machine is very easy in this LISP.

So you see, I'm not using the size of LISP expressions as my program-size measure, but I am using LISP to define the universal machine, and I'm also using it to produce the programs that I feed to the universal machine.

And my universal machine really works! Not just for theory—we've always had universal Turing machines that work for theoretical purposes—right? since the 1930's—but you can actually run interesting programs on it. And the fact that I'm using a very high-level language, LISP, as my basis, means that set theory is essentially built into this universal machine. For example, it's very easy to program set union and intersection in LISP. And the kind of algorithms that I want to do, which are the ones that I use to prove my incompleteness results, are very easy to express in this language. That's the point: my universal machine is good not only for proving theorems, but also for writing programs and running them on examples in a finite amount of time.

I recently spoke to a professor at the University of New Mexico in Albuquerque who teaches recursive function theory. He said that his computer science students felt it was very strange that they were running programs on computers all day long, but never in their class on recursive function and computability theory! They felt a kind of cognitive dissonance, it just didn't make sense. Maybe, he said, my approach using LISP would work better, because then students could run the programs. LISP is a version of recursive function theory that really works! We've learnt much better how to write software since 1930!

But the version of `try` that I used to analyze whether one can prove that a LISP expression is elegant is not quite right. So let me show you the real `try`, which has an additional argument. Now it's like this:

```
(try time-limit lisp-expression binary-data)
```

There's a time limit, there's a LISP expression that we're going to try to run, to evaluate, and now there's also binary data "on the side," raw binary data. The binary data is just a list of 0's and 1's—that's the easiest way to represent a bit string in LISP—for example

```
(1 0 1 0 1 1 1 1)
```

So there are two parentheses and there are blanks between the bits, but otherwise it's just like the bit strings that you've always known and loved.

Just as before, we're going to try to evaluate the given LISP expression in the given amount of time. The new wrinkle is that while it's being evaluated this LISP expression has access to raw binary data on the side. So I'm using `try` to do a lot of things at the same time! It does a time-limited evaluation, and it also gives binary data to a LISP expression.

So while we're running the LISP expression, there's a time limit, and the LISP expression can use a zero-argument primitive function that says, give me the next bit of the binary data. And it'll get the next bit, if there is one. The LISP expression can also use another primitive function to read an entire LISP S-expression from the binary data. How does it do it? Well, it reads eight bits at a time, sees the characters that it gets, paying special attention to blanks and parentheses, until it gets to a newline character `\n`. Then it stops. So you don't always read individual bits from the binary data, you can also read big chunks in if you want to.

Now what value do you get back from this `try`? Remember, you always get a value back, even if the LISP expression that you're trying never terminates. That's why you use `try` instead of `eval`. Well, it'll still be a triple.

```
(success/failure value/out-of-time/out-of-data captured-displays)
```

It'll start as before with `success` or `failure`. If the try was a success, that means that evaluation of the LISP expression completed, and the second element of the triple will be the value that this LISP expression returned. On the other hand, if the try was a failure, then the second element of the triple is going to tell us **why** the evaluation failed. I already told you one way that the evaluation can fail: you can run out of time. An evaluation might also fail in many other ways, for example if you try to apply a primitive function to arguments which are not of a suitable type. Since I didn't want to be bothered with that kind of problem, I made the semantics of my LISP extremely permissive, so that I don't have to put a lot of error messages here telling you what went wrong. In fact, the only way that a try can fail is if it ran out of time or it ran out of binary data. You run out of binary data if you use it all up and then try to keep reading it. It's okay to read the last bit of the binary data, but then it's not okay to ask for another bit.

What about the third element of the triple returned by `try`, the captured intermediate results? Now it's "captured displays." Let me tell you why.

Intermediate results are important when you're debugging a big LISP expression. The final value may not be enough to tell what went wrong. The cute LISP solution to this problem is to add a one-argument primitive function that's an identity function. I call it `display`.

```
(display X)
```

So from the point of view of pure mathematics, `display` is useless, it just returns the value of its argument X. But it's extremely useful, because `display` prints the value of its argument on your screen. And to debug a large LISP expression,

you just wrap `display`'s around interesting parts of the expression; that doesn't change the final value.

But in my LISP, in the game I'm playing, `display` is used for something much more important than debugging, it's used to put out theorems. `Display` enables a LISP expression to produce an infinite number of results. And that's very important because I want to model a formal axiomatic system, which is an unending computation that produces theorems, as a LISP S-expression. Normal LISP doesn't really have a way for a LISP expression to produce an infinite amount of output. So I had to create this mechanism in which `display` throws theorems and `try` catches them!

So in my LISP `display` has official status. It's not just a debugging mechanism. It's the way that a formal axiomatic system outputs each theorem, it's an important part of LISP.

So the last thing in the triple returned by a `try` is a list of captured displays. Every time that the LISP expression being tried tries to put something on your screen, it won't go there, it'll end up in this list instead. So if you're trying a formal axiomatic system, this will be a list of theorems.

That's it! This is all there is to it. It's the entire mechanism for programming algorithmic information theory in LISP.

The Halting Probability Ω

Well, I've shown you all the machinery, but what do I do with it! What's the next thing in the course on my web site? Well, there's not much time left! Just enough time for a quick summary!

I've shown you the basic tools that I need to program out my version of recursive function theory, my version of Turing and Gödel's incompleteness results. Basically, it's just `try`, that's all I have to add to LISP. It's what makes it possible for LISP to handle algorithmic information theory.

When I taught this course in Rovaniemi, I started with a historical introduction. Then I explained my LISP. Then I show off my universal Turing machine and run a bunch of simple programs on it. The next thing I do is I define the halting probability, it's capital omega.

$$\Omega$$

To define Ω you take my universal machine that has binary programs and produces LISP expressions, and you just feed it bits that you get by tossing a coin, that's independent tosses of a fair coin. And you ask, what is the halting probability? That's Ω. And I actually give a LISP program that calculates the halting probability in the limit from below. You'll find this program explained in my lecture transcript "An invitation to algorithmic information theory." It's in the DMTCS'96 *Proceedings.* When I gave that talk I was speaking three to five times faster than I am today—I don't know how I covered so much material! So if you want to see the LISP program to calculate Ω in the limit from below, it's in the DMTCS'96 *Proceedings,* or look at my Web site, where there are sample runs.

The next thing I do is I prove that Ω is irreducible algorithmic information. The precise result turns out to be this:

$$H(\Omega_N) > N - 8000.$$

What's Ω_N? Well, the halting probability is a real number, so write it in binary, and take the first N bits after the "decimal" point. So this inequality states that to get the first N bits of the halting probability, you need a program that's more than $N - 8000$ bits long. And the reason is that if you knew the first N bits of the halting probability, that would enable you to solve the halting problem for all programs up to N bits in size. That's how you prove that Ω is irreducible.

And then finally I get what I think is my most devastating incompleteness result, which says that a formal axiomatic system can't enable you to prove, to determine, more than

$$H(\text{FAS}) + 15328$$

bits of the halting probability. This follows from the previous inequality, the one that states that

$$H(\Omega_N) > N - 8000.$$

In other words, since a small program can't give you a lot of bits of Ω, a set of axioms can't enable enable you to determine substantially more bits of Ω than there are bits of axioms. That's what

$$H(\text{FAS})$$

is, it's the number of bits in the smallest program that makes my universal machine output all the theorems in the formal axiomatic system. In other words, it's the complexity of the formal axiomatic system, it's the number of bits in its axioms. So the more bits of Ω you want to determine, the more bits you have to add to your axioms!

My main theorem is a mathematical pun. Turing proved that the halting problem is undecidable. I prove that the halting probability is irreducible! Not only you can't compress bits of Ω into a program substantially smaller than the number of bits of Ω you calculate, you can't do it using reasoning either. Essentially the only way to get bits of Ω out of a formal axiomatic system, is if you just add those bits as axioms! So determining bits of Ω is a losing proposition! Mathematical reasoning doesn't help at all. Well, that's not quite right, because of the constant 15328.

$$H(\text{FAS}) + \mathbf{15328}$$

If it weren't for this, I'd say you get out exactly what you put in!

So if you want to prove more bits of Ω, essentially the only way to do it is to add them to your axioms. But you can prove **anything** by adding it to your axioms. And what Ω shows, is that sometimes—Ω's a fairly simple object from the point of view of non-constructive mathematics—in some relatively elementary branches of mathematics, the only way to get more out is to put it into the

axioms. These are situations in which mathematical reasoning is really useless, really impotent! Because if you can get something out of a set of axioms only by putting it in as a new postulate, why bother! So Ω is really a worst case, it's our worst nightmare come true, because it's

Irreducible Mathematical Information

Conclusion

So to end, let me try to state in words what I find intriguing about Ω. As I said, Ω shows that in some cases you're in big trouble! But let me emphasize the philosophical discontinuity.

The normal view of mathematics is that if something is true, it's true for a reason, right? In mathematics the reason that something is true is called a proof. And the job of the mathematician is to find proofs. So normally you think if something is true it's true for a reason. Well, what Ω shows you, what I've discovered, is that some mathematical facts are **true for no reason!** They're true **by accident!** And consequently they forever escape the power of mathematical reasoning. Each bit of Ω has got to be a 0 or a 1, but it's so delicately balanced, that we're never going to know which it is.

I used to believe that all of mathematical truth, all the infinite variety of mathematical truth, **could be compressed** into a small set of axioms and methods of reasoning that we could all agree on, and that we learn as mathematics students. I felt this deep in my soul, it's part of what makes mathematics beautiful, the sharpness, the clarity—it seemed inhuman, even superhuman! Unfortunately the existence of irreducible mathematical facts shows that in some cases there is absolutely no compression, no structure or pattern at all in mathematical truth. I don't know why anyone would want to prove what bits of Ω are. But if you wanted to do that, it would be completely hopeless. Because, you see, the bits of Ω aren't 0 or 1 for any particular reason—they've got to be one or the other, it's a specific Ω, there's a LISP program for calculating it in the limit from below—but it doesn't happen for a reason, it happens by accident. If God were willing to answer yes/no questions, each bit of Ω would require a separate question, because there are no correlations, there is no redundancy!

But I'm being too pessimistic. After all, Fermat's last theorem was just demonstrated. And I wouldn't be too surprised if another determined, brilliant individual were to prove the Riemann hypothesis. In fact, clever mathematicians do succeed in settling famous conjectures—remember the announcement that "four colors suffice"?

In this direction

Irreducible Mathematical Information

you can't go any farther, right? But I think that it's an interesting question to understand how come in spite of these results it is in fact possible to do mathematics so well? I think that the interesting question now is not to prove

incompleteness results, but to see how come mathematics is still so wonderful. It is! We can prove wonderful theorems, breathtaking theorems. And I think that it would be interesting now to try to understand better how this is possible.

I guess that's the story that I wanted to tell you. Thank you very much!