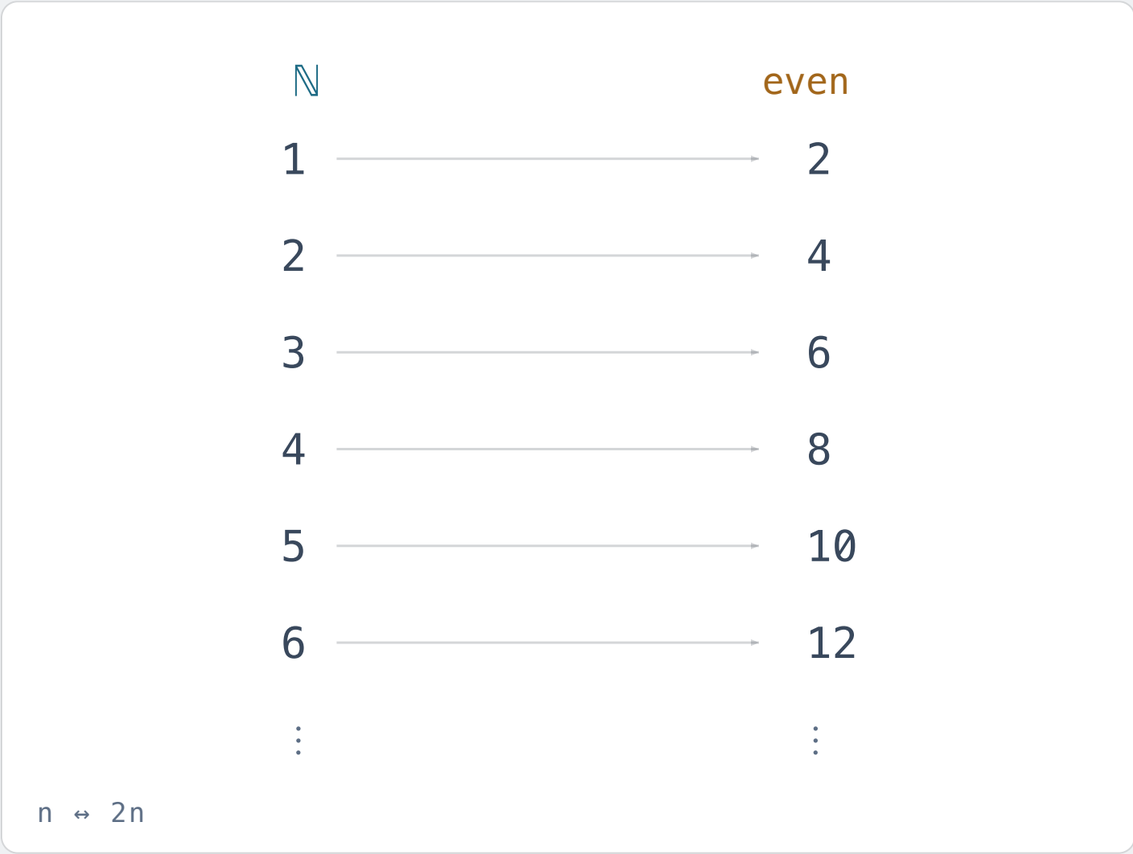
INTRODUCTION TO COMPUTER SCIENCE · WEEK 5 · STATION II

Countability

Everything you can write down can be listed. So a program is a number — and a compiler is arithmetic.

COUNTING WITHOUT NUMBERS

Two sets are the same size if you can pair them up.



No counting needed — just a perfect pairing. Every left has exactly one right, and nothing is left over. That is how you know your two hands have the same number of fingers without counting.

So there are as many even numbers as numbers: pair n with 2n. The part is as big as the whole. Galileo found this disturbing in 1638. Cantor made it the definition.

A set pairable with 1, 2, 3, … is called countable. It can be *listed*: first, second, third, and every element at some finite position.

NOTATION

Everything we can write down is countable.

1.	ε
2.	0
3.	1
4.	00
5.	01
6.	10
7.	11
8.	000
9.	001
10.	…

the enumeration of all texts

every finite text has a number

A program is a finite string of symbols. So is a proof, a specification, a sentence, a grammar, a score, a formula, a prompt.

List all strings of length 0, then length 1, then 2, then 3 … Every finite text appears at some finite position.

So all C* programs that will ever exist form a countable list. Same for all RISC-U binaries. Same for all proofs. Same for all sentences of English.

This is not saying notation is small. Countable is already infinite. It is saying notation is finite-and-discrete, and finite things line up.

GÖDELISIERUNG

Give every text a number, and notation becomes something you can compute with.

SYNTAX

```
if (x) halt();
```

↓

105102324012041321049710811640415 9

NUMBER

a program reading itself

text → number → text

If every text has a position on the list, every text has a number. The simplest version you already know: write the ASCII codes one after the other. A text is a very long integer.

Gödel used this in 1931 to make arithmetic talk about arithmetic. The method carries his name: Gödel numbering.

Once notation is a number, a program can read a program, run a program — and, once a program can read programs, read *itself*. Every computer you have ever used is this idea in metal: code is data.

IN THE TERMINAL

A binary is one number, 188,392 bytes long.

```
$ ./selfie -c selfie.c -o selfie.m
./selfie: 188392 bytes generated with 43492 instructions and 14424 bytes of data
./selfie: 188392 bytes with 43492 instructions and 14424 bytes of data written into selfie.m
$ od -A d -t x1 selfie.m | head -2
0000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
0000016 02 00 f3 00 01 00 00 00 00 00 01 00 00 00 00 00
$ ./selfie -l selfie.m -m 1
./selfie: 188392 bytes with 43492 instructions and 14424 bytes of data loaded from selfie.m
selfie.m { -c { source } | -o binary | [ -s | -S ] assembly | -l binary } [ ( -m | -d | -r | -y ) 0
```

The `-o` option writes the number to a file. The first four bytes, 7f 45 4c 46, spell ELF: the file format every Linux program uses, so this number runs on real RISC-V hardware too.

The `-l` option reads the number back and the `-m` option runs it — on an emulated machine, inside selfie. The program that just printed the synopsis is a number that selfie computed from a text that is selfie.

THE FIXED POINT

The compiler compiles its own source, and gets the same number twice.

selfie1.m = selfie2.m

```
$ make self-self-check
./selfie -c selfie.c -o selfie1.m -s selfie1.s -m 2 -c se
…
diff -q selfie1.m selfie2.m
diff -q selfie1.s selfie2.s
```

Stage one: a C compiler you did not write produces a selfie. Stage two: that selfie compiles selfie.c into selfie1.m. Stage three: selfie1.m, running on the emulator, compiles selfie.c into selfie2.m.

selfie1.m and selfie2.m are identical: 188,392 bytes, the same number. The compiler agrees with itself. What that does *not* prove is the subject of week 8.

CODE IS DATA

A compiler eats programs. An emulator runs programs. A model is trained on programs. And once a system can describe systems, it can describe itself.

Nothing on this deck needed anything but the fact that texts are numbers. The compiler is a function from one number, selfie.c, to another, selfie.m. The emulator is a function from a number and an input to what the number does.

Self-reference is not a trick. It is the price of being expressive enough to be useful — and the instrument with which, in two weeks, we find exactly where expressiveness ends.

ASIDE · WHAT A NUMBER CAN BE

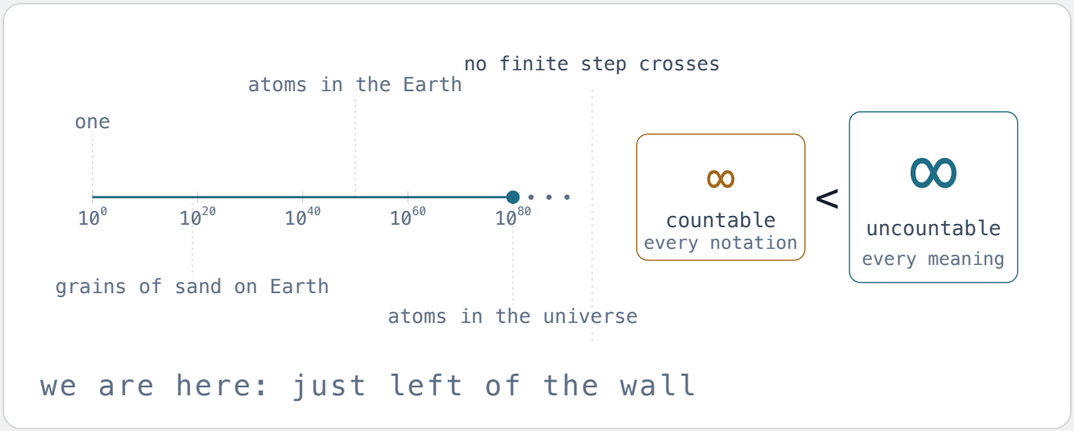
The same 188,392 bytes are a program to the emulator, data to the compiler that wrote them, and, next week, the tape of a universal machine. Nothing in the bytes says which. You have heard this before.

PREVIEW

What we want to talk about is not countable.

Programs: countable. What a program *does* — its answer on every input, an infinite sheet of yes and no — is that countable?

Week 7 proves that it is not, with an argument three lines long that is the most consequential of the twentieth century. Week 6 first builds the machine that runs the programs, so that we know exactly what a behaviour is.



BEFORE NEXT WEEK

Recommended exercises.

1. Read the Meaning chapter up to *Give every text a number*, and the Selfie chapter's three commands.
2. Write down the first sixteen strings over {0, 1} in the order of the enumeration. Where is 1011? What is at position 100?
3. Gödel-number the string `c = 7;` by writing its ASCII codes one after the other. Then decode 99611032611032555559.
4. Run `make self-self-check` and time it. Roughly how many instructions did the emulator execute? The output tells you.
5. Pair the natural numbers with the integers, negative ones included. Then with the fractions, if you dare.

NEXT WEEK

The machine: RISC-U, code and data in one memory, the emulator that runs every program written for it — including itself.