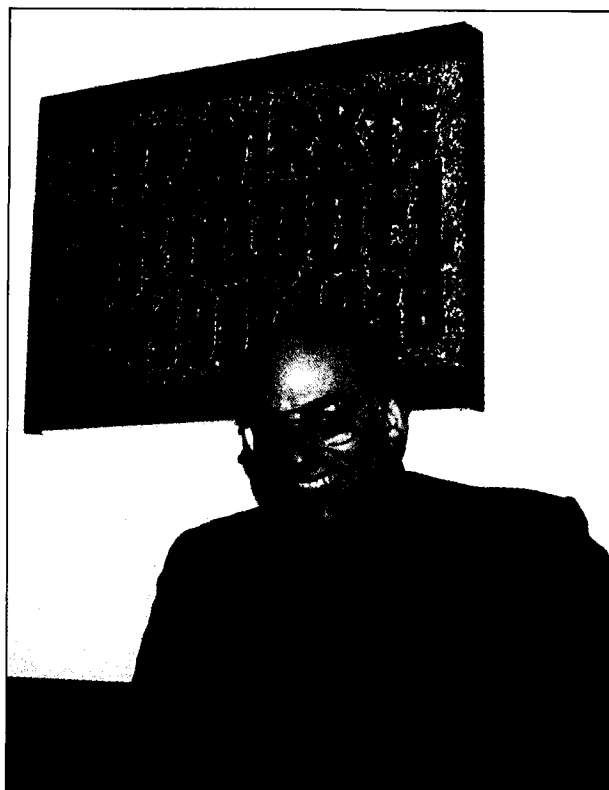


*The* **LIMITS** *of*  
**MATHEMATICS**



CHAITIN, VIENNA, 1991

A Course on  
Information  
Theory and  
the Limits of  
Formal  
Reasoning

# *The* **LIMITS** *of* **MATHEMATICS**

**Gregory J. CHAITIN**



Springer

Gregory J. Chaitin  
IBM Research Division  
Thomas J. Watson Research Center  
30 Saw Mill River Road  
Hawthorne, NY 10532  
USA

# Library of Congress Cataloging-in-Publication Data

Chaitin, Gregory J.

The limits of mathematics : a course on information theory & the limits of formal reasoning / Gregory J. Chaitin

p. cm.

Springer series in discrete mathematics and theoretical computer science  
Includes bibliographical references.

ISBN 981308359X

1. Computer science--Mathematics. 2. Information theory.

3. Reasoning.

QA76.9.M35 C48 1997

511.3 21--dc

97-36103

CIP

ISBN 981-3083-59-X

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on micro-films or in any other way, and storage in databanks or in any system now known or to be invented. Permission for use must always be obtained from the publisher in writing.

© Springer-Verlag Singapore Pte. Ltd. 1998  
2nd Reprint 1998

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Typesetting: Camera-ready by author  
Printed in Singapore  
SPIN 10699225 5 4 3 2

# Foreword

As a teenager, Greg created independently of Kolmogorov and Solomonoff, what we call today algorithmic information theory, a subject of which he is the main architect. His 1965 paper on gedanken experiments on automata, which he wrote when he was in high school, is still of interest today. He was also heavily involved in IBM, where he has worked for almost thirty years, on the development of RISC technology.

Greg's results are widely quoted. My favorite portrait of Greg can be found in John Horgan's—a writer for *Scientific American*—1996 book *The End of Science*. Greg has gotten many honors. He was a guest of distinguished people like Prigogine, the King and Queen of Belgium, and the Crown Prince of Japan.

Just to be brief, allow me to paraphrase Bette Davis in *All About Eve*. She said, “Fasten your seat belts, it's going to be a bumpy talk!” Ladies and Gentlemen, Greg Chaitin! [Laughter & Applause]

CRISTIAN CALUDE introducing GREGORY CHAITIN at the DMTCS'96 meeting at the University of Auckland.

# Preface

If a catastrophe were to suddenly threaten to destroy everything that I've ever written, this book is what I would try to preserve! What's here is basic, everything else is technical details. This book is the final version of a course on the limits of mathematics that I've given several times. It contains transcripts of my three favorite lectures, and in these transcripts I've tried to retain the flavor of a live performance in front of an audience. Facing an audience gives me a special energy that I don't get when I'm lecturing to a typewriter!

After these three lecture transcripts, comes a lot of LISP code, and then finally some Mathematica code.

For this course I had to invent a new LISP dialect. Unfortunately there is no manual for my LISP, but two of the lectures explain it, and there is a long lisp run, `examples.r`, that contains lots of comments and shows every feature of the language. So I think that what's here is equivalent to a tutorial plus a manual.

I've also included the code for my LISP interpreter. It's three-hundred lines of Mathematica, and it's in an Appendix at the end of the book.

Unfortunately software is ephemeral, it changes out from under you, and you have to run as fast as you can to stay in the same place! The code in the Appendix is Mathematica Version 2 code, not the latest version of Mathematica, which is Version 3. I also coded the interpreter in C, and C might be replaced by Java in the not too distant future! Since this course is now in its final form, I've decided to get out of the race, to freeze the software.

But you can understand the basic ideas in this book by reading the lectures without going through the LISP code. And you can read my LISP code and examine the LISP runs without having a LISP interpreter. So everything essential is here in this book. Being able to run the LISP interpreter is certainly helpful—that was the whole idea of my course, that it should be a “hands on” computer course—but I also gave this course a few times without computers, and that seemed to work too, even if it wasn’t as exciting.

You can also run my LISP code in your head, which is the way that mathematics is traditionally done. Then just think of it as a formalism, rather than as a programming language.

I’ve enormously enjoyed giving this course or overviews of it, in Maine, privately in my office, in Santa Fe and Albuquerque, in Romania on the Black Sea, in Abisko and Rovaniemi on the Arctic Circle, and in Helsinki, Copenhagen and Auckland. It was tremendous fun, and I’m very, very grateful for these invitations—they stimulated me to work through the ideas and produce many, many versions of the software. So I want to thank George Markowsky, John Casti, Cris Calude, Walter Meyerstein, Bernard Moret, Ed Angel, Veikko Keränen, Gautam Dasgupta, Tor Nørretranders and Anders Karlqvist.

I must also thank the Santa Fe Institute, which I have had the pleasure of visiting regularly. Two of the chapters in this book are lectures that I gave during those visits. And I still remember solving some of the key problems one sleepless night at John and Vivien Casti’s friendly home.

And none of this could have happened without IBM, which has supported my research for three decades. In particular, I’m grateful to my current management chain, Lee Nackman and Ambuj Goyal, for making it possible for me to finish this book. And I admire the very bright and determined participants in my courses, brave pioneers! “*Merci beaucoup*” and “*Tusind tak*,” everyone! Final thanks to Hans-Christian Reichel for the frontispiece photograph he took in the University of Vienna classroom where Gödel lectured.

GREGORY CHAITIN, April 1997

# Contents

Randomness in arithmetic and the decline and fall of reductionism in pure mathematics	1
Elegant LISP programs	29
An invitation to algorithmic information theory	57
The limits of mathematics	89
examples.r . . . . .	96
godel.r . . . . .	119
utm.r . . . . .	122
godel2.r . . . . .	126
omega.r . . . . .	129
omega2.r . . . . .	131
omega3.r . . . . .	132
godel3.r . . . . .	136
Appendix. LISP interpreter in Mathematica	141



“He thought he had **THE TRUTH!**”  
GWEN, the literary agent in  
Norman Chaitin’s play *Off Broadway*.

# RANDOMNESS IN ARITHMETIC AND THE DECLINE AND FALL OF REDUCTIONISM IN PURE MATHEMATICS

Bulletin of the European Association for Theoretical  
Computer Science No. 50 (June 1993), pp. 314–  
328

G. J. Chaitin, *chaitin@watson.ibm.com*

*Lecture given Thursday 22 October 1992 at a Mathematics – Computer  
Science Colloquium at the University of New Mexico. The lecture was  
videotaped; this is an edited transcript.*

## 1. Hilbert on the axiomatic method

Last month I was a speaker at a symposium on reductionism at Cambridge University where Turing did his work. I'd like to repeat the talk I gave there and explain how my work continues and extends Turing's.

Two previous speakers had said bad things about David Hilbert. So I started by saying that in spite of what you might have heard in some of the previous lectures, Hilbert was not a twit!

Hilbert's idea is the culmination of two thousand years of mathematical tradition going back to Euclid's axiomatic treatment of geometry, going back to Leibniz's dream of a symbolic logic and Russell and Whitehead's monumental *Principia Mathematica*. Hilbert's dream was to once and for all clarify the methods of mathematical reasoning. Hilbert wanted to formulate a formal axiomatic system which would encompass all of mathematics.

### Formal Axiomatic System

→

→

→

Hilbert emphasized a number of key properties that such a formal axiomatic system should have. It's like a computer programming language. It's a precise statement about the methods of reasoning, the postulates and the methods of inference that we accept as mathematicians. Furthermore Hilbert stipulated that the formal axiomatic system encompassing all of mathematics that he wanted to construct should be "consistent" and it should be "complete."

### Formal Axiomatic System

→ consistent

→ complete

→

Consistent means that you shouldn't be able to prove an assertion and the contrary of the assertion.

### Formal Axiomatic System

→ consistent  $A \neg A$

→ complete

→

You shouldn't be able to prove  $A$  and not  $A$ . That would be very embarrassing.

Complete means that if you make a meaningful assertion you should be able to settle it one way or the other. It means that either  $A$  or not  $A$  should be a theorem, should be provable from the axioms using the rules of inference in the formal axiomatic system.

### **Formal Axiomatic System**

→ consistent  $A \neg A$

→ complete  $A \neg A$

→

Consider a meaningful assertion  $A$  and its contrary not  $A$ . Exactly one of the two should be provable if the formal axiomatic system is consistent and complete.

A formal axiomatic system is like a programming language. There's an alphabet and rules of grammar, in other words, a formal syntax. It's a kind of thing that we are familiar with now. Look back at Russell and Whitehead's three enormous volumes full of symbols and you'll feel you're looking at a large computer program in some incomprehensible programming language.

Now there's a very surprising fact. Consistent and complete means only truth and all the truth. They seem like reasonable requirements. There's a funny consequence, though, having to do with something called the decision problem. In German it's the Entscheidungsproblem.

### **Formal Axiomatic System**

→ consistent  $A \neg A$

→ complete  $A \neg A$

→ decision problem

Hilbert ascribed a great deal of importance to the decision problem.

### **HILBERT**

#### **Formal Axiomatic System**

→ consistent  $A \neg A$

→ complete  $A \neg A$

→ decision problem

Solving the decision problem for a formal axiomatic system is giving an algorithm that enables you to decide whether any given meaningful

assertion is a theorem or not. A solution of the decision problem is called a decision procedure.

## HILBERT

### Formal Axiomatic System

→ consistent  $A \neg A$

→ complete  $A \neg A$

→ decision procedure

This sounds weird. The formal axiomatic system that Hilbert wanted to construct would have included all of mathematics: elementary arithmetic, calculus, algebra, everything. If there's a decision procedure, then mathematicians are out of work. This algorithm, this mechanical procedure, can check whether something is a theorem or not, can check whether it's true or not. So to require that there be a decision procedure for this formal axiomatic system sounds like you're asking for a lot.

However it's very easy to see that if it's consistent and it's complete that implies that there must be a decision procedure. Here's how you do it. You have a formal language with a finite alphabet and a grammar. And Hilbert emphasized that the whole point of a formal axiomatic system is that there must be a mechanical procedure for checking whether a purported proof is correct or not, whether it obeys the rules or not. That's the notion that mathematical truth should be objective so that everyone can agree whether a proof follows the rules or not.

So if that's the case you run through all possible proofs in size order, and look at all sequences of symbols from the alphabet one character long, two, three, four, a thousand, a thousand and one... a hundred thousand characters long. You apply the mechanical procedure which is the essence of the formal axiomatic system, to check whether each proof is valid. Most of the time, of course, it'll be nonsense, it'll be ungrammatical. But you'll eventually find every possible proof. It's like a million monkeys typing away. You'll find every possible proof, though only in principle of course. The number grows exponentially and this is something that you couldn't do in practice. You'd never get to proofs that are one page long.

But in principle you could run through all possible proofs, check which ones are valid, see what they prove, and that way you can sys-

tematically find all theorems. In other words, there is an algorithm, a mechanical procedure, for generating one by one every theorem that can be demonstrated in a formal axiomatic system. So if for every meaningful assertion within the system, either the assertion is a theorem or its contrary is a theorem, only one of them, then you get a decision procedure. To see whether an assertion is a theorem or not you just run through all possible proofs until you find the assertion coming out as a theorem or you prove the contrary assertion.

So it seems that Hilbert actually believed that he was going to solve once and for all, all mathematical problems. It sounds amazing, but apparently he did. He believed that he would be able to set down a consistent and complete formal axiomatic system for all of mathematics and from it obtain a decision procedure for all of mathematics. This is just following the formal, axiomatic tradition in mathematics.

But I'm sure he didn't think that it would be a practical decision procedure. The one I've outlined would only work in principle. It's exponentially slow, it's terribly slow! Totally impractical. But the idea was that if all mathematicians could agree whether a proof is correct and be consistent and complete, in principle that would give a decision procedure for automatically solving any mathematical problem. This was Hilbert's magnificent dream, and it was to be the culmination of Euclid and Leibniz, and Boole and Peano, and Russell and Whitehead.

Of course the only problem with this inspiring project is that it turned out to be impossible!

## **2. Gödel, Turing and Cantor's diagonal argument**

Hilbert is indeed inspiring. His famous lecture in the year 1900 is a call to arms to mathematicians to solve a list of twenty-three difficult problems. As a young kid becoming a mathematician you read that list of twenty-three problems and Hilbert is saying that there is no limit to what mathematicians can do. We can solve a problem if we are clever enough and work at it long enough. He didn't believe that in principle there was any limit to what mathematics could achieve.

I think this is very inspiring. So did John von Neumann. When he was a young man he tried to carry through Hilbert's ambitious program. Because Hilbert couldn't quite get it all to work, in fact he started off just with elementary number theory, 1, 2, 3, 4, 5, ..., not even with real numbers at first.

And then in 1931 to everyone's great surprise (including von Neumann's), Gödel showed that it was impossible, that it couldn't be done, as I'm sure you all know.

### **Gödel 1931**

This was the opposite of what everyone had expected. Von Neumann said it never occurred to him that Hilbert's program couldn't be carried out. Von Neumann admired Gödel enormously, and helped him to get a permanent position at the Institute for Advanced Study.

What Gödel showed was the following. Suppose that you have a formal axiomatic system dealing with elementary number theory, with 1, 2, 3, 4, 5 and addition and multiplication. And we'll assume that it's consistent, which is a minimum requirement—if you can prove false results it's really pretty bad. What Gödel showed was that if you assume that it's consistent, then you can show that it's incomplete. That was Gödel's result, and the proof is very clever and involves self-reference. Gödel was able to construct an assertion about the whole numbers that says of itself that it's unprovable. This was a tremendous shock. Gödel has to be admired for his intellectual imagination; everyone else thought that Hilbert was right.

However I think that Turing's 1936 approach is better.

### **Gödel 1931**

### **Turing 1936**

Gödel's 1931 proof is very ingenious, it's a real tour de force. I have to confess that when I was a kid trying to understand it, I could read it and follow it step by step but somehow I couldn't ever really feel that I was grasping it. Now Turing had a completely different approach.

Turing's approach I think it's fair to say is in some ways more fundamental. In fact, Turing did more than Gödel. Turing not only got as a corollary Gödel's result, he showed that there could be no decision procedure.

You see, if you assume that you have a formal axiomatic system for arithmetic and it's consistent, from Gödel you know that it can't be complete, but there still might be a decision procedure. There still might be a mechanical procedure which would enable you to decide if a given assertion is true or not. That was left open by Gödel, but Turing settled it. The fact that there cannot be a decision procedure is more fundamental and you get incompleteness as a corollary.

How did Turing do it? I want to tell you how he did it because that's the springboard for my own work. The way he did it, and I'm sure all of you have heard about it, has to do with something called the halting problem. In fact if you go back to Turing's 1936 paper you will not find the words "halting problem." But the idea is certainly there.

People also forget that Turing was talking about "computable numbers." The title of his paper is "On computable numbers, with an application to the Entscheidungsproblem." Everyone remembers that the halting problem is unsolvable and that comes from that paper, but not as many people remember that Turing was talking about computable real numbers. My work deals with computable and dramatically uncomputable real numbers. So I'd like to refresh your memory how Turing's argument goes.

Turing's argument is really what destroys Hilbert's dream, and it's a simple argument. It's just Cantor's diagonal procedure (for those of you who know what that is) applied to the computable real numbers. That's it, that's the whole idea in a nutshell, and it's enough to show that Hilbert's dream, the culmination of two thousand years of what mathematicians thought mathematics was about, is wrong. So Turing's work is tremendously deep.

What is Turing's argument? A real number, you know  $3.1415926 \dots$ , is a length measured with arbitrary precision, with an infinite number of digits. And a computable real number said Turing is one for which there is a computer program or algorithm for calculating the digits one by one. For example, there are programs for  $\pi$ , and there are algorithms for solutions of algebraic equations with integer coefficients. In fact most of the numbers that you actually find in analysis are computable. However they're the exception, if you know set theory, because the computable reals are denumerable and the reals are nondenumerable (you don't have to know what that means). That's the essence of



Turing's idea.

The idea is this. You list all possible computer programs. At that time there were no computer programs, and Turing had to invent the Turing machine, which was a tremendous step forward. But now you just say, imagine writing a list with every possible computer program.

$p_1$	<b>Gödel 1931</b>
$p_2$	<b>Turing 1936</b>
$p_3$	
$p_4$	
$p_5$	
$p_6$	
$\vdots$	

If you consider computer programs to be in binary, then it's natural to think of a computer program as a natural number. And next to each computer program, the first one, the second one, the third one, write out the real number that it computes if it computes a real (it may not). But if it prints out an infinite number of digits, write them out. So maybe it's 3.1415926 and here you have another and another and another:

$p_1$	3.1415926...	<b>Gödel 1931</b>
$p_2$	...	<b>Turing 1936</b>
$p_3$	...	
$p_4$	...	
$p_5$	...	
$p_6$	...	
$\vdots$		

So you make this list. Maybe some of these programs don't print out an infinite number of digits, because they're programs that halt or that have an error in them and explode. But then there'll just be a

blank line in the list.

$p_1$	3.1415926...	<b>Gödel</b> 1931
$p_2$	...	<b>Turing</b> 1936
$p_3$	...	
$p_4$	...	
$p_5$		
$p_6$	...	
$\vdots$		

It's not really important—let's forget about this possibility.

Following Cantor, Turing says go down the diagonal and look at the first digit of the first number, the second digit of the second, the third...

$p_1$	-. <u><math>d_{11}</math></u> $d_{12}d_{13}d_{14}d_{15}d_{16}$ ...	<b>Gödel</b> 1931
$p_2$	-. $d_{21}$ <u><math>d_{22}</math></u> $d_{23}d_{24}d_{25}d_{26}$ ...	<b>Turing</b> 1936
$p_3$	-. $d_{31}d_{32}$ <u><math>d_{33}</math></u> $d_{34}d_{35}d_{36}$ ...	
$p_4$	-. $d_{41}d_{42}d_{43}$ <u><math>d_{44}</math></u> $d_{45}d_{46}$ ...	
$p_5$		
$p_6$	-. $d_{61}d_{62}d_{63}d_{64}d_{65}$ <u><math>d_{66}</math></u> ...	
$\vdots$		

Well actually it's the digits after the decimal point. So it's the first digit after the decimal point of the the first number, the second digit after the decimal point of the second, the third digit of the third number, the fourth digit of the fourth, the fifth digit of the fifth. And it doesn't matter if the fifth program doesn't put out a fifth digit, it really doesn't matter.

What you do is you change these digits. Make them different. Change every digit on the diagonal. Put these changed digits together into a new number with a decimal point in front, a new real number. That's Cantor's diagonal procedure. So you have a digit which you choose to be different from the first digit of the first number, the second digit of the second, the third of the third, and you put these

together into one number.

$$\begin{array}{ll}
 p_1 & - .\underline{d_{11}}d_{12}d_{13}d_{14}d_{15}d_{16} \cdots & \text{Gödel 1931} \\
 p_2 & - .d_{21}\underline{d_{22}}d_{23}d_{24}d_{25}d_{26} \cdots & \text{Turing 1936} \\
 p_3 & - .d_{31}d_{32}\underline{d_{33}}d_{34}d_{35}d_{36} \cdots \\
 p_4 & - .d_{41}d_{42}d_{43}\underline{d_{44}}d_{45}d_{46} \cdots \\
 p_5 & \\
 p_6 & - .d_{61}d_{62}d_{63}d_{64}d_{65}\underline{d_{66}} \cdots \\
 & \vdots \\
 & . \neq d_{11} \neq d_{22} \neq d_{33} \neq d_{44} \neq d_{55} \neq d_{66} \cdots
 \end{array}$$

This new number cannot be in the list because of the way it was constructed. Therefore it's an uncomputable real number. How does Turing go on from here to the halting problem? Well, just ask yourself **why** can't you compute it? I've explained how to get this number and it looks like you could almost do it. To compute the  $N$ th digit of this number, you get the  $N$ th computer program (you can certainly do that) and then you start it running until it puts out an  $N$ th digit, and at that point you change it. Well what's the problem? That sounds easy.

The problem is, what happens if the  $N$ th computer program never puts out an  $N$ th digit, and you sit there waiting? And that's the halting problem—you cannot decide whether the  $N$ th computer program will ever put out an  $N$ th digit! This is how Turing got the unsolvability of the halting problem. Because if you could solve the halting problem, then you could decide if the  $N$ th computer program ever puts out an  $N$ th digit. And if you could do that then you could actually carry out Cantor's diagonal procedure and compute a real number which has to differ from any computable real. That's Turing's original argument.

Why does this explode Hilbert's dream? What has Turing proved? That there is no algorithm, no mechanical procedure, which will decide if the  $N$ th computer program ever outputs an  $N$ th digit. Thus there can be no algorithm which will decide if a computer program ever halts (finding the  $N$ th digit put out by the  $N$ th program is a special case). Well, what Hilbert wanted was a formal axiomatic system from which all mathematical truth should follow, only mathematical truth, and all mathematical truth. If Hilbert could do that, it would give us a mechanical procedure to decide if a computer program will ever halt. Why?

You just run through all possible proofs until you either find a proof that the program halts or you find a proof that it never halts. So if Hilbert's dream of a finite set of axioms from which all of mathematical truth should follow were possible, then by running through all possible proofs checking which ones are correct, you would be able to decide if any computer program halts. In principle you could. But you **can't** by Turing's very simple argument which is just Cantor's diagonal argument applied to the computable reals. That's how simple it is!

Gödel's proof is ingenious and difficult. Turing's argument is so fundamental, so deep, that everything seems natural and inevitable. But of course he's building on Gödel's work.

### 3. The halting probability and algorithmic randomness

The reason I talked to you about Turing and computable reals is that I'm going to use a different procedure to construct an uncomputable real, a much more uncomputable real than Turing does.

$p_1$	$-. \underline{d_{11}} d_{12} d_{13} d_{14} d_{15} d_{16} \cdots$	<b>Gödel 1931</b>
$p_2$	$-. d_{21} \underline{d_{22}} d_{23} d_{24} d_{25} d_{26} \cdots$	<b>Turing 1936</b>
$p_3$	$-. d_{31} d_{32} \underline{d_{33}} d_{34} d_{35} d_{36} \cdots$	uncomputable reals
$p_4$	$-. d_{41} d_{42} d_{43} \underline{d_{44}} d_{45} d_{46} \cdots$	
$p_5$		
$p_6$	$-. d_{61} d_{62} d_{63} d_{64} d_{65} \underline{d_{66}} \cdots$	
$\vdots$		
	$. \neq d_{11} \neq d_{22} \neq d_{33} \neq d_{44} \neq d_{55} \neq d_{66} \cdots$	

And that's how we're going to get into much worse trouble.

How do I get a much more uncomputable real? (And I'll have to tell you how uncomputable it is.) Well, not with Cantor's diagonal argument. I get this number, which I like to call  $\Omega$ , like this:

$$\Omega = \sum_{p \text{ halts}} 2^{-|p|}$$

This is just the halting probability. It's sort of a mathematical pun. Turing's fundamental result is that the halting problem is unsolvable—there is no algorithm that'll settle the halting problem. My fundamental result is that the halting probability is algorithmically irreducible or algorithmically random.

What exactly is the halting probability? I've written down an expression for it:

$$\Omega = \sum_{p \text{ halts}} 2^{-|p|}$$

Instead of looking at individual programs and asking whether they halt, you put all computer programs together in a bag. If you generate a computer program at random by tossing a coin for each bit of the program, what is the chance that the program will halt? You're thinking of programs as bit strings, and you generate each bit by an independent toss of a fair coin, so if a program is  $N$  bits long, then the probability that you get that particular program is  $2^{-N}$ . Any program  $p$  that halts contributes  $2^{-|p|}$ , two to the minus its size in bits, the number of bits in it, to this halting probability.

By the way there's a technical detail which is very important and didn't work in the early version of algorithmic information theory. You couldn't write this:

$$\Omega = \sum_{p \text{ halts}} 2^{-|p|}$$

It would give infinity. The technical detail is that no extension of a valid program is a valid program. Then this sum

$$\sum_{p \text{ halts}} 2^{-|p|}$$

turns out to be between zero and one. Otherwise it turns out to be infinity. It only took ten years until I got it right. The original 1960s version of algorithmic information theory is wrong. One of the reasons it's wrong is that you can't even define this number

$$\Omega = \sum_{p \text{ halts}} 2^{-|p|}$$

In 1974 I redid algorithmic information theory with “self-delimiting” programs and then I discovered the halting probability  $\Omega$ .

Okay, so this is a probability between zero and one

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

like all probabilities. The idea is you generate each bit of a program by tossing a coin and ask what is the probability that it halts. This number  $\Omega$ , this halting probability, is not only an uncomputable real—Turing already knew how to do that. It is uncomputable in the worst possible way. Let me give you some clues how uncomputable it is.

Well, one thing is it’s algorithmically incompressible. If you want to get the first  $N$  bits of  $\Omega$  out of a computer program, if you want a computer program that will print out the first  $N$  bits of  $\Omega$  and then halt, that computer program has to be  $N$  bits long. Essentially you’re only printing out constants that are in the program. You cannot squeeze the first  $N$  bits of  $\Omega$ . This

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

is a real number, you could write it in binary. And if you want to get out the first  $N$  bits from a computer program, essentially you just have to put them in. The program has to be  $N$  bits long. That’s irreducible algorithmic information. There is no concise description.

Now that’s an abstract way of saying things. Let me give a more concrete example of how random  $\Omega$  is. Émile Borel at the turn of this century was one of the founders of probability theory.

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

**Émile Borel**

QUESTION: Can I ask a very simple question before you get ahead?

ANSWER: Sure.

QUESTION: I can’t see why  $\Omega$  should be a probability. What if the two one-bit programs both halt? I mean, what

if the two one-bit programs both halt and then some other program halts. Then  $\Omega$  is greater than one and not a probability.

ANSWER: I told you no extension of a valid program is a valid program.

QUESTION: Oh right, no other programs can halt.

ANSWER: The two one-bit programs would be all the programs there are. That's the reason this number

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

can't be defined if you think of programs in the normal way.

So here we have Émile Borel, and he talked about something he called a normal number.

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

**Émile Borel** — normal reals

What is a normal real number? People have calculated  $\pi$  out to a billion digits, maybe two billion. One of the reasons for doing this, besides that it's like climbing a mountain and having the world record, is the question of whether each digit occurs the same number of times. It looks like the digits 0 through 9 each occur 10% of the time in the decimal expansion of  $\pi$ . It looks that way, but nobody can prove it. I think the same is true for  $\sqrt{2}$ , although that's not as popular a number to ask this about.

Let me describe some work Borel did around the turn of the century when he was pioneering modern probability theory. Pick a real number in the unit interval, a real number with a decimal point in front, with no integer part. If you pick a real number in the unit interval, Borel showed that with probability one it's going to be "normal." Normal means that when you write it in decimal each digit will occur in the limit exactly 10% of the time, and this will also happen in any other base. For example in binary 0 and 1 will each occur in the limit exactly 50% of the time. Similarly with blocks of digits. This was called an absolutely

normal real number by Borel, and he showed that with probability one if you pick a real number at random between zero and one it's going to have this property. There's only one problem. He didn't know whether  $\pi$  is normal, he didn't know whether  $\sqrt{2}$  is normal. In fact, he couldn't exhibit a single individual example of a normal real number.

The first example of a normal real number was discovered by a friend of Alan Turing's at Cambridge called David Champernowne, who is still alive and who's a well-known economist. Turing was impressed with him—I think he called him “Champ”—because Champ had published this in a paper as an undergraduate. This number is known as Champernowne's number. Let me show you Champernowne's number.

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

**Émile Borel** — normal reals

**Champernowne**

.01234567891011121314...99100101...

It goes like this. You write down a decimal point, then you write 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, then 10, 11, 12, 13, 14 until 99, then 100, 101. And you keep going in this funny way. This is called Champernowne's number and Champernowne showed that it's normal in base ten, only in base ten. Nobody knows if it's normal in other bases, I think it's still open. In base ten though, not only will the digits 0 through 9 occur exactly 10% of the time in the limit, but each possible block of two digits will occur exactly 1% of the time in the limit, each block of three digits will occur exactly .1% of the time in the limit, etc. That's called being normal in base ten. But nobody knows what happens in other bases.

The reason I'm saying all this is because it follows from the fact that the halting probability  $\Omega$  is algorithmically irreducible information that this

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

is normal in any base. That's easy to prove using ideas about coding and compressing information that go back to Shannon. So here we finally have an example of an absolutely normal number. I don't know



how natural you think it is, but it is a specific real number that comes up and is normal in the most demanding sense that Borel could think of. Champernowne's number couldn't quite do that.

This number  $\Omega$  is in fact random in many more senses. I would say it this way. It cannot be distinguished from the result of independent tosses of a fair coin. In fact this number

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

shows that you have total randomness and chaos and unpredictability and lack of structure in pure mathematics! The same way that all it took for Turing to destroy Hilbert's dream was the diagonal argument, you just write down this expression

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

and this shows that there are regions of pure mathematics where reasoning is totally useless, where you're up against an impenetrable wall. This is all it takes. It's just this halting probability.

Why do I say this? Well, let's say you want to use axioms to prove what the bits of this number  $\Omega$  are. I've already told you that it's uncomputable—right?—like the number that Turing constructs using Cantor's diagonal argument. So we know there is no algorithm which will compute digit by digit or bit by bit this number  $\Omega$ . But let's try to prove what individual bits are using a formal axiomatic system. What happens?

The situation is very, very bad. It's like this. Suppose you have a formal axiomatic system which is  $N$  bits of formal axiomatic system (I'll explain what this means more precisely later). It turns out that with a formal axiomatic system of complexity  $N$ , that is,  $N$  bits in size, you can prove what the positions and values are of at most  $N + c$  bits of  $\Omega$ .

Now what do I mean by formal axiomatic system  $N$  bits in size? Well, remember that the essence of a formal axiomatic system is a mechanical procedure for checking whether a formal proof follows the rules or not. It's a computer program. Of course in Hilbert's days there

were no computer programs, but after Turing invented Turing machines you could finally specify the notion of computer program exactly, and of course now we're very familiar with it.

So the proof-checking algorithm which is the essence of any formal axiomatic system in Hilbert's sense is a computer program, and just see how many bits long this computer program is.<sup>1</sup> That's essentially how many bits it takes to specify the rules of the game, the axioms and postulates and the rules of inference. If that's  $N$  bits, then you may be able to prove say that the first bit of  $\Omega$  in binary is 0, that the second bit is 1, that the third bit is 0, and then there might be a gap, and you might be able to prove that the thousandth bit is 1. But you're only going to be able to settle  $N$  cases if your formal axiomatic system is an  $N$ -bit formal axiomatic system.

Let me try to explain better what this means. It means that you can only get out as much as you put in. If you want to prove whether an individual bit in a specific place in the binary expansion of the real number  $\Omega$  is a 0 or a 1, essentially the only way to prove that is to take it as a hypothesis, as an axiom, as a postulate. It's irreducible mathematical information. That's the key phrase that really gives the whole idea.

### Irreducible Mathematical Information

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

**Émile Borel** — normal reals

**Champernowne**

.01234567891011121314...99100101...

Okay, so what have we got? We have a rather simple mathematical object that completely escapes us.  $\Omega$ 's bits have no structure. There is no pattern, there is no structure that we as mathematicians can comprehend. If you're interested in proving what individual bits of this number at specific places are, whether they're 0 or 1, reasoning is completely useless. Here mathematical reasoning is irrelevant and can get

---

<sup>1</sup>Technical Note: Actually, it's best to think of the complexity of a formal axiomatic system as the size in bits of the computer program that enumerates the set of all theorems.

nowhere. As I said before, the only way a formal axiomatic system can get out these results is essentially just to put them in as assumptions, which means you're not using reasoning. After all, anything can be demonstrated by taking it as a postulate that you add to your set of axioms. So this is a worst possible case—this is irreducible mathematical information. Here is a case where there is no structure, there are no correlations, there is no pattern that we can perceive.

## 4. Randomness in arithmetic

Okay, what does this have to do with randomness in arithmetic? Now we're going back to Gödel—I skipped over him rather quickly, and now let's go back.

Turing says that you cannot use proofs to decide whether a program will halt. You can't always prove that a program will halt or not. That's how he destroys Hilbert's dream of a universal mathematics. I get us into more trouble by looking at a different kind of question, namely, can you prove that the fifth bit of this particular real number

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

is a 0 or a 1, or that the eighth bit is a 0 or a 1. But these are strange-looking questions. Who had ever heard of the halting problem in 1936? These are not the kind of things that mathematicians normally worry about. We're getting into trouble, but with questions rather far removed from normal mathematics.

Even though you can't have a formal axiomatic system which can always prove whether a program halts or not, it might be good for everything else and then you could have an **amended** version of Hilbert's dream. And the same with the halting probability  $\Omega$ . If the halting problem looks a little bizarre, and it certainly did in 1936, well,  $\Omega$  is brand new and certainly looks bizarre. Who ever heard of a halting probability? It's not the kind of thing that mathematicians normally do. So what do I care about all these incompleteness results!

Well, Gödel had already faced this problem with his assertion which is true but unprovable. It's an assertion which says of itself that it's

unprovable. That kind of thing also never comes up in real mathematics. One of the key elements in Gödel's proof is that he managed to construct an **arithmetical** assertion which says of itself that it's unprovable. It was getting this self-referential assertion to be in elementary number theory which took so much cleverness.

There's been a lot of work building on Gödel's work, showing that problems involving computations are equivalent to arithmetical problems involving whole numbers. A number of names come to mind. Julia Robinson, Hilary Putnam and Martin Davis did some of the important work, and then a key result was found in 1970 by Yuri Matijasevič. He constructed a diophantine equation, which is an algebraic equation involving only whole numbers, with a lot of variables. One of the variables,  $K$ , is distinguished as a parameter. It's a polynomial equation with integer coefficients and all of the unknowns have to be whole numbers—that's a diophantine equation. As I said, one of the unknowns is a parameter. Matijasevič's equation has a solution for a particular value of the parameter  $K$  if and only if the  $K$ th computer program halts.

In the year 1900 Hilbert had asked for an algorithm which will decide whether a diophantine equation, an algebraic equation involving only whole numbers, has a solution. This was Hilbert's tenth problem. It was tenth in his famous list of twenty-three problems. What Matijasevič showed in 1970 was that this is equivalent to deciding whether an arbitrary computer program halts. So Turing's halting problem is exactly as hard as Hilbert's tenth problem. It's exactly as hard to decide whether an arbitrary program will halt as to decide whether an arbitrary algebraic equation in whole numbers has a solution. Therefore there is no algorithm for doing that and Hilbert's tenth problem cannot be solved—that was Matijasevič's 1970 result.

Matijasevič has gone on working in this area. In particular there is a piece of work he did in collaboration with James Jones in 1984. I can use it to follow in Gödel's footsteps, to follow Gödel's example. You see, I've shown that there's complete randomness, no pattern, lack of structure, and that reasoning is completely useless, if you're interested

in the individual bits of this number

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

Following Gödel, let's convert this into something in elementary number theory. Because if you can get into all this trouble in elementary number theory, that's the bedrock. Elementary number theory, 1, 2, 3, 4, 5, addition and multiplication, that goes back to the ancient Greeks and it's the most solid part of all of mathematics. In set theory you're dealing with strange objects like large cardinals, but here you're not even dealing with derivatives or integrals or measure, only with whole numbers. And using the 1984 results of Jones and Matijasevič I can indeed dress up  $\Omega$  arithmetically and get randomness in elementary number theory.

What I get is an exponential diophantine equation with a parameter. "Exponential diophantine equation" just means that you allow variables in the exponents. In contrast, what Matijasevič used to show that Hilbert's tenth problem is unsolvable is just a polynomial diophantine equation, which means that the exponents are always natural number constants. I have to allow  $X^Y$ . It's not known yet whether I actually need to do this. It might be the case that I can manage with a polynomial diophantine equation. It's an open question, I believe that it's not settled yet. But for now, what I have is an exponential diophantine equation with seventeen thousand variables. This equation is two-hundred pages long and again one variable is the parameter.

This is an equation where every constant is a whole number, a natural number, and all the variables are also natural numbers, that is, positive integers. (Actually **non-negative** integers.) One of the variables is a parameter, and you change the value of this parameter—take it to be 1, 2, 3, 4, 5. Then you ask, does the equation have a finite or infinite number of solutions? My equation is constructed so that it has a finite number of solutions if a particular individual bit of  $\Omega$  is a 0, and it has an infinite number of solutions if that bit is a 1. So deciding whether my exponential diophantine equation in each individual case has a finite or infinite number of solutions is exactly the

same as determining what an individual bit of this

$$0 < \Omega = \sum_{p \text{ halts}} 2^{-|p|} < 1$$

halting probability is. And this is completely intractable because  $\Omega$  is irreducible mathematical information.

Let me emphasize the difference between this and Matijasevič's work on Hilbert's tenth problem. Matijasevič showed that there is a polynomial diophantine equation with a parameter with the following property: You vary the parameter and ask, does the equation have a solution? That turns out to be equivalent to Turing's halting problem, and therefore escapes the power of mathematical reasoning, of formal axiomatic reasoning.

How does this differ from what I do? I use an exponential diophantine equation, which means I allow variables in the exponent. Matijasevič only allows constant exponents. The big difference is that Hilbert asked for an algorithm to decide if a diophantine equation has a solution. The question I have to ask to get randomness in elementary number theory, in the arithmetic of the natural numbers, is slightly more sophisticated. Instead of asking whether there is a solution, I ask whether there are a finite or infinite number of solutions—a more abstract question. This difference is necessary.

My two-hundred page equation is constructed so that it has a finite or infinite number of solutions depending on whether a particular bit of the halting probability is a 0 or a 1. As you vary the parameter, you get each individual bit of  $\Omega$ . Matijasevič's equation is constructed so that it has a solution if and only if a particular program ever halts. As you vary the parameter, you get each individual computer program.

Thus even in arithmetic you can find  $\Omega$ 's absolute lack of structure,  $\Omega$ 's randomness and irreducible mathematical information. Reasoning is completely powerless in those areas of arithmetic. My equation shows that this is so. As I said before, to get this equation I use ideas that start in Gödel's original 1931 paper. But it was Jones and Matijasevič's 1984 paper that finally gave me the tool that I needed.

So that's why I say that there is randomness in elementary number theory, in the arithmetic of the natural numbers. This is an impenetrable stone wall, it's a worst case. From Gödel we knew that we couldn't

get a formal axiomatic system to be complete. We knew we were in trouble, and Turing showed us how basic it was, but  $\Omega$  is an extreme case where reasoning fails completely.

I won't go into the details, but let me talk in vague information-theoretic terms. Matijasevič's equation gives you  $N$  arithmetical questions with yes/no answers which turn out to be only  $\log N$  bits of algorithmic information. My equation gives you  $N$  arithmetical questions with yes/no answers which are irreducible, incompressible mathematical information.

## 5. Experimental mathematics

Okay, let me say a little bit in the minutes I have left about what this all means.

First of all, the connection with physics. There was a big controversy when quantum mechanics was developed, because quantum theory is nondeterministic. Einstein didn't like that. He said, "God doesn't play dice!" But as I'm sure you all know, with chaos and nonlinear dynamics we've now realized that even in classical physics we get randomness and unpredictability. My work is in the same spirit. It shows that pure mathematics, in fact even elementary number theory, the arithmetic of the natural numbers, 1, 2, 3, 4, 5, is in the same boat. We get randomness there too. So, as a newspaper headline would put it, God not only plays dice in quantum mechanics and in classical physics, but even in pure mathematics, even in elementary number theory. So if a new paradigm is emerging, randomness is at the heart of it. By the way, randomness is also at the heart of quantum field theory, as virtual particles and Feynman path integrals (sums over all histories) show very clearly. So my work fits in with a lot of work in physics, which is why I often get invited to talk at physics meetings.

However the really important question isn't physics, it's mathematics. I've heard that Gödel wrote a letter to his mother who stayed in Europe. You know, Gödel and Einstein were friends at the Institute for Advanced Study. You'd see them walking down the street together. Apparently Gödel wrote a letter to his mother saying that even though Einstein's work on physics had really had a tremendous impact on how

people did physics, he was disappointed that his work had not had the same effect on mathematicians. It hadn't made a difference in how mathematicians actually carried on their everyday work. So I think that's the key question: How should you really do mathematics?

I'm claiming I have a much stronger incompleteness result. If so maybe it'll be clearer whether mathematics should be done the ordinary way. What is the ordinary way of doing mathematics? In spite of the fact that everyone knows that any finite set of axioms is incomplete, how do mathematicians actually work? Well suppose you have a conjecture that you've been thinking about for a few weeks, and you believe it because you've tested a large number of cases on a computer. Maybe it's a conjecture about the primes and for two weeks you've tried to prove it. At the end of two weeks you don't say, well obviously the reason I haven't been able to show this is because of Gödel's incompleteness theorem! Let us therefore add it as a new axiom! But if you took Gödel's incompleteness theorem very seriously this might in fact be the way to proceed. Mathematicians will laugh but physicists actually behave this way.

Look at the history of physics. You start with Newtonian physics. You cannot get Maxwell's equations from Newtonian physics. It's a new domain of experience—you need new postulates to deal with it. As for special relativity, well, special relativity is almost in Maxwell's equations. But Schrödinger's equation does not come from Newtonian physics and Maxwell's equations. It's a new domain of experience and again you need new axioms. So physicists are used to the idea that when you start experimenting at a smaller scale, or with new phenomena, you may need new principles to understand and explain what's going on.

Now in spite of incompleteness mathematicians don't behave at all like physicists do. At a subconscious level they still assume that the small number of principles, of postulates and methods of inference, that they learned early as mathematics students, are enough. In their hearts they believe that if you can't prove a result it's your own fault. That's probably a good attitude to take rather than to blame someone else, but let's look at a question like the Riemann hypothesis. A physicist would say that there is ample experimental evidence for the Riemann hypothesis and would go ahead and take it as a working assumption.

What is the Riemann hypothesis? There are many unsolved ques-



tions involving the distribution of the prime numbers that can be settled if you assume the Riemann hypothesis. Using computers people check these conjectures and they work beautifully. They're neat formulas but nobody can prove them. A lot of them follow from the Riemann hypothesis. To a physicist this would be enough: It's useful, it explains a lot of data. Of course a physicist then has to be prepared to say "Oh oh, I goofed!" because an experiment can subsequently contradict a theory. This happens very often.

In particle physics you throw up theories all the time and most of them quickly die. But mathematicians don't like to have to backpedal. But if you play it safe, the problem is that you may be losing out, and I believe you are.

I think it should be obvious where I'm leading. I believe that elementary number theory and the rest of mathematics should be pursued more in the spirit of experimental science, and that you should be willing to adopt new principles. I believe that Euclid's statement that an axiom is a self-evident truth is a big mistake. The Schrödinger equation certainly isn't a self-evident truth! And the Riemann hypothesis isn't self-evident either, but it's very useful.

So I believe that we mathematicians shouldn't ignore incompleteness. It's a safe thing to do but we're losing out on results that we could get. It would be as if physicists said, okay no Schrödinger equation, no Maxwell's equations, we stick with Newton, everything must be deduced from Newton's laws. (Maxwell even tried it. He had a mechanical model of an electromagnetic field. Fortunately they don't teach that in college!)

I proposed all this twenty years ago when I started getting these information-theoretic incompleteness results. But independently a new school on the philosophy of mathematics is emerging called the "quasi-empirical" school of thought regarding the foundations of mathematics. There's a book of Tymoczko's called *New Directions in the Philosophy of Mathematics* (Birkhäuser, Boston, 1986). It's a good collection of articles. Another place to look is *Searching for Certainty* by John Casti (Morrow, New York, 1990) which has a good chapter on mathematics. The last half of the chapter talks about this quasi-empirical view.

By the way, Lakatos, who was one of the people involved in this new movement, happened to be at Cambridge at that time. He'd left

Hungary.

The main schools of mathematical philosophy at the beginning of this century were Russell and Whitehead's view that logic was the basis for everything, the formalist school of Hilbert, and an "intuitionist" constructivist school of Brouwer. Some people think that Hilbert believed that mathematics is a meaningless game played with marks of ink on paper. Not so! He just said that to be absolutely clear and precise what mathematics is all about, we have to specify the rules determining whether a proof is correct so precisely that they become mechanical. Nobody who thought that mathematics is meaningless would have been so energetic and done such important work and been such an inspiring leader.

Originally most mathematicians backed Hilbert. Even after Gödel and even more emphatically Turing showed that Hilbert's dream didn't work, in practice mathematicians carried on as before, in Hilbert's spirit. Brouwer's constructivist attitude was mostly considered a nuisance. As for Russell and Whitehead, they had a lot of problems getting all of mathematics from logic. If you get all of mathematics from set theory you discover that it's nice to define the whole numbers in terms of sets (von Neumann worked on this). But then it turns out that there's all kinds of problems with sets. You're not making the natural numbers more solid by basing them on something which is more problematical.

Now everything has gone topsy-turvy. It's gone topsy-turvy, not because of any philosophical argument, not because of Gödel's results or Turing's results or my own incompleteness results. It's gone topsy-turvy for a very simple reason—the computer!

The computer as you all know has changed the way we do everything. The computer has enormously and vastly increased mathematical experience. It's so easy to do calculations, to test many cases, to run experiments on the computer. The computer has so vastly increased mathematical experience, that in order to cope, people are forced to proceed in a more pragmatic fashion. Mathematicians are proceeding more pragmatically, more like experimental scientists do. This new tendency is often called "experimental mathematics." This phrase comes up a lot in the field of chaos, fractals and nonlinear dynamics.

It's often the case that when doing experiments on the computer,

numerical experiments with equations, you see that something happens, and you conjecture a result. Of course it's nice if you can prove it. Especially if the proof is short. I'm not sure that a thousand page proof helps too much. But if it's a short proof it's certainly better than not having a proof. And if you have several proofs from different viewpoints, that's very good.

But sometimes you can't find a proof and you can't wait for someone else to find a proof, and you've got to carry on as best you can. So now mathematicians sometimes go ahead with working hypotheses on the basis of the results of computer experiments. Of course if it's physicists doing these computer experiments, then it's certainly okay; they've always relied heavily on experiments. But now even mathematicians sometimes operate in this manner. I believe that there's a new journal called the *Journal of Experimental Mathematics*. They should've put me on their editorial board, because I've been proposing this for twenty years based on my information-theoretic ideas.

So in the end it wasn't Gödel, it wasn't Turing, and it wasn't my results that are making mathematics go in an experimental mathematics direction, in a quasi-empirical direction. The reason that mathematicians are changing their working habits is the computer. I think it's an excellent joke! (It's also funny that of the three old schools of mathematical philosophy, logicist, formalist, and intuitionist, the most neglected was Brouwer, who had a constructivist attitude years before the computer gave a tremendous impulse to constructivism.)

Of course, the mere fact that everybody's doing something doesn't mean that they ought to be. The change in how people are behaving isn't because of Gödel's theorem or Turing's theorems or my theorems, it's because of the computer. But I think that the sequence of work that I've outlined does provide some theoretical justification for what everybody's doing anyway without worrying about the theoretical justification. And I think that the question of how we should actually do mathematics requires **at least** another generation of work. That's basically what I wanted to say—thank you very much!

## **Bibliography**

- [1] G. J. Chaitin, *Information-Theoretic Incompleteness*, World Scientific, 1992.
- [2] G. J. Chaitin, *Information, Randomness & Incompleteness*, second edition, World Scientific, 1990.
- [3] G. J. Chaitin, *Algorithmic Information Theory*, revised third printing, Cambridge University Press, 1990.

# ELEGANT LISP PROGRAMS<sup>1</sup>

in C. Calude, Theoretical Computer Science: People and Ideas, Springer Verlag Singapore, in press

G. J. Chaitin, IBM Research, P. O. Box 704, Yorktown Heights, NY 10598, chaitin@watson.ibm.com

## Abstract

*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.*

---

<sup>1</sup>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

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

FAS  $\longrightarrow$  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.

FAS (S-expression)  $\longrightarrow$  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 out-do 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  $H(\cdot)$  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 $\Omega$

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  $\Omega$  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  $\Omega$ . 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  $\Omega$  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  $\Omega$  is irreducible algorithmic information. The precise result turns out to be this:

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

What's  $\Omega_N$ ? 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  $\Omega$  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  $\Omega$ , a set of axioms can't enable enable you to determine substantially more bits of  $\Omega$  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  $\Omega$  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  $\Omega$  into a program substantially smaller than the number of bits of  $\Omega$  you calculate, you can't do it using reasoning either. Essentially the only way to get bits of  $\Omega$  out of a formal axiomatic system, is if you just add those bits as axioms! So determining bits of  $\Omega$  is a losing proposition! Mathematical reasoning doesn't help at all. Well, that's not quite right, because of the constant 15328.

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

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  $\Omega$ , 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  $\Omega$  shows, is that sometimes— $\Omega$ '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  $\Omega$  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  $\Omega$ . As I said,  $\Omega$  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  $\Omega$  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  $\Omega$  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  $\Omega$  are. But if you wanted to do that, it would be completely hopeless. Because, you see, the bits of  $\Omega$  aren't 0 or 1 for any particular reason—they've got to be one or the other, it's a specific  $\Omega$ , 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  $\Omega$  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!

# AN INVITATION TO ALGORITHMIC INFORMATION THEORY<sup>1</sup>

in D. Bridges, C. Calude, Proceedings of DMTC-S'96, Springer Verlag Singapore, 1997, pp. 1–23

G. J. Chaitin, IBM Research, P. O. Box 218, Yorktown Heights, NY 10598, [chaitin@watson.ibm.com](mailto:chaitin@watson.ibm.com)

## Abstract

*I'll outline the latest version of my limits of math course. The purpose of this course is to illustrate the proofs of the key information-theoretic incompleteness theorems of algorithmic information theory by means of algorithms written in a specially designed version of LISP. The course is now available at*

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

*together with a LISP interpreter written in Mathematica.*

---

<sup>1</sup>Lecture given Wednesday 24 April 1996 at a Computer Science Colloquium at the University of New Mexico. The lecture was videotaped; this is an edited transcript prepared for DMTC'S'96.



## Introduction

Hi everybody! It's a great pleasure being back in this beautiful, beautiful state. You guys were nice enough to invite me up a bunch of times in the past. And I've usually tried to explain general ideas. I thought this time I'd try to give a different kind of a talk.

I've been working for several years on a project, on a course I call "The Limits of Mathematics." And this course in the past three years has gone through a number of different versions. It has a lot of software attached to it. And I've been changing this software. I've been trying to explain this software to people as the course. In fact a few weeks from now I'll be giving this course in Rovaniemi, Finland, on the arctic circle, for two weeks.

The way this course is now is that it's on this Web site that I gave you guys the URL for in the abstract. It's an HTML document and it has a lot of LISP code in it, and just to make life worse this is not normal LISP. It's pretty close to pure LISP, you know, the heart of LISP that everybody learns. But I've had to add a few features to LISP. So this stuff is all available and you're welcome to play with it. The course is there.

And as general background I suggest my lecture on "Randomness in Arithmetic and the Decline and Fall of Reductionism in Pure Mathematics" which was published, among other places, by Oxford University Press in *Nature's Imagination*, that's the name of the volume. That's a talk that I gave here.

## Recursive function theory revisited

My work basically is doing recursive function theory at the kind of level it was done in the early days in the 1930's before it got technical. At the kind of level that Gödel and Turing were doing it where the concern was what are the limits of mathematical reasoning. And basically I've added two new things. This is sort of revisited sixty years later, this stuff. I've added two new things.

One new thing I add to the stew is that an idea that was missing which is significant is the idea of program-size complexity, or how big

a program is, how many bits of information there are. Not the time! In that sense it's just like the old recursive function theory and Turing machines. I don't care about time. But I do care about the size of a program in bits. Okay? So that's one new idea.

The other new idea I've added just in the past three years is that I want to really be able to run the programs on interesting examples on a computer. In other words, it's not enough to wave your hands and talk about an algorithm. I not only want a universal Turing machine that I can prove theorems with, but one that is fun to actually program and that you can run interesting examples on with a computer.

John McCarthy invented LISP not just for use in AI [artificial intelligence], not just for use as a practical language. If you look at his 1960 paper on LISP in the *Communications of the ACM*, he said, "This is a better universal Turing machine. Let's do recursive function theory that way!"<sup>2</sup> And the funny thing is that nobody except me has really I think taken that seriously. And the reason of course is that theoreticians didn't really much care about programming, about really playing with the computer. So the thing I've added is I think nowadays you have no excuse! If a theory has to do with the size of computer programs, you want to damn well see the programs, you want to be able to run them, it ought to be a programming language that is easy to use.

So I've done that using LISP because LISP is simple enough, LISP is in the intersection between theoretical and practical programming. Lambda calculus is even simpler and more elegant than LISP, but it's unusable. Pure lambda calculus with combinators  $S$  and  $K$ , it's beautifully elegant, but you can't really run programs that way, they're too slow. So I would say that I believe that LISP is a powerful language that can actually be used on a computer to run stuff in a finite amount of time, but on the other hand it's simple enough and elegant enough that you can prove theorems about it. So that's the kind of game that I've been playing. Okay, so you'll find this on my Web site.

---

<sup>2</sup>Of course these aren't McCarthy's words, it's my paraphrase.

## My new LISP & complexity measure

Okay, so let me leap in and give you the general idea. What is this big new thing you add to LISP, this change you make to LISP to be able to talk about program-size complexity? Well, you could just take LISP expressions and measure how big are they in characters, right? That's not a bad complexity measure. But it's not the best complexity measure.

The best complexity measure says, you have a LISP S-expression, which is a computer program, and then you have binary data on the side:

<code>lisp</code>	<code>binary</code>
<code>S-exp</code>	<code>data</code>

The problem with measuring the size of a LISP program is that due to LISP syntax the LISP S-expression is redundant. So you really want to add to the LISP S-expression, which is a powerful way to express an algorithm, some raw binary data. That's like having data on a tape in addition to your program. So in the data the bits can be 0 and 1 independently, and there's some way that the program can get access to the data, and then you look at the size of the whole thing.

So basically the universal Turing machine I'm using as my yardstick to talk about program size, to use to measure program size, has programs which are big binary programs:

**BINARY PROGRAM**

<code>lisp</code>	<code>binary</code>
<code>S-exp</code>	<code>data</code>

That sounds bad, right? We have this awful machine language in binary. You're not going to want to write in binary. Well, it's not so bad. The beginning of a program is a LISP S-expression converted to binary, 8 bits per character. And you know where the LISP S-expression ends, because by convention there is always a special ASCII character at the end.

```
BINARY PROGRAM
lisp      binary
S-exp     data
8 bits/char
```

So the universal Turing machine starts reading the binary tape bit by bit. It doesn't sort of gobble it all in and start running. It reads each bit of the program as it needs to—that's very important. And it starts off by reading a LISP S-expression, and then every time the LISP S-expression says "Read another bit," it gets it from what's left on the tape, that is, in the binary program. And if the S-expression attempts to read beyond the end of the binary data, the program is a failure and is aborted. So that's how it goes. And then you just add the total number of bits, which is 8 times the number of characters in the S-expression plus the number of bits in the data. That gives you the total number of bits in the program, okay?

So that's done because just using the size in characters of a LISP S-expression (or multiplying it by some factor to have it in bits) is no good. LISP syntax is pretty simple, but even so there's redundancy there. So you've got to add this extra binary data on the side.

## Programming my UTM in LISP

So let me show you my universal Turing machine programmed out in my new LISP. When I came here a year ago, I had a LISP, it was pretty awful. It was a LISP where atoms and variable names were only one character long. And it was a LISP where there was no arithmetic. The way you had to do integers was as lists of 0's and 1's and you had to define, you had to program out arithmetic. I thought this was cute, this is a fun game to play once in your life, but the problem is it's an obstacle if I want to explain this to anybody else! So now I have a LISP like all other LISP's with long names for atoms, and my universal Turing machine is now programmed like this.

Define  $U$  of  $p$ ,  $U$  is the function,  $p$  is the program, as follows:

```

define (U p)
  cadr try no-time-limit
        'eval read-exp
        p

```

Now this is it, this is my universal Turing machine that I described in words. Try is a primitive function in this LISP sort of like eval in normal LISP. Try has three arguments, here they are:

```

no-time-limit
'eval read-exp
p

```

So this, argument 2,

```
'eval read-exp
```

is a LISP expression that we're going to evaluate with a time limit. In fact there's no time limit here:

```
no-time-limit
```

You also have associated binary data. Try is the way that you give a LISP expression binary data. So this

```
p
```

is the binary data, it's the program for the universal Turing machine. It's a list of 0's and 1's. This is try's third argument.

Oh by the way, in normal LISP you have to put lots of parentheses. In my LISP I leave out a lot of them because in my LISP every built-in function has a fixed number of arguments, so I don't bother to write all the parentheses.

Back to the LISP code for my UTM.

```
(read-exp)
```

is a primitive function with no arguments. So we're trying to evaluate this

```
('(eval(read-exp)))
```

normally with a time limit, but in this case with no time limit. And `read-exp` reads a LISP expression from the beginning of the binary data `p`. And then we evaluate it, in other words, we run it. And while you're running it, if within the LISP expression that you read you ask for more binary data, you get it from what's left of the binary data.

It works! `Try` does exactly what it should, I added `try` to normal LISP just for this. And `try` is the real difference between normal LISP and my LISP. Okay, so this

```
(define (U p)
  (cadr (try no-time-limit
            ('(eval(read-exp)))
            p)))
```

is my universal Turing machine, which I think is not a very big, complicated program.

And the nice thing about this is that you see lots of examples in my course. What I do is I define LISP functions then I run them on examples. And there are lots of comments saying why I'm doing it and what it proves.

Actually there are two versions of each of these LISP runs, one of them with no comments and running it just on the case that counts, and another one with all the comments in the world and with lots of test cases for all the auxiliary functions. I do this because without comments a program is incomprehensible, but when you put all the comments in it becomes incomprehensible for another reason—it's so big that you can't find anything. So I present it both ways.

Okay, so one of the things you can do is you can actually get a program and run it on this machine. So how do you do it? Well, you take a LISP expression, and then you want to convert it to binary, so I have a primitive function for doing that, and then you concatenate it to the binary data that it needs, and then you feed this into the function `U` of `p` that we defined. And in my Web site I have runs of the universal Turing machine that show how it works.

## A simple program for $U$ that proves a theorem

Let me show you a particularly interesting example of this, and I'm also going to prove a theorem, a very simple theorem about my version of program-size complexity. This theorem says that the program-size complexity of a pair of objects is bounded by the sum of the individual program-size complexities of the two objects plus a constant:

$$H(x, y) \leq H(x) + H(y) + c.$$

What this means is this.  $H(x)$  is the size in bits of the smallest program for the universal machine  $U$  of  $p$  that calculates  $x$ ;  $x$  is an S-expression. And  $H(y)$  is the size of the one that calculates  $y$ . And putting them together and adding a fixed number of bits you get a program that calculates the pair  $(x, y)$ , which won't have a comma by the way, since we're doing LISP, which is without commas.

So how does this work? Well, let me show you what the program looks like. What you do is you take the minimum size program for  $x$ , the minimum size program for  $y$ , and I'll call that  $x^*$  and  $y^*$ , and you concatenate in front a prefix  $\phi_c$ , a magic prefix that's going to be  $c$  bits long, and I'll show you this prefix. This gives you

$$\phi_c x^* y^*.$$

Then when you give it to the universal machine  $U$  of  $p$  it's going to produce the pair  $(x, y)$ .

Now what is this prefix  $\phi_c$  that you're concatenating in front of these two bit strings  $x^*$  and  $y^*$ ? Actually you can put any program for  $x$  and any program for  $y$  here in  $\phi_c x^* y^*$ . I don't know the minimum ones. You'll see that this prefix works because if you give it any program  $x^*$  to calculate  $x$  and any program  $y^*$  to calculate  $y$ , this prefix  $\phi_c$  when you feed it to my universal machine  $U$  of  $p$  is going to end up being a program that calculates the pair  $(x, y)$ .

So what is this prefix  $\phi_c$ ? Well, the prefix is this. Start with

```
(cons (eval (read-exp))
      (cons (eval (read-exp))
            nil))
```

Here I'm putting in all the parentheses. Then convert this to a bit string  $\phi_c$ . So you write this expression and use a primitive function to convert it to a bit string and then you use `append` to concatenate these lists of bits and get this long bit string:

$$\phi_c x^* y^*.$$

And then you feed it into  $U$  and it produces  $(x, y)$ .

How does this

```
(cons (eval (read-exp))
      (cons (eval (read-exp))
            nil))
```

work? It says, “Read from the binary data.” The binary data when  $\phi_c$  is running is going to be  $x^*y^*$ , the rest of the program. So it's going to read and run the program  $x^*$  to calculate  $x$ , and that's going to give it  $x$ . And then it's going to read off of the rest of the binary data and run the program  $y^*$  to calculate  $y$ , and that's going to give it  $y$ . And then it's going to `cons` these two things  $x$  and  $y$  up into a pair. Okay?

So it's the number of bits in this S-expression  $\phi_c$

```
(cons (eval (read-exp))
      (cons (eval (read-exp))
            nil))
```

which is the constant  $c$  in

$$H(x, y) \leq H(x) + H(y) + c.$$

So count the number of characters here in  $\phi_c$ , multiply by 8, and that's the constant  $c$  in this theorem. Okay? So this is pretty straight-forward.

## “Computing” the halting probability $\Omega$

Now, what do I do next in my course on my Web site? Okay, so I've defined this universal Turing machine  $U$  of  $p$ . The next major step in my theory is to define the halting probability  $\Omega$ . This is a number that a



lot of people get excited about [laughter], the halting probability. Well, I'm going to write down a program to calculate the halting probability now. Let me actually write it down for you! Here it goes. Let's write this program out. So I'm going to write a LISP program as we go.

Okay, so we're going to define a function that counts how many programs halt, let's call it `count-halt`, which has two arguments, one is a prefix, and the other is `bits-left`.

```
define (count-halt prefix bits-left)
```

What's the idea? I'm going to count how many programs for my universal Turing machine halt... Oh, I should put in time... Let me put in time, there's another argument here:

```
define (count-halt time prefix bits-left)
```

So I'm going to count how many programs halt on my universal Turing machine that have this prefix with `bits-left` bits added on afterwards, within this amount of time. So this `count-halt` is the auxiliary function that I use. Okay?

So the first thing to show you is how you get from this the halting probability. So before I define the auxiliary function, let me show you how to define a function `omega`, the main function, that uses this auxiliary function. It's only one main function with one auxiliary function. Let's call it `omega` of  $n$ :

```
define (omega n)
```

This is the  $n$ th approximation, the  $n$ th lower bound on the halting probability. As  $n$  goes to infinity, this will give you the halting probability in the limit from below. The problem is it converges very, very, very, very slowly! [Laughter] Noncomputably slowly! Thanks for the laughs.

Let's define this function. Well, what I want to do is I want to cons up a rational number. And the way I do it is, I count how many programs halt within time  $n$  that have as prefix `nil`, the empty bit string, and that add  $n$  bits to it.

```
define (omega n)
  cons (count-halt n nil n)
```

And I cons that with division.

```
define (omega n)
  cons (count-halt n nil n)
  cons /
```

And cons that with, I use  $\wedge$  for power, 2 raised to the power  $n$ , and nil.

```
(define (omega n)
  (cons (count-halt n nil n)
    (cons /
      (cons ( $\wedge$  2 n)
        nil))))
```

This depends on the representation you pick for rational numbers; I just write out a fraction. And so (omega  $n$ ) is a triple, I'm creating a triple. So it's (the number of programs  $n$  bits in size that halt within time  $n$ ) divided by  $2^n$ . Okay?

So that's the main function. Now let me define the auxiliary function. The auxiliary function goes like this. You start off with the empty prefix nil and recursively you start adding bits until you get a full  $n$ -bit program and then you run it for time  $n$  and you see whether it halts. Now how do you program that out in LISP? Well, recursively it's very easy, you go like this.

Let's first see if bits-left is not equal to 0. If bits-left is greater than 0, then add a 0 bit to the prefix and subtract one from the number of bits left, see how many of these programs halt within time  $n$ , and add that to the same thing when you append a 1 to the prefix instead of a 0.

So we've got this:

```
define (count-halt time prefix bits-left)
  if > bits-left 0
  + (count-halt time (append prefix '(0)) (- bits-left 1))
    (count-halt time (append prefix '(1)) (- bits-left 1))
```

I'll put parentheses in occasionally and other times not. [Laughter] Well, actually in my LISP you don't put them in for primitive functions

and the interpreter adds them, and writes it out for you just so that you can check that it's what you meant.

So this is the recursion. This is going to make `count-halt` look at all  $n$ -bit programs. And what happens when it finally has no bits left to add? It started off with  $n$  bits to append to the prefix, and with the empty prefix. So now you've finally got an  $n$ -bit string, you get all possible  $n$ -bit strings. So then you take the  $n$ -bit string, and you try for time  $n$  `eval read-exp` applied to the  $n$ -bit data string which is the prefix. If this try is a success then `count-halt` is a 1 otherwise it's a 0.

So now we've got this:

```
define (count-halt time prefix bits-left)
  if > bits-left 0
  + (count-halt time (append prefix '(0)) (- bits-left 1))
    (count-halt time (append prefix '(1)) (- bits-left 1))
  if = success car (try time
                        'eval read-exp
                        prefix)
    1
    0
```

This recursion is going out and seeing whether all possible  $n$ -bit programs halt or not. When you've got the whole  $n$ -bit program, how do you see whether it halts or not? You try running it for the time that you were given, which is going to be  $n$  for omega of  $n$ . And this part

```
(try time
  'eval read-exp
  prefix)
```

does that. It's taking the prefix as the S-expression and the binary data of a program for my universal Turing machine. It's reading an S-expression off of the beginning of the prefix and it's running the S-expression with the rest of the prefix as binary data. Okay?

I don't really have the time to explain this properly. If you were giving this to students in a class you'd probably take a whole class to

explain this. And then the students would go and they would play with it and they would use it. And if you look at my Web site you'll find that I'm running `count-halt` on examples to show that it works. I also have a different definition of  $\Omega$  of  $n$  which is more traditional, which builds up the list of all  $n$ -bit strings, and then sees which halt, and then counts how many halt. But I think that what I have here is more elegant, it's really all you need.

Let me say it in words again.

What you see here in this version of LISP is the  $n$ th lower bound on  $\Omega$ . With increasing positive integers  $n$ ,  $\Omega$  of  $n$  gives you better and better lower bounds on the halting probability. The problem is that you never know how far out to go to get the halting probability with a given degree of accuracy. To get  $n$  bits of the halting probability right, you have to go out Busy Beaver function of  $n$ , that is basically the idea.

You look at all  $n$ -bit programs, as prefix you start off with the empty list of bits, and the bits left to add is  $n$  initially, and you're going to count how many of them halt within time  $n$ . That's what

```
(count-halt n nil n)
```

does.

## Discussion of the program for $\Omega$

Let's look at this conceptually and let's look at it pedagogically. Pedagogically, I certainly learned a lot writing this program, right? I had to invent the programming language, I had to write the interpreter! So I learned a lot from this. Is this a good way to transmit an idea to a student? I don't know! Reading somebody else's code is not great, right?

You really understand an algorithm when you've programmed it and tried it on examples and debugged it yourself. So I think that to use this in a class effectively you need exercises. You need to ask the students to do variations on what you've presented, so they make it their own. I don't do this here. This is a very concise version of the course.

If someone really wanted to use this as a course, they could start with this, which it is at least feasible that someone could understand. Before it wasn't. I could understand it. A few very committed people could sort of try to think they understood it, but I think there's more of a chance with this, it looks more like normal LISP. So that's from the pedagogic point of view.

Now let's look at this philosophically. The point is this. This program is  $\Omega$ 's definition. This is really it. So if this number is going to be very mind-boggling, you'd like to pin down very concretely how it's defined. And we have.

You'd like to pin down very concretely how  $\Omega$ 's defined before you start being astonished by it. Here  $\Omega$  is really very concrete and you can actually run ( $\omega$  0), ( $\omega$  1), ( $\omega$  2), ... Of course the time grows exponentially as  $n$  goes up, but maybe you can get to one interesting example. You can try variations of this program to convince yourself that you understand it. And you can try debugging `count-halt` separately by running examples to show that it works. That's how to convince yourself that this works: run lots of examples. I haven't tried giving a formal proof that this program is correct. I wonder if that would be an interesting project?

Okay, now I once tried explaining this to an astrophysicist who amazingly enough was briefly interested [laughter], a very bright guy. He was so bright that he could understand the previous version of the program for  $\Omega$ , the one in which the names of built-in functions are a single character, and where you had to program out arithmetic, you didn't have positive integers like we do here. He took the definition of ( $\omega$   $n$ ) and `count-halt`, printed them on a small sheet of paper, folded it up and put it in his wallet. He said that it was like a mantra, "I have  $\Omega$  in my pocket!" But it was  $\Omega$  for him only because he was very bright and he took the trouble to go through this course with me. I think that this new version is much easier to understand than the one that he put in his pocket.

## Proof that $\Omega$ is algorithmically irreducible

Okay, so what is the next thing that one does in this course on the limits of mathematics? I've tried to throw out all the inessential stuff and just give the main ideas. And now we'll start to see why this number  $\Omega$  is significant.

Well, the next thing I do in the course is I point out that  $(\omega_0)$ ,  $(\omega_1)$ ,  $(\omega_2)$ , ... is a monotone increasing sequence of rational numbers. So, this isn't constructive, but we all know that there is a real number which is the least upper bound of this sequence. So  $(\omega_n)$  defines in a nonconstructive way a real number  $\Omega$ . It's almost constructive. It's pretty close to being constructive. Really constructive would be if you could calculate each digit of  $\Omega$ , right? I'm getting better and better lower bounds, the only thing I don't have is I don't have a way to calculate how far out to go to get a given degree of accuracy.

Okay, so this defines a real number that I call  $\Omega$ . Now let's imagine this number being written in base-two binary, so there's going to be a 0, a binary point, and then there're going to be some bits, right?

$$\Omega = 0.011101\dots$$

And let's think of this bit string as a LISP S-expression, it's going to be a list of 0's and 1's separated by blanks. And let's define  $\Omega_N$  to be the first  $N$  bits of this, the LISP S-expression for the first  $N$  bits of  $\Omega$ . It's a list with  $N$  elements.

The theorem you want to show is that  $H(\Omega_N)$  is pretty complicated, that there's a lot of information in those first  $N$  bits of  $\Omega$ . In fact it's algorithmically irreducible. You cannot compress the first  $N$  bits of  $\Omega$  into a program substantially less than  $N$  bits in size. The exact result is that

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

This turns out to be the case.

Why is there a lot of information in the first  $N$  bits of  $\Omega$ ? It's not difficult to see why. If you knew the first  $N$  bits of  $\Omega$ , it solves the halting problem. It would enable you, slowly, but it would enable you to solve the halting problem for all programs up to  $N$  bits in size. So

once you do that, you run all the programs up to  $N$  bits in size that halt, you see what they calculate, and then you just put what they calculate together into a list. That gives you something which cannot be done by a program less than or equal to  $N$  bits in size.

And it takes 8000 bits to explain how to do this. There's a prefix  $\phi_{8000}$  which is 8000 bits long that if you put it in front of a program that calculates the first  $N$  bits of  $\Omega$ , this gives you a program to calculate something whose complexity is greater than  $N$ . This shows rather concretely that this inequality

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

has to follow. Okay?

So I'm giving my universal Turing machine  $U$  a program which starts with an 8000-bit prefix  $\phi_{8000}$ . That sounds like a lot, but in fact it's only a thousand characters of LISP. And then next to it I concatenate any program  $\Omega_N^*$  for calculating the first  $N$  bits of  $\Omega$ , if somehow I had one. I don't know where to get one! So this is what we've got:

$$U(\phi_{8000} \Omega_N^*).$$

And then you run this and what the machine  $U$  does because of  $\phi_{8000}$  is this. First it reads in and runs  $\Omega_N^*$  to calculate the first  $N$  bits of  $\Omega$ . Then it uses (omega  $n$ ) to get better and better lower bounds on  $\Omega$  until  $\Omega_N$  is correct, until it gets the first  $N$  bits of  $\Omega$  right. God, or an oracle, is giving us the program for the first  $N$  bits of  $\Omega$ !

Once  $U$  has calculated (omega  $n$ ) for  $n$  high enough that the first  $N$  bits of  $\Omega$  are correct, at that point it's seen all  $N$ -bit programs that are ever going to halt. In fact, if they ever halt, they halt within time  $n$ .

And then  $U$  sees what these programs produce, and it forms the list of the output of all programs up to  $N$  bits in size that halt. And to finish up everything,  $U$  outputs this list and halts. In other words, this list is the value of

$$U(\phi_{8000} \Omega_N^*).$$

It's precisely the list of all LISP S-expressions  $x$  with program-size complexity  $H(x) \leq N$ .

This list cannot itself have program-size complexity  $\leq N$ , because it can't be contained within itself. So it can't be the output of a program less than or equal to  $N$  bits in size, it's got to have program-size complexity greater than  $N$ . Therefore this program  $\phi_{8000} \Omega_N^*$  for producing the list

$$U(\phi_{8000} \Omega_N^*)$$

must be greater than  $N$  bits in size. In other words, (the size of any program that calculates the first  $N$  bits of  $\Omega$ ) plus 8000 has got to be greater than  $N$ . So you get this inequality

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

Okay?

And in my Web site I actually show you this program. I program out in LISP the algorithm that I just described in words. You program it out, it's not a big deal,  $\phi_{8000}$  is about one page of LISP, of my new LISP. But if you put in a lot of comments and run lots of examples then it's more than one page. Okay?

Martin Gardner had an explanation of this algorithm in an article in *Scientific American* on  $\Omega$  that you'll find in one of the Martin Gardner collections. In my Web site this algorithm is actually written out in LISP and you can run it on some examples. You can certainly run the auxiliary functions and other pieces of the algorithm and convince yourself that it works. But when you put it all together, you're not usually going to be able to run it on the real data  $\Omega_N$ , because the whole point of this is to show that the bits of  $\Omega$  are hard to know, they're algorithmically irreducible.

By the way, it's easy to see that

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

implies that  $\Omega$  is violently non-computable. Let's compare the two real numbers  $\Omega$  and  $\pi$ .  $\pi$  has the property that the string of its first  $N$  bits has very small program-size complexity. Given  $N$ , you can calculate the first  $N$  bits of  $\pi$ . So the first  $N$  bits of  $\pi$  only have about  $\log_2 N$  bits of complexity. But the first  $N$  bits of  $\Omega$  are irreducible, can be reduced at most 7999 bits, maybe down to  $N - 7999$ , we've proved that.



By the way, program-size irreducibility implies statistical randomness. So  $\Omega$ 's irreducibility implies that  $\Omega$  is a normal real number. In any base, its digits all have the same limiting relative frequency.

## Proof that you can't even deduce what the bits of $\Omega$ are

Okay, so the next thing you do with this, and this is the next (and last) program in this course, is you get an incompleteness result for  $\Omega$  from this. From this inequality

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

which states that  $\Omega$  is algorithmically irreducible. By the way, I don't show in the course that this implies statistical randomness. That's a detour. I want to go straight to the incompleteness result.

(But I think you can sort of wave your hands and try to convince people that irreducibility implies statistical randomness. Or you can actually go and develop algorithmic information theory in excruciating detail! But in my course my goal is to get to the major incompleteness results as quickly as possible, illustrating them with interesting LISP programs, not to develop the entire theory. Now you could also take the proof of every theorem in my Cambridge University Press book *Algorithmic Information Theory* and write down a LISP program for the algorithm in it. I once started doing that but I quickly gave it up. It was too much work, and you don't really want to see every algorithm in the theory in such detail. I'm concentrating on programming in LISP what I think are the fun algorithms in the theory, especially the ones connected with the basic incompleteness results, the ones I want to use to present what I think are the fundamental concepts in my theory.)

So the last program in this course is the one that yields an incompleteness result for  $\Omega$ .

Let's start with this question: How do I represent a formal axiomatic system in LISP? Well, this is the way I do it. Think of it as a black box which every now and then throws out a theorem. So it's a program that you start running, and it may or may not halt, normally it doesn't halt, and every now and then it prints out a theorem.

So why is this a formal axiomatic system? Well, the idea is that I don't care about the details of what's going on inside, what the axioms are, or what the logic used is like. What I care about is the criterion, which I think was enunciated rather clearly by Hilbert, that states that the essence of a formal axiomatic system is that there should be a proof-checking algorithm. So if that's the case, you can run through all possible proofs in size order, see which ones are correct, and print out all the theorems. Of course this is impractical. The time it takes to do it grows exponentially, and I wouldn't really want to run such a program.

But you can cheat and you can just have a LISP S-expression which every now and then outputs a theorem... It won't be the final value... Every now and then it uses a primitive function, I call it `display`, but you could also say "output intermediate result." So I think of a formal axiomatic system as a LISP S-expression that every now and then calls a particular new LISP primitive function which outputs an intermediate result. And that way it puts out a lot of LISP S-expressions, which are the theorems.

And you can cheat and instead of putting a real formal axiomatic system in there with a proof-checking algorithm and running through all possible proofs, which is terribly slow (and never halts), you can cheat and put in a little example of something that just `display`'s a few sample theorems and then halts. That way you can run a little example and debug algorithms which work with formal axiomatic systems.

How can algorithms work with these toy formal axiomatic systems? Let's say that you have a formal axiomatic system. We've agreed that it's a LISP program that is going to put out intermediate results using a new primitive function called `display`. Actually all normal LISP's have a function for outputting intermediate results; you use it for debugging, right? But it becomes more important in this framework. It plays an official role in my computerized version of algorithmic information theory, because `try` captures all the intermediate results. That's very important.

`Try` is a time-bounded execution/evaluation of a LISP expression. You use `try` to run a program with a time limit, giving the program raw binary data on the side, and **capturing all the intermediate output** as well as the final value (if any). That's very important. You

see, the new primitive function `try` that I added to LISP is the way that you can run a formal axiomatic system for a while and see what are the theorems that it produces. That's put in as a primitive function in this LISP. The value returned by `try` is a 3-element list  $(\alpha \beta \gamma)$ .  $\alpha$  is success or failure.  $\beta$  is the value of the expression being tried if the `try` is a success, and is `out-of-time` or `out-of-data` if the `try` is a failure. And  $\gamma$  is the list of captured `display`'s, the list of theorems. The theorems don't get displayed, they end up in this list instead. Okay?

Let me say by the way that this LISP started off as three-hundred lines of Mathematica. I invented this LISP using Mathematica as my programming tool; I wrote the LISP interpreter in Mathematica. That way I could play with my LISP and try it out as the design evolved. Mathematica is the most powerful programming language that I know. But it's slow. So I rewrote the interpreter for this LISP in C. And it's a thousand lines of C, it's a hundred times faster than in Mathematica, but the program is incomprehensible of course.

I think that the Mathematica code is pretty nice. The C code is incomprehensible! Like all good C programs, it's too clever. You can deal with it while you're writing it, and immediately afterwards it's already incomprehensible, even for the programmer!

The Mathematica code I have to say I really like. It's 300 lines of Mathematica. I think it's pretty understandable, I think it's pretty clean, and I think that if I were presenting this as a course I would go through the LISP interpreter with the students. I think it's important that they should see the code. It's only 300 lines of Mathematica, and they could make changes to the interpreter as a way of showing that they understand it. So it's there, the Mathematica code is there on my Web site. (So is the C code. Use it, but don't try to understand it!)

Okay, so what's the next (and the last) program? The next program works with a formal axiomatic system, which is just a mechanism for producing an infinite set of theorems. It's given to us as a binary program for  $U$ ,  $FAS^*$ , which is a LISP expression plus binary data, and we're going to measure its complexity in bits the same way that we did before. I'll prove that if the formal axiomatic system has program-size complexity  $N$ , then it can enable you to determine, to prove what is the value of at most  $N + 15328$  bits of the halting probability  $\Omega$ .

If a FAS has program-size complexity  $N$ , then it can enable you to determine at most  $N + 15328$  bits of  $\Omega$ .<sup>3</sup>

So how do I show this? Well, I have a Berry paradox kind of proof. The idea is that if you could prove a lot of bits of  $\Omega$ , then  $\Omega$  wouldn't be this irreducible

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

That would give you a way to compress  $\Omega$  into the axioms of the FAS. It would give you too concise a way to calculate  $\Omega$ . If you could prove what the bits of  $\Omega$  are, you'd do it systematically by searching through all possible proofs, and that would give you a way to calculate the bits of  $\Omega$ . That's all we're saying, that it would contradict this

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

So deduction and computation are very close, as Turing already noticed in his famous 1936 paper "On Computable Numbers..." where he proves an incompleteness result using the unsolvability of the halting problem. My argument is at the same level, it's analogous.

The new business here is roughly like the paradox of the first uninteresting positive integer. You know, that's sort of an interesting number. So if you could prove that a number's uninteresting, that would be an interesting fact about it, and "uninteresting" is the notion of algorithmically incompressible. So it turns out that you can't prove that an  $N$ -bit string is algorithmically incompressible if  $N$  is larger than the complexity of your axioms. You can't prove that an  $N$ -bit string is algorithmically irreducible if it has more bits than the axioms you're using for the proof. And similarly it turns out that with  $N$  bits of axioms you get at most  $N + 15328$  bits of  $\Omega$ .

That's the general idea. Here are the details.

I write out a 7328-bit program  $\phi_{7328}$ , it's about one page of LISP code. Why 7328 bits? Because you have to add that to the constant in the inequality for  $H(\Omega_N)$  to get the constant in our incompleteness result. So the difference between these two constants 8000 and 15328 is

---

<sup>3</sup>Perhaps one should refer to this as the *logical (or deductive) irreducibility* of  $\Omega$ , to distinguish it from the *computational (or algorithmic) irreducibility* of  $\Omega$ , namely the fact that  $H(\Omega_N) > N - 8000$ .

the size of the next program in this course, the last program. Divided by 8, you get the size of a LISP expression in characters.

And what this LISP program  $\phi_{7328}$  does is this. Using `try` it starts running the formal axiomatic system that you're assumed to be given that's  $N$  bits of code. So we're looking at this:

$$U(\phi_{7328} \text{ FAS}^* \dots).$$

The prefix  $\phi_{7328}$ , the program that outputs the theorems of the FAS, and some extra stuff that I'll explain later are concatenated and fed to  $U$ .

$\phi_{7328}$  starts running the formal axiomatic system using larger and larger time bounds, capturing the intermediate output, which are the theorems. And it looks at the theorems to see how many bits of  $\Omega$  it got. And I allow partial determinations where you get some of the bits but you leave holes with unknown bits between some of the bits that you do know.

And what  $\phi_{7328}$  does is it looks for a small set of axioms that enable you to prove substantially more bits of  $\Omega$  than there are in those axioms—at least 15329 bits more. And then it just fills in the holes, the missing bits, which costs just one bit per bit, one bit for each missing bit. So  $\phi_{7328}$ 's final output and the value of

$$U(\phi_{7328} \text{ FAS}^* \text{ missing bits of } \Omega)$$

will be one of the  $\Omega_N$ 's. For some  $N$ , it'll be the list of the first  $N$  bits of  $\Omega$ . By the way,  $\phi_{7328}$  may not need all the bits of  $\text{FAS}^*$ , because it only keeps a finite part of the potentially infinite computation for the formal axiomatic system.

So if you could use  $N$  bits of axioms to get essentially more than  $N$  bits of  $\Omega$ —more than  $N + 15328$  bits, in fact—then you could fill in the missing bits at a cost of one bit each, and you get into trouble with this inequality

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

That's the point.

So that's how it goes, the basic idea is straight-forward. And I actually have this 7328-bit LISP program  $\phi_{7328}$  and you can run it.

Well, you can certainly test the auxiliary functions. You can certainly run all the auxiliary functions on examples to convince yourself that they work. But to test the whole algorithm, its main function, you have to cheat, you don't give it a real formal axiomatic system, because if you really try to run through all possible proofs in size order it would take too long. So you cheat, you give the algorithm a LISP expression that just throws out a few test theorems that will put the algorithm  $\phi_{7328}$  that's running the formal axiomatic system through its paces.

That's how you convince yourself that this all works. And you may have to "tweak" things a little bit, you may have to slightly change the LISP code so that it works with your simple test cases. So you can convince yourself by running little examples that this would all work if you really gave it a genuine formal axiomatic system, say Zermelo-Fraenkel set theory.

I would like to have somebody program out Zermelo-Fraenkel set theory in my version of LISP, which is pretty close to a normal LISP as far as this task is concerned, just to see how many bits of complexity mathematicians normally assume. You see, the whole effort here has been to get these constants 8000 and 15328. And if you programmed ZF, you'd get a really sharp incompleteness result. It wouldn't say that you can get at most  $H(\text{ZF}) + 15328$  bits of  $\Omega$ , it would say, perhaps, at most 96000 bits! We'd have a much more definite incompleteness theorem. I hope that somebody will program ZF in LISP. I stop at this point [laughter], you know, programming fatigue! And you want to leave something for the students to do, right? So this is a little job for them. And if they're really clever maybe it'll put out the theorems in Zermelo-Fraenkel set theory reasonably quickly, not by running through all possible proofs in size order, which would take too long, but by doing some kind of tree search so that you can actually have something interesting happen in a reasonable amount of time.

## Discussion

Okay, so why is all this interesting? Well, some of you may know already, but let me wave my hands frantically in the last few minutes and say why.

This looks like a kind of programming madness, right? Up to now, it looks like what I've been telling you is, "Oh how much fun it was to program something that nobody's ever programmed before!" Or maybe just that this is neater than it was in the previous versions of my course. Yes, programming can be an obsession, it can ruin your life. You can sit at your terminal watching your life fall apart as you stay there hacking away until 4 in the morning! But is there any other justification for this except as a kind of drug? Well, I think so! I think this has some philosophical significance.

What's the point about this incompleteness theorem, about this technical result?

If a FAS has program-size complexity  $N$ , then it can enable you to determine at most  $N + 15328$  bits of  $\Omega$ .

Well, what it's really telling you is that you might get some bits of  $\Omega$ , the first ones, say. You might be able to calculate the first few bits of  $\Omega$ . In fact, in a different version of this theory, with a different UTM, (this was with my one-character LISP, my one character per atom LISP), I did get the first 7 bits of  $\Omega$ , and they were all 1 bits. You see, once your lower bound on  $\Omega$  is 127/128ths, you know that these bits can't change. The first 7 bits were 1's. And now I have a much friendlier LISP, but I lost this, I can no longer determine the first 7 bits of  $\Omega$  this way.

But anyway, you might be able to get some bits of  $\Omega$  without contradicting  $\Omega$ 's logical and computational irreducibility, without contradicting this incompleteness result and this inequality:

- You can determine at most  $H(\text{FAS}) + 15328$  bits of  $\Omega$ .
- $H(\Omega_N) > N - 8000$ .

But in spite of this, the basic point is that  $\Omega$  really shows that some areas of mathematics have no structure, have no pattern at all.

Let me put it this way. Normally you think that if something is true, it's true for a reason, right? In mathematics, the reason is called a proof, and the job of a mathematician is to find the reason that something is true, to find a proof. But the bits of  $\Omega$  are mathematical facts that are true for no reason, they're accidental!

Now this is a very specific  $\Omega$ , I've programmed it out. Imagine it being written in binary. And you ask, an individual bit, say the 33rd bit, is it a 0 or a 1? Let's say you're trying to prove which it is. And the answer is, you can't! The reason you can't is because, whether that particular bit is a 0 or a 1 is true for no reason, it's true by accident. It's so delicately balanced whether it's going to be a 0 or a 1, that we will never know!

It's like independent tosses of a fair coin. Independent tosses of a fair coin has got to come out heads or tails in each case, but there's no reason that it comes out one or the other, right? So it's exactly the same story with these mathematical facts, with the bits of  $\Omega$ . There is no pattern or structure in the sequence of bits of  $\Omega$ .

I don't know why anybody would want to try to determine bits of  $\Omega$ . Although people have played with the Busy Beaver function, which in a way is like trying to calculate the bits of  $\Omega$ . I don't know why you'd try to determine the bits of  $\Omega$ . But if you were to try to do this, what this incompleteness result shows you is that you're in big, big trouble! Because essentially the only way to prove what a bit of  $\Omega$  is, is to add the theorem that you want to prove as a new axiom. It's irreducible mathematical information. Now you can prove **anything** by adding it as a new axiom. The point here is that for  $\Omega$  that's essentially **the only way** to do it. No compression is possible.

So there is no structure, there is no pattern,  $\Omega$  has maximum entropy, it mirrors independent tosses of a fair coin. Now to a physicist what I'm saying sounds pretty reasonable, right?  $\Omega$  has maximum entropy, the bits of  $\Omega$  are completely uncorrelated. But to a mathematician this all sounds weird!

My new  $\Omega$  is a particular well-defined real number. It's a real number with a rather simple definition, I can even write the LISP program that defines  $\Omega$  on one computer screen. So you believe, thinking in Platonic terms, that each bit is either a 0 or a 1, even if I'm never going to know which. But it's black or white, right? And what I'm saying is that I think that it's really better to think that it's **grey**. It's really better to think that each bit of  $\Omega$  has probability one-half of being 0 or of being 1—even though it's a particular well-determined bit, because I've written out the program that defines this number. Defines it, not by enabling you to calculate it bit by bit—that would contradict  $\Omega$ 's



unknowability. But the program for  $\Omega$  does enable you to calculate better and better lower bounds on  $\Omega$ , so  $\Omega$  is **almost** a computable real number in the same sense that  $\pi$  is. Almost, but not quite!

So the game is like this. I'm trying very hard to be constructive, as constructive as possible. Writing out programs is a sure sign that you're a constructivist, right? [Laughter] I want to settle all the programming details. But I'm trying to be as constructive as possible about non-constructivity! I want to exhibit something that escapes the power of constructivity, of mathematical reasoning, that you can't calculate, but that's just over the border between the constructible and the non-constructible. And I think that  $\Omega$  is pretty damn good, it's just on the border.

Now this doesn't mean that all of mathematics falls down in a heap! But the normal notion of mathematics was that there were a small, finite set of axioms and rules of inference that we could all agree on, from which all the infinite mathematical truth would follow. This is the tradition that goes back to Euclid, to Leibniz, to Peano, to Frege, to Russell and Whitehead, to Hilbert. And Gödel showed that there was a problem. And Turing showed that there was a problem, using a different method involving computers. And I think that  $\Omega$  follows in that tradition and shows that the problem is even bigger. However I don't think this means that you should stop doing mathematics! So what is the significance of  $\Omega$  and of incompleteness? Should it affect how we actually do mathematics? I'll give you my opinion.

Of course the nature of mathematics has been discussed for a long time! Every generation of mathematicians have their own answer. But let me share with you my own feelings about this, my tentative conclusions.

There's a word that a philosopher coined that's very good. He says that there's an emerging new school, and emerging new *quasi-empirical* view of the foundations of mathematics. One talks about the formalist school, the logicist school, and the intuitionist school. Well, Thomas Tymoczko has a book, *New Directions in the Philosophy of Mathematics*, which has a whole bunch of articles, including two of mine, and he thinks that all these articles tend to support a new quasi-empirical view of the foundations of mathematics.

Now what does quasi-empirical mean? I'll tell you what it means

to me. Quasi-empirical means that **pure math ain't that different from physics!** The normal notion of pure math is that mathematicians have some kind of direct pipeline to God's thoughts, to absolute truth! But poor physicists! You know, they tried Newtonian mechanics, it looks good for a while, then Einstein shows it's all wrong. Then—surprise!—quantum mechanics shows that Einstein was wrong! And now there's superstring theory. And is it right? Is it wrong? And mathematicians laugh and say, "Oh those poor physicists! It's such a messy subject! They always have to backpedal, they don't know what they're doing! It's all so tentative!"

Well, I think that mathematics and physics are not really that different!—Physicists love it when I say this! [Laughter]

Let me try explaining this another way. Euclid said that mathematics is based on self-evident truths. But my impression is that maybe axioms are not self-evident truths. I don't believe in self-evident truths. Maybe it's more like in physics. Maybe mathematics should be done more like physics, where you're willing to add new axioms because they're useful, not because they're self-evident. And then of course you have to be prepared to say "I goofed!" and remove an axiom, which mathematicians don't like to have to do.

This may sound completely crazy, but in fact it's not just my opinion. Gödel makes very similar remarks in Volume II of his *Collected Works*, in his essay on "Russell's Mathematical Logic." This essay was originally published in the Paul Arthur Schilpp volume *The Philosophy of Bertrand Russell*.

I talked about new axioms to a mathematician once, and he replied, "Okay, I'm willing to add the Riemann hypothesis as a new axiom if you can prove to me that it doesn't follow from the usual axioms." Well, that's hard to do, because if the Riemann hypothesis were false, then there would be a numerical counter-example that one could easily verify that shows that it's false. So if you could show that the Riemann hypothesis is beyond the power of the usual axioms, that would imply that the Riemann hypothesis is true!

So these ideas are very controversial.

But the whole point of algorithmic information theory, the whole point of my information-theoretic approach to incompleteness, is that sometimes to get more information out of a set of axioms, you've just

got to put more in. So let's put more axioms in! Physicists have always done that.

I had these ideas a long time ago. I proved my first information-theoretic incompleteness theorem in 1970. Although it's only in the past two or three years that I discovered how to actually program the algorithm in my original proof and run it on examples. And I was recently surprised to discover or rediscover that there are highly relevant quotes from Einstein and from Gödel. Let me throw them into this discussion, let me end with that.

Einstein has a very nice remark that I angered some mathematicians with. But what do they care, he's only a physicist, right? [Laughter] In his essay "Remarks on Bertrand Russell's Theory of Knowledge" in the Paul Arthur Schilpp volume *The Philosophy of Bertrand Russell*, Einstein says that "the series of integers is obviously an invention of the human mind, a self-created tool which simplifies the ordering of certain sensory experiences." So you can see that Einstein's attitude is very empirical.

I think that Einstein's position is that the positive integers are not *a priori*, they're not God given, we invent them like we invent all of physics. But the positive integers look more *a priori* than other concepts because they've been around longer, we invented them a long time ago. After all, when an idea has been around for a few thousand years, it's not surprising that people think that it's obvious, that it's just common sense, that it's "a necessary tool of thought." The other extreme is the latest field theory in physics. That looks a lot more tentative. It hasn't been here long, and it'll probably be shot down next week, right? And there are probably thirteen different versions! But in Einstein's view there is no fundamental difference. The positive integers have been around for a long time, but they're still just an invention.

So I like this quote from Einstein, but it doesn't convince mathematicians.

Then there are some very interesting remarks made by Gödel. You can find them in Gödel's *Collected Works*. Gödel's philosophical position is the exact opposite of Einstein's. Gödel believed in the Platonic universe of mathematical ideas, he believed that the positive integers are just as real as tables and chairs. There are an infinity of positive

integers, and they're out there somewhere. They're in the Platonic universe of mathematical ideas, that's where mathematical objects are!

I don't know! I used to laugh at all of this when I was a kid. But think about it seriously. You're young, you're trying to learn mathematics, you're doing elementary number theory, and you don't really start to worry about the fact that elementary number theory presupposes arbitrarily large positive integers, and how do they fit in the universe? Imagine a positive integer that is

$$10^{10^{10^{10}}}$$

digits long. Does it exist? In what sense does it exist? You don't care—right?—you prove theorems about it, you know that it would be commutative, right?  $a + b$  is going to be equal to  $b + a$  even if neither number fits in the universe! [Laughter] But then later on in life you start to worry about this! [Laughter]

So I'm not sure if the positive integers exist anymore. But Gödel thinks that they do, and that philosophical position is associated with the name of Plato. But it's really just the classical mathematical position that the positive integers really exist, that an infinity of them are really out there. And starting from that, Gödel comes to a very surprising conclusion. Since the positive integers are just as real as tables and chairs, you can do experiments with them by doing calculations, and if you see a pattern, you can just go ahead like a scientist would in dealing with electrons. Since integers are just as real as electrons, why can't we use the same kinds of methods that scientists use?

Here are Gödel's exact words, taken from a previously unpublished manuscript \*1951 that is in Volume III of his *Collected Works*:

“If mathematics describes an objective world just like physics, there is no reason why inductive methods should not be applied in mathematics just the same as in physics.”

And in his essay “What is Cantor's Continuum Problem?” in Volume II of his *Collected Works*, Gödel says that maybe we'll come up with new versions of set theory, maybe we'll come up with new ideas about sets the same way that physicists come up with new ideas about physical objects. The justification for these new principles would be

that they're useful, that they help us to organize our mathematical experience, just as the ultimate justification for physical principles is that they help us to organize our physical experience.

I think it's very funny! Here's Einstein, who's a diehard empiricist, and here's Gödel, who's a diehard Platonist, and they sort of come to the same conclusion! But they were buddies at the Institute for Advanced Study, so it's not surprising that they influenced each other. And the information-theoretic viewpoint that I've explained to you today leads me in the very same direction.

Another funny thing is that I don't believe that all this work on the foundations of mathematics has changed at all the way mathematicians actually work. Gödel's incompleteness theorem was initially very shocking. But then mathematicians noticed that the kind of assertion that Gödel constructed that's true but unprovable is not the kind of assertion that you deal with in your everyday work as a mathematician. And I think it's fair to say that about  $\Omega$  too. The longer I live with  $\Omega$ , the more natural it looks to me. But a skeptic would say, "I don't care about the bits of  $\Omega$ , so what if there's trouble there!"

But I do think that something else is making a difference in how mathematicians carry on their everyday work. It's the fact that the computer has so vastly expanded mathematical experience, that we just have to deal with it somehow. A lot of physicists do experimental work on the computer now. There's even a journal called *Experimental Mathematics*.

The computer has so vastly increased mathematical experience, that how do we keep up with all of it? Well, the answer is that sometimes you see that something seems to be the case, and it'd be nice if you could prove it, but for the moment you can't, so you sort of conjecture it. And if you're doing mathematical physics, then you're on the borderline between math and physics, and it's certainly okay to behave in this way. But if you're a mathematician, if you're just over the border, then it's not so clear anymore.

Well, I once had a conversation like this with a mathematician, and the Riemann hypothesis came up, and he said, "It works fine the way we do things now. You have a paper, and you say the paper is 'modulo the Riemann hypothesis.' So why do you need to call it a new axiom?" And he has a point. But I do think that if a principle is

very helpful for a long time, and nobody shoots it down, why not call it a new axiom? But one shouldn't do this too quickly. One has to be careful, the way physicists are, although I sometimes wonder how careful physicists really are! [Laughter]

Okay, that's the idea. And if you go to my Web site you'll find this course there, together with the Mathematica code for my LISP interpreter. So you're welcome to play with it, and you're welcome to send me e-mail about it, and I'd be pleased as Punch if one of you tried giving my course on the limits of mathematics to real students in a normal university setting.

Anyway, you guys have been nice enough to invite me for many years to give talks, and it's been very stimulating to talk with you, and for the moment I think I've exhausted this train of thought. Of course, I've been thinking that since I was fifteen years old, but fortunately every now and then I get a new idea! [Laughter] But for now, that's all I have to say, and maybe someday I'll have more to say, and I'll be lucky enough to come out here again and have a chance to kick it around with you! Thank you very much! [Applause]

# THE LIMITS OF MATHEMATICS

**Journal of Universal Computer Science Vol. 2, No. 5 (28 May 1996), pp. 270–305**

**G. J. Chaitin, IBM Research, P. O. Box 704, Yorktown Heights, NY 10598, [chaitin@watson.ibm.com](mailto:chaitin@watson.ibm.com)**

## Introduction

In a remarkable development, I have constructed a new definition for a self-delimiting universal Turing machine (UTM) that is easy to program and runs very quickly. This provides a new foundation for algorithmic information theory (AIT), which is the theory of the size in bits of programs for self-delimiting UTM's. Previously, AIT had an abstract mathematical quality. Now it is possible to write down executable programs that embody the constructions in the proofs of theorems. So AIT goes from dealing with remote idealized mythical objects to being a theory about practical down-to-earth gadgets that one can actually play with and use.

This new self-delimiting UTM is implemented via software written in a new version of LISP that I invented especially for this purpose. This LISP was designed by writing an interpreter for it in Mathematica that

was then translated into C. I have tested this software by running it on IBM RS/6000 workstations with the AIX version of UNIX.

Using this new software and the latest theoretical ideas, it is now possible to give a self-contained “hands on” mini-course presenting very concretely my latest proofs of my two fundamental information-theoretic incompleteness theorems. The first of these theorems states that an  $N$ -bit formal axiomatic system cannot enable one to exhibit any specific object with program-size complexity greater than  $N + c$ . The second of these theorems states that an  $N$ -bit formal axiomatic system cannot enable one to determine more than  $N + c'$  scattered bits of the halting probability  $\Omega$ .

Most people believe that anything that is true is true for a reason. These theorems show that some things are true for no reason at all, i.e., accidentally, or at random.

As is shown in this course, the algorithms considered in the proofs of these two theorems are now easy to program and run, and by looking at the size in bits of these programs one can actually, for the first time, determine exact values for the constants  $c$  and  $c'$ .

I used this approach and software in an intensive short course on the limits of mathematics that I gave at the University of Maine in Orono in the summer of 1994. I also lectured on this material during a stay at the Santa Fe Institute in the spring of 1995, and at a meeting at the Black Sea University in Romania in the summer of 1995. A summary of the approach that I used on these three occasions will appear under the title “A new version of algorithmic information theory” in a forthcoming issue of the new magazine *Complexity*, which has just been launched by the Santa Fe Institute and Wiley. A less technical discussion of the basic ideas that are involved “How to run algorithmic information theory on a computer” will also appear in *Complexity*.

After presenting this material at these three different places, it became obvious to me that it is extremely difficult to understand it in its original form. So next time, at the Rovaniemi Institute of Technology in the spring of 1996, I am going to use the new, more understandable software in this report; everything has been redone in an attempt to make it as easy to understand as possible.

For their stimulating invitations, I thank Prof. George Markowsky of the University of Maine, Prof. Cristian Calude of the University of



Auckland, Prof. John Casti of the Santa Fe Institute, and Prof. Veikko Keränen of the Rovaniemi Institute of Technology. And I am grateful to IBM for supporting my research for almost thirty years, and to my current management chain at the IBM Research Division, Dan Prener, Christos Georgiou, Eric Kronstadt, Jeff Jaffe, and Jim McGroddy.

This report includes the LISP runs `*.r` used to present the information-theoretic incompleteness theorems of algorithmic information theory. This report does not include the software used to produce these LISP runs. To obtain the software for this course via e-mail, please send requests to `chaitin@watson.ibm.com`.

## The New Idea

Here is a quick summary of this new LISP, in which atoms can now either be words or unsigned decimal integers. First of all, comments are written like this: `[comment]`. Each LISP primitive function has a fixed number of arguments. `'` is QUOTE, `=` is EQ, and `atom`, `car`, `cdr`, `cadr`, `caddr`, `cons` are provided with their usual meaning. We also have `lambda`, `define`, `let`, `if` and `display` and `eval`. The notation `"` indicates that an S-expression with explicit parentheses follows, not what is usually the case in this LISP, an M-expression, in which the parentheses for each primitive function are implicit. `nil` denotes the empty list `()`, and the logical truth values are `true` and `false`. For dealing with unsigned decimal integers we have `+`, `-`, `*`, `^`, `<`, `>`, `<=`, `>=`, `base10-to-2`, `base2-to-10`.

So far this is fairly standard. The new idea is this. We define our standard self-delimiting universal Turing machine as follows. Its program is in binary, and appears on a tape in the following form. First comes a LISP expression, written in ASCII with 8 bits per character, and terminated by an end-of-line character `'\n'`. The TM reads in this LISP expression, and then evaluates it. As it does this, two new primitive functions `read-bit` and `read-exp` with no arguments may be used to read more from the TM tape. Both of these functions explode if the tape is exhausted, killing the computation. `read-bit` reads a single bit from the tape. `read-exp` reads in an entire LISP expression, in 8-bit character chunks, until it reaches an end-of-line character `'\n'`.

This is the only way that information on the TM tape may be accessed, which forces it to be used in a self-delimiting fashion. This is because no algorithm can search for the end of the tape and then use the length of the tape as data in the computation. If an algorithm attempts to read a bit that is not on the tape, the algorithm aborts.

How is information placed on the TM tape in the first place? Well, in the starting environment, the tape is empty and any attempt to read it will give an error message. To place information on the tape, one must use the primitive function `try` which tries to see if an expression can be evaluated.

Consider the three arguments  $\alpha$ ,  $\beta$  and  $\gamma$  of `try`. The meaning of the first argument is as follows. If  $\alpha$  is `no-time-limit`, then there is no depth limit. Otherwise  $\alpha$  must be an unsigned decimal integer, and gives the depth limit (limit on the nesting depth of function calls and re-evaluations). The second argument  $\beta$  of `try` is the expression to be evaluated as long as the depth limit  $\alpha$  is not exceeded. And the third argument  $\gamma$  of `try` is a list of bits to be used as the TM tape.

The value  $\nu$  returned by the primitive function `try` is a triple. The first element of  $\nu$  is `success` if the evaluation of  $\beta$  was completed successfully, and the first element of  $\nu$  is `failure` if this was not the case. The second element of  $\nu$  is `out-of-data` if the evaluation of  $\beta$  aborted because an attempt was made to read a non-existent bit from the TM tape. The second element of  $\nu$  is `out-of-time` if evaluation of  $\beta$  aborted because the depth limit  $\alpha$  was exceeded. These are the only possible error flags, because this LISP is designed with maximally permissive semantics. If the computation  $\beta$  terminated normally instead of aborting, the second element of  $\nu$  will be the result produced by the computation  $\beta$ , i.e., its value. That's the second element of the list  $\nu$  produced by the `try` primitive function.

The third element of the value  $\nu$  is a list of all the arguments to the primitive function `display` that were encountered during the evaluation of  $\beta$ . More precisely, if `display` was called  $N$  times during the evaluation of  $\beta$ , then the third element of  $\nu$  will be a list of  $N$  elements. The  $N$  arguments of `display` appear in the third element of  $\nu$  in chronological order. Thus `try` can not only be used to determine if a computation  $\beta$  reads too much tape or goes on too long (i.e., to greater depth than  $\alpha$ ), but `try` can also be used to capture all the output that

$\beta$  displayed as it went along, whether the computation  $\beta$  aborted or not.

In summary, all that one has to do to simulate a self-delimiting universal Turing machine  $U(p)$  running on the binary program  $p$  is to write

```
try no-time-limit 'eval read-exp p
```

This is an M-expression with parentheses omitted from primitive functions. (Recall that all primitive functions have a fixed number of arguments.) With the parentheses supplied, it becomes the S-expression

```
(try no-time-limit ('(eval(read-exp))) p)
```

This says that one is to read a complete LISP S-expression from the TM tape  $p$  and then evaluate it without any time limit and using whatever is left on the tape  $p$ .

Some more primitive functions have also been added. The 2-argument function `append` denotes list concatenation, and the 1-argument function `bits` converts an S-expression into the list of the bits in its ASCII character string representation. These are used for constructing the bit strings that are then put on the TM tape using `try`'s third argument  $\gamma$ . We also provide the 1-argument functions `size` and `length` that respectively give the number of characters in an S-expression, and the number of elements in a list. Note that the functions `append`, `size` and `length` could be programmed rather than included as built-in primitive functions, but it is extremely convenient and much much faster to provide them built in.

Finally a new 1-argument identity function `debug` with the side-effect of outputting its argument is provided for debugging. Output produced by `debug` is invisible to the "official" display and `try` output mechanism. `debug` is needed because `try  $\alpha$   $\beta$   $\gamma$`  suppresses all output  $\theta$  produced within its depth-controlled evaluation of  $\beta$ . Instead `try` collects all output  $\theta$  from within  $\beta$  for inclusion in the final value  $\nu$  that `try` returns, namely  $\nu = (\text{success/failure, value of } \beta, \theta)$ .

## Course Outline

The course begins by explaining with examples my new LISP. See `examples.r`.

Then the theory of LISP program-size complexity is developed a little bit. LISP program-size complexity is extremely simple and concrete. In particular, it is easy to show that it is impossible to prove that a self-contained LISP expression is elegant, i.e., that no smaller expression has the same value. To prove that an  $N$ -character LISP expression is elegant requires a formal axiomatic system that itself has at least LISP complexity  $N - 410$ . See `godel.r`.

Next we define our standard self-delimiting universal Turing machine  $U(p)$  using

```
cadr try no-time-limit 'eval read-exp p
```

as explained in the previous section.

Next we show that

$$H(x, y) \leq H(x) + H(y) + c$$

with  $c = 432$ . Here  $H(\dots)$  denotes the size in bits of the smallest program that makes our standard universal Turing machine compute  $\dots$ . Thus this inequality states that the information needed to compute the pair  $(x, y)$  is bounded by a constant  $c$  plus the sum of the information needed to compute  $x$  and the information needed to compute  $y$ . Consider

```
cons eval read-exp
cons eval read-exp
nil
```

This is an M-expression with parentheses omitted from primitive functions. With all the parentheses supplied, it becomes the S-expression

```
(cons (eval (read-exp))
(cons (eval (read-exp))
nil))
```

$c = 432$  is just 8 bits plus 8 times the size in characters of this LISP S-expression. See `utm.r`.

Consider a binary string  $x$  whose size is  $|x|$  bits. In `utm.r` we also show that

$$H(x) \leq 2|x| + c$$

and

$$H(x) \leq |x| + H(|x|) + c'$$

with  $c = 1106$  and  $c' = 1024$ . As before, the programs for doing this are exhibited and run.

Next we turn to the self-delimiting program-size complexity  $H(X)$  for infinite r.e. sets  $X$ . This is defined to be the size in bits of the smallest LISP expression  $\xi$  that executes forever without halting and outputs the members of the r.e. set  $X$  using the LISP primitive `display`, which is an identity function with the side-effect of outputting the value of its argument. Note that this LISP expression  $\xi$  is allowed to read additional bits or expressions from the TM tape using the primitive functions `read-bit` and `read-exp` if  $\xi$  so desires. But of course  $\xi$  is charged for this; this adds to  $\xi$ 's program size.

It is in order to deal with such unending expressions  $\xi$  that the LISP primitive function for time-limited evaluation `try` captures all output from `display` within its second argument  $\beta$ .

Now consider a formal axiomatic system  $A$  of complexity  $N$ , i.e., with a set of theorems  $T_A$  that considered as an r.e. set as above has self-delimiting program-size complexity  $H(T_A) = N$ . We show that  $A$  cannot enable us to exhibit a specific S-expression  $s$  with self-delimiting complexity  $H(s)$  greater than  $N + c$ . Here  $c = 4872$ . See `gode12.r`.

Next we show two different ways to calculate the halting probability  $\Omega$  of our standard self-delimiting universal Turing machine in the limit from below. See `omega.r` and `omega2.r`. The first way of doing this, `omega.r`, is straight-forward. The second way to calculate  $\Omega$ , `omega2.r`, uses a more clever method. Using the clever method as a subroutine, we show that if  $\Omega_N$  is the first  $N$  bits of the fractional part of the base-two real number  $\Omega$ , then

$$H(\Omega_N) > N - c$$

with  $c = 8000$ . Again this is done with a program that can actually be run and whose size gives us a value for  $c$ . See `omega3.r`.

Consider again the formal axiomatic system  $A$  with complexity  $N$ , i.e., with self-delimiting program-size complexity  $H(T_A) = N$ . Using the lower bound of  $N - c$  on  $H(\Omega_N)$  established in `omega3.r`, we show that  $A$  cannot enable us to determine more than the first  $N + c'$  bits of  $\Omega$ . Here  $c' = 15328$ . In fact, we show that  $A$  cannot enable us to determine more than  $N + c'$  bits of  $\Omega$  even if they are scattered and we leave gaps. See `godel3.r`.

Last but not least, the philosophical implications of all this should be discussed, especially the extent to which it tends to justify experimental mathematics. This would be along the lines of the discussion in my talk transcript “Randomness in arithmetic and the decline and fall of reductionism in pure mathematics.”

This concludes our “hands-on” mini-course on the information-theoretic limits of mathematics.

## Bibliography

Here is a useful collection of hand-outs for this course:

- [1] G. J. Chaitin, “Randomness in arithmetic and the decline and fall of reductionism in pure mathematics,” in J. Cornwell, *Nature’s Imagination*, Oxford University Press, 1995, pp. 27–44.
- [2] G. J. Chaitin, “The Berry paradox,” *Complexity* 1:1 (1995), pp. 26–30.
- [3] G. J. Chaitin, “A new version of algorithmic information theory,” *Complexity*, to appear.
- [4] G. J. Chaitin, “How to run algorithmic information theory on a computer,” *Complexity*, to appear.

## examples.r

LISP Interpreter Run

[ Test new lisp & show how it works ]

aa [ initially all atoms eval to self ]

expression aa  
value aa

nil [ except nil = the empty list ]

expression nil  
value ()

'aa [ quote = literally ]

expression (' aa)  
value aa

'(aa bb cc) [ delimiters are ' " ( ) [ ] blank \n ]

expression (' (aa bb cc))  
value (aa bb cc)

(aa bb cc) [ what if quote omitted?! ]

expression (aa bb cc)  
value aa

'car '(aa bb cc) [ here effect is different ]

expression (' (car (' (aa bb cc))))  
value (car (' (aa bb cc)))

car '(aa bb cc) [ car = first element of list ]

expression (car (' (aa bb cc)))  
value aa

car '((a b)c d)

```
expression (car (' ((a b) c d)))  
value      (a b)
```

```
car '(aa)
```

```
expression (car (' (aa)))  
value      aa
```

```
car aa [ ignore error ]
```

```
expression (car aa)  
value      aa
```

```
cdr '(aa bb cc) [ cdr = rest of list ]
```

```
expression (cdr (' (aa bb cc)))  
value      (bb cc)
```

```
cdr '((a b)c d)
```

```
expression (cdr (' ((a b) c d)))  
value      (c d)
```

```
cdr '(aa)
```

```
expression (cdr (' (aa)))  
value      ()
```

```
cdr aa [ ignore error ]
```

```
expression (cdr aa)  
value      aa
```

```
cadr '(aa bb cc) [ combinations of car & cdr ]
```

```
expression (car (cdr (' (aa bb cc))))  
value      bb
```



```
caddr '(aa bb cc)
```

```
expression (car (cdr (cdr (' (aa bb cc)))))  
value      cc
```

```
cons 'aa '(bb cc) [ cons = inverse of car & cdr ]
```

```
expression (cons (' aa) (' (bb cc)))  
value      (aa bb cc)
```

```
cons'(a b)'(c d)
```

```
expression (cons (' (a b)) (' (c d)))  
value      ((a b) c d)
```

```
cons aa nil
```

```
expression (cons aa nil)  
value      (aa)
```

```
cons aa ()
```

```
expression (cons aa ())  
value      (aa)
```

```
cons aa bb [ ignore error ]
```

```
expression (cons aa bb)  
value      aa
```

```
("cons aa) [ supply nil for missing arguments ]
```

```
expression (cons aa)  
value      (aa)
```

```
("cons '(aa) '(bb) '(cc)) [ ignore extra arguments ]
```

```
expression (cons (' (aa)) (' (bb)) (' (cc)))  
value      ((aa) bb)
```

```
atom ' aa [ is-atomic? predicate ]
```

```
expression (atom (' aa))
```

```
value      true
```

```
atom '(aa)
```

```
expression (atom (' (aa)))
```

```
value      false
```

```
atom '( )
```

```
expression (atom (' ( )))
```

```
value      true
```

```
= aa bb [ are-equal-S-expressions? predicate ]
```

```
expression (= aa bb)
```

```
value      false
```

```
= aa aa
```

```
expression (= aa aa)
```

```
value      true
```

```
= '(a b)'(a b)
```

```
expression (= (' (a b)) (' (a b)))
```

```
value      true
```

```
= '(a b)'(a x)
```

```
expression (= (' (a b)) (' (a x)))
```

```
value      false
```

```
if true x y [ if ... then ... else ... ]
```

```
expression (if true x y)
```

```
value      x
```

```
if false x y
```

```
expression (if false x y)
```

```
value      y
```

```
if xxx x y [ anything not false is true ]
```

```
expression (if xxx x y)
```

```
value      x
```

```
[ display intermediate results ]
```

```
cdr display cdr display cdr display '( a b c d e )
```

```
expression (cdr (display (cdr (display (cdr (display (' (a b  
c d e))))))))
```

```
display    (a b c d e)
```

```
display    (b c d e)
```

```
display    (c d e)
```

```
value      (d e)
```

```
('lambda(x y)x 1 2) [ lambda expression ]
```

```
expression ((' (lambda (x y) x)) 1 2)
```

```
value      1
```

```
('lambda(x y)y 1 2)
```

```
expression ((' (lambda (x y) y)) 1 2)
```

```
value      2
```

```
('lambda(x y)cons y cons x nil 1 2)
```

```
expression ((' (lambda (x y) (cons y (cons x nil)))) 1 2)
```

```
value      (2 1)
```

```
(if true "car "cdr '(a b c)) [ function expressions ]
```

```
expression ((if true car cdr) (' (a b c)))
value      a
```

```
(if false "car "cdr '(a b c))
```

```
expression ((if false car cdr) (' (a b c)))
value      (b c)
```

```
('lambda()cons x cons y nil) [ function with no arguments ]
```

```
expression ((' (lambda () (cons x (cons y nil))))
value      (x y)
```

[ Here is a way to create an expression and then  
evaluate it in the current environment. EVAL (see  
below) does this using a clean environment instead. ]

```
(display
cons "lambda cons nil cons display 'cons x cons y nil nil)
```

```
expression ((display (cons lambda (cons nil (cons (display ('
      (cons x (cons y nil)))) nil))))
display    (cons x (cons y nil))
display    (lambda () (cons x (cons y nil)))
value      (x y)
```

```
[ let ... be ... in ... ]
```

```
let x a cons x cons x nil [ first case, let x be ... in ... ]
```

```
expression ((' (lambda (x) (cons x (cons x nil)))) a)
value      (a a)
```

```
x
```

```
expression x
value      x
```

```
[ second case, let (f x) be ... in ... ]
```

```
let (f x) if atom display x x (f car x)
(f '(((a)b)c))
```

```
expression (((' (lambda (f) (f (' (((a) b) c))))) (' (lambda (
x) (if (atom (display x)) x (f (car x))))))
display      (((a) b) c)
display      ((a) b)
display      (a)
display      a
value        a
```

```
f
```

```
expression f
value      f
```

```
append '(a b c) '(d e f) [ concatenate-list primitive ]
```

```
expression (append (' (a b c)) (' (d e f)))
value      (a b c d e f)
```

```
[ define "by hand" temporarily ]
```

```
let (cat x y) if atom x y cons car x (cat cdr x y)
(cat '(a b c) '(d e f))
```

```
expression (((' (lambda (cat) (cat (' (a b c)) (' (d e f)))))
(' (lambda (x y) (if (atom x) y (cons (car x) (cat
(cdr x) y))))))
value      (a b c d e f)
```

```
cat
```

```
expression cat
value      cat
```

```
[ define "by hand" permanently ]
```

```
define (cat x y) if atom x y cons car x (cat cdr x y)
```

```
define      cat
value      (lambda (x y) (if (atom x) y (cons (car x) (cat (c
dr x) y)))))
```

```
cat
```

```
expression  cat
value      (lambda (x y) (if (atom x) y (cons (car x) (cat (c
dr x) y)))))
```

```
(cat '(a b c) '(d e f))
```

```
expression (cat (' (a b c)) (' (d e f)))
value      (a b c d e f)
```

```
define x (a b c) [ define atom, not function ]
```

```
define      x
value      (a b c)
```

```
cons x nil
```

```
expression (cons x nil)
value      ((a b c))
```

```
define x (d e f)
```

```
define      x
value      (d e f)
```

```
cons x nil
```

```
expression (cons x nil)
value      ((d e f))
```

```
size abc [ size of S-expression in characters ]
```

```
expression (size abc)
```

value            3

size ' ( a b c )

expression (size (' (a b c)))

value            7

length ' ( a b c ) [ number of elements in list ]

expression (length (' (a b c)))

value            3

length display bits ' a [ S-expression --> bits ]

expression (length (display (bits (' a))))

display        (0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0)

value            16

length display bits ' abc [ extra character is \n ]

expression (length (display (bits (' abc))))

display        (0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0)  
                 0 0 0 1 0 1 0)

value            32

length display bits nil

expression (length (display (bits nil)))

display        (0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 1 0 1 0)

value            24

length display bits ' (a)

expression (length (display (bits (' (a)))))

display        (0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 1 0 1 0 0 1 0  
                 0 0 0 1 0 1 0)

value            32

[ plus ]

+ abc 15 [ not number --> 0 ]

expression (+ abc 15)  
value 15

+ '(abc) 15

expression (+ (' (abc)) 15)  
value 15

+ 10 15

expression (+ 10 15)  
value 25

- 10 15 [ non-negative minus ]

expression (- 10 15)  
value 0

- 15 10

expression (- 15 10)  
value 5

\* 10 15 [ times ]

expression (\* 10 15)  
value 150

^ 10 15 [ power ]

expression (^ 10 15)  
value 1000000000000000

< 10 15 [ less than ]

expression (< 10 15)  
value true



< 10 10

expression (< 10 10)  
value false

> 15 10 [ greater than ]

expression (> 15 10)  
value true

> 10 10

expression (> 10 10)  
value false

<= 15 10 [ less than or equal ]

expression (<= 15 10)  
value false

<= 10 10

expression (<= 10 10)  
value true

>= 10 15 [ greater than or equal ]

expression (>= 10 15)  
value false

>= 10 10

expression (>= 10 10)  
value true

= 10 15 [ equal ]

expression (= 10 15)

value            false

= 10 10

expression    (= 10 10)

value            true

[ here not number isn't considered zero ]

= abc 0

expression    (= abc 0)

value            false

= 0003 3 [ other ways numbers are funny ]

expression    (= 3 3)

value            true

000099 [ leading zeros removed ]

expression    99

value            99

[ and numbers are constants ]

let x b cons x cons x nil

expression    ((' (lambda (x) (cons x (cons x nil)))) b)

value            (b b)

let 99 45 cons 99 cons 99 nil

expression    ((' (lambda (99) (cons 99 (cons 99 nil)))) 45)

value            (99 99)

define 99 45

define            99

value            45

```
cons 99 cons 99 nil
```

```
expression (cons 99 (cons 99 nil))  
value      (99 99)
```

```
[ decimal<-->binary conversions ]
```

```
base10-to-2 255
```

```
expression (base10-to-2 255)  
value      (1 1 1 1 1 1 1 1)
```

```
base10-to-2 256
```

```
expression (base10-to-2 256)  
value      (1 0 0 0 0 0 0 0)
```

```
base10-to-2 257
```

```
expression (base10-to-2 257)  
value      (1 0 0 0 0 0 0 1)
```

```
base2-to-10 '(1 1 1 1)
```

```
expression (base2-to-10 (' (1 1 1 1)))  
value      15
```

```
base2-to-10 '(1 0 0 0 0)
```

```
expression (base2-to-10 (' (1 0 0 0 0)))  
value      16
```

```
base2-to-10 '(1 0 0 0 1)
```

```
expression (base2-to-10 (' (1 0 0 0 1)))  
value      17
```

```
[ illustrate eval & try ]
```

```
eval display '+ display 5 display 15
```

```
expression (eval (display (' (+ (display 5) (display 15))))
display    (+ (display 5) (display 15))
display    5
display    15
value      20
```

```
try 0 display '+ display 5 display 15 nil
```

```
expression (try 0 (display (' (+ (display 5) (display 15))))
           nil)
display    (+ (display 5) (display 15))
value      (success 20 (5 15))
```

```
try 0 display '+ debug 5 debug 15 nil
```

```
expression (try 0 (display (' (+ (debug 5) (debug 15)))) nil)
display    (+ (debug 5) (debug 15))
debug      5
debug      15
value      (success 20 ())
```

```
[ eval & try use initial variable bindings ]
```

```
cons x nil
```

```
expression (cons x nil)
value      ((d e f))
```

```
eval 'cons x nil
```

```
expression (eval (' (cons x nil)))
value      (x)
```

```
try 0 'cons x nil nil
```

```
expression (try 0 (' (cons x nil)) nil)
value      (success (x) ())
```

```
define five! [ to illustrate time limits ]
let (f x) if = display x 0 1 * x (f - x 1)
    (f 5)
```

```
define      five!
value      ((' (lambda (f) (f 5))) (' (lambda (x) (if (= (display x) 0) 1 (* x (f (- x 1)))))))
```

```
eval five!
```

```
expression (eval five!)
display    5
display    4
display    3
display    2
display    1
display    0
value      120
```

```
[ by the way, numbers can be big: ]
```

```
let (f x) if = x 0 1 * x (f - x 1)
    (f 100) [ one hundred factorial! ]
```

```
expression ((' (lambda (f) (f 100))) (' (lambda (x) (if (= x
0) 1 (* x (f (- x 1)))))))
value      93326215443944152681699238856266700490715968264381
62146859296389521759999322991560894146397615651828
62536979208272237582511852109168640000000000000000
00000000
```

```
[ time limit is nesting depth of re-evaluations
  due to function calls & eval & try ]
```

```
try 0 five! nil
```

```
expression (try 0 five! nil)
value      (failure out-of-time ())
```

try 1 five! nil

expression (try 1 five! nil)  
value (failure out-of-time ())

try 2 five! nil

expression (try 2 five! nil)  
value (failure out-of-time (5))

try 3 five! nil

expression (try 3 five! nil)  
value (failure out-of-time (5 4))

try 4 five! nil

expression (try 4 five! nil)  
value (failure out-of-time (5 4 3))

try 5 five! nil

expression (try 5 five! nil)  
value (failure out-of-time (5 4 3 2))

try 6 five! nil

expression (try 6 five! nil)  
value (failure out-of-time (5 4 3 2 1))

try 7 five! nil

expression (try 7 five! nil)  
value (success 120 (5 4 3 2 1 0))

try no-time-limit five! nil

expression (try no-time-limit five! nil)  
value (success 120 (5 4 3 2 1 0))

```
define two* [ to illustrate running out of data ]
```

```
  let (f x) if = 0 x nil
      cons * 2 display read-bit (f - x 1)
  (f 5)
```

```
define      two*
value      (((' (lambda (f) (f 5))) (' (lambda (x) (if (= 0 x)
      nil (cons (* 2 (display (read-bit))) (f (- x 1)))
      ))))
```

```
try 6 two* '(1 0 1 0 1)
```

```
expression (try 6 two* (' (1 0 1 0 1)))
value      (failure out-of-time (1 0 1 0 1))
```

```
try 7 two* '(1 0 1 0 1)
```

```
expression (try 7 two* (' (1 0 1 0 1)))
value      (success (2 0 2 0 2) (1 0 1 0 1))
```

```
try 7 two* '(1 0 1)
```

```
expression (try 7 two* (' (1 0 1)))
value      (failure out-of-data (1 0 1))
```

```
try no-time-limit two* '(1 0 1)
```

```
expression (try no-time-limit two* (' (1 0 1)))
value      (failure out-of-data (1 0 1))
```

```
try 18
```

```
'let (f x) if = 0 x nil
      cons * 2 display read-bit (f - x 1)
  (f 16)
```

```
bits 'a
```

```
expression (try 18 (' (((' (lambda (f) (f 16))) (' (lambda (x)
      (if (= 0 x) nil (cons (* 2 (display (read-bit)))
```

```

                (f (- x 1)))))) (bits (' a)))
value          (success (0 2 2 0 0 0 0 2 0 0 0 0 2 0 2 0) (0 1 1
                0 0 0 0 1 0 0 0 0 1 0 1 0))

```

```

[ illustrate nested try's ]
[ most constraining limit wins ]

```

```

try 20
'cons abcdef try 10
'let (f n) (f display + n 1) (f 0) [infinite loop]
nil nil

expression (try 20 (' (cons abcdef (try 10 (' ((' (lambda (f)
                (f 0))) (' (lambda (n) (f (display (+ n 1)))))))
                nil))) nil)
value       (success (abcdef failure out-of-time (1 2 3 4 5 6
                7 8 9)) ())

```

```

try 10
'cons abcdef try 20
'let (f n) (f display + n 1) (f 0) [infinite loop]
nil nil

expression (try 10 (' (cons abcdef (try 20 (' ((' (lambda (f)
                (f 0))) (' (lambda (n) (f (display (+ n 1)))))))
                nil))) nil)
value       (failure out-of-time ())

```

```

try 10
'cons abcdef try 20
'let (f n) (f debug + n 1) (f 0) [infinite loop]
nil nil

expression (try 10 (' (cons abcdef (try 20 (' ((' (lambda (f)
                (f 0))) (' (lambda (n) (f (debug (+ n 1))))))) ni
                1))) nil)
debug      1
debug      2
debug      3

```



```
debug      4
debug      5
debug      6
debug      7
debug      8
value      (failure out-of-time ())
```

```
try no-time-limit
'cons abcdef try 20
'let (f n) (f display + n 1) (f 0) [infinite loop]
nil nil
```

```
expression (try no-time-limit (' (cons abcdef (try 20 (' (('
  (lambda (f) (f 0))) (' (lambda (n) (f (display (+
    n 1))))))) nil))) nil)
value      (success (abcdef failure out-of-time (1 2 3 4 5 6
  7 8 9 10 11 12 13 14 15 16 17 18 19)) ())
```

```
try 10
'cons abcdef try no-time-limit
'let (f n) (f display + n 1) (f 0) [infinite loop]
nil nil
```

```
expression (try 10 (' (cons abcdef (try no-time-limit (' (('
  (lambda (f) (f 0))) (' (lambda (n) (f (display (+
    n 1))))))) nil))) nil)
value      (failure out-of-time ())
```

[ illustrate read-bit & read-exp ]

read-bit

```
expression (read-bit)
value      out-of-data
```

read-exp

```
expression (read-exp)
value      out-of-data
```

```
try 0 'read-bit nil
```

```
expression (try 0 (' (read-bit)) nil)
value      (failure out-of-data ())
```

```
try 0 'read-exp nil
```

```
expression (try 0 (' (read-exp)) nil)
value      (failure out-of-data ())
```

```
try 0 'read-exp bits 'abc
```

```
expression (try 0 (' (read-exp)) (bits (' abc)))
value      (success abc ())
```

```
try 0 'read-exp bits '(abc def)
```

```
expression (try 0 (' (read-exp)) (bits (' (abc def))))
value      (success (abc def) ())
```

```
try 0 'read-exp bits '(abc(def ghi)jkl)
```

```
expression (try 0 (' (read-exp)) (bits (' (abc (def ghi) jkl)
)))
value      (success (abc (def ghi) jkl) ())
```

```
try 0 'cons read-exp cons read-bit nil bits 'abc
```

```
expression (try 0 (' (cons (read-exp) (cons (read-bit) nil)))
            (bits (' abc)))
value      (failure out-of-data ())
```

```
try 0 'cons read-exp cons read-bit nil append bits 'abc '(0)
```

```
expression (try 0 (' (cons (read-exp) (cons (read-bit) nil)))
            (append (bits (' abc)) (' (0))))
value      (success (abc 0) ())
```

```
try 0 'cons read-exp cons read-bit nil append bits 'abc '(1)
```

```
expression (try 0 (' (cons (read-exp) (cons (read-bit) nil)))
              (append (bits (' abc)) (' (1))))
```

```
value      (success (abc 1) ())
```

```
try 0 'read-exp bits '(a b c)
```

```
expression (try 0 (' (read-exp)) (bits (' (a b c))))
```

```
value      (success (a b c) ())
```

```
try 0 'cons read-exp cons read-exp nil bits '(a b c)
```

```
expression (try 0 (' (cons (read-exp) (cons (read-exp) nil)))
              (bits (' (a b c))))
```

```
value      (failure out-of-data ())
```

```
try 0 'cons read-exp cons read-exp nil
      append bits '(a b c) bits '(d e f)
```

```
expression (try 0 (' (cons (read-exp) (cons (read-exp) nil)))
              (append (bits (' (a b c))) (bits (' (d e f))))))
```

```
value      (success ((a b c) (d e f)) ())
```

```
bits 'a [ to get characters codes ]
```

```
expression (bits (' a))
```

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

```
try 0 'read-exp '(0 1 1 0 0 0 0 1) ['a' but no \n character]
```

```
expression (try 0 (' (read-exp)) (' (0 1 1 0 0 0 0 1)))
```

```
value      (failure out-of-data ())
```

```
try 0 'read-exp '(0 1 1 0 0 0 0 1 0 0 0 0 1 0 1)[0 missing]
```

```
expression (try 0 (' (read-exp)) (' (0 1 1 0 0 0 0 1 0 0 0 0
1 0 1)))
```

```
value      (failure out-of-data ())
```

```

try 0 'read-exp '(0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0) [okay]

expression (try 0 (' (read-exp)) (' (0 1 1 0 0 0 0 1 0 0 0 0
      1 0 1 0)))
value      (success a ())

[ if we get to \n reading 8 bits at a time,
  we will always interpret as a valid S-expression ]
try 0 'read-exp
      '(0 0 0 0 1 0 1 0) [nothing in record; only \n]

expression (try 0 (' (read-exp)) (' (0 0 0 0 1 0 1 0)))
value      (success () ())

try 0 'read-exp '(1 1 1 1 1 1 1 1 [unprintable character]
      0 0 0 0 1 0 1 0) [is deleted]

expression (try 0 (' (read-exp)) (' (1 1 1 1 1 1 1 1 0 0 0 0
      1 0 1 0)))
value      (success () ())

bits () [ to get characters codes ]

expression (bits ())
value      (0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 1 0 1 0)

[ three left parentheses==>three right parentheses supplied ]
try 0 'read-exp '(0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0
      0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0)

expression (try 0 (' (read-exp)) (' (0 0 1 0 1 0 0 0 0 0 1 0
      1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 1 0)))
value      (success (((()))) ())

[ right parenthesis 'a'==>left parenthesis supplied ]
try 0 'read-exp '(0 0 1 0 1 0 0 1 0 1 1 0 0 0 0 1
      0 0 0 0 1 0 1 0) [ & extra 'a' ignored ]

```

```
expression (try 0 (' (read-exp)) (' (0 0 1 0 1 0 0 1 0 1 1 0
                                0 0 0 1 0 0 0 0 1 0 1 0)))
value      (success () ())
```

```
[ 'a' right parenthesis==>'a' is seen & parenthesis ]
try 0 'read-exp '(0 1 1 0 0 0 0 1 0 0 1 0 1 0 0 1
                  0 0 0 0 1 0 1 0) [ is ignored ]
```

```
expression (try 0 (' (read-exp)) (' (0 1 1 0 0 0 0 1 0 0 1 0
                                1 0 0 1 0 0 0 0 1 0 1 0)))
value      (success a ())
```

End of LISP Run

Elapsed time is 16 seconds.

## **godel.r**

LISP Interpreter Run

```
[[[
  Show that a formal system of lisp complexity
  H_lisp (FAS) = N cannot enable us to exhibit
  an elegant S-expression of size greater than N + 410.
  An elegant lisp expression is one with the property
  that no smaller S-expression has the same value.
  Setting: formal axiomatic system is never-ending
  lisp expression that displays elegant S-expressions.
]]]
```

[Here is the key expression.]

define expression

```
let (examine x)
  if atom x false
  if < n size car x car x
  (examine cdr x)
```

```
let fas 'display ^ 10 430 [insert FAS here preceeded by ']
```

```
let n + 410 size fas
```

```
let t 0
```

```
let (loop)
  let v try t fas nil
  let s (examine caddr v)
  if s eval s
  if = success car v failure
  let t + t 1
  (loop)
```

```
(loop)
```

```
define      expression
value      ((' (lambda (examine) ((' (lambda (fas) ((' (lambda
a (n) ((' (lambda (t) ((' (lambda (loop) (loop)))
(' (lambda () ((' (lambda (v) ((' (lambda (s) (if
s (eval s) (if (= success (car v)) failure ((' (la
mbda (t) (loop))) (+ t 1)))))) (examine (car (cdr
(cdr v)))))) (try t fas nil)))))) 0))) (+ 410 (s
ize fas)))) (' (display (^ 10 430)))) (' (lambda
a (x) (if (atom x) false (if (< n (size (car x))
(car x) (examine (cdr x))))))
```

```
[Size expression.]
```

```
size expression
```

```
expression (size expression)
```

```
value      430
```

```
[Run expression & show that it knows its own size
and can find something bigger than it is.]
```

```
eval expression
```

```
expression (eval expression)
```

```
expression ((' (lambda (examine) ((' (lambda (fas) ((' (lambda
a (n) ((' (lambda (t) ((' (lambda (loop) (loop)))
(' (lambda () ((' (lambda (v) ((' (lambda (s) (if
s (eval s) (if (= success (car v)) failure ((' (la
mbda (t) (loop))) (+ t 1)))))) (examine (car (cdr
(cdr v)))))) (try t fas nil)))))) 0))) (+ 410 (s
```

```

size fas)))))) (' (display (~ 10 429)))))) (' (lambda
a (x) (if (atom x) false (if (< n (size (car x)))
(car x) (examine (cdr x)))))))
value      failure

```

End of LISP Run

Elapsed time is 2 seconds.

## utm.r

LISP Interpreter Run

```

[[[
First steps with my new construction for
a self-delimiting universal Turing machine.
We show that
     $H(x,y) \leq H(x) + H(y) + c$ 
and determine  $c$ .
Consider a bit string  $x$  of length  $|x|$ .
We also show that
     $H(x) \leq 2|x| + c$ 
and that
     $H(x) \leq |x| + H(\text{the binary string for } |x|) + c$ 
and determine both these  $c$ 's.
]]]

```

```

[
Here is the self-delimiting universal Turing machine!
]
define (U p) cadr try no-time-limit 'eval read-exp p

define      U
value      (lambda (p) (car (cdr (try no-time-limit (' (eval
(read-exp))) p))))

(U bits 'cons x cons y cons z nil)

```



```
expression (U (bits (' (cons x (cons y (cons z nil))))))
value      (x y z)
```

```
(U append bits 'cons a debug read-exp
      bits '(b c d)
)
```

```
expression (U (append (bits (' (cons a (debug (read-exp))))
      (bits (' (b c d)))))
debug      (b c d)
value      (a b c d)
```

```
[
  The length of alpha in bits is the
  constant c in  $H(x) \leq 2|x| + 2 + c$ .
]
```

```
define alpha
let (loop) let x read-bit
            let y read-bit
            if = x y
              cons x (loop)
            nil
(loop)
```

```
define      alpha
value      ((' (lambda (loop) (loop))) (' (lambda () ((' (lambda (x) ((' (lambda (y) (if (= x y) (cons x (loop)) nil))) (read-bit)))) (read-bit)))))
```

```
length bits alpha
```

```
expression (length (bits alpha))
value      1104
```

```
(U
  append
    bits alpha
    '(0 0 1 1 0 0 1 1 0 1)
)
```

```
expression (U (append (bits alpha) (' (0 0 1 1 0 0 1 1 0 1)))
           )
value      (0 1 0 1)
```

```
(U
  append
    bits alpha
    '(0 0 1 1 0 0 1 1 0 0)
)
```

```
expression (U (append (bits alpha) (' (0 0 1 1 0 0 1 1 0 0)))
           )
value      out-of-data
```

```
[
  The length of beta in bits is the
  constant c in  $H(x,y) \leq H(x) + H(y) + c$ .
]
```

```
define beta
cons eval read-exp
cons eval read-exp
  nil
```

```
define      beta
value      (cons (eval (read-exp)) (cons (eval (read-exp)) ni
1))
```

```
length bits beta
```

```
expression (length (bits beta))
value      432
```

```
(U
  append
    bits beta
  append
    bits 'cons a cons b cons c nil
    bits 'cons x cons y cons z nil
```

```
)

expression (U (append (bits beta) (append (bits (' (cons a (c
ons b (cons c nil)))))) (bits (' (cons x (cons y (c
ons z nil))))))))
value      ((a b c) (x y z))
```

```
(U
  append
    bits beta
  append
    append bits alpha '(0 0 1 1 0 0 1 1 0 1)
    append bits alpha '(1 1 0 0 1 1 0 0 1 0)
)
```

```
expression (U (append (bits beta) (append (append (bits alpha
) (' (0 0 1 1 0 0 1 1 0 1))) (append (bits alpha)
(' (1 1 0 0 1 1 0 0 1 0))))))
value      ((0 1 0 1) (1 0 1 0))
```

```
[
  The length of gamma in bits is the
  constant c in  $H(x) \leq |x| + H(|x|) + c$ 
]
```

```
define gamma
let (loop k)
  if = 0 k nil
  cons read-bit (loop - k 1)
(loop base2-to-10 eval read-exp)
```

```
define      gamma
value      ((' (lambda (loop) (loop (base2-to-10 (eval (read-
exp)))))) (' (lambda (k) (if (= 0 k) nil (cons (re
ad-bit) (loop (- k 1))))))
```

```
length bits gamma
```

```
expression (length (bits gamma))
value      1024
```

```

(U
  append
    bits gamma
  append
    [Arbitrary program for U to compute number of bits]
    bits' '(1 0 0 0)
    [That many bits of data]
    '(0 0 0 0 0 0 0 1)
)

expression (U (append (bits gamma) (append (bits (' (' (1 0 0
                                             0)))) (' (0 0 0 0 0 0 0 1)))))
value      (0 0 0 0 0 0 0 1)

```

End of LISP Run

Elapsed time is 19 seconds.

## **godel2.r**

LISP Interpreter Run

```

[[[
  Show that a formal system of complexity N
  can't prove that a specific object has
  complexity > N + 4872.
  Formal system is a never halting lisp expression
  that output pairs (lisp object, lower bound
  on its complexity). E.g., (x 4) means
  that x has complexity H(x) greater than or equal to 4.
]]]

```

[Here is the prefix.]

define pi

let (examine pairs)

```

if atom pairs false
if < n cadr car pairs
  car pairs
  (examine cdr pairs)

```

```
let t 0
```

```
let fas nil
```

```

let (loop)
  let v try t 'eval read-exp fas
  let n + 4872 length fas
  let p (examine caddr v)
  if p car p
  if = car v success failure
  if = cadr v out-of-data
    let fas append fas cons read-bit nil
    (loop)
  if = cadr v out-of-time
    let t + t 1
    (loop)
  unexpected-condition

```

```
(loop)
```

```

define      pi
value      ((' (lambda (examine) ((' (lambda (t) ((' (lambda
(fas) ((' (lambda (loop) (loop))) (' (lambda () (('
' (lambda (v) ((' (lambda (n) ((' (lambda (p) (if
p (car p) (if (= (car v) success) failure (if (= (
car (cdr v)) out-of-data) ((' (lambda (fas) (loop)
)) (append fas (cons (read-bit) nil))) (if (= (car
(cdr v)) out-of-time) ((' (lambda (t) (loop))) (+
t 1)) unexpected-condition)))))) (examine (car (c
dr (cdr v))))) (+ 4872 (length fas)))) (try t (
' (eval (read-exp))) fas)))))) nil))) 0))) (' (la
mbda (pairs) (if (atom pairs) false (if (< n (car
(cdr (car pairs)))) (car pairs) (examine (cdr pair
s)))))))

```

[Size pi.]

length bits pi

expression (length (bits pi))

value 4872

[Size pi + fas.]

length

append bits pi

bits 'display '(xyz 9999)

expression (length (append (bits pi) (bits (' (display (' (xyz 9999)))))))

value 5072

[Here pi finds something suitable.]

cadr try no-time-limit 'eval read-exp

append bits pi

bits 'display '(xyz 5073)

expression (car (cdr (try no-time-limit (' (eval (read-exp)))  
(append (bits pi) (bits (' (display (' (xyz 5073)  
))))))))

value xyz

[Here pi doesn't find anything suitable.]

cadr try no-time-limit 'eval read-exp

append bits pi

bits 'display '(xyz 5072)

expression (car (cdr (try no-time-limit (' (eval (read-exp)))  
(append (bits pi) (bits (' (display (' (xyz 5072)  
))))))))

value failure

End of LISP Run

Elapsed time is 153 seconds.

## omega.r

LISP Interpreter Run

```
[[[[ Omega in the limit from below! ]]]]
```

```
define (all-bit-strings-of-size k)
  if = 0 k ' (())
  (extend-by-one-bit (all-bit-strings-of-size - k 1))
```

```
define      all-bit-strings-of-size
value      (lambda (k) (if (= 0 k) (' (())) (extend-by-one-bit
      (all-bit-strings-of-size (- k 1)))))
```

```
define (extend-by-one-bit x)
  if atom x nil
  cons append car x '(0)
  cons append car x '(1)
  (extend-by-one-bit cdr x)
```

```
define      extend-by-one-bit
value      (lambda (x) (if (atom x) nil (cons (append (car x)
      (' (0))) (cons (append (car x) (' (1))) (extend-by-one-bit
      (cdr x))))))
```

```
define (count-halt p)
  if atom p 0
  +
  if = success car try t 'eval read-exp car p
  1 0
  (count-halt cdr p)
```

```
define      count-halt
value      (lambda (p) (if (atom p) 0 (+ (if (= success (car
      (try t (' (eval (read-exp))) (car p)))) 1 0) (count-halt
      (cdr p)))))
```

```
t-halt (cdr p))))))
```

```
define (omega t) cons (count-halt (all-bit-strings-of-size t))
                      cons /
                      cons ^ 2 t
                      nil
```

```
define      omega
value      (lambda (t) (cons (count-halt (all-bit-strings-of-
size t)) (cons / (cons (^ 2 t) nil)))))
```

```
(omega 0)
```

```
expression (omega 0)
value      (0 / 1)
```

```
(omega 1)
```

```
expression (omega 1)
value      (0 / 2)
```

```
(omega 2)
```

```
expression (omega 2)
value      (0 / 4)
```

```
(omega 3)
```

```
expression (omega 3)
value      (0 / 8)
```

```
(omega 8)
```

```
expression (omega 8)
value      (1 / 256)
```

End of LISP Run

Elapsed time is 38 seconds.



**omega2.r**

LISP Interpreter Run

[[[[ Omega in the limit from below! ]]]]

```

define (count-halt prefix bits-left-to-extend)
  if = bits-left-to-extend 0
  if = success car try t 'eval read-exp prefix
    1 0
  + (count-halt append prefix '(0) - bits-left-to-extend 1)
    (count-halt append prefix '(1) - bits-left-to-extend 1)

define      count-halt
value      (lambda (prefix bits-left-to-extend) (if (= bits-left-to-extend 0) (if (= success (car (try t (' (eval (read-exp))) prefix))) 1 0) (+ (count-halt (append prefix '(0))) (- bits-left-to-extend 1)) (count-halt (append prefix '(1)) (- bits-left-to-extend 1)))))

define (omega t) cons (count-halt nil t)
              cons /
              cons ^ 2 t
              nil

define      omega
value      (lambda (t) (cons (count-halt nil t) (cons / (cons (^ 2 t) nil)))))

(omega 0)

expression (omega 0)
value      (0 / 1)

(omega 1)

expression (omega 1)
value      (0 / 2)

```

(omega 2)

expression (omega 2)  
value (0 / 4)

(omega 3)

expression (omega 3)  
value (0 / 8)

(omega 8)

expression (omega 8)  
value (1 / 256)

End of LISP Run

Elapsed time is 33 seconds.

## omega3.r

LISP Interpreter Run

```
[[[
Show that
     $H(\Omega_n) > n - 8000.$ 
 $\Omega_n$  is the first  $n$  bits of  $\Omega$ ,
where we choose
     $\Omega = xxx0111111...$ 
instead of
     $\Omega = xxx1000000...$ 
if necessary.
]]]
```

[Here is the prefix.]

define pi

```

let (count-halt prefix bits-left-to-extend)
  if = bits-left-to-extend 0
  if = success car try t 'eval read-exp prefix
    1 0
  + (count-halt append prefix '(0) - bits-left-to-extend 1)
    (count-halt append prefix '(1) - bits-left-to-extend 1)

let (omega t) cons (count-halt nil t)
  cons /
  cons ^ 2 t
  nil

let w eval read-exp

let n length w

let w cons base2-to-10 w
  cons /
  cons ^ 2 n
  nil

let (loop t)
  if (<=rat w (omega t))
    (big nil n)
    (loop + t 1)

let (<=rat x y)
  <= * car x caddr y * caddr x car y

let (big prefix bits-left-to-add)
  if = 0 bits-left-to-add
  cons cadr try t 'eval read-exp prefix
    nil
  append (big append prefix '(0) - bits-left-to-add 1)
    (big append prefix '(1) - bits-left-to-add 1)

(loop 0)

```

```

define      pi
value      ((' (lambda (count-halt) ((' (lambda (omega) ((' (
lambda (w) ((' (lambda (n) ((' (lambda (w) ((' (la
mbda (loop) ((' (lambda (<=rat) ((' (lambda (big)
(loop 0))) (' (lambda (prefix bits-left-to-add) (i
f (= 0 bits-left-to-add) (cons (car (cdr (try t ('
(eval (read-exp))) prefix))) nil) (append (big (a
ppend prefix (' (0))) (- bits-left-to-add 1)) (big
(append prefix (' (1))) (- bits-left-to-add 1))))
)))) (' (lambda (x y) (<= (* (car x) (car (cdr (c
dr y)))) (* (car (cdr (cdr x))) (car y)))))) ('
(lambda (t) (if (<=rat w (omega t)) (big nil n) (l
oop (+ t 1)))))) (cons (base2-to-10 w) (cons / (
cons (^ 2 n) nil)))) (length w))) (eval (read-e
xp)))) (' (lambda (t) (cons (count-halt nil t) (c
ons / (cons (^ 2 t) nil)))))) (' (lambda (prefix
bits-left-to-extend) (if (= bits-left-to-extend 0
) (if (= success (car (try t (' (eval (read-exp)))
prefix))) 1 0) (+ (count-halt (append prefix (' (
0))) (- bits-left-to-extend 1)) (count-halt (appen
d prefix (' (1))) (- bits-left-to-extend 1))))))

```

[Run pi.]

```
cadr try no-time-limit 'eval read-exp
```

```
append bits pi
```

```
bits '
```

```
[Program to compute first n = 8 bits of Omega]
```

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

```
expression (car (cdr (try no-time-limit (' (eval (read-exp)))
(append (bits pi) (bits (' (' (0 0 0 0 0 0 0 1)))
))))))
```

```
value      (out-of-data out-of-data out-of-data out-of-data o
ut-of-data out-of-data out-of-data out-of-data out
-of-data out-of-data () out-of-data out-of-data ou
t-of-data out-of-data out-of-data out-of-data out-
of-data out-of-data out-of-data out-of-data out-of
-data out-of-data out-of-data out-of-data out-of-d
ata out-of-data out-of-data out-of-data out-of-dat
```

[illegible]

of-data out-of-data out-of-data out-of-data out-of-  
 -data out-of-data out-of-data out-of-data out-of-d  
 ata out-of-data out-of-data out-of-data out-of-dat  
 a out-of-data out-of-data out-of-data out-of-data  
 out-of-data out-of-data out-of-data out-of-data ou  
 t-of-data out-of-data out-of-data out-of-data out-  
 of-data out-of-data out-of-data out-of-data out-of  
 -data out-of-data out-of-data out-of-data out-of-d  
 ata out-of-data out-of-data out-of-data out-of-dat  
 a out-of-data out-of-data out-of-data out-of-data  
 out-of-data out-of-data out-of-data out-of-data ou  
 t-of-data out-of-data out-of-data out-of-data out-  
 of-data out-of-data out-of-data out-of-data out-of  
 -data out-of-data out-of-data out-of-data out-of-d  
 ata out-of-data out-of-data out-of-data out-of-dat  
 a out-of-data)

[Size pi.]

length bits pi

expression (length (bits pi))

value 8000

End of LISP Run

Elapsed time is 148 seconds.

## godel3.r

LISP Interpreter Run

[[[

Show that a formal system of complexity  $N$   
 can't determine more than  $N + 8000 + 7328$   
 $= N + 15328$  bits of  $\Omega$ .

Formal system is a never halting lisp expression  
 that outputs lists of the form  $(1\ 0\ X\ 0\ X\ X\ X\ 1\ 0)$ .  
 This stands for the fractional part of  $\Omega$ ,

and means that these 0,1 bits of Omega are known.  
X stands for an unknown bit.  
]]]

[Here is the prefix.]

define pi

```
let (number-of-bits-determined w)
  if atom w 0
  + (number-of-bits-determined cdr w)
    if = X car w
      0
      1
```

```
let (supply-missing-bits w)
  if atom w nil
  cons if = X car w
    read-bit
    car w
  (supply-missing-bits cdr w)
```

```
let (examine w)
  if atom w false
  [if < n (number-of-bits-determined car w)]
  [ Change n to 1 here so will succeed. ]
  if < 1 (number-of-bits-determined car w)
    car w
    (examine cdr w)
```

let t 0

let fas nil

```
let (loop)
  let v try t 'eval read-exp fas
  let n + 8000 + 7328 length fas
  let w (examine caddr v)
  if w (supply-missing-bits w)
```

```

if = car v success failure
if = cadr v out-of-data
  let fas append fas cons read-bit nil
  (loop)
if = cadr v out-of-time
  let t + t 1
  (loop)
unexpected-condition

```

```
(loop)
```

```

define      pi
value      ((' (lambda (number-of-bits-determined) ((' (lambda
a (supply-missing-bits) ((' (lambda (examine) ((' (lambda
(lambda (t) ((' (lambda (fas) ((' (lambda (loop) (
loop))) (' (lambda () ((' (lambda (v) ((' (lambda
(n) ((' (lambda (w) (if w (supply-missing-bits w)
(if (= (car v) success) failure (if (= (car (cdr v)
)) out-of-data) ((' (lambda (fas) (loop))) (append
fas (cons (read-bit) nil))) (if (= (car (cdr v))
out-of-time) ((' (lambda (t) (loop))) (+ t 1)) une
xpected-condition)))))) (examine (car (cdr (cdr v)
)))))) (+ 8000 (+ 7328 (length fas)))))) (try t ('
(eval (read-exp))) fas)))))) nil))) 0))) (' (lam
bda (w) (if (atom w) false (if (< 1 (number-of-bit
s-determined (car w))) (car w) (examine (cdr w))))
)))) (' (lambda (w) (if (atom w) nil (cons (if (=
X (car w)) (read-bit) (car w)) (supply-missing-bi
ts (cdr w)))))) (' (lambda (w) (if (atom w) 0 (
+ (number-of-bits-determined (cdr w)) (if (= X (ca
r w)) 0 1))))))

```

```
[Size pi.]
```

```
length bits pi
```

```
expression (length (bits pi))
```

```
value      7328
```

```
[Run pi.]
```



```
cadr try no-time-limit 'eval read-exp
append bits pi
append [Toy formal system with only one theorem.]
      bits 'display '(1 X 0)
      [Missing bit of omega that is needed.]
      '(1)
```

```
expression (car (cdr (try no-time-limit (' (eval (read-exp)))
      (append (bits pi) (append (bits (' (display (' (1
      X 0)))) (' (1)))))) (' (1))))))
value      (1 1 0)
```

End of LISP Run

Elapsed time is 94 seconds.

# APPENDIX. LISP INTERPRETER IN MATHEMATICA

(\*\*\*\*\* lisp.m: LISP interpreter \*\*\*\*\*)

(\*\*\*\*\* INSTRUCTIONS FOR USING THIS LISP INTERPRETER \*\*\*\*\*)

To run the lisp.m interpreter, first enter Mathematica using the command math. To load the interpreter, enter

```
<< lisp.m
```

To run a LISP program xyz.l and produce xyz.r, enter

```
run @ "xyz"
```

To run several programs, enter

```
run /@ {"xxx","yyy","zzz"}
```

To run the eight LISP programs in the course, enter

```
runall
```

Type Exit to exit from Mathematica.

Here is how to run the programs that compute the halting probability  $\Omega$  in the limit from below:

```
math
```

```
<< lisp.m
```

```
run /@ {"omega","omega2"}
```

```
run @ "omega3"
```

```
Exit
```

Reference: Wolfram, Mathematica, 2nd Ed., Addison-Wesley, 1991.

\*\*\*\*\*)

```
getbit[] :=
```

```
Block[ {x},
```

```
trouble = False; (* Mma bug bypass *)
```

```

If[ atom@ tape, (trouble = True; Throw@ "out-of-data")];
x = car@ tape;
tape = cdr@ tape;
If[ x === 0, 0, 1 ]
]

getchar[] := FromCharacterCode[
  128*getbit[] + 64*getbit[] + 32*getbit[] + 16*getbit[] +
  8*getbit[] + 4*getbit[] + 2*getbit[] + getbit[]
]

getrecord[] :=
Block[ { c, line = "", str },
  inputbuffer2 = {};
  While[ "\n" != (c = getchar[]),
    line = line <> c ];
  If[ trouble, Throw@ "out-of-data"]; (* Mma bug bypass *)
  (* keep only printable ASCII codes *)
  line = FromCharacterCode@
  Cases[ ToCharacterCode@ line, n_Integer /; 32 <= n < 127 ];
  str = StringToStream@ line;
  inputbuffer2 = ReadList[str, Word, TokenWords->{"(", ")"}];
  Close@ str;
  (* convert strings of digits to integers *)
  inputbuffer2 = If[DigitQ@#, ToExpression@#, #]& /@ inputbuffer2 ;
]

getexp[rparenokay_:False] :=
Block[ { w, d, l = {} },
  (* supply unlimited number of )'s if tokens run out *)
  If[ inputbuffer2 == {}, w = ")",
    w = First@ inputbuffer2;
    inputbuffer2 = Rest@ inputbuffer2 ];
  Switch[
    w,
    ")", Return@ If[rparenokay, ")", {}],
    "(",
  (While[ "]" != (d = getexp[True]),
    AppendTo[l, d]
  ];
  Return@ l
),
_, w
]

```

]

```

atom[x_] :=
  MatchQ[ x, {}|_String|_Integer ]
car[x_] :=
  If[ atom@ x, x, First@ x ]
cdr[x_] :=
  If[ atom@ x, x, Rest@ x ]
cons[x_,y_] :=
  If[ MatchQ[y,_String|_Integer], x, Prepend[y,x] ]

eval[e2_,a_,d2_] :=

Block[ {e = e2, d = d2, f, args, x, y, z},
  If[ MatchQ[e,_Integer], Return@ e ];
  If[ atom@ e, Block[ {names,values,pos},
    {names,values} = a;
    pos = Position[names,e,{1}];
    Return@ If[ pos == {}, e, values[[pos[[1,1]]]] ]
  ];
  f = eval[car@ e,a,d];
  e = cdr@ e;
  Switch[
    f,
    "", Return@ car@ e,
    "if", Return@
  If[
    eval[car@ e,a,d] != "false",
    eval[car@cdr@ e,a,d],
    eval[car@cdr@cdr@ e,a,d]
  ]
  ];
  args = eval[#,a,d]& /@ e;
  x = car@ args;
  y = car@cdr@ args;
  z = car@cdr@cdr@ args;
  Switch[
    f,
    "read-bit", Return@ getbit[],
    "read-exp", Return@ (getrecord[]; getexp[]),
    "bits", Return@ Flatten[ ( Rest@ IntegerDigits[256 + #, 2] )& /@
      ToCharacterCode[ output@x <> "\n" ] ],
    "car", Return@ car@ x,
    "cdr", Return@ cdr@ x,

```

```

"cons", Return@ cons[x,y],
"size", Return@ StringLength@ output@ x,
"length", Return@ Length@ x,
"+", Return@ (nmb@x + nmb@y),
"-", Return@ If[ nmb@x < nmb@y, 0, nmb@x - nmb@y ],
"*, Return@ (nmb@x * nmb@y),
"^", Return@ (nmb@x ^ nmb@y),
"<", Return@ If[nmb@x < nmb@y, "true", "false"],
">", Return@ If[nmb@x > nmb@y, "true", "false"],
">=", Return@ If[nmb@x >= nmb@y, "true", "false"],
"<=", Return@ If[nmb@x <= nmb@y, "true", "false"],
"base10-to-2", Return@ IntegerDigits[nmb@x, 2],
"base2-to-10", Return@ Fold[ (2 #1 + If[#2==0,0,1])&, 0, x ],
"append", Return@ Join[ If[atom@x,{},x], If[atom@y,{},y] ],
"atom", Return@ If[ atom@ x, "true", "false" ],
"=", Return@ If[ x === y, "true", "false" ],
"display", Return@ (AppendTo[out,x];
                    If[ display, print[ "display", output@ x ] ];
                    x),
"debug", Return@ (print[ "debug", output@ x ];
                  x)
];
If[ d == 0, Throw@ "out-of-time" ];
d--;
Switch[
f,
"eval", Return@ eval[x,,d],
"try", Return@
Block[ {out = {}, tape = z, display = False, xx},
If[ x === "no-time-limit", xx = Infinity, xx = nmb@ x ];
If[ xx < d,
Catch@ {"success",eval[y,,xx],out} //
If[ # === "out-of-time", {"failure",#,out}, # ] & ,
Catch@ {"success",eval[y,,d],out} //
If[ # === "out-of-time", Throw@ #, # ] &
] //
If[ # === "out-of-data", {"failure",#,out}, # ] & ] (* end block *)
]; (* end switch *)
f = cdr@ f;
eval[ car@cdr@ f, bind[car@ f,args,a], d ]
]

nmb[n_Integer] := n
nmb[_] := 0

```

```

bind[vars_?atom,args_,a_] :=
  a

bind[vars_,args_,a_] :=
Block[ {names,values,pos},
  {names,values} = bind[ cdr@ vars, cdr@ args, a ];
  pos = Position[names, car@ vars, {1}];
  {Prepend[Delete[names,pos], car@ vars],
   Prepend[Delete[values,pos], car@ args]}
]

eval[e_] :=
(
  out = tape = {};
  display = True;
  print[ "expression", output@ e ];
  Catch[ eval[ e, {names,defs}, Infinity ] ]
)

eval[e_,,d_] := eval[e,{{"nil"},{}}],d]

run[fn_] := run[fn, "lisp.m", ".r"]

word2[]:=
Block[ {w,line,str},
While[
  inputbuffer == {},
  line = Read[i,Record];
  If[ line == EndOfFile, Abort[] ];
  Print@ line;
  WriteString[o,line,"\n"];
  (* keep only printable ASCII codes *)
  line = FromCharacterCode@
Cases[ ToCharacterCode@ line, n_Integer /; 32 <= n < 127 ];
  str = StringToStream@ line;
  inputbuffer = ReadList[str, Word,
                        TokenWords->{"(", ")", "[", "]", "'", "\""}];
  Close@ str
];
w = First@ inputbuffer;
inputbuffer = Rest@ inputbuffer;
(* convert string of digits to integer *)
If[DigitQ@w, ToExpression@w, w]

```

]

```
word[] :=
Block[ {w},
While[ True,
  w = word2[];
  If[ w != "[", Return@ w ];
  While[ word[] != "]" ]
]
]
```

```
get[sexp_:False,rparenokay_:False] :=
```

```
Block[ {w = word[], d, l = {}, name, def, body, varlist},
Switch[
  w,
  ")"", Return@ If[rparenokay,""),{}],
  "(",
  While[ "]" != (d = get[sexp,True]),
  AppendTo[l,d]
];
Return@ l
];
If[ sexp, Return@ w ];
Switch[
  w,
  "\"", get[True],
  "cadr",
  {"car",{"cdr",get[]}},
  "caddr",
  {"car",{"cdr",{"cdr",get[]}}},
  "let",
  {name,def,body} = {get[],get[],get[]};
  If[
    !MatchQ[name,{_String|_Integer}],
    varlist = Rest@ name;
    name = First@ name;
    def = {"'",{"lambda",varlist,def}}
  ];
  {"'",{"lambda",{name},body}},def},
  "read-bit"|"read-exp",
    {w},
  "car"|"cdr"|"atom"|"'"|"display"|"eval"|"bits"|"debug"|"length"|"size"|"base2-to-10"|"base10-to-2",
```

```

        {w,get[]},
"cons"|"="|"lambda"|"append"|"define"|
"+"|"-"|"*"|"^"|"<"|">"|"<="|">=",
        {w,get[],get[]},
"if"|"let"|"try", {w,get[],get[],get[]},
], w
]

(* output S-exp *)
output2[x_String] := x<>" "
output2[x_Integer] := ToString[x]<>" "
output2[{x___}] :=
Block[ {s},
  s = StringJoin["(", output2 /@ {x}];
  If[ StringTake[s,-1] == " ", s = StringDrop[s,-1] ];
  s <> ")" "
]
output[x_] := StringDrop[ output2@x ,-1 ]

blanks = StringJoin@ Table[" ",{12}]

print[x_,y_] := (print2[x,StringTake[y,50]];
  print["",StringDrop[y,50]]) /; StringLength[y] > 50
print[x_,y_] := print2[x,y]
print2[x_,y_] := print3[StringTake[x<>blanks,12]<>y]
print3[x_] := (Print[x]; WriteString[o,x,"\n"])

let[n_,d_] :=
(
  print[ "define", output@ n ];
  print[ "value", output@ d ];
  PrependTo[names,n];
  PrependTo[defs,d];
)

run[fn_,whoami_,outputsuffix_] :=
(
  inputbuffer = {};
  names = {"nil"}; defs = {};
  t0 = SessionTime[];
  o = OpenWrite[fn<>outputsuffix];
  i = OpenRead[fn<>".l"];
  print3["LISP Interpreter Run"];

```



```

print3@ "";
CheckAbort[
While[True,
(print3@ "";
Replace[#, {
{"define", {func_, vars___}, def_} :> let[func, {"lambda", {vars}, def}],
{"define", var_, def_} :> let[var, def],
_ :> print[ "value", output@ eval@ # ]
}]
)& @ get[];
print3@ ""
],
];
print3@ "End of LISP Run";
print3@ "";
print3@ StringForm[
"Elapsed time is `` seconds.",
Round[SessionTime[]-t0]
];
Close@ i;
Close@ o
)

runall := run /@ {"examples", "godel", "utm", "godel2",
                  "omega", "omega2", "omega3", "godel3"}

$RecursionLimit = $IterationLimit = Infinity
SetOptions[$Output, PageWidth->Infinity];

```

<p>Springer Series in <b>Discrete Mathematics and Theoretical Computer Science</b></p>
--

## Editors

Douglas S. Bridges  
Department of Mathematics  
University of Waikato  
Private Bag 3105  
Hamilton  
New Zealand  
e-mail: douglas@waikato.ac.nz

Cristian S. Calude  
Department of Computer Science  
University of Auckland  
Private Bag 92019  
Auckland  
New Zealand  
e-mail: cristian@cs.auckland.ac.nz

## Advisory Editorial Board

John Casti  
Santa Fe Institute

Arto Salomaa  
Turku University

Edsger W. Dijkstra  
University of Texas at Austin

R. L. Graham  
AT&T Research  
Murray Hill

Gregory J. Chaitin  
IBM Research Division

Juris Hartmanis  
Cornell University

Helmut Jürgensen  
University of Western Ontario  
University of Potsdam

Joseph Goguen  
Oxford University

Grzegorz Rozenberg  
Leiden University

Anil Nerode  
Cornell University

Springer-Verlag Singapore announces a new series, *Discrete Mathematics and Theoretical Computer Science*, produced in cooperation with the Centre for Discrete Mathematics and Theoretical Computer Science of the Universities of Auckland and Waikato, New Zealand. This series will bring to the research community information about the latest developments on the interface between mathematics and computing, especially in the areas of artificial intelligence, combinatorial optimization, computability and complexity, and theoretical computer vision. It will focus on research monographs and proceedings of workshops and conferences, aimed at graduate students and professional researchers, and on textbooks, primarily for the advanced undergraduate or lower graduate level.

For details of forthcoming titles, please contact the publisher at:

Springer-Verlag Singapore Pte. Ltd.  
#04-01 Cencon I  
1 Tannery Road  
Singapore 347719  
Tel: (65) 842 0112  
Fax: (65) 842 0107  
e-mail: ianjs@cyberway.com.sg

Springer Series in

**Discrete Mathematics and Theoretical Computer Science**

D.S. Bridges, C.S. Calude, J. Gibbons, S. Reeves,  
I.H. Witten (Eds.), *Combinatorics, Complexity and  
Logic*. Proceedings, 1996. viii, 422 pages.

L. Groves, S. Reeves (Eds.), *Formal Methods  
Pacific '97*. Proceedings, 1997. viii, 320 pages.

G.J. Chaitin, *The Limits of Mathematics: A Course  
on Information Theory and the Limits of Formal  
Reasoning*, 1998. xii, 148 pages.