Chapter 8

Introduction to Turing Machines

In this chapter we change our direction significantly. Until now, we have been interested primarily in simple classes of languages and the ways that they can be used for relatively constrained problems, such as analyzing protocols, searching text, or parsing programs. Now, we shall start looking at the question of what languages can be defined by any computational device whatsoever. This question is tantamount to the question of what computers can do, since recognizing the strings in a language is a formal way of expressing any problem, and solving a problem is a reasonable surrogate for what it is that computers do.

We begin with an informal argument, using an assumed knowledge of C programming, to show that there are specific problems we cannot solve using a computer. These problems are called "undecidable." We then introduce a venerable formalism for computers, called the Turing machine. While a Turing machine looks nothing like a PC, and would be grossly inefficient should some startup company decide to manufacture and sell them, the Turing machine long has been recognized as an accurate model for what any physical computing device is capable of doing.

In Chapter 9, we use the Turing machine to develop a theory of "undecidable" problems, that is, problems that no computer can solve. We show that a number of problems that are easy to express are in fact undecidable. An example is telling whether a given grammar is ambiguous, and we shall see many others.

8.1 Problems That Computers Cannot Solve

The purpose of this section is to provide an informal, C-programming-based introduction to the proof of a specific problem that computers cannot solve. The particular problem we discuss is whether the first thing a C program prints

is `hello, world`. Although we might imagine that simulation of the program would allow us to tell what the program does, we must in reality contend with programs that take an unimaginably long time before making any output at all. This problem — not knowing when, if ever, something will occur — is the ultimate cause of our inability to tell what a program does. However, proving formally that there is no program to do a stated task is quite tricky, and we need to develop some formal mechanics. In this section, we give the intuition behind the formal proofs.

8.1.1 Programs that Print "Hello, World"

In Fig. 8.1 is the first C program met by students who read Kernighan and Ritchie's classic book.[1] It is rather easy to discover that this program prints `hello, world` and terminates. This program is so transparent that it has become a common practice to introduce languages by showing how to write a program to print `hello, world` in those languages.

```
main()
{
    printf("hello, world\n");
}
```

Figure 8.1: Kernighan and Ritchie's hello-world program

However, there are other programs that also print `hello, world`; yet the fact that they do so is far from obvious. Figure 8.2 shows another program that might print `hello, world`. It takes an input n, and looks for positive integer solutions to the equation $x^n + y^n = z^n$. If it finds one, it prints `hello, world`. If it never finds integers x, y, and z to satisfy the equation, then it continues searching forever, and never prints `hello, world`.

To understand what this program does, first observe that `exp` is an auxiliary function to compute exponentials. The main program needs to search through triples (x, y, z) in an order such that we are sure we get to every triple of positive integers eventually. To organize the search properly, we use a fourth variable, `total`, that starts at 3 and, in the while-loop, is increased one unit at a time, eventually reaching any finite integer. Inside the while-loop, we divide `total` into three positive integers x, y, and z, by first allowing x to range from 1 to `total-2`, and within that for-loop allowing y to range from 1 up to one less than what x has not already taken from `total`. What remains, which must be between 1 and `total-2`, is given to z.

In the innermost loop, the triple (x, y, z) is tested to see if $x^n + y^n = z^n$. If so, the program prints `hello, world`, and if not, it prints nothing.

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 1978, Prentice-Hall, Englewood Cliffs, NJ.

```
int exp(int i, n)
/* computes i to the power n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

Figure 8.2: Fermat's last theorem expressed as a hello-world program

If the value of n that the program reads is 2, then it will eventually find combinations of integers such as `total` $= 12$, $x = 3$, $y = 4$, and $z = 5$, for which $x^n + y^n = z^n$. Thus, for input 2, the program *does* print `hello, world`.

However, for any integer $n > 2$, the program will never find a triple of positive integers to satisfy $x^n + y^n = z^n$, and thus will fail to print `hello, world`. Interestingly, until a few years ago, it was not known whether or not this program would print `hello, world` for some large integer n. The claim that it would not, i.e., that there are no integer solutions to the equation $x^n + y^n = z^n$ if $n > 2$, was made by Fermat 300 years ago, but no proof was found until quite recently. This statement is often referred to as "Fermat's last theorem."

Let us define the *hello-world problem* to be: determine whether a given C program, with a given input, prints `hello, world` as the first 12 characters that it prints. In what follows, we often use, as a shorthand, the statement about a program that it prints `hello, world` to mean that it prints `hello, world` as the first 12 characters that it prints.

It seems likely that, if it takes mathematicians 300 years to resolve a question

Why Undecidable Problems Must Exist

While it is tricky to prove that a specific problem, such as the "hello-world problem" discussed here, must be undecidable, it is quite easy to see why almost all problems must be undecidable by any system that involves programming. Recall that a "problem" is really membership of a string in a language. The number of different languages over any alphabet of more than one symbol is not countable. That is, there is no way to assign integers to the languages such that every language has an integer, and every integer is assigned to one language.

On the other hand programs, being finite strings over a finite alphabet (typically a subset of the ASCII alphabet), *are* countable. That is, we can order them by length, and for programs of the same length, order them lexicographically. Thus, we can speak of the first program, the second program, and in general, the ith program for any integer i.

As a result, we know there are infinitely fewer programs than there are problems. If we picked a language at random, almost certainly it would be an undecidable problem. The only reason that most problems *appear* to be decidable is that we rarely are interested in random problems. Rather, we tend to look at fairly simple, well-structured problems, and indeed these are often decidable. However, even among the problems we are interested in and can state clearly and succinctly, we find many that are undecidable; the hello-world problem is a case in point.

about a single, 22-line program, then the general problem of telling whether a given program, on a given input, prints `hello, world` must be hard indeed. In fact, any of the problems that mathematicians have not yet been able to resolve can be turned into a question of the form "does this program, with this input, print `hello, world`? Thus, it would be remarkable indeed if we could write a program that could examine any program P and input I for P, and tell whether P, run with I as its input, would print `hello, world`. We shall prove that no such program exists.

8.1.2 The Hypothetical "Hello, World" Tester

The proof of impossibility of making the hello-world test is a proof by contradiction. That is, we assume there is a program, call it H, that takes as input a program P and an input I, and tells whether P with input I prints `hello, world`. Figure 8.3 is a representation of what H does. In particular, the only output H makes is either to print the three characters `yes` or to print the two characters `no`. It always does one or the other.

If a problem has an algorithm like H, that always tells correctly whether an instance of the problem has answer "yes" or "no," then the problem is said to

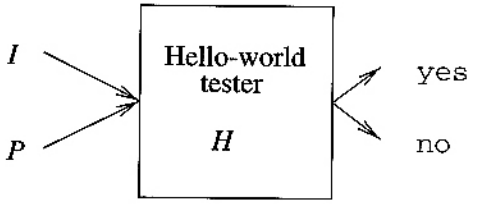

Figure 8.3: A hypothetical program H that is a hello-world detector

be "decidable." Otherwise, the problem is "undecidable." Our goal is to prove that H doesn't exist; i.e., the hello-world problem is undecidable.

In order to prove that statement by contradiction, we are going to make several changes to H, eventually constructing a related program called H_2 that we show does not exist. Since the changes to H are simple transformations that can be done to any C program, the only questionable statement is the existence of H, so it is that assumption we have contradicted.

To simplify our discussion, we shall make a few assumptions about C programs. These assumptions make H's job easier, not harder, so if we can show a "hello-world tester" for these restricted programs does not exist, then surely there is no such tester that could work for a broader class of programs. Our assumptions are:

1. All output is character-based, e.g., we are not using a graphics package or any other facility to make output that is not in the form of characters.

2. All character-based output is performed using `printf`, rather than `putchar()` or another character-based output function.

We now assume that the program H exists. Our first modification is to change the output `no`, which is the response that H makes when its input program P does not print `hello, world` as its first output in response to input I. As soon as H prints "n," we know it will eventually follow with the "o."[2] Thus, we can modify any `printf` statement in H that prints "n" to instead print `hello, world`. Another `printf` statement that prints an "o" but not the "n" is omitted. As a result, the new program, which we call H_1, behaves like H, except it prints `hello, world` exactly when H would print `no`. H_1 is suggested by Fig. 8.4.

Our next transformation on the program is a bit trickier; it is essentially the insight that allowed Alan Turing to prove his undecidability result about Turing machines. Since we are really interested in programs that take other programs as input and tell something about them, we shall restrict H_1 so it:

a) Takes only input P, not P and I.

[2]Most likely, the program would put `no` in one `printf`, but it could print the "n" in one `printf` and the "o" in another.

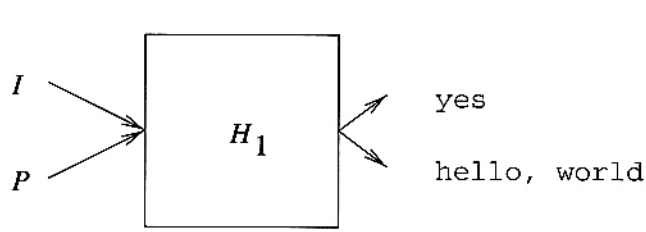

Figure 8.4: H_1 behaves like H, but it says `hello, world` instead of `no`

b) Asks what P would do if its input were its own code, i.e., what would H_1 do on inputs P as program and P as input I as well?

The modifications we must perform on H_1 to produce the program H_2 suggested in Fig. 8.5 are as follows:

1. H_2 first reads the entire input P and stores it in an array A, which it "malloc's" for the purpose.[3]

2. H_2 then simulates H_1, but whenever H_1 would read input from P or I, H_2 reads from the stored copy in A. To keep track of how much of P and I H_1 has read, H_2 can maintain two cursors that mark positions in A.

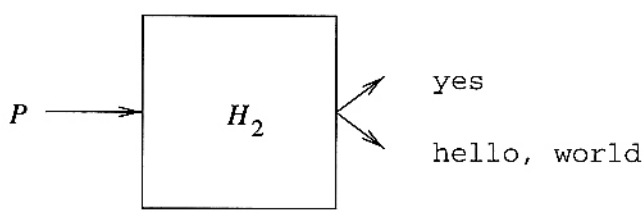

Figure 8.5: H_2 behaves like H_1, but uses its input P as both P and I

We are now ready to prove H_2 cannot exist. Thus, H_1 does not exist, and likewise, H does not exist. The heart of the argument is to envision what H_2 does when given itself as input. This situation is suggested in Fig. 8.6. Recall that H_2, given any program P as input, makes output `yes` if P prints `hello, world` when given itself as input. Also, H_2 prints `hello, world` if P, given itself as input, does not print `hello, world` as its first output.

Suppose that the H_2 represented by the box in Fig. 8.6 makes the output `yes`. Then the H_2 in the box is saying about its input H_2 that H_2, given itself

[3]The UNIX `malloc` system function allocates a block of memory of a size specified in the call to `malloc`. This function is used when the amount of storage needed cannot be determined until the program is run, as would be the case if an input of arbitrary length were read. Typically, `malloc` would be called several times, as more and more input is read and progressively more space is needed.

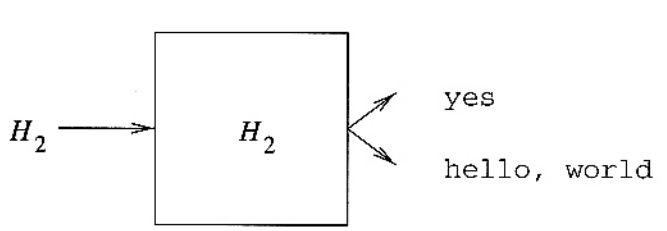

Figure 8.6: What does H_2 do when given itself as input?

as input, prints `hello, world` as its first output. But we just supposed that the first output H_2 makes in this situation is `yes` rather than `hello, world`.

Thus, it appears that in Fig. 8.6 the output of the box is `hello, world`, since it must be one or the other. But if H_2, given itself as input, prints `hello, world` first, then the output of the box in Fig. 8.6 must be `yes`. Whichever output we suppose H_2 makes, we can argue that it makes the other output.

This situation is paradoxical, and we conclude that H_2 cannot exist. As a result, we have contradicted the assumption that H exists. That is, we have proved that no program H can tell whether or not a given program P with input I prints `hello, world` as its first output.

8.1.3 Reducing One Problem to Another

Now, we have one problem — does a given program with given input print `hello, world` as the first thing it prints? — that we know no computer program can solve. A problem that cannot be solved by computer is called *undecidable*. We shall give the formal definition of "undecidable" in Section 9.3, but for the moment, let us use the term informally. Suppose we want to determine whether or not some other problem is solvable by a computer. We can try to write a program to solve it, but if we cannot figure out how to do so, then we might try a proof that there is no such program.

Perhaps we could prove this new problem undecidable by a technique similar to what we did for the hello-world problem: assume there is a program to solve it and develop a paradoxical program that must do two contradictory things, like the program H_2. However, once we have one problem that we know is undecidable, we no longer have to prove the existence of a paradoxical situation. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. The strategy is suggested in Fig. 8.7; the technique is called the *reduction* of P_1 to P_2.

Suppose that we know problem P_1 is undecidable, and P_2 is a new problem that we would like to prove is undecidable as well. We suppose that there is a program represented in Fig. 8.7 by the diamond labeled "decide"; this program prints `yes` or `no`, depending on whether its input instance of problem P_2 is or is not in the language of that problem.[4]

[4]Recall that a problem is really a language. When we talked of the problem of deciding