Self-Reference and Computability

In this lecture we will explore the deep connection between proofs and computation. At the heart of this connection is the notion of self-reference, and it has far-reaching consequences for the limits of computation (the Halting Problem) and the foundations of logic in mathematics (Gödel's incompleteness theorem).

The Liar's Paradox

Recall that propositions are statements that are either true or false. We saw in an earlier lecture that some statements are not well defined or too imprecise to be called propositions. But here is a statement that is problematic for more subtle reasons:

"All Cretans are liars."

So said a Cretan in antiquity, thus giving rise to the so-called liar's paradox which has amused and confounded people over the centuries. Why? Because if the statement above is true, then the Cretan was lying, which implies the statement is false. But actually the above statement isn't really a paradox; it simply yields a contradiction if we assume it is true, but if it is false then there is no problem.

A true formulation of this paradox is the following statement:

"This statement is false."

Is the statement above true? If the statement is true, then what it asserts must be true; namely that it is false. But if it is false, then it must be true. So it really is a paradox, and we see that it arises because of the self-referential nature of the statement. Around a century ago, this paradox found itself at the center of foundational questions about mathematics and computation.

We will now study how this paradox relates to computation. Before doing so, let us consider another manifestation of the paradox, created by the great logician Bertrand Russell. In a village with just one barber, every man keeps himself clean-shaven. Some of the men shave themselves, while others go to the barber. The barber proclaims:

"I shave all and only those men who do not shave themselves."

It seems reasonable then to ask the question: Does the barber shave himself? Thinking more carefully about the question though, we see that, assuming that the barber's statement is true, we are presented with the same self-referential paradox: a logically impossible scenario. If the barber does not shave himself, then according to what he announced, he shaves himself. If the barber does shave himself, then according to his statement he does not shave himself!

Self-Replicating Programs

Can we use self-reference to design a program that outputs itself? To illustrate the idea, let us consider how we can do this if we could write the program in English. Consider the following instruction:

```
Print out the following sentence:
```

EECS 70 离散数学与概率论 2020秋季

Note 12

自指与可计算性

在本次讲座中，我们将探讨证明与计算之间的深刻联系。这种联系的核心概念是自指，它对计算的极限（停机问题）以及数学逻辑的基础（哥德尔不完备性定理）有着深远的影响。

说谎者悖论

回想一下，命题是具有真假值的语句。我们在之前的讲座中看到，有些语句定义不清或过于模糊，不能称为命题。但这里有一个由于更微妙原因而存在问题的语句：

所有克里特人都是说谎者。

古时候一位克里特人如此说道，从而产生了所谓的说谎者悖论，几个世纪以来一直令人感到有趣和困惑。为什么？因为如果上面的语句为真，那么这位克里特人就在撒谎，这意味着该语句是假的。但实际上上述语句并不真正构成一个悖论；它只是在我们假设其为真时导致矛盾，而如果它是假的，则没有问题。

这个悖论的一个确切表述是以下语句：这个语句是假的。

上面的语句是真的吗？如果该语句为真，那么它所断言的内容必须为真；也就是说它是假的。但如果它是假的，那么它又必须为真。所以这确实是一个悖论，我们看到它的产生源于语句的自指性质。大约一个世纪前，这个悖论成为关于数学和计算基础问题的核心。

我们现在将研究这个悖论与计算的关系。在此之前，让我们考虑由著名逻辑学家伯特兰·罗素提出的另一个表现形式。在一个只有一个理发师的村庄里，每个男人都保持干净的脸面。有些人自己刮胡子，其他人则去理发师那里。理发师宣称：

我给所有且仅给那些不自己刮胡子的男人刮胡子。

那么很自然地会问一个问题：理发师给自己刮胡子吗？更仔细地思考这个问题后，我们意识到，假设理发师的说法是真的，我们就面临同样的自指悖论：一个逻辑上不可能的情况。如果理发师不给自己刮胡子，那么根据他的声明，他应该给自己刮胡子。如果理发师给自己刮胡子，那么根据他的说法，他就不该给自己刮胡子！

自复制程序

我们能否利用自指设计一个输出自身的程序？为了说明这个想法，让我们考虑一下如果可以用英语写程序该如何实现。考虑以下指令：

打印下面这句话：

```
"Print out the following sentence:"
```

If we execute the instruction above (interpreting it as a program), then we will get the following output:

> Print out the following sentence:

Clearly this is not the same as the original instruction above, which consists of two lines. We can try to modify the instruction as follows:

```
Print out the following sentence twice:
    "Print out the following sentence twice:"
```

Executing this modified instruction yields the output which now consists of two lines:

> Print out the following sentence twice:
>
> Print out the following sentence twice:

This almost works, except that we are missing the quotes in the second line. We can fix it by modifying the instruction as follows:

```
Print out the following sentence twice, the second time in quotes:
    "Print out the following sentence twice, the second time in quotes:"
```

Then we see that when we execute this instruction, we get exactly the same output as the instruction itself:

> Print out the following sentence twice, the second time in quotes:
>
> "Print out the following sentence twice, the second time in quotes:"

Quines and the Recursion Theorem

In the above section we have seen how to write a self-replicating program in English. But can we do this in a real programming language? In general, a program that prints itself is called a *quine*,[1] and it turns out we can always write quines in any programming language.

As another example, consider the following pseudocode:

```
(Quine "s")
    (s "s")
```

The pseudocode above defines a program `Quine` that takes a string `s` as input, and outputs `(s "s")`, which means we run the string `s` (now interpreted as a program) on itself. Now consider executing the program `Quine` with input `"Quine"`:

```
(Quine "Quine")
```

By definition, this will output

> (Quine "Quine")

which is the same as the instruction that we executed!

This is a simple example, but how do we construct quines in general? The answer is given by the *recursion theorem*. The recursion theorem states that given any program $P(x,y)$, we can always convert it to another program $Q(x)$ such that $Q(x) = P(x,Q)$, i.e., Q behaves exactly as P would if its second input is the descrip-

[1] Quine is named after the philosopher and logician Willard Van Orman Quine, as popularized in the book *"Gödel, Escher, Bach: An Eternal Golden Braid"* by Douglas Hofstadter.

打印下面这句话：

如果我们执行上面的指令（将其解释为程序），那么我们会得到以下输出：

打印下面这句话：

显然这与上面原始的两条指令不同。我们可以尝试进一步修改指令如下：

将下面这句话打印两遍：将下面这句话打印两遍：

执行这个修改后的指令会产生如下两行输出：

将下面这句话打印两遍：将下面这句话打

印两遍：

这几乎可行，只是第二行缺少引号。我们可以通过修改指令来解决这个问题：

将下面这句话打印两遍，第二次加引号：将下面这句话打印两遍，第二次加引号：

然后我们发现，当我们执行这条指令时，得到的输出与指令本身完全相同：

将下面这句话打印两遍，第二次加引号：将下面这句话打印两遍，第二

次加引号：

Quines 与递归定理

在上一节中，我们已经看到如何用英语编写一个自复制程序。但我们能否在真正的编程语言中做到这一点？一般来说，打印出自身的程序被称为 quine[1]，而事实证明，我们可以在任何编程语言中编写 quines。

再举一个例子，考虑下面的伪代码：

(Quine s) (s s)

上面的伪代码定义了一个名为 Quine 的程序，它接受字符串 s 作为输入，并输出(s s)，意味着我们将字符串 s（现在被解释为程序）作用于其自身。现在考虑以 Quine 作为输入来执行程序 Quine：

(Quine Quine)

根据定义，这将输出

(Quine Quine)

这与我们执行的指令完全相同！

这是一个简单的例子，但一般情况下我们如何构造 quines？答案由递归定理给出。递归定理指出，对于任意程序 P(x,y)，我们总能将其转换为另一个程序 Q(x)，使得 Q(x) = P(x,Q)，即 Q 的行为与 P 在其第二个输入为 Q 的描述时完全相同。

1“Quine”这个名字来源于哲学家和逻辑学家威拉德·范·奥曼·奎因，由道格拉斯·霍夫施塔特在《哥德尔、埃舍尔、巴赫：一条永恒的金带》一书中推广。

tion of the program Q. In this sense we can think of Q as a "self-aware" version of P, since Q essentially has access to its own description.

The Halting Problem

Are there tasks that a computer cannot perform? For example, we would like to ask the following basic question when compiling a program: does it go into an infinite loop? In 1936, Alan Turing showed that there is no program that can perform this test. The proof of this remarkable fact is very elegant and combines two ingredients: self-reference (as in the liar's paradox), and the fact that we cannot separate programs from data. In computers, a program is represented by a string of bits just as integers, characters, and other data are. The only difference is in how the string of bits is interpreted.

We will now examine the Halting Problem. Given the description of a program and its input, we would like to know if the program ever halts when it is executed on the given input. In other words, we would like to write a program `TestHalt` that behaves as follows:

$$\texttt{TestHalt(P,x)} = \begin{cases} \text{"yes"}, & \text{if program } P \text{ halts on input } x \\ \text{"no"}, & \text{if program } P \text{ loops on input } x \end{cases}$$

Why can't such a program exist? First, let us use the fact that a program is just a bit string, so it can be input as data. This means that it is perfectly valid to consider the behavior of `TestHalt(P,P)`, which will output "yes" if P halts on P, and "no" if P loops forever on P. We now prove that such a program cannot exist.

Theorem: *The Halting Problem is uncomputable; i.e., there does not exist a computer program TestHalt with the behavior specified above on all inputs* (P,x).

Proof: Assuming for contradiction the existence of the program `TestHalt`, use it to construct the following program:

```
Turing(P)
  if TestHalt(P,P) = "yes" then loop forever
  else halt
```

So if the program P when given P as input halts, then `Turing(P)` loops forever; otherwise, `Turing(P)` halts. Assuming we have the program `TestHalt`, we can easily use it as a subroutine in the above program `Turing`.

Now let us look at the behavior of `Turing(Turing)`. There are two cases: either it halts, or it does not. If `Turing(Turing)` halts, then it must be the case that `TestHalt(Turing,Turing)` returned "no." But by definition of `TestHalt`, that would mean that `Turing(Turing)` should not have halted. In the second case, if `Turing(Turing)` does not halt, then it must be the case that `TestHalt(Turing, Turing)` returned "yes," which would mean that `Turing(Turing)` should have halted. In both cases, we arrive at a contradiction which must mean that our initial assumption, namely that the program `TestHalt` exists, was wrong. Thus, `TestHalt` cannot exist, so it is impossible for a program to check if any general program halts! □

What proof technique did we use? This was actually a proof by diagonalization, the same technique that we used in the previous lecture to show that the real numbers are uncountable. Why? Since the set of all computer programs is countable (they are, after all, just finite-length strings over some alphabet, and the set of all finite-length strings is countable), we can enumerate all programs as follows (where P_i represents the

Q 的程序描述。从这个意义上讲，我们可以把 Q 看作是 P 的一种自我感知版本，因为 Q 本质上可以访问它自身的描述。

停机问题

是否存在计算机无法完成的任务？例如，我们在编译程序时希望能提出这样一个基本问题：它是否会进入无限循环？1936年，艾伦·图灵证明了没有任何程序能够完成这种测试。这个惊人事实的证明非常优美，并结合了两个要素：自指（如说谎者悖论），以及我们无法将程序与数据分离的事实。在计算机中，程序由一串比特表示，就像整数、字符和其他数据一样。唯一的区别在于这串比特的解释方式。

我们现在来研究停机问题。给定一个程序及其输入的描述，我们想知道该程序在给定输入上执行时是否会最终停止。换句话说，我们希望编写一个程序 TestHalt，使其行为如下：

$$\texttt{TestHalt(P,x)} = \begin{cases} \text{yes，如果程序 P 在输入 x 上停止；} \\ \text{no，如果程序 P 在输入 x 上无限循环} \end{cases}$$

为什么这样的程序不可能存在？首先，我们利用程序本身只是一个比特串这一事实，因此它可以作为数据输入。这意味着考虑 TestHalt(P,P) 的行为是完全合理的，当 P 在输入 P 上停止时输出 yes，当 P 在输入 P 上无限循环时输出 no。我们现在证明这样的程序不可能存在。

定理：停机问题是不可计算的；也就是说，不存在一个计算机程序 TestHalt 能在所有输入 (P,x) 上表现出上述行为。

证明：假设存在程序 TestHalt，用它来构造以下程序：

```
Turing(P)
    如果 TestHalt(P,P) = yes，则永远循环，否则停止
```

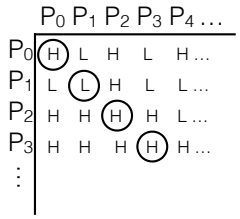
因此，如果程序P在输入P时停机，则Turing(P)永远循环；否则Turing(P)停机。假设我们拥有程序TestHalt，我们可以很容易地将其作为子程序用于上述程序Turing中。

现在让我们考察Turing(Turing)的行为。有两种情况：它要么停机，要么不停机。如果Turing(Turing)停机，那么TestHalt(Turing,Turing)必定返回no。但根据TestHalt的定义，这意味着Turing(Turing)本不该停机。第二种情况，如果Turing(Turing)不停机，那么TestHalt(Turing, Turing)必定返回yes，这意味着Turing(Turing)本该停机。在这两种情况下，我们都得到了矛盾，说明最初的假设——即程序TestHalt存在——是错误的。因此TestHalt不可能存在，也就不可能有程序能够检查任意通用程序是否会停机！

□

我们使用了什么证明技术？这实际上是通过对角化证明，与上一讲中我们用来证明实数不可数所使用的技术相同。为什么？因为所有计算机程序的集合是可数的（毕竟它们只是某个字母表上的有限长度字符串，而所有有限长度字符串的集合是可数的），因此我们可以将所有程序枚举如下（其中Pi表示第

i^{th} program):



The $(i, j)^{\text{th}}$ entry in the table above is H if program P_i halts on input P_j, and L (for "Loops") if it does not halt. Now if the program `Turing` exists it must occur somewhere on our list of programs, say as P_n. But this cannot be, since if the n^{th} entry in the diagonal is H, meaning that P_n halts on P_n, then by its definition `Turing` loops on P_n; and if the entry is L, then by definition `Turing` halts on P_n. Thus the behavior of `Turing` is different from that of P_n, and hence `Turing` does not appear on our list. Since the list contains all possible programs, we must conclude that the program `Turing` does not exist. And since `Turing` is constructed by a simple modification of `TestHalt`, we can conclude that `TestHalt` does not exist either. Hence the Halting Problem cannot be solved.

In fact, there are many more questions we would like to answer about programs but cannot. For example, we cannot know if a program ever outputs anything or if it ever executes a specific line. We also cannot check if two programs produce the same output. And we cannot check to see if a given program is a virus. These issues are explored in greater detail in the advanced course CS172 (Computability and Complexity).

The Easy Halting Problem

As noted above, the key idea in establishing the uncomputability of the Halting Problem is self-reference: Given a program P, we ran into trouble when deciding whether $P(P)$ halts. But in practice, how often do we want to execute a program with its own description as input? Is it possible that if we disallow this kind of self-reference, we can solve the Halting Problem?

For example, given a program P, what if we ask instead the question: "Does P halt on input 0?" This looks easier than the Halting Problem (hence the name Easy Halting Problem), since we only need to check whether P halts on a specific input 0, instead of an arbitrary given input (such as P itself). However, it turns out this seemingly easier problem is still uncomputable! We prove this claim by showing that if we could solve the Easy Halting Problem, then we could also solve the Halting Problem itself; since we know the Halting Problem is uncomputable, this implies the Easy Halting Problem must also be uncomputable.

Specifically, suppose we have a program `TestEasyHalt` that solves the Easy Halting Problem:

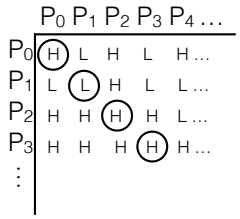
$$\texttt{TestEasyHalt(P)} = \begin{cases} \text{"yes"}, & \text{if program } P \text{ halts on input 0} \\ \text{"no"}, & \text{if program } P \text{ loops on input 0} \end{cases}$$

Then we can use `TestEasyHalt` as a subroutine in the following algorithm that solves the Halting Problem:

```
Halt(P,x)
  construct a program P' that, on input 0, returns P(x)
  return TestEasyHalt(P')
```

The algorithm `Halt` constructs another program P', which depends on both the original program P and the

第i个程序）：



上表中第(i, j)项是H表示程序Pi在输入Pj上停机，L（表示循环）表示不停机。现在如果程序Turing存在，它必定出现在我们的程序列表中，设为Pn。但这不可能，因为如果对角线上的第n项是H，意味着Pn在Pn上停机，那么根据定义Turing在Pn上会循环；而如果该项是L，则根据定义Turing应在Pn上停机。因此Turing的行为与Pn不同，故Turing不在我们的列表中。由于列表包含了所有可能的程序，我们必须得出结论：程序Turing不存在。而由于Turing是由TestHalt经过简单修改构造而来，因此我们也可以断定TestHalt不存在。因此停机问题是不可解的。

事实上，我们还想回答许多关于程序的其他问题，但都无法做到。例如，我们无法知道一个程序是否会输出任何内容，或是否会执行某一行特定代码。我们也不能检查两个程序是否产生相同的输出，也无法判断某个给定程序是否是病毒。这些问题在高级课程CS172（可计算性与复杂性）中有更详细的探讨。

简单停机问题

如前所述，确立停机问题不可计算性的关键思想是自指：给定一个程序P，我们在判断P(P)是否会停机时遇到了麻烦。但在实践中，我们有多频繁需要以程序自身的描述作为输入来执行程序？如果我们禁止这类自指，是否有可能解决停机问题?

例如，给定一个程序P，如果我们改为提问：P在输入0上会停机吗？这看起来比停机问题更容易（因此称为“简单停机问题”），因为我们只需要检查P是否在特定输入0上停机，而不是在任意给定输入（如P自身）上停机。然而，事实证明，这个看似更简单的问题仍然是不可计算的！我们通过证明：如果我们能解决简单停机问题，那么我们也能解决停机问题本身；而我们知道停机问题是不可计算的，因此简单停机问题也必然是不可计算的。具体来说，假设我们有一个程序TestEasyHalt能解决简单停机问题：

$$\texttt{TestEasyHalt(P)} = \begin{cases} \text{yes, if program P halts on input 0} \\ \text{no, if program P loops on input 0} \end{cases}$$

那么我们可以将TestEasyHalt作为子程序用于以下解决停机问题的算法中：

```
Halt(P,x)
```

construct a program Pthat, on input 0, returns P(x) return TestEasyHalt(P)

算法Halt构造另一个程序P0，该程序依赖于原始程序P和

original input x, such that when we call $P'(0)$ we return $P(x)$. Such a program P' can be constructed very simply as follows:

```
P'(y)
   return P(x)
```

That is, the new program P' ignores its input y and always returns $P(x)$. (Note that the descriptions of P and of x are "hard-wired" into P'.) Then we see that $P'(0)$ halts if and only if $P(x)$ halts. Therefore, if we have such a program `TestEasyHalt`, then `Halt` will correctly solve the Halting Problem. Since we know there cannot be such a program `Halt`, we conclude `TestEasyHalt` does not exist either.

The technique that we use here is called a *reduction*. Here we are reducing one problem "Does P halt on x?" to another problem "Does P' halt on 0?", in the sense that if we know how to solve the second problem, then we can use that knowledge to construct an answer for the first problem. This implies that the second problem is actually as difficult as the first, despite the apparently simpler description of the second problem.

Godel's Incompleteness Theorem

In 1900, the great mathematician David Hilbert posed the following two questions about the foundation of logic in mathematics:

1. Is arithmetic consistent?
2. Is arithmetic complete?

To understand the questions above, we recall that mathematics is a formal system based on a list of axioms (for example, Peano's axioms of the natural numbers, axiom of choice, etc.) together with rules of inference. The axioms provide the initial list of true statements in our system, and we can apply the rules of inference to prove other true statements, which we can again use to prove other statements, and so on.

The first question above asks whether it is possible to prove both a proposition P and its negation $\neg P$. If this is the case, then we say that arithmetic is *inconsistent*; otherwise, we say arithmetic is *consistent*. If arithmetic is inconsistent, meaning there are false statements that can be proved, then the entire arithmetic system will collapse because from a false statement we can deduce anything, so every statement in our system will be vacuously true.

The second question above asks whether every true statement in arithmetic can be proved. If this is the case, then we say that arithmetic is *complete*. We note that given a statement, which is either true or false, it can be very difficult to prove which one it is. As a real-world example, consider the following statement, which is known as Fermat's Last Theorem:

$$(\forall n \geq 3)\ \neg(\exists x, y, z \in \mathbb{Z}^+)(x^n + y^n = z^n).$$

This theorem was first stated by Pierre de Fermat in 1637,[2] but it has eluded proofs for centuries until it was finally proved by Andrew Wiles in 1994.

In 1928, Hilbert formally posed the questions above as the Entscheidungsproblem. Most people believed that the answer would be "yes," since ideally arithmetic should be both consistent and complete. However, in 1930 Kurt Gödel proved that the answer is in fact "no": Any formal system that is sufficiently rich to formalize arithmetic is either inconsistent (there are false statements that can be proved) or incomplete (there are true statements that cannot be proved). Gödel proved his result by exploiting the deep connection

[2] Along with the famous note: "I have discovered a truly marvelous proof of this, which this margin is too narrow to contain."

原始输入x，使得当我们调用P0(0)时返回P(x)。这样的程序P0可以非常简单地构造如下：

```
P(y)
  return P(x)
```

也就是说，新程序P0忽略其输入y，总是返回P(x)。（注意P和x的描述被硬编码进P0中。）于是我们看到P0(0)停机当且仅当P(x)停机。因此，如果我们有这样的程序TestEasyHalt，那么Halt就能正确解决停机问题。由于我们知道这样的程序Halt不可能存在，因此我们得出结论：TestEasyHalt也不存在。

这里使用的技术称为归约。我们将一个问题“P在x上会停机吗？”归约为另一个问题“P0在0上会停机吗？”，意思是如果我们知道如何解决第二个问题，就可以利用该知识来构造第一个问题的答案。这表明第二个问题实际上与第一个问题一样难，尽管第二个问题的描述看起来更简单。

哥德尔的不完备性定理

1900年，伟大的数学家大卫·希尔伯特提出了关于数学逻辑基础的以下两个问题：

1. 算术是一致的吗?
2. 算术是完备的吗?

为了理解上述问题，我们回顾数学是一个基于一组公理（例如自然数的皮亚诺公理、选择公理等）以及推理规则的形式系统。公理提供了系统中初始的真命题列表，我们可以应用推理规则来证明其他真命题，这些新证明的命题又可用于证明更多命题，如此继续下去。

上面第一个问题问的是：是否有可能同时证明一个命题P及其否定¬P。如果可能，则称算术是不一致的；否则称算术是一致的。如果算术是不一致的，意味着存在可以被证明的假命题，那么整个算术系统就会崩溃，因为从一个假命题可以推出任何结论，从而使系统中的每个命题都空洞地为真。

上面第二个问题问的是：算术中每一个真命题是否都可以被证明？如果如此，则称算术是完备的。我们注意到，给定一个命题，它要么为真要么为假，但要证明其真假可能极为困难。一个现实世界的例子是下面这个被称为费马大定理的命题：

$$(\forall n \geqslant 3)\ \neg(\exists x,y,z \in Z+)(xn + yn = zn)。$$

这个定理最早由皮埃尔·德·费马于1637年提出，2但几个世纪以来一直未被证明，直到1994年由安德鲁·怀尔斯最终证明。

1928年，希尔伯特正式提出了上述问题，即判定性问题（Entscheidungsproblem）。大多数人认为答案会是肯定的，因为理想情况下算术应当既一致又完备。然而，1930年库尔特·哥德尔证明了事实恰恰相反：任何足够丰富以形式化算术的形式系统，要么是不一致的（存在可以被证明的假命题），要么是不完备的（存在无法被证明的真命题）。哥德尔通过揭示证明与算术之间的深刻联系来证明他的结果。

2以及著名的注释：我发现了一个真正奇妙的证明，可惜页边太窄写不下。

between proofs and arithmetic. Actually Gödel's theorem also embodies a deep connection between proofs and computation, which was illuminated after Turing formalized the definition of computation in 1936 via the notion of Turing machines and computability.

In the rest of this note, we will first sketch the essence of Gödel's proof, and then we will outline an easier proof of the theorem using what we know about the Halting Problem.

Sketch of Gödel's Proof

Suppose we have a formal system F, which consists of a list of axioms and rules of inference, and assume F is sufficiently expressive that we can use it to express all of arithmetic.

Now suppose we can write the following statement:

$$S(F) = \text{“This statement is not provable in } F\text{.”}$$

Once we have this statement, there are two possibilities:

1. Case 1: $S(F)$ is provable. Then the statement $S(F)$ is true, but by inspecting the content of the statement itself, we see that this implies $S(F)$ should not be provable. Thus, F is inconsistent in this case.

2. Case 2: $S(F)$ is not provable. By construction, this means the statement $S(F)$ is true. Thus, F is incomplete in this case, since there is a true statement (namely, $S(F)$) that is not provable.

To complete the proof, it now suffices to construct such a statement $S(F)$. This is the difficult part of Gödel's proof, which requires a clever encoding (a so-called "Gödel numbering") of symbols and propositions as natural numbers.

Proof via the Halting Problem

Let us now see how we can prove Gödel's result by reduction to the Halting Problem. Here we proceed by contradiction: Suppose arithmetic is both consistent and complete; we will use this assumption to solve the Halting Problem, which we have seen is impossible.

Recall that in the Halting Problem we want to decide whether a given program P halts on a given input x. For fixed P and x, let $S_{P,x}$ denote the proposition "P halts on input x." The key observation is that this proposition can be phrased as a statement in arithmetic. The form of the statement $S_{P,x}$ will be

$$\exists z(z \text{ encodes a valid halting execution sequence of } P \text{ on input } x).$$

Although the details require some work, your programming intuition should hopefully convince you that such a statement can be written, in a fairly mechanical way, using only the language of standard arithmetic, with the usual operators, connectives and quantifiers: basically the statement just has to check, step by step, that the string z (encoded as a very long integer in binary) lists out the sequence of states that a computer would go through when running program P on input x.

Now let us assume, for contradiction, that arithmetic is both consistent and complete. This means that, for any (P,x), the statement $S_{P,x}$ is either true or false, and that there must exist a proof in arithmetic of either $S_{P,x}$ or its negation, $\neg S_{P,x}$ (and not both). But now recall that a proof is simply a finite binary string. Therefore,

在证明和算术之间。实际上，哥德尔定理还体现了一种深刻的联系，即证明与计算之间的联系，这种联系在图灵于1936年通过图灵机和可计算性的概念形式化定义计算后得到了阐明。

在本笔记的其余部分，我们将首先概述哥德尔证明的核心思想，然后利用我们对停机问题的了解，勾勒出该定理的一个更简单的证明。

哥德尔证明的概要

假设我们有一个形式系统F，它由一组公理和推理规则组成，并且假设F具有足够的表达能力，能够表达全部算术。

现在假设我们可以写出如下语句：

$$S(F) = \text{这个语句在F中不可证明。}$$

一旦我们有了这个语句，就有两种可能性：

1. 情况1：S(F)可证明。那么语句S(F)为真，但通过检查该语句本身的内容，我们发现这暗示S(F)不应是可证明的。因此，在这种情况下F是不一致的。

2. 情况2：S(F)不可证明。根据构造，这意味着语句S(F)为真。因此，在这种情况下F是不完备的，因为存在一个为真的语句（即S(F)）无法被证明。

为了完成证明，现在只需构造这样一个语句S(F)。这是哥德尔证明中的困难部分，需要一种巧妙的编码（所谓的哥德尔编号），将符号和命题编码为自然数。

通过停机问题的证明

现在让我们看看如何通过归约到停机问题来证明哥德尔的结果。我们采用反证法：假设算术系统既一致又完备；我们将利用这一假设来解决停机问题，而我们知道这是不可能的。

回想一下，在停机问题中，我们希望判断一个给定的程序P是否会在给定输入x上停机。对于固定的P和x，令SP,x表示命题“P在输入x上停机”。关键观察是，这个命题可以被表述为算术中的一个语句。命题SP,x的形式为

$$\text{存在z，使得z编码了程序P在输入x上的一次有效的停机执行序列。}$$

尽管细节需要一些工作，但你的编程直觉应该足以让你相信，这种语句可以用标准算术语言相当机械地写出，仅使用通常的运算符、连接词和量词：基本上该语句只需逐步检查字符串z（以很长的二进制整数编码）是否列出了计算机在输入x上运行程序P时所经历的状态序列。

现在让我们假设，为了引出矛盾，算术系统既是相容的又是完备的。这意味着，对于任意(P,x)，命题SP,x要么为真，要么为假，并且在算术系统中必然存在对SP,x或其否定¬SP,x的证明（但不能两者都有）。但现在回想一下，一个证明只是一个有限的二进制字符串。因此，

there are only countably many possible proofs, so we can enumerate them one by one and search for a proof of $S_{P,x}$ or $\neg S_{P,x}$. The following program performs this task:

```
Search(P,x)
  for every proof q:
    if q is a proof of S_{P,x} then output "yes"
    if q is a proof of ¬S_{P,x} then output "no"
```

The program Search takes as input the program P, and proceeds to check every possible proof until it finds either one that proves $S_{P,x}$, or one that proves $\neg S_{P,x}$. By assumption, we know that one of these proofs always exists, so the program Search will terminate in finite time, and it will correctly solve the Halting Problem. On the other hand, since we have already established that the Halting Problem is uncomputable, such a program Search cannot exist. Therefore, our initial assumption must be wrong, so it is not true that arithmetic is both consistent and complete.

Note that in the argument above we rely on the fact that, given a proof, we can construct a program that mechanistically checks whether it is a valid proof of a given proposition. This is a manifestation of the intimate connection between proofs and computation.

只有可数多个可能的证明，因此我们可以逐个枚举并寻找SP,x或¬SP,x的证明。以下程序执行此任务：

```
Search(P,x)
    对于每一个证明q：
        如果q是SP,x的证明，则输出yes；如果q是¬SP,x的证
        明，则输出no
```

程序Search以程序P为输入，然后逐一检查每一个可能的证明，直到找到一个能够证明SP,x的证明，或一个能够证明¬SP,x的证明为止。根据假设，我们知道其中一种证明总是存在的，因此程序Search将在有限时间内终止，并正确地解决停机问题。然而，由于我们已经确定停机问题是不可计算的，因此这样的程序Search不可能存在。因此，我们最初的假设一定是错误的，所以算术系统不可能同时具有一致性和完备性。

请注意，在上述论证中，我们依赖于这样一个事实：给定一个证明，我们可以构造一个程序，机械地检查它是否是某个命题的有效证明。这体现了证明与计算之间紧密的联系。