Contents

Introduction

Chapter 1
MACHINES RUN **LOOPS**

Chapter 2
MACHINES GET **LARGE**

Chapter 3
MACHINES ARE **LIVING**

Chapter 4
MACHINES ARE **INCOMPLETE**

Chapter 5
MACHINES CAN BE **INSTRUMENTED**

Chapter 6
MACHINES AUTOMATE **IMBALANCE**

Acknowledgments

Notes

Index

About the Author

John Maeda is an American technologist, designer, engineer, artist, investor, author and teacher. He was recently appointed Chief Experience Officer at Publicis Sapient, the technology consulting and delivery arm of communications and marketing conglomerate Publicis. He has held positions with Automattic, the parent company of WordPress.com, the venture capital firm Kleiner Perkins, led research at the MIT Media Lab, and served as president of the Rhode Island School of Design. Named as one of the seventy-five most influential people of the twenty-first century by *Esquire*, Maeda draws on his diverse background as an MIT-trained engineer, award-winning designer, and executive leader to bring people and ideas together at scale. He is the author of several celebrated books, including *The Laws of Simplicity* and *Redesigning Leadership*. He has appeared as a speaker all over the world, from Davos to Beijing to São Paulo to New York, and his talks for TED.com have received millions of views. He was born, raised, and lives in the United States of America.

To my mother, Elinor “Yumi” Maeda, who taught me how to speak human. She taught me how to treat people the way they love to be treated—in the Hawaiian way.

To my father, Yoji Maeda, who taught me how to work like a machine. He taught me how to craft products that customers love and come back for—in the Japanese way.

Together they worked harder than any people I know so that I could attend any college in the world. Thanks to the two of them, I learned how to speak machine. And most importantly, thanks to Yumi, I haven’t forgotten how to speak human too.

Introduction

It was a typical cold, snowy New England winter day on December 17, 2004, when I started a WordPress "web log" on the topic of simplicity. There was a whimsical motivation to turning my research at MIT in that direction—namely, how the letters MIT occurred in perfect sequence in the words SIMPLICITY and COMPLEXITY. But there was a less Dr. Seussian motivation at play as well, which I wrote about in my first blog entry:

> I have always been interested in how the computer (which is an object of great complexity) and design (which is traditionally about simplicity) tend to mix poorly together like the proverbial "oil and water."[1]

Subsequently, that blog turned into a book titled *The Laws of Simplicity*, which was rapidly translated into fourteen languages. Why was it unusually impactful? I think because it arrived at a time when computing technology was just starting to impact everyday lives back in the pre-iPhone era. That book's overwhelming momentum and the concurrent rise of Apple's successful fusing of design and technology oddly drove me to head in the opposite direction of computing's inherent complexities and instead toward designing for simplicity.

I wanted to somehow get closer to the essence of design and move away from computers the way I had done once prior in my early career—back in the nineties, when I was a practicing graphic designer in Japan with a mismatched MIT pedigree. I'd somehow managed to escape the "T" (Technology) of MIT as an engineering student, and then made a U-turn into the thick of it as an MIT Media Lab professor leading the intersection of design and advanced computing technologies. Perhaps it was dealing with the weight of earning tenure that made me feel stuck somewhere in the future of design. I wanted to reconnect with the classics. I think it was a mix of not knowing what to do with the MBA I'd

earned as a part-time hobby and an overwhelming mood in 2008 of Barack Obama's "Yes, we can," combined with my desire to rekindle the past, that resulted in my becoming the sixteenth president of the esteemed "temple" of the art and design world: Rhode Island School of Design, or RISD.

A series of later milestones, which built upon the work I had long led at the MIT Media Lab, gave me a reputation as a fierce defender of design. These included, for instance, appearing before Congress to encourage putting an "A," for Art, in STEM education, turning it into STEAM, and later launching the "Design in Tech Reports" while working in Silicon Valley at venture capital firm Kleiner Perkins. So in 2019, when a popular business magazine announced in a headline that I'd said, "In reality, design is not that important," it did not come as a surprise to me that I would be dragged through the internet mud by all lovers of design.

There were many short- and long-form responses to the interview that cast my responses as every shade of clueless and ignorant. I knew that the controversy had hit a peak when my idol, Hartmut Esslinger, the design force behind Apple's original design language, started coming after me on social media.[2] If I had earned some sort of "badge" in the design world, the internet was now ruling that it was my public duty to turn that badge in, and for a period of time I would be unwelcome at any Temple of Design out there. How did I feel? Terrible.

My words had been taken out of context from a twenty-minute phone interview—and, frankly, when the article came out, I immediately admired the editorial team's choice of headline as brilliant clickbait. Apparently, my interview was the highest-performing article on their website for quite some time, as evidenced by the myriad spears and cleavers that were continuously being lobbed in my direction. What stung the most was knowing that few people had really read the entire interview, so the headline was all that stuck in anyone's mind. To them, I had completely demeaned the work designers do every day. So I needed to be punished.

Here's the reality: I honestly don't believe that design is the most important matter today. Instead, I believe we should focus first on understanding computation. Because when we combine design with computation, a kind of magic results; when we combine business with computation, great financial opportunities can emerge. What is computation? That's the question I would get

asked anytime I stepped off the MIT campus when I was in my twenties and thirties, and then whenever I left any technology company I worked with in my forties and fifties.

Computation is an invisible, alien universe that is infinitely large and infinitesimally detailed. It's a kind of raw material that doesn't obey the laws of physics, and it's what powers the internet at a level that far transcends the power of electricity. It's a ubiquitous medium that experienced software developers and the tech industry control to a degree that threatens the sovereignty of existing nation-states. Computation is not something you can fully grasp after training in a "learn to code" boot camp, where the mechanics of programming can be easily learned. It's more like a foreign country with its own culture, its own set of problems, and its own language—but where knowing the language is not enough, let alone if you have only a minimal understanding of it.

There's been a conscious push in all countries to promote a greater understanding of how computers and the internet work. However, by the time a technology-centered educational program is launched, it is already outdated. That's because the pace of progress in computing hasn't moved at the speed of humans—it's been moving at the exponential speed of the internet's evolution. Back in 1999, when a BBC interviewer made a dismissive comment about the internet, the late musician David Bowie presciently offered an alternate interpretation: "It's an alien life form ... and it's just landed here."[3] Since the landing of this alien life form, the world has not been the same—and design as it has conventionally been defined by the Temple of Design no longer feels to me like the foundational language of the products and services worlds. Instead, it's ruled by new laws that are governed by the rising Temple of Tech in a way that intrinsically excludes folks who are less technically literate.

A new form of design has emerged: computational design. This kind of design has less to do with the paper, cotton, ink, or steel that we use in everything we physically craft in the real world, and instead has more to do with the bytes, pixels, voice, and AI that we use in everything we virtually craft in the digital world powered by new computing technologies. It's the text bubble that pops up on your screen with a message from your loved one, or the perfect photo you shot in the cold rain with your hands trembling and yet which came out perfectly, or the friendly "Here you go, John" that you hear when you ask your smart stereo to play your favorite

Bowie tunes. These new kinds of interactions with our increasingly intelligent devices and surroundings require a fundamental understanding of how computing works to maximize what we can make.

So I came to wonder if I could find a way for more nontechie people to start building a basic understanding of computation. And then, with that basic conceptual grounding, to show how computation is transforming the design of products and services. For much of the twentieth century, computation by itself was useful only to the military to calculate missile trajectories. But in the twenty-first century, it is *design* that has made computation relevant to business and, more so, to our everyday lives. Design matters a lot when it is leveraged with a deep understanding of computation and the unique set of possibilities it brings. But achieving an intuitive understanding of an invisible alien universe doesn't come so easily.

This book is the result of a six-year journey I have traveled away from "pure" design and into the heart of what is impacting design the most: computation. I will take you on a tour through the minds and cultures of computing machines from how they once existed in a simpler form to how they've evolved into the much more complex forms we know today. Keep in mind that this book is not designed to turn you into a computer science genius—I've vastly simplified, and in some cases oversimplified, technical concepts in a way that will surely raise some experts' eyebrows and might cause them to cringe. But I hope that, armed with even my rough approximations, you will learn how computation has expanded both in technical capability and in sociocultural impact—something you may regard simultaneously as hugely impressive and terribly frightening.

Computation brings its share of problems, but most of them have to do with us—how we use it—rather than by the underlying technology itself. We've entered an era in which the computing machines we use today are powered not just by electricity and mathematics, but by our every action and with insights gained in real time as we use them. In the future we'll have only ourselves to blame for how computation evolves, but we're more likely to succumb to a victim mentality if we remain ignorant as to what is really going on. So it will become only natural to want to pin the blame on a handful of tech company leaders, if not all of them—a fairly likely scenario, since fear of the invisible or unknown is far

more powerful than any fear that has physical form, like a pack of wild wolves or a threatening tornado. The intangible, invisible alien force that is the internet represents the perfect object of fear that already lives in your neighborhood and teaches your children while secretly seeking to do you all harm, which explains why the eeriness of tech is so widely portrayed in TV and movies today.[4]

I have always believed that being curious is better than being afraid—for when we are curious we get inventive, whereas when we are afraid we get destructive. Something about my experience between the Temple of Design and the Temple of Tech has kept me curious all these years. When I think more deeply about it, it's the many career failures I've fortunately experienced in between my few successes that drive me to still remain a little hungry and foolish. But to be honest, I'm just like anyone else—tired, a little lazy, and all too eager to wait for a hero to rise who will protect me and fight for all of us. There's a common lack of understanding of what computation fundamentally can and cannot do. Rather than give away your power of understanding to someone else, I invite you to be curious about the computational universe.

Perhaps I wrote this book for you. Perhaps *you* are the hero the world has been waiting for. Perhaps you are one of the many who will find a way to wield the power of computation with inventiveness and wonder. Those kinds of heroes are now desperately needed in order to advance computation beyond what it is today in its superpowerful, albeit running with the conflicted conscience of a teenager, form. Being new to the computational universe, you just might discover something that we first-generation techies have not yet been able to imagine. When you find it and make it into a success, it will set an example for the rest of us. I wish that heroic moment upon you someday, but first let me start you on the path to speaking the language of the machine.

the F1 race continued with no need for the cars to make a pit stop, we'd think to ourselves, *Magic!*

But a computer running a program, if left powered up, can sit in a loop and run forever, never losing energy or enthusiasm. It's a metamechanical machine that never experiences surface friction and is never subject to the forces of gravity like a real mechanical machine—so it runs in complete perfection. This property is the first thing that sets computational machines apart from the living, tiring, creaky, squeaky world.

I first came in touch with the power of computational loops in 1979, when I was in seventh grade. I had just encountered my first computer. That in itself was unusual for someone with my background growing up on the poor side of town; thanks to desegregation efforts motivated by the civil rights movement, I was bused to a school an hour away from my home that was much better than the run-down schools in my neighborhood.

Commodore was a big name in computing at the time. When I say big, of course I mean big for maybe a few thousand people in the United States and Europe. Personal computers weren't really personal back then, because the average family could not afford one. The Commodore PET (Personal Electronic Transactor) was manufactured in the United States with a small screen displaying text in only fluorescent green, a tiny tactile keyboard, and a tape cassette drive for storage. It had 8 kilobytes of total memory, and its processing speed was 1 megahertz. In comparison, the average mobile phone has 8 gigabytes and runs at 2 gigahertz, which is a millionfold increase in memory and a thousandfold increase in speed.

There was no internet yet, so you couldn't search for anything. There was no Microsoft, so you couldn't do your schoolwork on a word processor or spreadsheet. There was no touch screen or mouse, so you couldn't directly interact with what was on the monitor. There were no color or grayscale pixels to display images with, so you couldn't visually express information. There was only one font, and text was only uppercase. You would "navigate" the computer screen by pressing the cursor keys: up, down, left, right. And any functionality that you wanted it to have, you would need to type in a program to create it yourself or type it in line by line from a book or magazine.

As you can imagine, the computer sat in the classroom generally unused—it was not only useless, it was soulless as an experience.

No expressive or informative images. No stereo sound or the latest tunes. No utility or empowerment with an amazing set of apps. It just blinked at you, constantly, with its cursor rectangle—awaiting you to type in instructions for it to follow. And when you did raise the courage to type something into it, you would likely be rewarded with, in all caps, SYNTAX ERROR—which essentially translated to YOU'RE WRONG. I DON'T UNDERSTAND.

It's no surprise that the computer attracted only a few kinds of students—perhaps those who grew up a bit lower on the empathy scale (like me), or those who could tolerate the crushing blow of being told they were wrong with each keystroke. My friend Colin, whose parents worked with computers at Boeing, showed me my first program. He rapidly typed the following program into the PET, free from any syntax errors:

```
10 PRINT "COLIN"
20 GOTO 10
```

Then he told me to go ahead and type RUN ... What happened next astounded me. The computer began printing "COLIN" continuously. I asked when it was going to stop. Colin said, "Never." This worried me. He then proceeded to break the program with CONTROL-C. And then the blinking text prompt came back.

Colin then retyped the first line of code, but this time with one space and a semicolon.

```
10 PRINT "COLIN ";
```

And he typed RUN to display the following:

```
COLIN COLIN COLIN COLIN COLIN COLIN COLIN
COLIN COLIN COLIN COLIN COLIN COLIN COLIN
COLIN COLIN COLIN COLIN COLIN COLIN COLIN
COLIN COLIN COLIN COLIN COLIN COLIN COLIN
COLIN COLIN COLIN COLIN ...
```

And again, it kept printing and scrolling by. I tried it myself, and typed:

```
10 PRINT "MAEDA "...
```

to experience the incredibly affirming display of my name being said forever and ever:

```
MAEDA MAEDA MAEDA MAEDA MAEDA MAEDA MAEDA
MAEDA MAEDA MAEDA MAEDA MAEDA MAEDA MAEDA
MAEDA MAEDA ...
```

Thereafter, I would perform this magic "say my name" trick for anyone who was curious about the computer. I did it for my classmate Jessica, who I kind of had a thing for. The limits of my computer expertise became evident when she asked, "What else can you do?" *Uh-oh.*

But my curiosity was piqued. I started to read *Byte* magazine (one of maybe two computing magazines available). Since there was close to no software available, it was really important to learn to write programs. *Byte* regularly included entire computer programs printed out across many pages and ready to be manually typed in to a computer yourself—the only problem was that I didn't have a computer to regularly use.

Luckily, my mother, Elinor, was always forward-looking and hopeful that her children could do bigger and better things in life. She set aside enough money from our small, family-operated tofu shop in Seattle to buy me an Apple II computer and an Epson line printer. To express my gratitude to her, I wanted my first computer program to help her in some way at the tofu shop. Thus I set out to write a monthly billing program that I hoped could save her some time. It would manually take in our regular customers' orders each week and print out an invoice at the end of the month.

I was a fast typist by the tenth grade, and so with zeal I wrote this program that I felt could help my mother. It took me maybe three months of programming every day after school. My conundrum was figuring out how to deal with leap years—I figured that if I made an input routine for 365 days in a year, I would run into a problem every four years. In the end I saw that as a bridge to cross when I came to it, so instead I just kept typing and typing until I had completed all 365 input routines (there were no text editors with copy and paste functions). It was a manual, laborious project. I recall the deep satisfaction I felt when my mother first used it to print invoices for the month.

Shortly after this moment of success, my tenth-grade math teacher, Mr. Moyer, encouraged me to come to his after-school

computer club. I had gained a reputation for being skilled at computer programming—perhaps what I should really say is that I was a computer nerd. Having successfully written my first thousand-line computer program, I thought it would be beneath me to go to Mr. Moyer's club meeting as I would surely be too much of an expert compared with the others attending. But on the day I showed up, I vividly remember Mr. Moyer talking about LOOPS using a command called FOR ... NEXT. Upon learning about it, I broke into a sweat—the kind of sweat you feel when you've done something terribly stupid.

When I got home that evening, I looked at my long computer program, which was 365 distinct input statements of the form:

```
10 DIM T(365), A(365) : HOME
100 REM GET THE NUMBER OF TOFU AND SUSHI
  AGE FOR EACH DAY OF THE YEAR
110 REM COMMENTS LIKE THIS ARE HOW
  PROGRAMMERS TALK TO THEMSELVES
120 PRINT "IT'S DAY 1"
130 PRINT "HOW MANY TOFU"
140 INPUT T(1)
150 PRINT "TOFU ORDER IS", T(1)
160 PRINT "HOW MANY DOZEN SUSHI AGE"
170 INPUT A(1)
180 PRINT "SUSHI AGE ORDER IS", A(1)
290 PRINT "CONTINUE? HIT 0 TO EXIT OR 1 TO
  CONTINUE"
200 INPUT ANSWER
210 IF (ANSWER = 0) GOTO 9999
220 PRINT "IT'S DAY 2"
230 PRINT "HOW MANY TOFU"
240 INPUT T(2)
250 PRINT "TOFU ORDER IS", T(2)
260 PRINT "HOW MANY DOZEN SUSHI AGE"
270 INPUT A(2)
280 PRINT "SUSHI AGE ORDER IS", A(2)
```

```
290 PRINT "CONTINUE? HIT 0 TO EXIT OR 1 TO
  CONTINUE"
300 INPUT ANSWER
310 IF (ANSWER = 0) GOTO 9999
320 PRINT "IT'S DAY 3"
330 PRINT "HOW MANY TOFU"
340 INPUT T(3)
350 PRINT "TOFU ORDER IS", T(3)
360 PRINT "HOW MANY DOZEN SUSHI AGE"
370 INPUT A(3)
380 PRINT "SUSHI AGE ORDER IS", A(3)
390 PRINT "CONTINUE? HIT 0 TO EXIT OR 1 TO
  CONTINUE"
400 INPUT ANSWER
410 IF (ANSWER = 0) GOTO 9999
420 PRINT "IT'S DAY 4"
430 PRINT "HOW MANY TOFU"
440 INPUT T(4)
450 PRINT "TOFU ORDER IS", T(4)
460 PRINT "HOW MANY DOZEN SUSHI AGE"
470 INPUT A(4)
480 PRINT "SUSHI AGE ORDER IS", A(4)
490 PRINT "CONTINUE? HIT 0 TO EXIT OR 1 TO
  CONTINUE"
500 INPUT ANSWER
510 IF (ANSWER = 0) GOTO 9999
520 PRINT "IT'S DAY 5"
530 PRINT "HOW MANY TOFU"
540 INPUT T(5)
550 PRINT "TOFU ORDER IS", T(5)
560 PRINT "HOW MANY DOZEN SUSHI AGE"
570 INPUT A(5)
580 PRINT "SUSHI AGE ORDER IS", A(5)
590 PRINT "CONTINUE? HIT 0 TO EXIT OR 1 TO
  CONTINUE"
```

you see on-screen with an app is closer to the fast-food drive-thru sign that you drive up to and speak into—which has nothing inside it except a tethered connection to a bustling food factory just a few car lengths away. Just as you can learn nothing about how the actual restaurant works by taking apart the drive-thru's microphone box, the pixels on a computer screen don't tell us anything about the computational machine that it is connected to. Contrast that with cracking open the hardware you're running that app on—although its internals would be a bit confusing, you would still find the screen, the battery or power supply, and a few other recognizable parts. That's because when it comes to something that exists in the physical world, you can touch and understand it, to a degree, when you crack it open. Machines in the real world are made up of wires, gears, and hoses that kind of make sense, whereas machines in the digital world are made up of "bits" or "zeroes and ones," which are completely invisible.

But what about the program codes from the previous section, where I produced an invoice for my parents' tofu shop? The software, written in BASIC, is something you can read with your eyes or ears. Is software visible? Yes and no. On the one hand, program code is what lies at the heart of software and you can read it, but that's like confusing the recipe for cake with the cake itself. The software is what comes alive inside the machine due to the program codes—it's the cake, not the recipe. This can be a difficult conceptual leap.

Understanding the distinction between software running on a computer within its "mind" versus the actual program code being fed into the computer is useful because it lets you conceptualize what is really happening with computation. It can free you from believing that computer code is just, well, computer code—which on the surface is all you can generally see. But what computer code can represent is where the true potential lies. We could say the same for the words you are reading on this page in the sense that they spark intangible ideas in your mind, so it's not the actual words you are experiencing but the invisible ideas that underlie them instead. And in the same way that you know how powerful your imagination can become when fed with the right literary fuel (hopefully that includes this book), then your mind is empowered to do things you previously thought were impossible. That's what happens when a well-crafted computer program is brought to life with a finger tap or a double click—an alternate, invisible

consciousness instantly manifests, just like the magical moment when hydrating a completely desiccated sponge.

Computing machines can freely imagine within "cyberspace," a term coined by William Gibson in the novel *Neuromancer* in 1984, the same year I started at MIT:

> Cyberspace, a consensual hallucination, experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts ... Unthinkable complexity. Lines of light ranged in the nonspace of the mind, clusters and constellations of data. Like city lights receding ...

Trippy. But accurate, or at least close to what I've viscerally experienced within the invisible world "inside" the machine when writing code—note that the quotes around the word "inside" are important because there's nothing that lies within the visible shell of an app. There's a nether universe that computational machines can easily tap into that precedes the internet, and now because of the internet and the ubiquity of network-enabled devices, that universe has expanded wildly beyond what any of the lucky nerds like me who were there at the beginning could ever have expected. Gibson's "consensual hallucination, experienced daily by billions," can on the one hand allude to Facebook or any social media network today, or a shared multiplayer video game in a lush three-dimensional virtual world—or else in the less concrete direction of the "unthinkable complexity" that Gibson refers to, which is a better characterization of what I conjure within the poetry of his description of "lines of light ranged in the nonspace of the mind, clusters and constellations of data."

As you can tell, I'm unusually passionate about this subject, and quite eager for you to understand it with me. Whether by creating an art installation in Kyoto in 1993 to try to visualize the inner workings of a computer as a literal discotheque of people posing as computer parts, or by projecting on nine large screens in a dark gallery billions of chaotic particles buzzing about like bees for my 2005 exhibition at the Cartier Foundation in Paris, I've wanted more people to experience what digital consciousness can feel like. Why? Because I believe to speak machine, you need to "live" the world of the machine too. And, unfortunately, it is by nature invisible.

One way to understand it is by becoming a master computer programmer, but that's not the desired path for everyone. So as we move forward through the chapters of the book, try to keep Gibson's image of cyberspace as an apt representation of how native machine speakers collectively "see" the invisible.

Before we make the jump fully back into cyberspace, let's briefly dip into the topic of the final chapter of this book—Machines Automate Imbalance—to examine another invisible aspect of the history of computation that will be of great benefit to know. Just like any efficient computational machine, any given history of human beings will be repeated over and over until it becomes perceived as fact. So before you get too excited about machines, let's embrace more of the invisible by looking back at when human beings were the fully visible machinery of computation. And in the process you will have the opportunity to rewrite computing's history by properly including the many professional women who were unfairly made invisible.

3. Human computers are the original computing machines.

The first computers were not machines, but humans who worked with numbers—a definition that goes back to 1613, when English author Richard Braithwaite described "the best arithmetician that ever breathed" as "the truest computer of times."[1] A few centuries later, the 1895 *Century Dictionary* defined "computer" as follows:[2]

> One who computes; a reckoner; a calculator; specifically, one whose occupation is to make arithmetical calculations for mathematicians, astronomers, geodesists, etc. Also spelled computor.

At the beginning and well into the middle of the twentieth century, the word "computer" referred to a person who worked with pencil and paper. There might not have been many such human computers if the Great Depression hadn't hit the United States. As a means to create work and stimulate the economy, the Works Progress Administration started the Mathematical Tables Project, led by mathematician Dr. Gertrude Blanch, whose

objective was to employ hundreds of unskilled Americans to hand-tabulate a variety of mathematical functions over a ten-year period. These calculations were for the kinds of numbers you'd easily access today on a scientific calculator, like the natural constant *ex* or the trigonometric sine value for an angle, but they were instead arranged in twenty-eight massive books used to look up the calculations as expressed in precomputed, tabular form. I excitedly purchased one of these rare volumes at an auction recently, only to find that Dr. Blanch was not listed as one of the coauthors—so if conventional computation has the problem of being invisible, I realized that human computation had its share of invisibility problems too.

Try to imagine many rooms filled with hundreds of people with a penchant for doing math, all performing calculations with pencil and paper. You can imagine how bored these people must have been from time to time, and also how they would have needed breaks to eat or use the bathroom or just go home for the evening. Remember, too, that humans make mistakes sometimes—so someone who showed up to work late after partying too much the night prior might have made a miscalculation or two that day. Put most bluntly, in comparison with the computers we use today, the human computers were comparatively slow, at times inconsistent, and would make occasional mistakes that the digital computer of today would never make. But until computing machines came along to replace the human computers, the world needed to make do. That's where Dr. Alan Turing and the Turing machine came in.

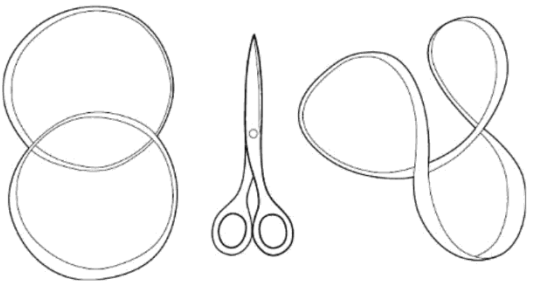
The idea for the Turing machine arose from Dr. Turing's seminal 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem," which describes a way to use the basic two acts of writing and reading numbers on a long tape of paper, along with the ability to write or read from anywhere along that tape of paper, as a means to describe a working "computing machine." The machine would be fed a state of conditions that would determine where the numbers on the tape would be written or rewritten based on what it could read—and in doing so, calculations could be performed. Although an actual computing machine could not be built with technology available back then, Turing had invented the ideas that underlie all modern computers. He claimed that such a machine could universally enable any calculation to be performed by storing the programming codes onto the processing tape itself. This is exactly how all computers

work today: the memory that a computer uses to make calculations happen is also used to store the computer codes.

Instead of many human computers working with numbers on paper, Alan Turing envisioned a machine that could tirelessly calculate with numbers on an infinitely long strip of paper, bringing the exact same enthusiasm to doing a calculation once, or 365 times, or even a billion times—without any hesitation, rest, or complaint. How could a human computer compete with such a machine? Ten years later, the ENIAC (Electronic Numerical Integrator and Computer), built for the US Army, would be one of the first working computing machines to implement Turing's ideas.[3] The prevailing wisdom of the day was that the important work of the ENIAC was the creation of the hardware—that credit being owned by ENIAC inventors John Mauchly and John Presper Eckert. The perceived "lesser" act of programming the computer—performed by a primary team of human computers comprising Frances Elizabeth Snyder Holberton, Frances Bilas Spence, Ruth Lichterman Teitelbaum, Jean Jennings Bartik, Kathleen McNulty Mauchly Antonelli, and Marlyn Wescoff Meltzer—turned out to be essential and vital to the project, and yet the women computers of ENIAC were long uncredited.[4]

As computation could be performed on subsequently more powerful computing machines than the ENIAC and human computers started to disappear, the actual act of computing gave way to writing the set of instructions for making calculations onto perforated paper cards that the machines could easily read. In the late 1950s, Dr. Grace Hopper invented the first "human readable" computer language, which made it easier for people to speak machine. The craft of writing these programmed instructions was first referred to as "software engineering" by NASA scientist Margaret Hamilton at MIT in the 1960s. Around this time, Gordon Moore, a pioneering engineer in the emerging semiconductor industry, predicted that computing power would double approximately every year, and the so-called Moore's law was born. And a short two decades later I would be the lucky recipient of a degree at MIT in the field that Hamilton had named, but with computers having become by then many thousands of times more powerful—Moore's exponential prediction turned out to be right.

To remain connected to the humanity that can easily be rendered invisible when typing away, expressionless, in front of a metallic box, I try to keep in mind the many people who first



You can imagine that, as a kid believing in magic, I totally flipped out. It felt like I'd stumbled upon some sort of dark magic. I wondered if I had unleashed the same eerie power I thought I had discovered when I could see through my hand. You can take this one step further by doing as John Barth did with his "Frame-Tale": on one side of a thin page, write, "ONCE UPON A TIME," and on the other side, "THERE WAS A STORY THAT BEGAN," and then tape the ends together with a twist. Start reading the story as you trace the curve of the Möbius strip. You find that the story never ends. That's the exact feeling that recursion instills in folks who appreciate it—it's a simple-looking loop, but with a literal twist. And with just that twist, it enters a different world.

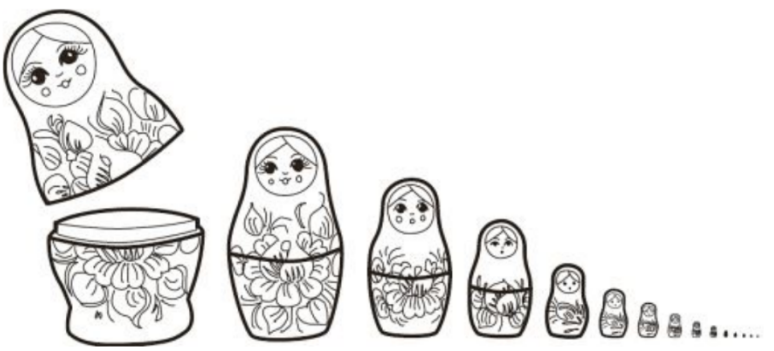
When it comes to writing recursion into computer programs, it's as simple as defining an idea as directly related to the idea itself. When first learning how to draw a tree, you see that this is reflected in nature. A tree starts with a vertical line that has a few lines popping out of its top. To proceed to draw the tree further, you take each of the lines at the top and repeat the act of adding more lines to pop out of each top. And so forth. In the end you get a tree with a lot of subbranches created by simply using the same method you started with. In other words, a tree branch is composed of tree branches. The part is defined by the part itself. You see this played out in the exact opposite direction from the sky into the ground underneath a tree with its root system—so nature paints with recursion quietly and obviously.

Another way to consider the magic of recursion is to look at a proprietary operating system called Unix that renegade programmers attempted to completely rewrite from scratch in the 1980s. Rather than let Unix be controlled by its owner, AT&T, MIT's Richard Stallman wanted such an important software system to be free of any constraints. He called his endeavor the GNU Project.[5] The name GNU epitomizes the idea of recursion, as it signifies "(G)NU's (N)ot (U)nix." Pause for a moment as you read that sentence. So the "U" makes sense as standing for "(U)nix," and the "N" makes sense as standing for "(N)ot." It's when you get to the "G" as standing for "(G)NU" that things get a bit weird. If we try to expand out the "G" and spell it out successively, you can see the infinite nature of this expression:

```
[G]NU's Not Unix
[[G]NU's Not Unix]NU's Not Unix
[[[G]NU's Not Unix]NU's Not Unix]NU's Not
  Unix
[[[[G]NU's Not Unix]NU's Not Unix]NU's Not
  Unix]NU's Not Unix
[[[[[G]NU's Not Unix]NU's Not Unix]NU's
  Not Unix]NU's Not Unix]NU's Not Unix
...
```

Relatedly, a physical metaphor that comes close is a Russian matryoshka doll—a doll that has a smaller but identical version of itself nested inside it, and on and on. That's because we can say that a matryoshka doll is made from another matryoshka doll, and so on and so forth. But even with the world record matryoshka doll set, you will run out of dolls by the time you pull out the fifty-first

one; when it comes to computational matryoshka dolls, there are no specific limits to how deeply they can be nested within each other unless a "base case" is set by the programmer explicitly. Imagine opening a doll after smaller doll after smaller doll after even a doll that is the size of a rice kernel, and you can still continue to find another doll inside the doll inside the doll.



On a more practical level beyond playing with dolls, mathematically minded folks can take the idea of recursion and create elegant expressions of concepts that look nothing like the messier tofu shop accounting program that I showed you before. Recursion differs from the brute force expression of a loop which likens itself to more like a conveyor belt on an assembly line. You lay out all the tasks, and then you tell each task to perform in sequence. And then you GOTO the beginning of the list and do it over again, like on an assembly line. Recursion is stylistically different in nature where you define the task to be performed in terms of itself—like laying out the steps to make a large pot of curry on an assembly line where one key ingredient is a smaller pot of curry. You end up with an assembly line that vanishes inside the process of making the smaller pot of curry, which in turn will require a smaller pot of curry, and so forth—you simply disappear inside the thing you're creating. It's not a concept for the fainthearted. The central idea is to express the definition of something with a definition itself, which is a vaguely imaginable idea that doesn't have a home in the real world but is completely native in the realm of cyberspace.

So now you know there are certain forms of elegantly expressing oneself in the computational world that are akin to the question-inducing power of art as we know it in our physical world. Along those lines, computational thinkers have

appreciation for a kind of highly conceptual art that isn't yet at the Metropolitan Museum of Fine Art. When programmers say "code is poetry," they really mean it. Recursion is an unusually compact way to express complex ideas that can be infinite in nature and are deeply paradoxical, like what happens when you try to unpack "GNU." In computation, it becomes possible to build an enigma into actual working machinery, but even before the computer era, recursion was a captivating philosophical concept. Or, expressed succinctly by Michael Corballis in his entire book on the topic of recursion from a humanist's perspective:[6]

> **Recursion** (rĭ-kûr'-zhən) *noun*. If you still don't get it, see **recursion**.

5. Loops are indestructible unless a programmer has made an error.

Think back to how we got started in this chapter with a loop that let us count to one billion:

```
top = 1000000000
i = 0
while i < top: i = i + 1
```

I just timed this running on my computer and it completed in under a minute. Keep in mind that by the time this book is printed and you have it in your hands, computing machines will have become even faster. The counter runs unimpeded—much like if you were behind the wheel of a fast car on a road that extended forever and hit the gas pedal. But what if there were a big rock a few miles down the road that—with your stereo blasting and your adrenaline on fire—you could easily fail to notice while speeding? That's right. *Blam!* Your car will likely wipe out, and hopefully you'll have your seat belt on.

There's a difference I'd like you to consider when thinking about the counting loop above as compared with my car analogy. The car will start to accelerate and at some point hit top speed. As it encounters the rock you will violently experience an "ouch" moment for at least a few seconds. You'll have time to regain your

senses and then step out of the fire and debris, hopefully with only a few minor flesh wounds.

The computational process, once initiated, will run at top speed from the very moment it comes alive. And if it were to hit some kind of error, it would immediately stop. The entire world it lived within would vanish in that same instant too. In the instant when the computing machine has been involuntarily stopped, it's an absolute catastrophe. Because when computation is doing its thing, you can't see it doing its thing. But when it's not working, it will either complain to you explicitly or it will simply freeze. I'm sure you've seen this happen before. Some message is flashed on your screen or the computer screen simply goes blank. You've usually had no warning at all before it's about to happen—and it usually makes you a bit unhappy, or even angry. Just search for "computer rage" on the internet and you'll feel in good company.

Before we go into why computers crash, let's consider what it feels like for the computer. The best analogy I can think of involves the many "epic" domino setup experts you see online who painstakingly lay out thousands of domino tiles and then turn on the video camera to watch the dominos fall in perfect sequence until one tile has been placed incorrectly and ... FAIL. The embarrassment and shock are real, and there's only one recourse: go back and fix everything, right from the very start.

It takes just one misplaced domino to destroy hundreds of items moving in perfect sequence together—this is what it's like when the software application you're running comes to a complete halt. And it's with the same straightforward attitude, with all the little dominos strewn everywhere across the floor, that a programmer needs to gleefully say: "It needs to be fixed." Much like the expert domino placer's disciplined patience to fix and redo everything, a programmer must behave in exactly the same manner. If computer programmers became uncontrollably angry each time a piece of software crashed, they wouldn't get any work done. Because software crashes a lot, you'll tend to find that people who write software professionally have an unusually high tolerance for catastrophes while also having little tolerance for minor mistakes that could easily be avoided.

Imagine a job where every few minutes you're likely to be told, by a computer, that you did something wrong. The more complex the program or computing system upon which it is running, the more things that can go wrong. These fall into three categories:

PENGUIN BOOKS

UK | USA | Canada | Ireland | Australia
India | New Zealand | South Africa

Penguin Books is part of the Penguin Random House group of companies whose addresses can be found at global.penguinrandomhouse.com.

First published in the United States of America by Portfolio/Penguin 2019
First published in Great Britain by Penguin Business 2019

Follow us on LinkedIn: linkedin.com/company/penguinbusiness

ISBN: 978-0-241-97661-6