A Proof of Undecidability

Proof of Theorem 2 For the sake of deriving a contradiction, assume that there is a programs that decides if its input has property SELFLOOPING. If such a program existed we could re-write it as a function `boolean selflooping()` that decides whether or not its input has property SELFLOOPING. We could then write the program `diagonal`[4] shown in Figure 1.4.

Does the program in Figure 1.4 itself have property SELFLOOPING?

Claim 1 It does not.

Proof of Claim 1 Assume it did. Then, when running the program with its own source code as input, the function `selflooping` would return `TRUE` to the main program, making it terminate and hence not have property SELFLOOPING, a contradiction. □

Claim 2 It does.

Proof of Claim 2 Assume it did not. Then, when running the program with its own source code as input, the function `selflooping` would return `FALSE` to the main program, sending it into an infinite loop and hence have property SELFLOOPING, a contradiction. □

These two claims form a contradiction, proving our initial assumption wrong; there cannot be a program that decides if its input has property SELFLOOPING. □

[4] Why `diagonal` is a fitting name for this program will become clear in the next section.

	digits →											
numbers ↓				$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{10^3}$	$\frac{1}{10^4}$	$\frac{1}{10^5}$	$\frac{1}{10^6}$	$\frac{1}{10^7}$	$\frac{1}{10^8}$	$\cdots$
d	$=$	0	.	1	1	9	1	1	9	9	9	$\cdots$
r_1	$=$	0	.	**6**	2	8	4	1	1	0	4	$\cdots$
r_2	$=$	0	.	0	**8**	8	4	1	1	3	8	$\cdots$
r_3	$=$	0	.	8	4	**1**	1	0	8	4	1	$\cdots$
r_4	$=$	0	.	1	0	0	**0**	0	1	0	0	$\cdots$
r_5	$=$	0	.	0	9	9	9	**9**	0	9	9	$\cdots$
r_6	$=$	0	.	0	4	6	2	6	**1**	4	6	$\cdots$
r_7	$=$	0	.	1	1	0	0	2	1	**1**	0	$\cdots$
r_8	$=$	0	.	4	4	9	4	3	4	4	**1**	$\cdots$
$\vdots$												

Figure 1.5: The diagonal in the proof of Theorem 4

Diagonalization

The proof of Theorem 2 was a special kind of proof by contradiction called a "diagonalization proof." To understand that aspect of the proof, let's look at a classical theorem with a diagonalization proof.

Theorem 4 (Cantor, 1873) *The real numbers are not countable.*

(A set is countable if there is a function from the natural numbers onto the set. In other words, there is a sequence $r_1, r_2, r_3, r_4, r_5, \ldots$ that contains all members of the set.)

Proof (Cantor, 1890/1891) [5] Assume the real numbers are countable.

Then there is a sequence that contains all reals and a subsequence r_1, r_2, $r_3, \ldots$ that contains all reals from the interval $(0, 1]$. Consider the infinite matrix that contains in row i a decimal expansion of r_i, one digit per column. (See Figure 1.5 for an illustration.) Consider the number d whose i^{th} digit is 9 if the matrix entry in row i and column i is 1, and whose i^{th} digit is 1 otherwise.

The number d thus defined differs from every number in the sequence r_1, r_2, $r_3, \ldots$, contradicting the assumption that all reals from $[0, 1)$ were in the sequence.[6] □

[5] Cantor gave a different first proof in 1873, according to Kline, p.997.

[6] There is a subtlety in this proof stemming from the fact that two different decimal expansions can represent the same number: $0.10000\ldots = 0.09999\ldots$. Why is this of concern? And why is the proof ok anyway?

programs ↓		p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	$\cdots$
`diagonal`	:	1	1	0	1	1	0	0	0	$\cdots$
p_1	:	**0**	0	0	0	1	1	0	0	$\cdots$
p_2	:	0	**0**	0	0	1	1	0	0	$\cdots$
p_3	:	0	0	**1**	1	0	0	0	1	$\cdots$
p_4	:	1	0	0	**0**	0	1	0	0	$\cdots$
p_5	:	0	0	0	0	**0**	0	0	0	$\cdots$
p_6	:	0	0	0	0	0	**1**	0	0	$\cdots$
p_7	:	1	1	0	0	0	1	**1**	0	$\cdots$
p_8	:	0	0	0	0	0	0	0	**1**	$\cdots$
$\vdots$										

Figure 1.6: The diagonal in the proof of Theorem 2

How does this proof of Cantor's Theorem relate to our proof of Theorem 2? Programs are finite strings and hence countable. Let $p_1, p_2, p_3, \ldots$ be a sequence of all programs. (Unlike a sequence of all reals, such a sequence does exist since programs are finite strings.) Consider the infinite boolean matrix which has a 0 in row i and column j if program i, when run with the source code of program j as input, does not terminate; and a 1 if it does. (See Figure 1.6 for an illustration.)

The program `diagonal.cpp` of Figure 1.4, which we assumed to exist at the start of the proof of Theorem 2, then differs from each program in the sequence, contradicting the fact that all programs were in the sequence. The reason is that if program p_i does terminate on input p_i, then `diagonal.cpp` does not terminate on input p_i, and vice versa.

Other proof techniques? It is remarkable that without diagonalization, nobody would know how to prove Theorem 2 nor any of the other undecidability results in this chapter.

This monopoly of one proof technique will cause problems in the study of computational complexity in Chapter 4, which injects a concern about efficiency into the discussion. Diagonalization does not work nearly as well in that context. Without any other technique available, we will not be able to prove some very basic conjectures.

Heuristics Every computational problem, whether unsolvable in theory or not, is open to a heuristic approach. The theory presented here does not address the feasibility of heuristic methods. It does suggest that heuristics would be the way to go if one wanted to create software tools to solve in some practically useful sense any of the problems which this theory shows "unsolvable."

Self-reference in other domains

Digressing for a moment from our focus on what programs can and cannot do, programs that process programs are not the only example of self-referencing causing trouble, and far from the oldest one. Here are a few others.

An example known to Greek philosophers over 2000 years ago is the self-referencing statement that says "This statement is a lie." This may look like a pretty silly statement to consider, but it is true that the variant that says "This is a theorem for which there is no proof," underlies what many regard as the most important theorem of 20^{th}-century mathematics, Gödel's Incompleteness Theorem. More on that in Section 1.8 on page 25.

A (seemingly) non-mathematical example arises when you try to compile a "Catalog of all catalogs," which is not a problem yet, but what about a "Catalog of all those catalogs that do *not* contain a reference to themselves?" Does it contain a reference to itself? If we call catalogs that contain a reference to themselves "self-promoting," the trouble stems from trying to compile a "catalog of all non-self-promoting catalogs." Note the similarity to considering a program that halts on all those programs that are "selflooping."

1.5 Transformations

SELFLOOPING is not the only undecidable property of programs. To prove other properties undecidable, we will exploit the observation that they have strong connections to SELFLOOPING, connections such that the undecidability of SELFLOOPING "rubs off" on them. Specifically, we will show that there are "transformations" of SELFLOOPING into other problems.

Before we apply the concept of transformations to our study of (un)decidability, let's look at some examples of transformations in other settings.

Consider the Unix command

```
sort
```

It can be used like

```
sort < phonebook
```

to sort the lines of a text file. "`sort`" solves a certain problem, called SORTING, a problem that is so common and important that whole chapters of books have been written about it (and a solution comes with every computer system you might want to buy).

But what if the problem you want to solve is not SORTING, but the problem of SORTING-THOSE-LINES-THAT-CONTAIN-THE-WORD-"Boulder", STLTCTWB for short? You could write a program "`stltctwb`" to solve STLTCTWB and use it like

```
stltctwb < phonebook
```

But probably you would rather do

```
cat phonebook | grep Boulder | sort
```

This is an example of a transformation. The command "`grep Boulder`" *transforms* the STLTCTWB problem *into* the SORTING problem. The command "`grep Boulder | sort`" solves the STLTCTWB problem. (The "`cat phonebook | `" merely provides the input.)

The general pattern is that if "`b`" is a program that solves problem B, and

```
a_to_b | b
```

solves problem A, then

```
a_to_b
```

is called a *transformation of problem A into problem B.*

The reason for doing such transformations is that they are a way to solve problems. If we had to formulate a theorem about the matter it would read as follows. (Let's make it an "Observation." That way we don't have to spell out a proof.)

Observation 1 *If there is a program that transforms problem A into problem B and there is a program that solves problem B, then there is a program that solves problem A.*

Since at the moment we are more interested in proving that things cannot be done than in proving that things can be done, the following rephrasing of the above observation will be more useful to us.

A problem is *solvable* if there exists a program for solving it.

Observation 2 *If there is a program that transforms problem A into problem B and problem A is unsolvable, then problem B is unsolvable as well.*

"Solvable" or "decidable?"

In good English, "problems" get "solved" and "questions" get "decided." Following this usage, we speak of "solvable and unsolvable problems" and "decidable and undecidable questions." For the substance of our discussion this distinction is of no consequence. For example, *solving* the SELFLOOPING *problem* for a program x is the same as *deciding* whether or programs have the *property* SELFLOOPING. In either case, the prefix "un-" means that there is no program to do the job.

Transforming binary addition into Boolean satisfiability (SAT)

For another example of a transformation, consider the following two problems.

Let BINARY ADDITION be the problem of deciding, given a string like

$$1\,0\,+\,1\,1\,=\,0\,1\,1 \tag{1.7}$$

whether or not it is a correct equation between binary numbers. (This one isn't.) Let's restrict the problem to strings of the form

$$x_2\ x_1\ +\ y_2\ y_1\ \ =\ \ z_3\ z_2\ z_1 \tag{1.8}$$

with each x_i, y_i, and z_i being 0 or 1. Let's call this restricted version 2-ADD.

Let Boolean Satisfiability (SAT) be the problem of deciding, given a Boolean expression, whether or not there is an assignment of truth values to the variables in the expression which makes the whole expression *true*.

A *transformation of* 2-ADD *to* SAT is a function t such that for all strings α,

$$\begin{array}{c}\alpha \text{ represents a correct addition of two binary numbers}\\ \Leftrightarrow\\ t(\alpha) \text{ is a satisfiable Boolean expression.}\end{array}$$

How can we construct such a transformation? Consider the Boolean circuit in Figure 1.7. It adds up two 2-bit binary numbers. Instead of drawing the picture we can describe this circuit equally precisely by the Boolean expression

$$\begin{array}{llllllll}
((& x_1 & \oplus & y_1 &) \Leftrightarrow & z_1 &) & \wedge\\
((& x_1 & \wedge & y_1 &) \Leftrightarrow & c_2 &) & \wedge\\
((& x_2 & \oplus & y_2 &) \Leftrightarrow & z_2' &) & \wedge\\
((& x_2 & \wedge & y_2 &) \Leftrightarrow & c_3' &) & \wedge\\
((& c_2 & \oplus & z_2' &) \Leftrightarrow & z_2 &) & \wedge\\
((& c_2 & \wedge & z_2' &) \Leftrightarrow & c_3'' &) & \wedge\\
((& c_3' & \vee & c_3'' &) \Leftrightarrow & z_3 &) &
\end{array}$$

This suggests transformation of 2-ADD to SAT that maps the string (1.7) to the expression

$$\begin{array}{llllllll}
((& 0 & \oplus & 1 &) \Leftrightarrow & 1 &) & \wedge\\
((& 0 & \wedge & 1 &) \Leftrightarrow & c_2 &) & \wedge\\
((& 1 & \oplus & 1 &) \Leftrightarrow & z_2' &) & \wedge\\
((& 1 & \wedge & 1 &) \Leftrightarrow & c_3' &) & \wedge\\
((& c_2 & \oplus & z_2' &) \Leftrightarrow & 1 &) & \wedge\\
((& c_2 & \wedge & z_2' &) \Leftrightarrow & c_3'' &) & \wedge\\
((& c_3' & \vee & c_3'' &) \Leftrightarrow & 0 &) &
\end{array}$$

which, appropriately, is not satisfiable. There are four variables to choose values for, c_2, z_2', c_3', and c_3'', but no choice of values for them will make the whole expression true. Note that every single one of the seven "conjuncts" can be made true with some assignment of truth values to c_2, z_2', c_3', and c_3'', but not all seven can be made true with one and the same assignment. In terms of the circuit this means that if the inputs and outputs are fixed to reflect the values given in (1.7), at least one gate in the circuit has to end up with inconsistent values on its input and output wires, or at least one wire has to have different values at its two ends. The circuit is "not satisfiable."

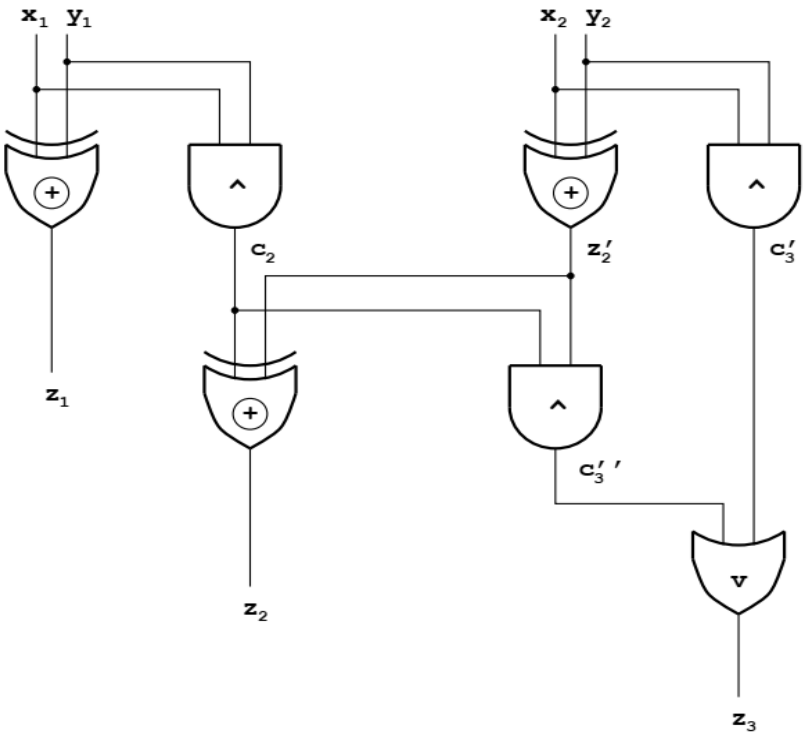

Figure 1.7: A circuit for adding two 2-bit binary numbers

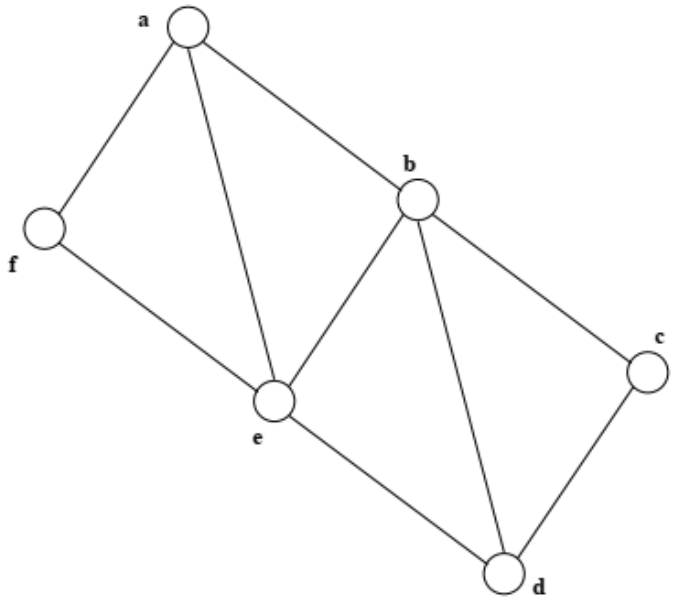

Figure 1.8: A graph of activities and conflicts

Note that transformations do not always go from a harder problem to an easier one.[7] In fact, the opposite is closer to the truth. The destination problem can be easier than the source problem only to the extent of the computational effort expended in carrying out the transformation. But the destination problem can be much harder than the source problem. For example, you can transform ODDLENGTH to LOOPING, but not LOOPING to ODDLENGTH.

Transforming GRAPH 3-COLORABILITY **into** SAT

Graphs like the one shown in Figure 1.8 can be used to represent scheduling problems. Each node represents an activity, e.g. a meeting of some group, and an edge between two nodes means that the two activities must not be scheduled at the same time, e.g. because there is a person who needs to attend both.

Since activities a, b, and e are all conflicting with each other, there is no way to schedule all the activities into fewer than three time slots. Are three slots enough? Instead of activities being scheduled into time slots, graph theorists talk about "nodes" being "colored." The question then becomes, are three different colors enough to color all the nodes, without adjacent nodes getting the same color? This problem is known as "GRAPH 3-COLORABILITY," or "3-COLOR" for short.

The idea behind the transformation is to make the variable a_{GREEN} (a_{RED}, a_{BLUE}) true if and only if the vertex a gets colored green (red, blue). The statement that a and b get different colors can then be made in the "language of Boolean expressions" as

$$\overline{(a_{\text{GREEN}} \wedge b_{\text{GREEN}})} \wedge \overline{(a_{\text{RED}} \wedge b_{\text{RED}})} \wedge \overline{(a_{\text{BLUE}} \wedge b_{\text{BLUE}})} \qquad (1.9)$$

We need to make one such statement for each of the nine edges in the graph. The conjunction of these nine statements does not yet complete the transformation because it could always be satisfied by making all the variables *false*, a choice that corresponds to avoiding color conflicts by not coloring the nodes at all. We can complete the transformation by insisting that each node get exactly one color assigned to it. For node a this can be done by the expression

$$(a_{\text{GREEN}} \wedge \overline{a_{\text{RED}}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge a_{\text{RED}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge \overline{a_{\text{RED}}} \wedge a_{\text{BLUE}}) \qquad (1.10)$$

[7] "Harder" and "easier" mean "needing more and less of a computational resource." Most often the resource we care about is computing time. "Harder" and "easier" do not mean harder or easier to program. Programming effort is not addressed by this theory.

In general, to carry out the transformation for any graph $G = G(V, E)$, we just have to make a statement like (1.10) for every node of the graph and a statement like (1.9) for every edge, i.e., we have to write this expression E_G:

$$\bigwedge_{x \in V} ((x_{\text{GREEN}} \wedge \overline{x_{\text{RED}}} \wedge \overline{x_{\text{BLUE}}}) \vee (\overline{x_{\text{GREEN}}} \wedge x_{\text{RED}} \wedge \overline{x_{\text{BLUE}}}) \vee (\overline{x_{\text{GREEN}}} \wedge \overline{x_{\text{RED}}} \wedge x_{\text{BLUE}}))$$

$$\bigwedge$$

$$\bigwedge_{(x,y) \in E} (\overline{(x_{\text{GREEN}} \wedge y_{\text{GREEN}})} \wedge \overline{(x_{\text{RED}} \wedge y_{\text{RED}})} \wedge \overline{(x_{\text{BLUE}} \wedge y_{\text{BLUE}})})$$

This expression E_G is satisfiable if and only if the graph G is 3-colorable.

Editing programs To make Observation 2 useful for proving undecidability results, we need to find transformations from one property of programs into another.

Lemma 1 *There is a program* `edit` *which, given as input a program x, creates as output a program y with the following two properties.*

1. *y never reads any input; and*
2. *y, on any input (which it ignores — see previous item), runs like x would on input x.*

Proof We can achieve this transformation "by hand" in a session with our favorite text editor. As an example, if we start with x being the program "`x.cpp`" from Figure 1.2 we could create the program in Figure 1.10. We can even do this in a way where the editing commands do not depend on the program x that is being editing. Therefore we could take a copy of the editor and hard-code those commands into it (or, preferably and equivalently, put the commands into a "script file" or a "macro" for our editor[8]). □

[8] Better yet, we can simplify this editing task greatly by making y the one-line program "`( x < x.cpp )`," which has all the right properties. To see that it ignores its input, do "`( x < x.cpp ) < input`."

```
void main( void )
{
    x_on_x_no_input();
}

void x_on_x_no_input( void )  // runs like x, but
{                             // instead of reading
    ...                       // input it ‘‘reads’’
}                             // from a string constant
```

Figure 1.9: The result of editing program x

1.6 The Undecidability of All I/0-Properties

Recall that LOOPING is the property that there exists an input on which the program will not terminate.

Theorem 5 LOOPING *is undecidable.*

Proof The program edit of Lemma 1 transforms SELFLOOPING into LOOPING. By Observation 2 this proves LOOPING undecidable. □

What other properties of programs are often of interest? The prime consideration usually is whether or not a program "works correctly." The exact meaning of this depends on what you wanted the program to do. If you were working on an early homework assignment in Programming 101, "working correctly" may have meant printing the numbers 1, 2, 3, . . . , 10.

Let ONE TO TEN be the problem of deciding whether or not a program prints the numbers 1, 2, 3, . . . , 10.

Theorem 6 ONE TO TEN *is undecidable.*

Proof Figure 1.11 illustrates a transformation of SELFLOOPING into the negation of property ONE TO TEN.

Note that unlike the function x_on_x_no_input in the previous proof, the function x_on_x_no_input_no_output suppresses whatever output the program x might have generated when run on its own source code as input. This is necessary because otherwise the transformation would not work in the case of programs x that print the numbers 1, 2, 3, . . . , 10.

Since the negation of ONE TO TEN is undecidable, so is the property ONE TO TEN. □

How many more undecidability results can we derive this way?

Figure 1.13 shows an example of an "input-output table" IO_x of a program x. For every input, the table shows the output that the program produces. If the program loops forever on some input and keeps producing output, the entry in the right-hand column is an infinite string.

```
        #define EOF NULL  // NULL is the proper end marker since
                          // we are ``reading'' from a string.

        void main( void )
        {
            x_on_x_no_input();
        }

        char mygetchar( void )
        {
            static int i = 0;
            static char* inputstring =
// start of input string ////////////////////////////////
"\
\n\
        void main( void )\n\
        {\n\
            char c;\n\
\n\
            c = getchar();\n\
            while (c != EOF)\n\
            {\n\
               while (c == 'X'); // loop forever if an 'X' is found\n\
               c = getchar();\n\
            }\n\
        };\n\
";
// end of input string //////////////////////////////////

            return( inputstring[i++] );
        }

        void x_on_x_no_input( void )
        {
            char c;

            c = mygetchar();
            while (c != EOF)
            {
               while (c == 'X'); // loop forever if an 'X' is found
               c = mygetchar();
            }
        }
```

Figure 1.10: The result of editing the program "x.cpp" of Figure 1.2

```
void main( void )
{
    x_on_x_no_input_no_output();
    one_to_ten();
}

void x_on_x_no_input_no_output( void )
{                               // runs like x, but
                                // instead of reading
    ...                         // input it ‘‘reads’’
                                // from a string constant,
                                // and it does not
}                               // generate any output

void one_to_ten( void )
{
    int i;
    for (i = 1; i <= 10; i++)
        cout << i << endl;
}
```

Figure 1.11: Illustrating the proof of Theorem 6

```
void main( void )
{
    x_on_x_no_input_no_output();
    notPI();
}

void x_on_x_no_input_no_output( void )
{
    ...
}

void notPI( void )          // runs like a program
{                           // that does not have
    ...                     // property PI
}
```

Figure 1.12: The result of the editing step `rice` on input `x`

Inputs	*Outputs*
λ	`Empty input file.`
0	`Okay.`
1	
00	`Yes.`
01	`Yes. Yes. Yes. ...`
10	`No.`
11	
000	`Yes. No. Yes. No. Yes. No. ···`
001	`Segmentation fault.`
010	`Core dumped.`
⋮	⋮

Figure 1.13: The "input-output function" of a lousy program

These tables are not anything we would want to store nor do we want to "compute" them in any sense. They merely serve to define what an "input-output property" is. Informally speaking, a property of programs is an input-output property if the information that is necessary to determine whether or not a program x has the property is always present in the program's input-output table IO_x. For example, the property of making a recursive function call on some input, let's call it RECURSIVE, is not an input-output property. Even if you knew the whole input-output table such as the one in Figure 1.13, you would still not know whether the program was recursive. The property of running in linear time is not an input-output property, nor is the property of always producing some output within execution of the first one million statements, nor the property of being a syntactically correct C program. Examples of input-output properties include

- the property of generating the same output for all inputs,
- the property of not looping on any inputs of length less that 1000,
- the property of printing the words `Core dumped` on some input.

More formally, a property Π of programs is an *input-output property* if the following is true:

> For any two programs a and b, if $IO_a = IO_b$ then either both a and b have property Π, or neither does.

A property of programs is *trivial* if either all programs have it, or if none do. Thus a property of programs is *nontrivial* if there is at least one program that has the property and one that doesn't.[9]

[9]There are only two trivial properties of programs. One is the property of being a program

Theorem 7 (Rice's Theorem) *No nontrivial input-output property of programs is decidable.*

Proof Let Π be a nontrivial input-output property of programs. There are two possibilities. Either the program

```
main () { while (TRUE); }
```
(1.11)

has the property Π or it doesn't. Let's first prove the theorem for properties Π which this program (1.11) does have.

Let `notPI` be a function that runs like a program that does not have property Π. Consider the program that is outlined in Figure 1.12. It can be created from x by a program, call it `rice`, much like the program in Figure 1.9 was created from x by the program `edit` of Lemma 1.

The function `rice` transforms SELFLOOPING into Π, which proves Π undecidable.

What if the infinite-loop program in (1.11) does not have property Π? In that case we prove Rice's Theorem for the negation of Π — by the argument in the first part of this proof — and then observe that if the negation of a property is undecidable, so is the property itself. □