1 0

0 »BITWISE

1 0

^=A LIFE

10 1N CODE

1 «DAVID

0 AUERBACH

0

1

Grateful acknowledgment is made to Schocken Books, a division of Penguin Random House LLC, for permission to reprint an excerpt of "In the Penal Colony" from *The Completed Stories* by Franz Kafka, edited by Nahum N. Glatzer, copyright © 1946, 1947, 1948, 1954, 1958, 1971 by Penguin Random House LLC. Reprinted by permission of Schocken Books, a division of Penguin Random House LLC. All rights reserved.

Some material in the chapters "Logo and Love" and "Chat Wars" first appeared, in a different form, in *Slate* and *n + 1*.

This page constitutes an extension of this copyright page.

Library of Congress Cataloging-in-Publication Data

Name: Auerbach, David (David B.), author.

Title: Bitwise : a life in code / David Auerbach.

Description: First edition. New York : Pantheon Books, 2018. Includes bibliographical references and index.

Identifiers: LCCN 2017055983. ISBN 9781101871294 (hardcover : alk. paper). ISBN 9781101871300 (ebook).

Subjects: LCSH: Computer science—Philosophy. Computer science—Social aspects. Auerbach, David (David B.)—Philosophy. Computer scientists—United States—Biography.

Classification: LCC QA76.167 .A84 2018 | DDC 004—dc23 | LC record available at lccn.loc.gov/2017055983

Ebook ISBN 9781101871300

www.pantheonbooks.com

Cover design by Tyler Comrie

v5.3.2

ep

Contents

Cover
Title Page
Copyright
Dedication
Introduction

Part I
- Chapter 1: Logo and Love
- Chapter 2: Chat Wars
- Chapter 3: Binaries
- Interlude: Foreign Tongues

Part II
- Chapter 4: Naming of Parts
- Chapter 5: Self-Approximations
- Chapter 6: Games Computers Play
- Interlude: Adventures with Text

Part III
- Chapter 7: Big Data
- Chapter 8: Programming My Child
- Chapter 9: Big Human
- Epilogue: The Reduction of Language, the Flattening of Life

Acknowledgments
Notes
Further Reading
Works Cited

Illustration Credits
A Note About the Author

INTRODUCTION

Thoughtfulness means: not everything is as obvious as it used to be.

—HANS BLUMENBERG

COMPUTERS always offered me a world that made sense. As a child, I sought refuge in computers as a safe, contemplative realm far from the world. People confused me. Computers were precise and comprehensible. On the one hand, the underspecified and elusive world of human beings; on the other, the regimented world of code.

I had tried to make sense of the real world, but couldn't. Many programmers can. They navigate relationships, research politics, and engage with works of art as analytically and surgically as they do code. But I could not determine the algorithms that ran the human world. Programming computers from a young age taught me to organize thoughts, break down problems, and build systems. But I couldn't find any algorithms sufficient to capture the complexities of human psychology and sociology.

Computer algorithms are sets of exact instructions. Imagine describing how to perform a task precisely, whether it's cooking or dancing or assembling furniture, and you'll quickly realize how much is left implicit and how many details we all take for granted without giving it a second thought. Computers don't possess that knowledge, yet computer systems today have evolved imperfect pictures of ourselves and our world. There is a gap between those pictures and reality. The smaller the gap, the more useful computers become to us. A self-driving car that can only distinguish between empty space and solid objects operates using a primitive image of the world. A car that can distinguish between human and nonhuman objects possesses a more

sophisticated picture, which makes it better able to avoid deadly errors. As the gap closes, we can better trust computers to *know* our world. Computers can even trick us into thinking the gap is smaller than it really is. This book is about that gap, how it is closing, and how *we* are changing as it closes. Computers mark the latest stage of the industrial revolution, the next relocation of our experience from the natural world to an artificial and man-made one. This computed world is as different from the "real" world as the factory town is from the rural landscape.

Above all, this book is the story of my own attempt to close that gap. I was born into a world where the personal computer did not yet exist. By the time I was old enough to program, it did, and I embraced technology. In college, I gained access to the internet and the nascent "World Wide Web," back in the days when AOL was better known than the internet itself. I studied literature, philosophy, and computer science, but only the latter field offered a secure future. So after college I took a job as a software engineer at Microsoft before moving to Google's then-tiny New York office. I took graduate classes in literature and philosophy on the side, and I continued to write, even as the internet ballooned and our lives gradually transitioned to being online all the time. As a coder and a writer, I always kept a foot in each world. For years, I did not understand how they could possibly converge. But neither made sense in isolation. I studied the humanities to understand logic and programming, and I studied the sciences to understand language and literature.

A "bitwise operator" is a computer instruction that operates on a sequence of bits (a sequence of 1s and 0s, "bit" being short for "binary digit"), manipulating the individual bits of data rather than whatever those bits might represent (which could be anything). To look at something bitwise is to say, "I don't care what it means, just crunch the data." But I also think of it as signifying an understanding of the hidden layers of data structures and algorithms beneath the surface of the worldly data that computers store. It's not enough to be worldwise if

computers are representing the world. We must be bitwise as well—and be able to translate our ideas between the two realms.

This book traces an outward path—outward from myself and my own history, to the social realm of human psychology, and then to human populations and their digital lives. Computers and the internet have flattened our local, regional, and global communities. Technology shapes our politics: in my lifetime, we have gone from Ronald Reagan, the movie star president, to Donald Trump, the tweeting president. We are bombarded with worldwide news that informs our daily lives. We form virtual groups with people halfway around the world, and these groups coordinate and act in real time. Our mechanisms of reason and emotion cannot process all this information in a systematic and rational way. We evolved as mostly nomadic creatures living in small communities, not urban-dwelling residents connected in a loose but extensive mesh to every other being on the planet. It's nothing short of astounding that the human mind copes with this drastic change in living. But we don't think quite right for our world today, and we are attempting to off-load that work to computers, to mixed results.

Computers paradoxically both mitigate and amplify our own limitations. They give us the tools to gain a greater perspective on the world. Yet if we feed them our prejudices, computers will happily recite those prejudices back to us in quantitative and apparently objective form. Computers can't know us—not yet, anyway—but we think they do. We see ourselves differently in their reflections.

We are also, in philosopher Hans Blumenberg's term, "creatures of deficiency." We are cursed to be aware of our poverty of understanding and the gaps between our constructions of the world and the world itself, but we can learn to constrain and quantify our lack of understanding. Computers may either help us understand the gaps in our knowledge of the world and ourselves, or they may exacerbate those gaps so thoroughly that we forget that they are even there. Today they do both.

PART I

1

LOGO AND LOVE

The Turtle

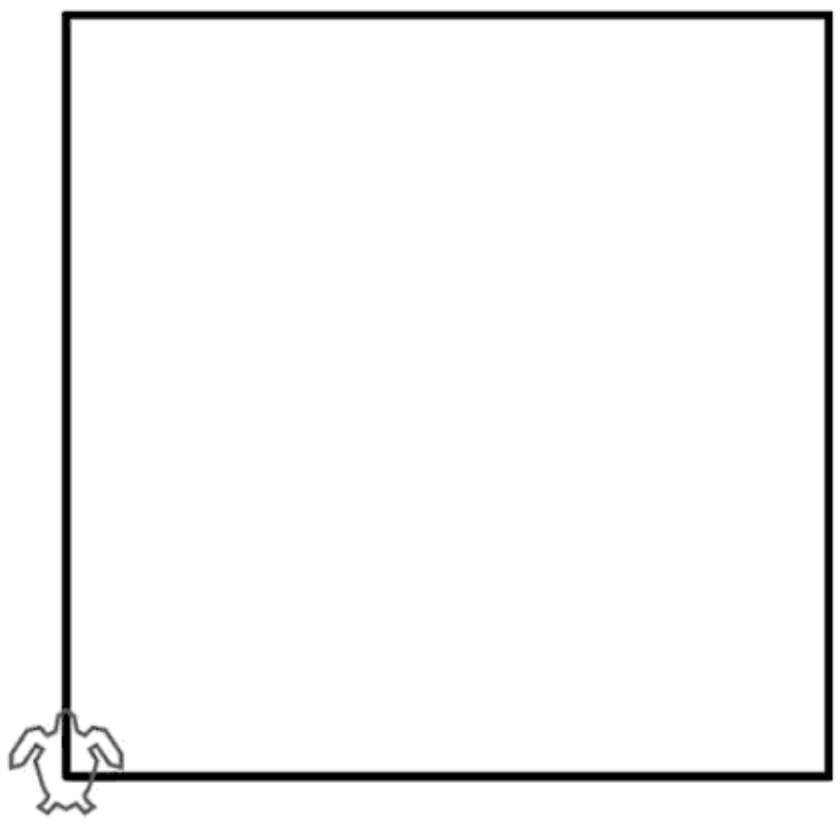
> I found particular pleasure in such systems as the differential gear….I fell in love with the gears.
>
> —SEYMOUR PAPERT

WE ARE DRIVEN TO DISCOVER how things work, but I was often disappointed to find out that one thing or another didn't work more neatly. The television, the automobile, and the human body seemed like they could be more organized, more elegant. Computers, however, did not disappoint me.

Like so many software engineers, I was a shy and awkward child, and I understood computers long before I understood people. The precision, clarity, and reliability that computers promised, particularly in the 1980s when they were so much simpler than they are today, provided a refuge for many children who did not easily integrate into the social fabric of their peers. But a computer was not merely something that I could play with; it was something I could program and control, and with which I could create a new world. Computers are now moving toward virtual reality and photorealistic games, but back then computers displayed only a screen of text and primitive monochrome graphics, which were nonetheless enough to support something that remains more fundamentally powerful than the sharpest graphics: code.

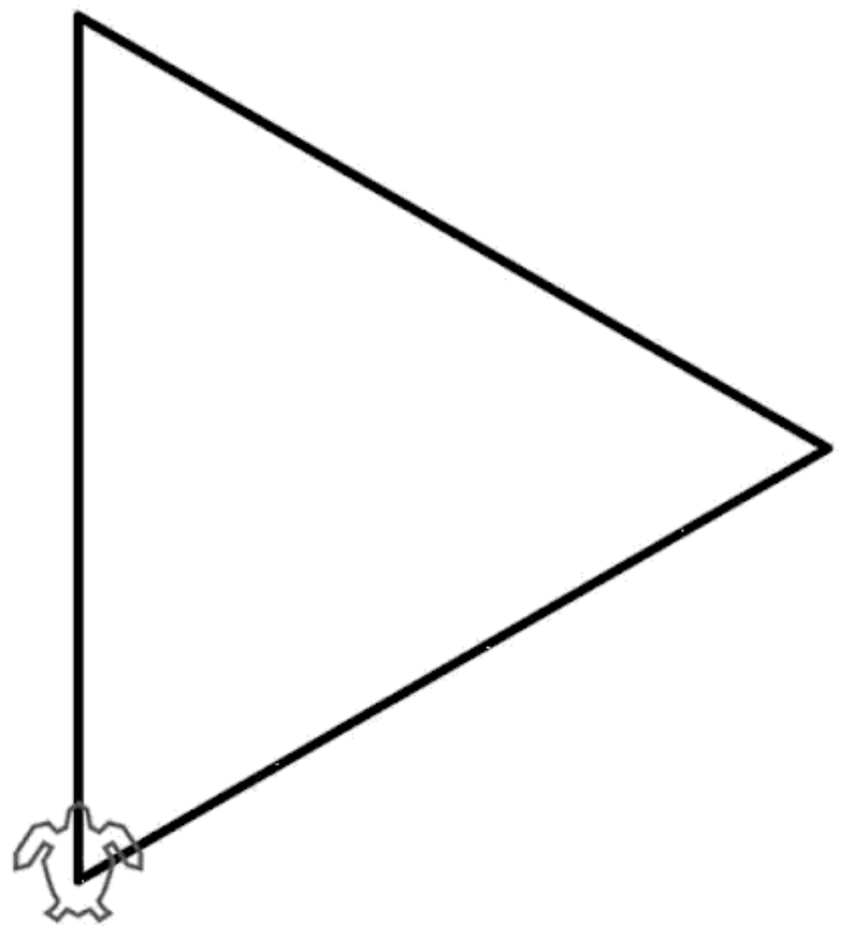
My first computer language was Logo, a graphical language developed in 1967 by Wally Feurzeig, Seymour Papert, and Cynthia Solomon and intended as an educational tool. I learned it at a computer class for kids at our local rec center in the suburbs of Los Angeles when I was seven. Armed with Logo, I could write instructions (in the form of a program) for a triangular "turtle" on the screen, which would then draw lines and shapes based on those instructions. The screen was monochrome, green text and lines on a black background.

The first "program" I wrote was a single line of code: drawing a square.



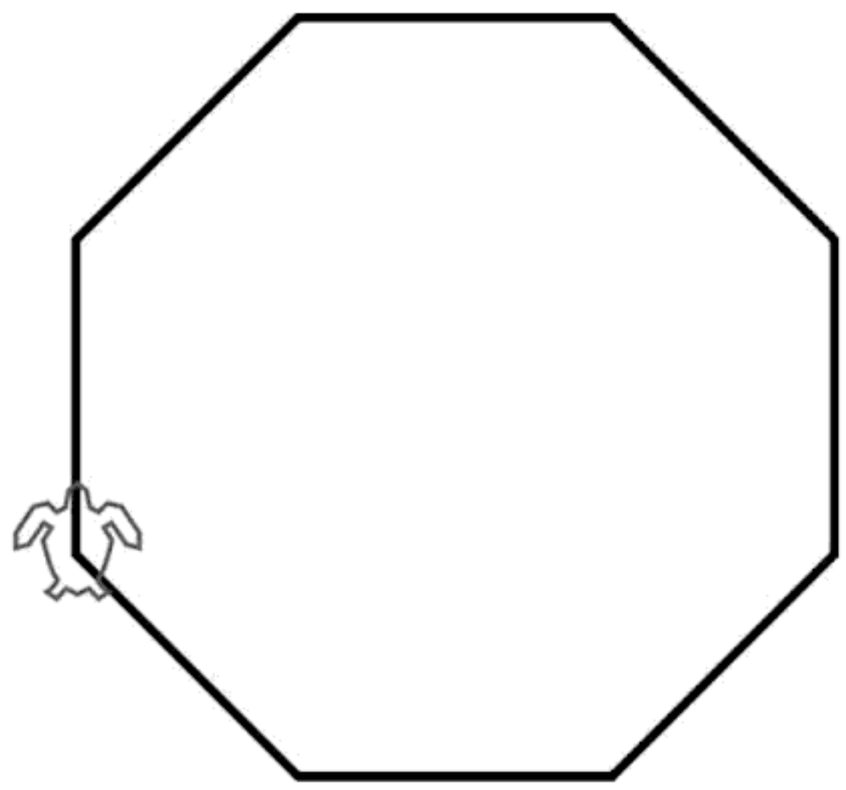
repeat 4 [forward 50 right 90]

That is, go forward 50 pixels, turn right by 90 degrees, and then repeat those two steps a total of four times. At the end of it, the turtle would be back where it started, having drawn out a square. By changing the angle and the number of repeats, I could draw a variety of polygons. A triangle:



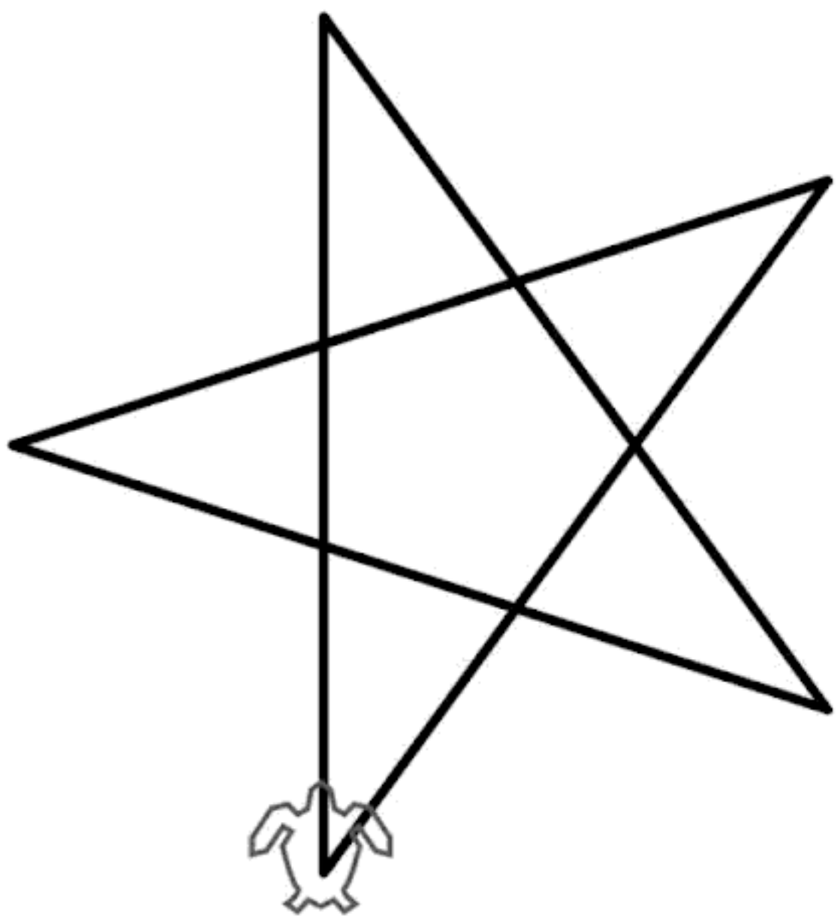
repeat 3 [forward 50 right 120]

An octagon:



repeat 8 [forward 50 right 45]

A pentagram:



repeat 5 [forward 50 right 144]

I could not draw a pentagram by hand, at least not well. The turtle drew it perfectly. The 144-degree angle felt like secret knowledge to me. I hadn't realized that the program did not need to be any more complex than that for a square or an octagon. Sometimes I boosted the number of repeats so that the turtle would continue to zip along the pentagram's lines like a bullet train.

These single-line programs are all algorithms. The word "algorithm" is a derivation of the name of ninth-century Persian mathematician Muhammad ibn Mūsā al-Khwārizmī. An algorithm is, informally speaking, the set of rules or instructions specifying the path from a specified problem ("Draw a pentagram with sides of length 50") to the solution to that problem (the visual display of the pentagram itself). Algorithms can become increasingly general, specified with variables rather than constants ("Draw a polygon with *n* sides of length *m*").

Algorithms hooked me. My own experience suggests that some people's brains are more tuned in to this way of thinking, just as some people are more attuned to mathematics or languages. I am not a visual or a verbal person: I was rejected from kindergarten because I couldn't draw. But these kinds of assemblages of instructions made intuitive sense, and I thought they were beautiful. Instead of just having the thing itself, I had the recipe

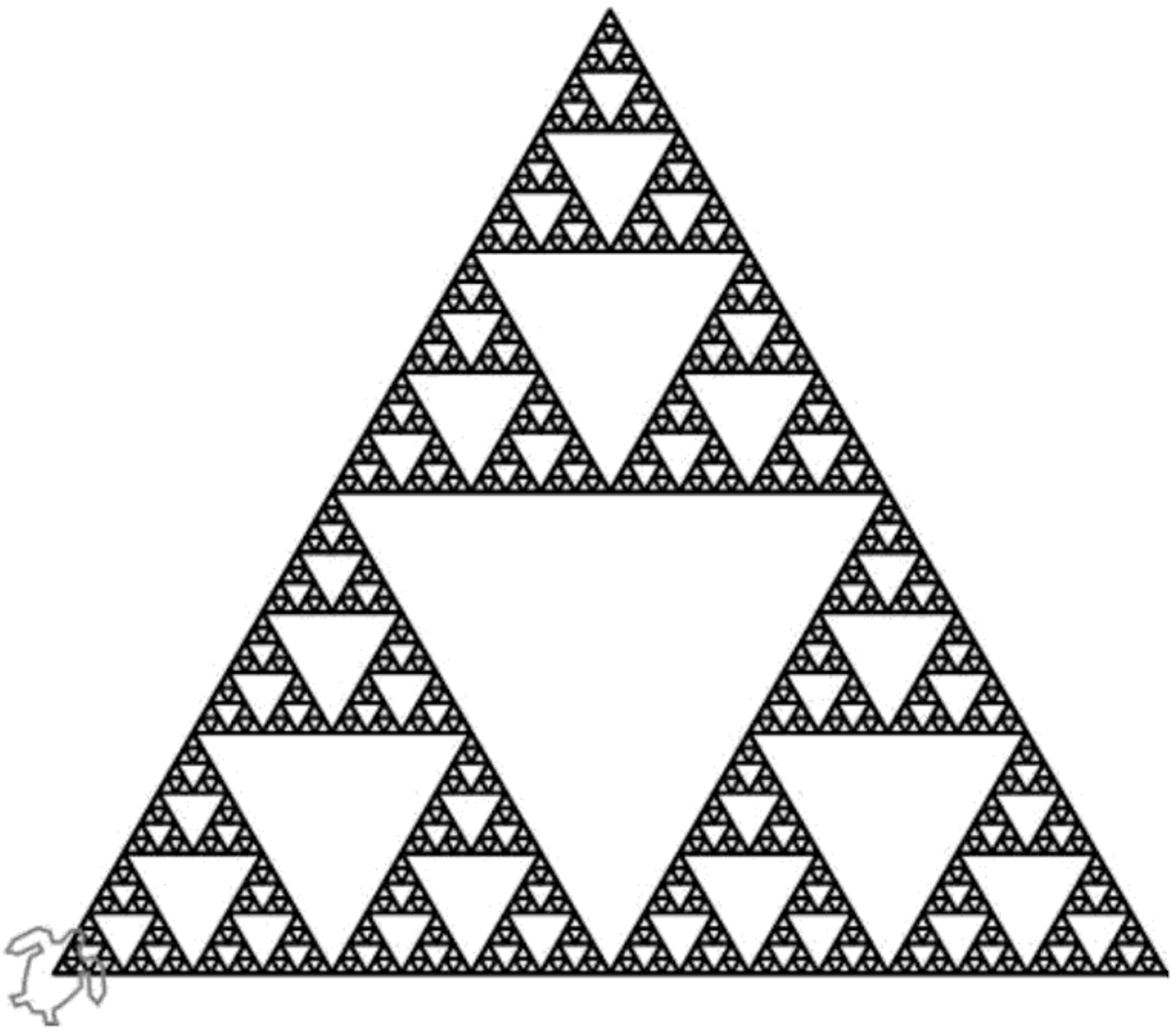
for the thing and, moreover, could make the recipe increasingly general so that reams of problems could be solved by twiddling the dials on a single recipe. That, in essence, is computer programming.

—

Simple algorithms can produce beautifully complex results. Here is a Logo program of half a dozen lines, **`sierpinskiTriangle`**, which draws a fractal triangle.

```
to sierpinskiTriangle :length :depth
   if :depth < 1 [ stop ]
   repeat 3 [
 sierpinskiTriangle :length/2 :depth-1
 forward :length
 right 120
   ]
end
```

Invoking the program with the command **`sierpinskiTriangle 500 7`** will cause the turtle to draw the following graphic:



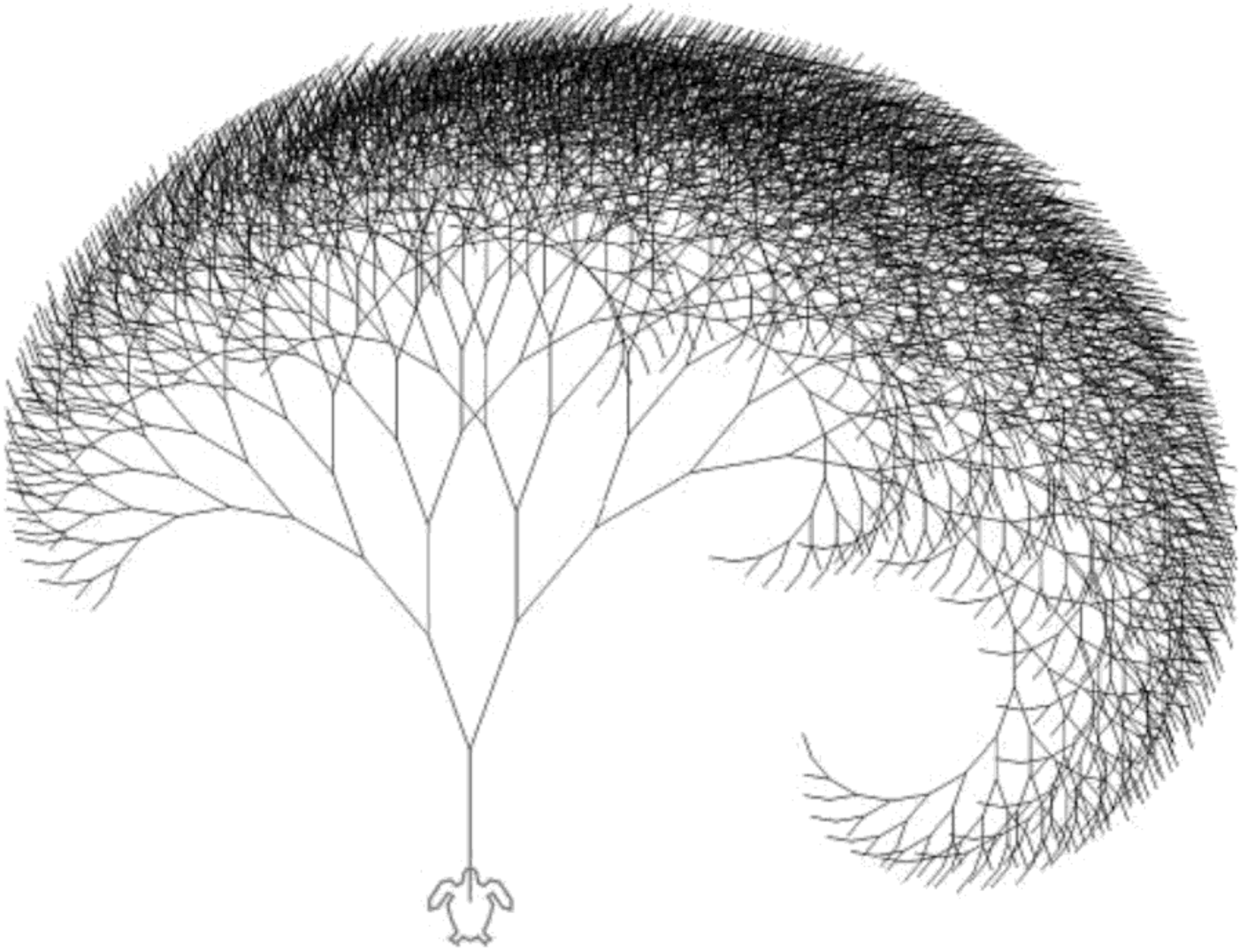
You can get this fractal pattern out of six lines of code because **`sierpinskiTriangle`** is doing one thing over and over again: drawing a triangle made out of three triangles. But every time it draws one of those triangles, it first draws three smaller triangles *inside* that triangle—in other words, it does the same thing, just smaller. So the code calls itself, in a process called *recursion*.

Here is another example of recursion, a program to draw a tree:

```
to tree :level :size :scale :angle
   if :level > 0 [
 fd :size
lt :angle
tree :level - 2 :size * :scale * :scale
:scale :angle
rt :angle
rt :angle
tree :level - 1 :size * :scale :scale
:angle
lt :angle
```

```
    bk :size
       ]
  end
```

Invoking this program with **`tree 18 100 .9 20`** produces this graphic:



This amazed me. It seemed impossible. How could a dozen lines of code produce such a beautiful and complex pattern? How had I instructed this computer to draw more capably and more beautifully than my hand could? I wanted to understand how such a great effect could stem from such a small set of instructions, and I wanted to author the programs that created such effects. My confusion led to my desire to understand. My wonder led to my desire to create.

Many people find their calling in a moment of sheer awe. The awe stems from not just the beauty and elegance but the sheer seeming *impossibility* of a past creation or discovery. For a writer, this could occur on reading a particular line of Shakespeare or Zhuangzi. For a mathematician, it may be found when studying

the proof of the irrationality of the square root of 2 or the supremely elegant unity of Euler's equation, which joins five fundamental mathematical constants through addition, multiplication, and exponentiation:

$$e^{i\pi} + 1 = 0$$

I was impressed and perplexed by this equation when I first saw it. The relation of the constants isn't obvious. I found it beautiful, yet it did not impel me to devour books of mathematics. I could appreciate the elegance of Euler's equation without wishing to dissolve my identity in the world of mathematics. Not so with the world of computers.

Plato believed that the core impulse to philosophizing lies in *aporia*, the point at which, in struggling to understand a phenomenon or answer a question, we come up against a seemingly irresolvable contradiction. The force of this contradiction can make us reassess the totality of what we thought we knew and reformulate it in a revolutionary way—for example, by saying, as Copernicus did in 1543, "Yet at rest in the middle of all things is the Sun."

Elsewhere, in the *Theaetetus,* Plato writes that philosophy begins in wonder (*thaumazein*), the awe-inspiring excitement that I felt on seeing the turtle-drawn tree. Aristotle, Plato's stolid successor, played down *aporia*. Perhaps this helped him to generate answers far more readily than Plato did. Aristotle produced systems of earthly and celestial motion, attempts at basic biology, and classifications of the various peoples and humors of the world.[*1] Plato's works, instead, tend to dwell more on how easily our minds are misled, and ask how we can be certain of anything. We care and work hard to understand and master a discipline through a combination of wonder and confusion. In his 1938 book *Experience and Prediction,* the philosopher of science Hans Reichenbach described the human condition as one not just of profound ignorance, but also illusion:

> We walk through the world as the spectator walks through a great factory: he does not see the details of machines and working operations, or the comprehensive connections between the different departments which determine the working processes on a large scale….We see the polished surface of our table as a smooth plane; but we know that it is a network of atoms with interstices much larger than the mass particles, and the microscope already shows not the atoms but the fact that the apparent smoothness is not better than the "smoothness" of the peel of a shriveled apple. We see the iron stove before us as a model of rigidity, solidity, immovability; but we know that its particles perform a violent dance, and that it resembles a swarm of dancing gnats more than the picture of solidity we attribute to it. We see the moon as a silvery disk in the celestial vault, but we know it is an enormous ball suspended in open space. We hear the voice coming from the mouth of a singing girl as a soft and continuous tone, but we know that this sound is composed of hundreds of impacts a second bombarding our ears like a machine gun….We do not see the things, not even the concreta, as they are but in a distorted form; we see a substitute world—not the world as it is, objectively speaking.

This "substitute world" that we see is, in short, a lie. Our brains take sense data and inaccurately analogize it into forms that are already familiar to us. But as children growing up, this substitute world works quite well. It is manageable and legible to us, since we engage with the world in a functional and effective fashion. The world as it is only grants flashes of strangeness to a child to suggest that reality might be quite different. What really goes on inside our bodies? How did this world come to be? What is death? These questions don't often present themselves, because we know to be productive with our time rather than diving into

what David Hume called the "deepest darkness" of paralyzing skepticism. Yet much joy and satisfaction can be found in chasing after the secrets and puzzles of the world. I felt that joy first with computers. In them I found a world strictly divided between the program and the output. The instructions and the execution. The simple code and the beautiful tree.

I remain a terrible artist, barely able to draw a human figure. But I fell in love with the concepts of algorithmic programming: instructions, branches, variables, functions. I saw how a program could generate the simulated world of the output. Recursion was too tricky for my seven-year-old self to wrap my head around, but I wanted badly to understand it, and I was convinced that I could. My wonder at the power of programming, the ability to create merely through simple lines of text and numbers, drove me.

The complexity of life is all around us, but we grow numb to what we see of it, even while so much lies outside our immediate experience: microworlds of cells, atoms, particles, as well as the macrocosmos of our universe containing far more galaxies than the Earth has people (approximately two trillion galaxies by NASA's 2016 estimate). Programming abstracted away the uncertainty of the world and laid its principles out before me. Notions of elegance and beauty drive programmers just as much as they do mathematicians and poets. What mattered is that I *felt* the jump from the programmatically simple to the aesthetically complex.

On a computer, that jump is clean, elegant, and definitive. One popular philosophical fable concerns the myth that the Earth is supported on the shell of a gigantic turtle.[*2] "What is supporting the turtle?" asks the philosopher. It's "turtles all the way down," comes the reply. There is no final answer available to us, only more questions. Programming offered a stopping point with its artificial world, a final answer. In Logo, there was just the turtle, just the one.

The first turtle I worked with was a simple triangle, not the

waddling shape you see in the pictures above. Later, the program LogoWriter made an appearance at my school. It was a frillier version of Logo, which replaced the triangle with a turtle shape closer to what I've used here. I disliked the turtle-shaped turtle (and I still do). LogoWriter added bells and whistles, but the representation of the turtle *as a turtle* had no functional impact whatsoever on the workings of Logo. It was a superfluous cosmetic change that drew attention away from what was truly remarkable about Logo: the relationship between the program and its execution. The turtle, whether triangle-shaped or turtle-shaped, was already abstracted away in my mind, just a point to designate where drawing would next originate.

Even as I coded on Logo Writer, that tree still puzzled me. I could not understand the concept behind recursion, the powerful technique that allowed the `tree` program to draw such a complicated pattern with so few lines of code. I wouldn't figure it out until my teens, when I would also learn what a powerful role it played in all of computer science and indeed in conceptual thinking in general. Recursion, in a nutshell, is use of a single piece of code to tackle a problem by breaking it down into subproblems *of the same form*—like drawing a branch of a tree that is itself a smaller tree. It is envisioning the world as an ornate yet fundamentally elegant fractal. Recursion reflects the efficient, parsimonious instinct of computer programming, which is to get a lot out of a little.

The Assembly

Why would you want more than machine language?

—JOHN VON NEUMANN

Programming isn't a wholly abstract exercise. Programming requires hardware, which only became available to the average

home in the 1980s, and it had its own subculture too. In the pre-internet days, there was a secret lore surrounding computers, and much of it revolved around the Apple II.

My first computer was an Apple IIe. It was, by far, the most popular home computer of its age, thanks not only to Apple's partnerships with educators but also to Apple's focus on making a *general-purpose* computer for consumers and hobbyists. Other consumer-oriented computers, like the Commodore 64 or the Atari 400, were dedicated simply to running the primitive software of the time. Apple's computers used Chuck Peddle's ubiquitous (and *cheap*) 6502 processor and sat somewhere in between those casual machines and professional PCs like IBMs. This owes primarily to Apple creator Steve Wozniak's background in the hobbyist and computer club community and his dedication to building a computer that could be both accessible and powerful.[*3] Wozniak was not trained in academia or research labs. He came out of vaguely countercultural groups who got into PCs with the same fervency that others get into coin collecting, cars, or Dungeons & Dragons.[*4]

There was a totality to the Apple IIe that no longer exists on computers today, or even mobile devices. It offered the sense of being *close* to the fundamental machinery of the system. The Apple IIe did not have a hard drive. Turn it on without a floppy in the drive and you'd just see "Apple][" frozen at the top of the monitor. I had to boot a floppy disk containing Apple DOS, the disk operating system, where I could program in Applesoft BASIC, as did many others around that time.

I remember the first programs I tinkered with on the Apple IIe. There was *Lemonade Stand,* a multiplayer accounting game originally created by Bob Jamison back in 1973, then ported to the Apple IIe in 1979 by Charlie Kellner. Playing it, I set prices and budgeted for advertising, depending on the weather. If prices were too high, people wouldn't buy your lemonade. If prices were too low, you wouldn't make a profit. After a few days of play, your mother stops giving you free sugar; a few days after that,

the price of lemonade mix goes up. If it rained, everything was destroyed for that day and you took a total loss. If construction crews were present on the street, they would pay *any* price for your lemonade. I changed the code so *everyone* would pay whatever price you set. I made a killing because I could change the rules. Then I changed the code so that for the second player only, people would never buy lemonade at any price, and I asked my mother to play it against me. I won. She was baffled, then simultaneously impressed and annoyed (a reaction that is every child's dream) when I told her I'd changed the code.

```
$$ LEMONSVILLE DAILY FINANCIAL REPORT $$

   DAY 5                          STAND 1

  47   GLASSES SOLD
$.15   PER GLASS                INCOME $7.05

  50   GLASSES MADE
  3    SIGNS MADE              EXPENSES $2.45

                  PROFIT   $4.60
                  ASSETS   $10.20

PRESS SPACE TO CONTINUE, ESC TO END...
```

Profiting from a heavy markup in *Lemonade Stand*

BASIC was a less elegant language next to Logo. But it was *native* to the Apple IIe. With a few mysterious commands named PEEK, POKE, and CALL, I could tinker directly with the guts of the Apple IIe. These commands let you access the physical Random Access Memory (RAM) of the machine, the immediate, transient short-term storage of the computer. **`PEEK(49200)`** would make the speaker *click*, a thrilling sound when the single loud beep was the only sound easily available to a BASIC programmer. **`POKE(49384,0)`** would start the disk drive

motor spinning, good for scaring someone into thinking their disk was being formatted. Other PEEKs and POKEs allowed for manipulation of text to make characters disappear and reappear and move around—things that weren't easy to do in BASIC proper. You could also crash and reboot your machine, which was otherwise nearly impossible in BASIC. POKEs and CALLs were powerful stuff. PEEK was (mostly) safe.

Peeks, Pokes and Pointers

Apple® Zero-Page

DECIMAL — HEX

32 **Text Window Left-Edge** (0-39 / normal is 0) $20
Example: **POKE 32, X** freezes the left X columns of text.
Warning: Don't let PEEK(32)+PEEK(33) exceed the screen width.

33 **Text Window Width** (1-40 or 1-80 / normal is 40 or 80) $21
Note: **POKE 33,33** scrunches listings to remove extra spaces.

34 **Text Window Top-Edge** (0-22 / normal is 0) $22

35 **Text Window Bottom** (1-24 / normal is 24) $23

36 **Horizontal Cursor-Position** (0-39) $24
Examples: If **PEEK(36)=X**, then the cursor is in column X+1.
POKE 36,X puts the cursor in column X+1 (useful with 80-columns for positioning the cursor beyond the 40-column limit of HTAB).
Note: **POKE 1403,X** works similarly—and more predictably.

37 **Vertical Cursor-Position** (0-23) $25
Examples: If **PEEK(37)=Y**, then the cursor is on text line Y+1.

43 **Boot Slot *16** (after boot) $2B

44 **Lo-Res Line End-Point** $2C

48 **Lo-Res COLOR *17** $30

50 **Text Output Format** $32
POKE 50, 63=INVERSE, **POKE 50, 255**=NORMAL,
POKE 50, 127=FLASH (for ASCII 64-95).

51 **Prompt-Character** $33
Note: **POKE 51,8: GOTO line#** will prevent a false "Not Direct Command" message caused by an immediate **GOTO line#** command.

76-79 **Random-Number Field** $4E.4F

103-104 **Start of Applesoft Program** $67.68
To Load a program at a non-standard location LOC—
POKE LOC-1, 0: POKE 103, LOC-INT(LOC/256): POKE 104, INT(LOC/256) Then LOAD PROGRAM.
Note: **FP** (DOS 3.3 only) sets start-of-program to normal 2049 ($801).

105-106 **LOMEM:** $69.6A
Note: LOMEM is the Start of Variable-Space, equivalent to End-of-Program (approx.) unless changed with the **LOMEM:** command.

107-108 **Start of Array-Space** $6B.6C

109-110 **End of Array-Space** $6D.6E

111-112 **Start of String-Storage** $6F.70

115-116 **HIMEM** $73.74
Note: HIMEM-1 is the highest address available for use by an Applesoft program. May be changed with the **HIMEM:** command.

117-118 **Line-Number Being Executed** $75.76

119-120 **Line-No. Where Program Stopped** $77.78

121-122 **Address of Line Executing** $79.7A

123-124 **Current DATA Line-Number** $7B.7C

125-126 **Next DATA Address** $7D.7E

127-128 **INPUT or DATA Address** $7F.80

129-130 **Last-Used Variable Name** $81.82

131-132 **Last-Used-Variable Address** $83.84

175-176 **End of Applesoft Program** $AF.B0

214 **RUN Flag** $D6
Example: **POKE 214, 255** makes any command RUN a program.

216 **ONERR Flag** $D8
Example: **POKE 216, 0** cancels the ONERR function.

218-219 **Line-Number of ONERR Error** ... $DA.DB

220-221 **ONERR Error Address** ..... $DC.DD

222 **ONERR Error Code** ..... $DE

DOS 3.3 and PRODOS	APPLESOFT
1: Language Not Available[1]	0: ?Next Without For
2 or 9: Range Error	16: ?Syntax Error (FP)
3: No Device Connected[1]	22: ?Return Without Gosub
4: Write-Protected	42: ?Out of Data
5: End of Data	53: ?Illegal Quantity
6: File[1] or Path[2] Not Found	69: ?Overflow
7: Volume Mismatch[1]	77: ?Out of Memory
8: I/O Error	90: ?Undef'd Statement
9: Disk Full	107: ?Bad Subscript
10: File Locked	120: ?Redim'd Array
11: Syntax Error[1] or Invalid Option[2]	133: ?Division by Zero
12: No Buffers Available	163: ?Type Mismatch
13: File Type Mismatch	176: ?String Too Long
14: Program Too Large	191: ?Formula Too Complex
15: Not Direct Command	224: ?Undef'd Function
17: Directory Full[1]	254: ?Re-Enter
18: File Not Open[1]	255: (control-C interrupt)
19: Duplicate File Name[1]	
20: File Busy[1]	[1]DOS 3.3 only
21: File(s) Still Open[1]	[2]ProDOS only

224-225 **X of Last HPLOT** (0-279) $E0.E1

226 **Y of Last HPLOT** (0-191) $E2

228 **HCOLOR Code** $E4
0=0, 42=1, 85=2, 127=3, 128=4, 170=5, 213=6, 255=7

230 **Hi-Res Plotting Page** $E6
POKE 230,32 selects Page 1. **POKE 230,96** selects Page 3.
POKE 230,64 selects Page 2.

231 **SCALE** $E7
Note: **SCALE=0** is equivalent to a SCALE of 256.

232-233 **Shape Table Start Address** $E8.E9

234 **Hi-Res Collision-Check** ..... $EA
Example: XDRAW a shape. If **PEEK(234)=0** then the shape started at a non-black hi-res point.

241 **SPEED** $F1
Note: **PEEK(241)** is 255 minus the current SPEED.

243 **FLASH Mask** $F3

249 **ROT** $F9

Display Switches

DECIMAL (with negative equivalent) — HEX

49232 (-16304) **Graphics** $C050
49233 (-16303) **Text** $C051
49234 (-16302) **Full-Graphics** $C052
49235 (-16301) **Split-Screen** $C053
49236 (-16300) **Page One** $C054
49237 (-16299) **Page Two** $C055
49238 (-16298) **Lo-Res** $C056
49239 (-16297) **Hi-Res** $C057
Note: Activate display switches by Poking each location.
Example: **POKE 49232,0** switches to Graphics display.

Keyboard, etc.

DECIMAL (with negative equivalent) — HEX

49152 (-16384) **Read Keyboard** ... $C000
49168 (-16368) **Clear Keyboard** ... $C010
Example: 10 KEY=**PEEK(49152)**: IF KEY<128 THEN 10
20 **POKE 49168, 0**
30 PRINT "KEY "; CHR$(KEY-128)

49200 (-16336) **Click Speaker** $C030
Example: FOR A=1 TO 99: BUZZ=**PEEK(49200)**: NEXT

49249 (-16287) **Button #0** $C061
Paddle-0 Button (or Open (left) Apple key.*

49250 (-16286) **Button #1** $C062
Paddle-1 Button (or Closed (right) Apple key.*

49251 (-16285) **Button #2** $C063
*Example: If **PEEK(49249+P)** is greater than 127, then Paddle Button #P is being pressed—or it's not connected.

DOS 3.3 Pokes

(assume DOS loaded in main memory)

POKE 40193, PEEK(40193)–N: CALL 42964
Moves DOS buffers down N*256 bytes.

POKE 44452,N+1: POKE 44605,N
Allows N file names before catalog pause.

POKE 44460,88: POKE 44461,252
Clears screen before catalog.

POKE 44505,234: POKE 44506,234
Exposes deleted file names in catalog.

POKE 44596, 234: POKE 44597, 234: POKE 44598, 234 Cancels catalog pause.

POKE 49197,234: POKE 49198,234: POKE 49199, 234 Prevents language card reload.

POKE 49384,0 Stops drive motor.

POKE 49385,0 Starts drive motor.

Notes

Apple's main memory consists of 65,536 bytes, numbered zero to 65535. Every byte has a value in the range 0-255.

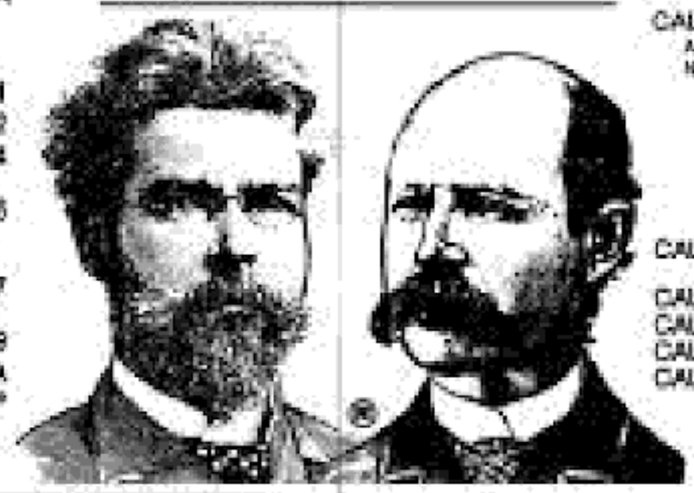
- You may Peek (look at) the value in byte number-B with the command— **PRINT PEEK(B)**
- You can usually Poke a new value-V into byte-B with the command— **POKE B,V**

Values higher than 255 must be stored in two bytes:

- To look at the value in consecutive bytes B1-B2— **PRINT PEEK(B1)+PEEK(B2)*256**
- To Poke a new value V (0-65535) into bytes B1-B2— **POKE B1, V-INT(V/256)*256** and **POKE B2, INT(V/256)**

Note: Since almost any memory location can be Peeked or Poked, program listings can reveal thousands of Peeks and Pokes not listed on this chart. Pokes are often used to write machine-language routines that may be activated with the CALL command—the possibilities are infinite.

Let A=PEEK(64435) and B=PEEK(64448).
If A=6 and B=0 then Apple IIc.
If A=6 and (B>223 AND B<240) then Apple IIe.
If A<>6 then Apple II or II+.



Page-3 DOS Vectors

DECIMAL — HEX

976-978 **Re-enter-DOS Vector** $3D0.3D2

1010-1012 **Reset Vector** $3F2.3F4
Example: POKE 1012,0 makes Reset boot.
(POKE 1012,56 to restore normal Reset function.)

1013-1015 **Ampersand Vector** $3F5.3F7
Examples: POKE 1013, 185: POKE 1014, 214 makes "&" LIST.
POKE 1014, 170: POKE 1015, 165 makes "&" CATALOG.
POKE 1014, 18: POKE 1015, 217 makes "&" RUN.

1016-1018 **Control-Y Vector** $3F8.3FA

DOS 3.3 Locations

DECIMAL — HEX

(All values assume DOS is loaded in main memory.)

42350 **Catalog-Routine** $A58E
Example: CALL 42350 catalogs a disk.

40514 **Greeting Program Run-Flag** $9E42
POKE 40514,52 and (NOT a disk. When booted, DOS will attempt to BRUN the greeting program. POKE 40514,20 for $9E4C.

43140-43271 **Commands** $A884.A907

43378-43582 **Error Messages** $A972.AA3E

43616-43617 **Last Bload Length** $AA60.AA61

43634-43635 **Last Bload Start** $AA72.AA73

43624 **Drive-Number** $AA68
Example: **POKE 43624, 0** changes disk input/output to Drive 0.

43626 **Slot-Number** $AA6A
Example: **POKE 43626, 5** changes disk input/output to Slot 5.

43660 **Control-D Command Character** $AA8C

44033 **Catalog Track Number** $AC01

45991-45998 **File-Type Codes** $B3A7.B3AE

45999-46010 **Disk Volume Heading** $B3AF.B3BA

46017 **Disk Volume Number** $B0C1

ProDOS™ Locations

DECIMAL — HEX

46944 **Slot/Drive Value** $BF30
If **PEEK(46944)** is greater than 127 then Drive 2, otherwise Drive 1.

47313-47422 **Commands** $B8D1.B93E

46840-46841 **Last Bload Length** $B6C8.B6C9

46926-46926 **Last Bload Start** $B6B9.B6BA

Useful Calls

DECIMAL (add 65536 for positive equivalent) — HEX

CALL-22153 **Reconnect DOS 3.3** $99D9F
CALL-3100 **Reveal hi-res page 1** $F3E4
CALL-3088 **Clear hi-res screen to black** $F3F2
CALL-3082 **Clear hi-res to last color Hplotted** $F3F6
Example: HGR2: HCOLOR=5: HPLOT 0,0: **CALL-3082**
CALL-2613 **Hi-res coordinates to Zero-Page** $F5CB
Example: The X and Y starting coordinates of the next shape table DRAW or XDRAW may be determined with a **CALL-2613**. Then X=PEEK(224)+PEEK(225)*256 and Y=PEEK(226).
CALL-1439 **Pseudo-Reset** $FA82
CALL-1370 **Boot** $FAA6
CALL-1321 **Display all registers** $FAD7
CALL-1184 **Clear screen and print "Apple ..."** $FB90
CALL-1036 **Move cursor right** $FBF4
CALL-1008 **Move cursor left** $FC10
CALL-998 **Move cursor up** $FC1A
CALL-958 **Clear text from cursor to bottom** $FC42
CALL-922 **Move cursor down** $FC96
CALL-868 **Clear text-line from cursor to right** ... $FC9C
CALL-756 **Wait for any keypress** $FD0C
CALL-678 **Wait for a Return keypress** $FD6A
CALL-657 **Better input: commas/colons o.k.** $FD6F
10 PRINT "NAME (LAST, FIRST) ";: **CALL -657**
20 A$="": FOR X=512 TO 767: IF PEEK(X)<>141 THEN A$=A$+CHR$(PEEK(X)-128): NEXT X
CALL-455 **Memory move** $FE2C
A Basic memory move: C6 & C8 are the Old-location Start & End, and 42 is the New Start. GOSUB 5000 to execute the move—
5000 N=C6: LOC=60: GOSUB 5020
N=C8: LOC=62: GOSUB 5020
N=42: LOC=66: GOSUB 5020
5010 POKE 768, 169: POKE 769, 0: POKE 770, 76:
POKE 771, 44: POKE 772, 254: CALL 768: RETURN
5020 POKE LOC, N-INT(N/256)*256
POKE LOC+1, INT(N/256): RETURN
CALL-415 **Disassembler** $FE81
Note: Poke start address at locations 58-59 before Call.
CALL-211 **Ring bell and print "ERR"** $FF2D
CALL-198 **Ring bell** $FF3A
CALL-151 **Enter monitor** $FF69
CALL-144 **Scan input buffer** $FF70
This example uses CALL -144 to execute a machine language routine from Basic (will not work in a quote pulmo):
100 A$="3083: A9 C2 20 ED FD 18 69 01 C8 D0 D0 F8 60 3000:D8"
110 FOR X=1 TO LEN(A$): POKE 511+X, ASC(MID$(A$,X,1))+128: NEXT
120 POKE 72,0: CALL -144

6/84

Beagle Bros Micro Software Inc.

Beagle Bros makes useful and entertaining Utilities, Games and Publications for all versions of Apple II® Computers.

"APPLE" is a Registered Trade Mark of Apple Computer, Inc.

These numbers were arbitrary to outsiders, and they were not

particularly publicized. Before the internet, programmers had to learn this kind of esoteric knowledge haphazardly from books, magazines, and other enthusiasts. There was a thrill of discovery that can't be re-created now that most information can be found with a simple web search.[*5] I would find a particular piece of Apple lore, then think about it until the next time I got on the computer to try it out. I discovered much, as many did then, through the charts produced by Bert Kersey's Beagle Bros software company. There is a certain set of people, myself included, for whom this chart will inspire an overpowering nostalgia.

Part of this nostalgia owes to Kersey's signature design and clip art, which distinguished Beagle Bros from other vendors. Part of it owes to the sheer intrigue around the secret details contained on the chart: this was hidden knowledge! Kersey captured the mystique:

> Pokes are often used to write machine-language routines that may be activated with the CALL command—the possibilities are infinite.

Even novelties were fascinating because they revealed unsuspected capabilities of the Apple IIe. The program, which appeared in a Beagle Bros catalogue, was pretty much impossible to parse if read.

```
1 HOME: LIST: BUZZ=49200
2 A$="!/-"+CHR$(92): FOR A=1 TO 48:
B=PEEK(BUZZ):FOR C=1 TO A: NEXT:
X$=MID$ (A$,A-INT(A/4)*4+1,1): VTAB 3:
HTAB 10: PRINT X$X$X$: NEXT: GOTO 2
```

If typed in and executed, it would print itself and then make a varispeed buzz as the characters in the first line appeared to spin around in time with the buzzing. Who would think of such a thing? My nostalgia for this ephemera also owes to the tangibility

into a language called 6502 assembly for Apple IIe CPUs. The clock speed of a processor, given in cycles per second, or hertz, dictates just how fast a CPU chip could execute individual assembly instructions.[*7] Vastly more daunting than BASIC, I didn't dare touch 6502 assembly as a kid. Assembly language grants access to the physical memory of the computer and allows one to specify numerical operation codes (opcodes) that are actually understood by the hardware in the CPU. In assembly, there is almost no distance between the programmer and the hardware.

Here's some assembly for a "Hello world!" program (one that just displays "Hello world!" and exits) in Apple II 6502 assembly:

```
COUT   gequ    $FDED           ;The Apple II character output func.

       keep    HelloWorld

main   start
       ldx     #0              ;Offset to the first character
loop   lda     msg,x           ;Get the next character
       cmp     #0              ;End of the string?
       beq     done            ;->Yes!
       jsr     COUT            ;Print it out
       inx                     ;Move on to the next character
       jmp     loop            ;And continue printing
done   rts                     ;All finished!
msg    dc      c'Hello world.'
       dc      h'0D'
       dc      h'00'
       end
```

And here it is in C:

```
int main() {
    printf("Hello world!\n");
    return 0;
}
```

And here it is in Applesoft BASIC:

```
10 PRINT "HELLO WORLD!"
```

In the eighties, many programmers coded directly in assembly. Programs were simpler and performance was critical. But as computers got larger and more complex, it became unfeasible to code in assembly.[*8] Programmers need to learn a different assembly language for different processors (as with the Apple II's 6502, the Macintosh's 68000, and the PC's 8086), which is horrendously inefficient. More efficient was to use a CPU-independent higher-level language. All the languages we hear about today, from C++ to Java to Ruby to Python, are higher-level languages. A compiler takes the code written in these languages and translates it into the assembly code for a particular processor.

Until I learned assembly in college, and how language compiler programs translated higher-level programming languages into assembly, computers remained partly opaque to me. That gap in my knowledge bothered me, because even though I had far more direct control over those lower layers, I couldn't understand them. When I took a compilers class in college, the infrastructure of the computer opened up to me. There was no longer a miracle in between my code and its execution. I could see the whole picture, finally, and it was beautiful.

The Split

I renounce any systematic approach and the demand for exact proof. I will only say what I think, and make clear why I think it. I comfort myself with the thought that even significant works of science were born of similar distress.

I want to develop an image of the world, the real background, in order to be able to unfold my unreality before

it.

—ROBERT MUSIL

When I was a teenager, programming lost its allure. The "real world," such as it was, had drawn my attention away from what now looked to be the sterile, hermetic world of computers. It was the late eighties. The web did not exist in any accessible form, nor were computers part of most people's daily lives. I was part of the very last generation to grow up in such a world. People only a few years younger than me would have the nascent public internet and the web to dig into and explore. I had online bulletin board systems (BBSs) and such, but they were strictly cordoned off from my everyday existence, the exclusive preserve of hobbyists, eccentrics, and freaks. And I was miserable in my small suburban enclave. For many programmers, computers held the answer to such misery. They continue to provide the mesmeric escape from the dreary everyday routines of teenage and adult years. I don't have a clear explanation as to why computers failed to offer me solace as they did for many others. Something kept me from locking in completely to the brain-screen bond that kept many teen programmers up all night coding games or hacking copy protection. Literature became my refuge instead.

My parents had raised me on science fiction, the standard literary junk food of computer geeks, but I felt increasingly drawn to explorations of human emotion and existential crisis. At a point of typical thirteen-year-old despair, I devoured the complete works of Kurt Vonnegut over the course of two weeks. They touched me. Vonnegut led me to explore increasingly "deep" fiction.[*9]

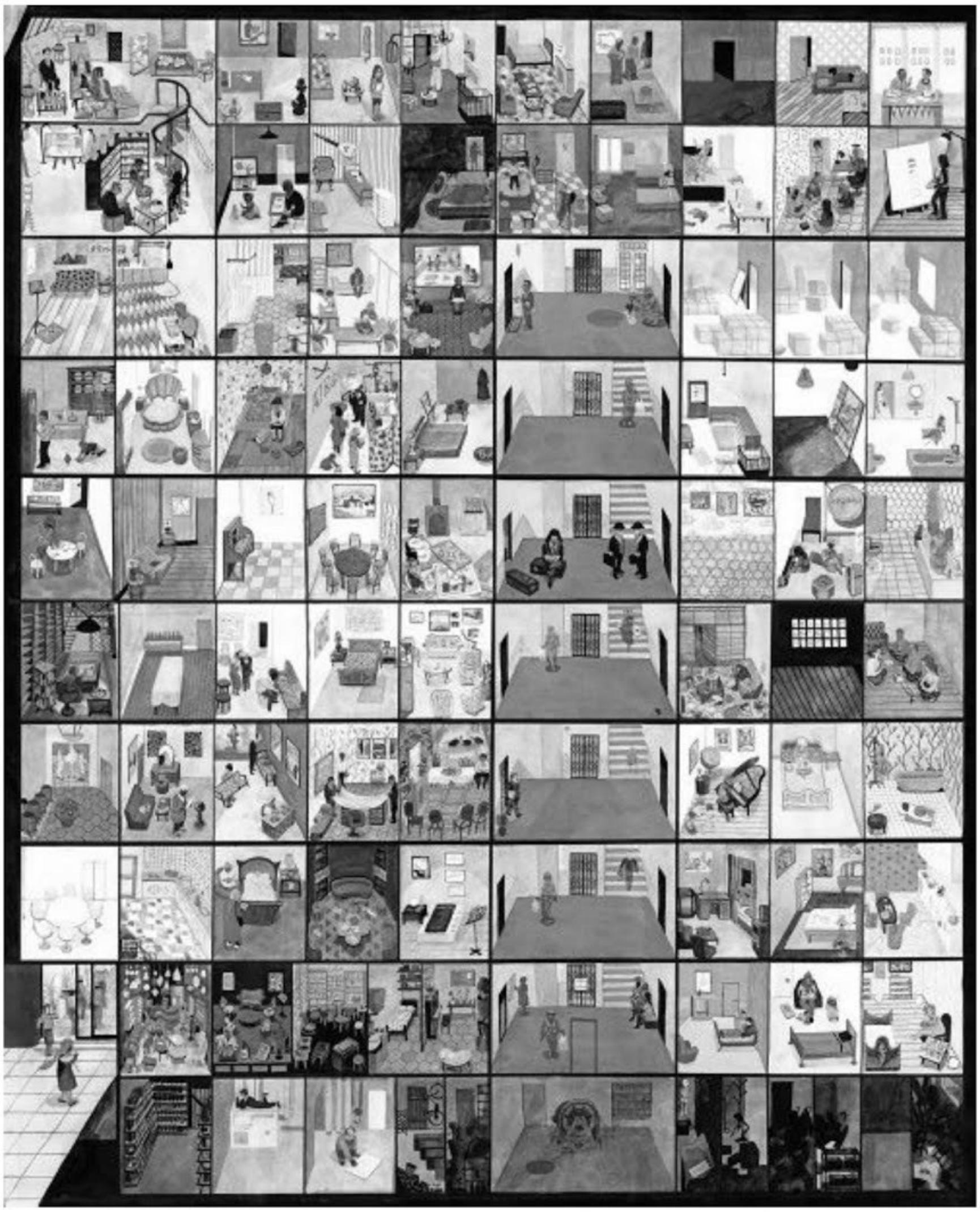
My high school physics teacher introduced my class to James Joyce, whom he considered the greatest author of the twentieth century. He told us that *Ulysses* was dauntingly complex and that *Finnegans Wake* was simply incomprehensible. The difficulty and obscurity of *Ulysses* intrigued me as a teenager much as that

Logo tree program had as a child. How could a book of fiction be "difficult"? Did it too hold a kind of programmatic complexity to it?

I had stumbled on the writers of the Oulipo, the French-dominated group specializing in experimental works of "potential literature," after Martin Gardner, amateur mathematical enthusiast, had published several articles on the group in his column in *Scientific American.*[*10] Their most famous members—Raymond Queneau, Georges Perec, Italo Calvino, and Harry Mathews—specialized in the innovation of literature through the use of formal constraints. One of the most infamous was Perec's novel *La disparition* (*A Void* in Gilbert Adair's English translation), which contains not a single *e* in its three hundred pages. That kind of constraint—leaving out a letter or set of letters—is called a lipogram. The poet Jean Lescure's "S+7" method replaces every noun in a text with the seventh noun following it in the dictionary. "Lend me your ears" becomes "Lend me your easels." Raymond Queneau's *One Hundred Thousand Billion Poems* is a set of ten sonnets with the exact same rhyme scheme and rhyme sounds. By mixing and matching lines, there are ten possible first lines, ten possible second lines, and so on, resulting in 10^{14} possible sonnets.

Georges Perec's mighty *Life: A User's Manual* describes the inhabitants of the ninety-nine rooms of a 10x10 apartment building (one corner is missing), where each successive room is a chess knight's move away from the current room, and each room is visited only once.[*11] The intricate structuring, I later discovered, had close ties to computer science. Perec was fascinated by a mathematical technique called the Graeco-Latin square, a device that pairs up two sets of elements so that each pair occurs only once. The pairs are distributed in a square so that each element also occurs only once in every row and column. Perec used (and abused) Graeco-Latin squares to structure his works: the original plan for *Life: A User's Manual* was to utilize over twenty squares to determine what objects, characters, times,

furniture, clothing, and music to place into each chapter. Finding these squares was a devilish task: in the eighteenth century, Euler thought no 10x10 Graeco-Latin squares existed, and one was only found in 1959, with the assistance of a computer.



Brecht Evens's rendition of the 10x10 apartment building in Georges Perec's *Life: A User's Manual*.

Perec, unable to construct squares of sufficient sizes himself, wrote to Indra Chakravarti, one of the authors of the 1960 paper "On Methods of Constructing Sets of Mutually Orthogonal Latin Squares Using a Computer," who provided Perec with two 12x12 squares. One of Chakravarti's coauthors was computer scientist Donald Knuth, who would go on to write the monumental bible

inaccuracy and error.

These kinds of opposing tendencies have been noticed by many. Neither side is specific to science or the humanities. Both the analytic and the heuristic exist within any domain of study, whether literature, logic, or sport. In 1905, mathematician, astronomer, and writer Henri Poincaré distinguished two types of mathematicians:

> The one sort are above all preoccupied with logic; to read their works, one is tempted to believe they have advanced only step by step, after the manner of a Vauban who pushes on his trenches against the place besieged, leaving nothing to chance. The other sort are guided by intuition and at the first stroke make quick but sometimes precarious conquests, like bold cavalrymen of the advance guard….Logic, which alone can give certainty, is the instrument of demonstration; intuition is the instrument of invention.

Intuition, Poincaré says, is both necessary and fallible. I call it "heuristic" because it is the often-unconscious art of selecting which facts are relevant, which phenomena are linked, and which shortcuts to take. Seventy years later, mathematician Mark Kac spoke of two species of genius, scientific and otherwise: the "ordinary" and the "magician." The difference, Kac says, is that even after we understand what a magician like Richard Feynman has done, we still have no idea how they got there.

Heuristics may seem like inferior mental shortcuts compared to the exactness of an algorithm, but in the 1950s, Nobel economic laureate and polymath Herbert Simon promoted heuristics as a necessary tool for coping with a world too complex to understand analytically. For Simon, heuristics were a necessary mechanism for dealing with the limitations we face in solving any problem: limitations of time, of knowledge, of brainpower. Psychologist Gerd Gigerenzer goes further,

emphasizing that a failed complex analysis often generates *worse* results than a simple, heuristic decision. We bring our biases to problems not because we are flawed but because we would be utterly lost without them: "Without bias, a mind could not function well in our uncertain world."

Heuristics are indeed necessary for human functioning, but we must be cautious in how we apply them and their biases. As we'll see later, when heuristics are carelessly translated to computers, trouble follows.

Bits and pieces inevitably slip through the cracks of heuristics. Those lost fragments, too complex to be captured by formal analysis or heuristic shorthand, fascinated me as much as the formal systems. When Oulipian writers wove formal abstractions into human joy and grief, as Jacques Roubaud did in *The Great Fire of London* and Perec did in *W, or The Memory of Childhood,* they brought out the gaps between those abstractions and the irreducible complexity of reality. The formal and the analytic, in their hands, became a heuristic tool in itself. Oulipian techniques offered me unorthodox tools for connecting with the world of human will and emotion.

The Oulipians played great games as well, but sometimes the game seemed to take precedence over the human significance of the story. If one is going to generate stories out of an arrangement of tarot cards, as Italo Calvino did in *The Castle of Crossed Destinies,* will the results pierce the human heart? At times yes, at times no, and so I found myself increasingly drawn to the less constrained writing of Virginia Woolf and Herman Melville.

But it was at the station of James Joyce where I moored my boat. I read *Ulysses* with a brilliant and sympathetic teacher. The book was damnably hard,[*12] with incomprehensible passages of Irish dialect, Catholic theology, and gutter obscenity. Joyce's ideas ranged from puerile to abstruse to profound, freely mixed together with no easily grasped logic. Joyce's book was meticulously written, yet his plan remained opaque to me. That

challenge kindled a similar curiosity in me as Logo had done years prior. But while computational concepts, however difficult, resolved themselves clearly, Joyce's *Ulysses* opened itself up to a myriad of interpretations. The characters of *Ulysses*—Stephen Dedalus, Molly Bloom, and Leopold Bloom—possessed lives that were laden with tragedy, loss, and pain. Stephen's loss of his mother, the Blooms' loss of their son, and the wayward wandering of their daughter—to me they were matters of the highest importance.

Yet for all of *Ulysses*'s rich messiness, it had been rigorously structured, even overstructured, by Joyce. Joyce distributed several schemas purporting to lay out the plan of the book, describing chapter-by-chapter parallels with episodes of the *Odyssey,* as well as the symbols and organs of the body that dominated each chapter. Yet Joyce's own words make it clear that the schema is not the be-all and end-all of the book. It was only one way (or eight ways) of seeing the novel, and Joyce had worked himself to exhaustion to ensure that no one interpretation or analysis could be final. The overlaid structures contradicted one another. A character could be a hero or a villain, or a success or a failure, depending on what prismatic structure the reader applied. No single one was correct. This was Joyce's way of drawing out what was lost in those gaps, by providing not one but *many* conflicting heuristics for understanding the book. Joyce's goal, as I came to see it, was not just to leave in ambiguity but to pile on contradiction upon contradiction. To enrich rather than reduce.

The formal patterns of *Ulysses* were fascinating to me, but not in and of themselves. Rather, they disguised and then revealed clues to the deepest puzzles of existence, those half-shown to us in dim light as the knotted tendrils of human feeling. Computers could not compare.

The Join

> It is only a frivolous love that cannot survive intellectual definition; great love prospers with understanding.
>
> —LEO SPITZER

I met my wife Nina when I was eighteen. We wouldn't get married for ten years, because who could possibly trust their eighteen-year-old self to be competent at choosing a partner? Our meeting had been a freak accident. I was visiting friends at Harvard, who were hosting an end-of-Passover pizza dinner at Pizzeria Uno. Seven or eight of us squeezed around a small table, and the person next to me was Nina. Nina wanted to be a poet. I wanted to be a novelist. We were both programmers. Nina and I each decided that the other was more interesting than anyone else at the table. I did not see her again for six months.

Over the summer, we exchanged emails daily while I did data entry and programming for a factory that made self-locking fasteners. We exchanged adolescent angst and book and movie recommendations. She liked the Cocteau Twins. I liked the Gang of Four. We made cassette mix tapes for each other. We eventually met up again and got together. Both of us had grown up as science and math geeks, yet both of us were enamored of literature and emotional quandries. She gave me James Agee and Walker Evans's *Let Us Now Praise Famous Men*. I gave her Jorge Luis Borges's *Ficciones*. We looked for the supposed plate memorializing William Faulkner's Quentin Compson on the Charles River Bridge in Cambridge. We never found it, though we did nearly freeze.[*13] A friend told me she knew Nina was special because she had convinced me to wear rainbow shoelaces. She had a far better ear for language and music than me, and she made me see beauty where I had only been looking for rigor and strain.

	TITLE	SCENE	HOUR	ORGAN
1.	Telemachus	The Tower	8 a.m.	
2.	Nestor	The School	10 a.m.	
3.	Proteus	The Strand	11 a.m.	
4.	Calypso	The House	8 a.m.	Kidney
5.	Lotus-eaters	The Bath	10 a.m.	Genitals
6.	Hades	The Graveyard	11 a.m.	Heart
7.	Aeolus	The Newspaper	12 noon	Lungs
8.	Lestrygonians	The Lunch	1 p.m.	Esophagus
9.	Scylla and Charybdis	The Library	2 p.m.	Brain
10.	Wandering Rocks	The Streets Room	3 p.m.	Blood
11.	Sirens	The Concert	4 p.m.	Ear
12.	Cyclops	The Tavern	5 p.m.	Muscle
13.	Nausicaa	The Rocks	9 p.m.	Eye, Nose
14.	Oxen of the Sun	The Hospital	10 p.m.	Womb
15.	Circe	The Brothel	12 midnight	Locomotor Apparatus
16.	Eumaeus	The Shelter	1 a.m.	Nerves
17.	Ithaca	The House	2 a.m.	Skeleton
18.	Penelope	The Bed		

	ART	COLOUR	SYMBOL	TECHNIC
1.	Theology	White, gold	Heir	Narrative (young)
2.	History	Brown	Horse	Catechism (personal)
3.	Philology	Green	Tide	Monologue (male)
4.	Economics	Orange	Nymph	Narrative (mature)
5.	Botany, Chemistry		Eucharist	Narcissism
6.	Religion	White, black	Caretaker	Incubism
7.	Rhetoric	Red	Editor	Enthymemic
8.	Architecture		Constables	Peristaltic
9.	Literature		Stratford, London	Dialectic
10.	Mechanics		Citizens	Labyrinth
11.	Music		Barracks	*Fuga per canonem*
12.	Politics		Fenian	Gigantism
13.	Painting	Grey, blue	Virgin	Tumescence, Detumescence
14.	Medicine	White	Mothers	Embryonic
15.	Magic		Whore	Hallucinative
16.	Navigation		Sailors	Narrative (old)
17.	Science		Comets	Catechism (impersonal)
18.	Flesh		Earth	Monologue (female)

One version of the patterns of *Ulysses*, as set down by Joyce.

If there was an algorithm for our feelings, it had to be reverse engineered. Our emotions and reactions often seize us with so much force that they wipe away any possibility of detached cognition. Love is irrational, it is passionate, it is madness. When I met Nina, I was scared. My previous relationship had ended badly and left me shell-shocked. It took months, if not longer, for me to separate Nina from my past.

There is a standard progression to many relationships: There is the initial crush, during which we are quick to overlook the flaws of the other person. When the rush of hormones and infatuation fades, we see what we previously couldn't. The other person reveals their flaws, their peccadilloes, and their small failures. Then begins the real work of negotiating. We remember the feeling of the initial crush. Maybe we wonder how things changed and why it is that the other person can no longer provoke that same level of positive joy. If we're cynical, we see that our minds become high on chemicals and gratifying delusions—tiny and harmless perversions of the true picture of

and inept judgments of our younger selves. We waited a decade to get married, to accumulate enough evidence that the code was now robust enough to keep the product running smoothly. But even foundations do not last forever. Regular maintenance and upgrades are crucial, lest a once-healthy system decline into a creaky machine.

3. BUGS NEVER DISAPPEAR.

They only hibernate. Our worst fights, and in particular our most trivial fights, always came after nine p.m., though it took us ages to figure that out. One of us—and if I'm being honest, it was me far more often than it was Nina—would get bent out of shape over an unkind word, a logistical screwup, or some other domestic misdemeanor. Some friend was annoying, some bill needed paying, or one or the other of us had failed in one of those very specific ways that only has meaning within the long-established habits of a relationship. These fights, which always began with some tiny offense and inflated into competing indictments of how the original dispute stood for some bigger problem, were dumb. Raw emotion kept the momentum going past any sense of rationality and perspective. And they always started after nine p.m., running on the fumes of fatigue and confusion. After each one was settled, we bemoaned the waste of our voices and our nerves. Nina figured out one solution, which was just to walk out. Then I figured out the other, which was not to discuss anything too heavy, or even make a pointed criticism, after nine p.m. Over a decade down the line, that little discovery has probably saved us hours of wear and tear on our cortisol levels and amygdalae.

Bugs can seem evanescent. I saw server crashes that appear out of nowhere and, just as mysteriously, seem to disappear. Bugs *never* disappear. If you haven't fixed it, it's a dead certainty the enigmatic bug will return. Bad fights may abruptly dissipate for calmer times, but underlying issues fester, only to explode later if they aren't excavated.