Entropy and FatFinger: Challenging the Compulsiveness of Code with Programmatic Anti-Styles

Daniel Temkin

Daniel Temkin
<leo@danieltemkin.com>

See <mitpressjournals.org/toc/leon/51/4> for supplemental files associated with this issue.

ABSTRACT

Coding, the translating of human intent into logical steps, reinforces a compulsive way of thinking, as described in Joseph Weitzenbaum's "Science and the Compulsive Programmer" (1976). Two projects by the author, Entropy (2010) and FatFinger (2017), challenge this by encouraging gestural approaches to code. In the Entropy programming language, data becomes slightly more approximate each time it is used, drifting from its original values, forcing programmers to be less precise. FatFinger, a Javascript dialect, allows the programmer to misspell code and interprets it as the closest runnable variation, strategically guessing at the programmer's intent.

We Are All Trash Coders

The psychological situation the compulsive programmer finds himself in while so engaged is strongly determined by two factors: first, he knows that he can make the computer do anything he wants it to do; and second, the computer constantly displays undeniable evidence of his failures to him.
—Joseph Weizenbaum, *Science and the Compulsive Programmer* [1]

We may all be compulsive programmers, or "computer bums" as Weizenbaum describes them. It has long been accepted that perfect code is not possible; that all code contains bugs, relies on libraries that have bugs, runs on operating systems with lurking bugs, waiting to be revealed. The modern approach is to manage error, rather than attempting to eliminate it. Yet when one writes code, there remains a nagging sense that there's a "right" way to write it, an approach that's always just out of reach, a perfect expression of our intent in discrete rational steps. As Wendy Chun describes, this ideal and our journey toward it is central to what makes coding pleasurable. "Every error leads to another," so that "programming becomes a technique, a game without a goal and thus without an end. . . . Hacking reveals the extent to which source code can become a fetish: something endless that always leads us pleasurably, as well as anxiously, astray" [2].

In a recent reddit programming thread on "controversial opinions," the top (most upvoted) comment was the opinion that programmers make up architectural problems out of boredom. One of Weizenbaum's telltale signs of compulsive coding is finding new faux-problems to solve. The second most popular was "Most programmers are bad at what they do, especially the ones that think they aren't," with many responders admitting to being bad themselves—bad, yet better than their contemporaries: "i'm a trash coder and i'm one of the best in the world." Is writing code as hard as this? What is a good coder, and why are they so rare (if they exist at all) [3]?

If coding is so hard already, it might be surprising that over-architecture would be one of the biggest issues for programmers, to the point that the idea of software architecture needs to be defended [4]. However, these two points are connected. Precision is the dominant aesthetic of code, and architecture is the aesthetic of control. It gives code the look of exactitude, of well-organized thought, with the hope that actual clarity will emerge. If control of the machine is illusory, if we can't easily identify the faults in our code, we try to design it away, sometimes going too far. We write clean-looking code, obey the latest software patterns, to invoke a feeling of control, yet never driving out the bugs, which lurk in our own programs, their libraries, their compilers, operating systems and the physical hardware below. We're all trash coders.

My two computer art projects are programmatic anti-styles: attempts to work against received ideas of how code performs and how it should be written. Entropy (2010) and FatFinger (2017) address the compulsiveness of code by working against the norms of programming. They make it impossible to indulge the illusion of control. They ask the programmer to give up precision, to use broader gestures in how intent is expressed to the machine, bugs and all: in FatFinger through the way code is written, and in Entropy through how it performs.

Entropy

At first glance at its code, Entropy appears a conventional language, a C clone with a nod to Pascal. However, the behavior of Entropy is very different from mainstream languages. Each piece of data, whether assigned to a variable or used as a constant, is a limited resource. Each time they are accessed, they become less precise.

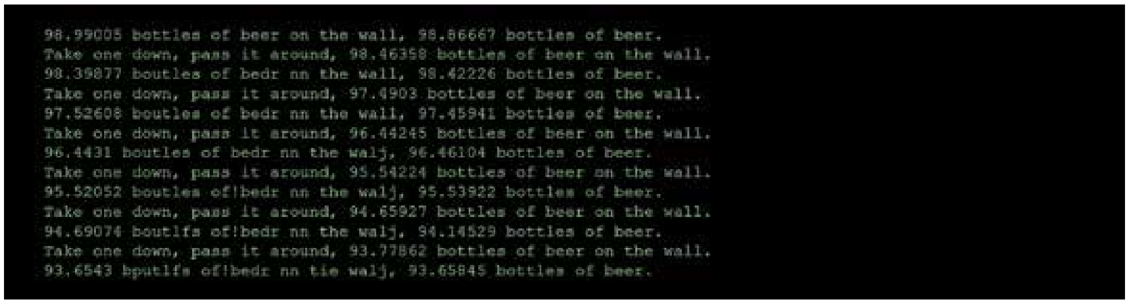
The name Entropy refers to Claude Shannon's concept of information entropy, or uncertainty. A (possibly apocryphal) story tells that von Neumann suggested the term to Shannon because it's a "solid physics word" and that "no one really knows what entropy really is, so in a debate you will always have the advantage" [5]. This seemed particularly fitting for a language dealing with ambiguity of meaning.

The approach an Entropy programmer takes to get her program to run is very different from programming with traditional languages. She needs to abandon the pursuit of precision—often working against years of habit—in order to program effectively. The programmer has, at best, a short window to get an idea across before the program's data corrodes. Well-architected code is possible, but pointless. The program itself is not altered, so the next time it runs, when variables are redeclared and flushed with data, that data is in its original condition, only to decay again.

This is accomplished by storing all data as floating-point numbers or floats. Integers are stored as floats and rounded back to ints when they are used. Characters, ordinarily stored as integers (in Unicode, the character "A" is stored as the number 65) are floats as well (65.0). Each time they are accessed, to change the value, to print it, to compare it, or any other use, a small amount of variation is introduced, sometimes remaining at 65.0, but as much as 0.5, enough to tip an A to a B. The mutation rate is configurable in the compiler, although the default of 0.5 cap is rarely altered. This means that the logic of the program is the same as how the programmer wrote it; the looping, functions, the overall program flow, are not altered, only the use of the data in the program that is affected.

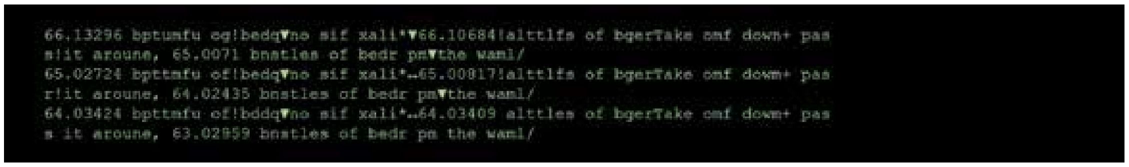
Seeing Entropy in action gives a better sense of what this means. The *99 Bottles of Beer on the Wall* program is a common second program for new languages, after programmers learn to write "Hello, World!" to the screen and want to see how iterating functions in the language. At the beginning of the run of the *99 Bottles* program in Entropy, we are already approximating, although it's only the iterator that looks odd on the first line, printing as 98.99005 (in this particular run,) rather than 99 (Fig. 1). The third line begins to have other recognizable breakdowns in the text, although the meaning is still clear. By 67, it becomes a challenge to read (Fig. 2), and by the end, it has collapsed into textual chaos. This is the end of the program (it looks like it got to eight and then died) (Fig. 3).

This jumble of random text is a frequent ending point for Entropy programs, as the more the same text is used, the more "off" it goes. The iterator (the counter 99, 98, 97, etc.) may slowly drift upward, but sooner or later, the program will halt; the force of the decrementation is stronger than the variable drift, as one is directed and the other erratic. Some programs might have far more radical failures in execution, but Entropy programs are generally more stable, in terms of actual halting, than one might expect.



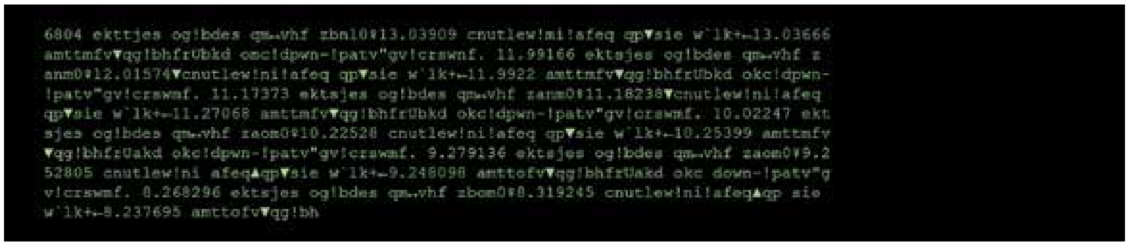
```
98.99005 bottles of beer on the wall, 98.86667 bottles of beer.
Take one down, pass it around, 98.46358 bottles of beer on the wall.
98.39877 boutles of bedr nn the wall, 98.42226 bottles of beer.
Take one down, pass it around, 97.4903 bottles of beer on the wall.
97.52608 boutles of bedr nn the wall, 97.45941 bottles of beer.
Take one down, pass it around, 96.44245 bottles of beer on the wall.
96.4431 boutles of bedr nn the walj, 96.46104 bottles of beer.
Take one down, pass it around, 95.54224 bottles of beer on the wall.
95.52052 boutles of!bedr nn the walj, 95.53922 bottles of beer.
Take one down, pass it around, 94.65927 bottles of beer on the wall.
94.69074 boutlfs of!bedr nn the walj, 94.14529 bottles of beer.
Take one down, pass it around, 93.77862 bottles of beer on the wall.
93.6543 bputlfs of!bedr nn tie walj, 93.65845 bottles of beer.
```

Fig. 1. *99 Bottles* begins. (Screen capture by Daniel Temkin)



```
66.13296 bptumfu og!bedq▼no sif xali*▼66.10684!alttlfs of bgerTake omf down+ pas
s!it aroune, 65.0071 bnstles of bedr pm▼the waml/
65.02724 bpttmfu of!bedq▼no sif xali*↔65.00817!alttlfs of bgerTake omf dowm+ pas
r!it aroune, 64.02435 bnstles of bedr pm▼the waml/
64.03424 bpttmfu of!bddq▼no sif xali*↔64.03409 alttles of bgerTake omf dowm+ pas
s it aroune, 63.02959 bnstles of bedr pm the waml/
```

Fig. 2. *99 Bottles* with 67 left. (Screen capture by Daniel Temkin)



```
6804 ekttjes og!bdes qm↔vhf zbnl0♀13.03909 cnutlew!mi!afeq qp▼sie w`lk+↔13.03666
amttmfv▼qg!bhfrUbkd omc!dpwn-!patv"gv!crswnf. 11.99166 ektsjes og!bdes qm↔vhf z
anm0♀12.01574▼cnutlew!ni!afeq qp▼sie w`lk+↔11.9922 amttmfv▼qg!bhfrUbkd okc!dpwn-
!patv"gv!crswmf. 11.17373 ektsjes og!bdes qm↔vhf zanm0♀11.18238▼cnutlew!ni!afeq
qp▼sie w`lk+↔11.27068 amttmfv▼qg!bhfrUbkd okc!dpwn-!patv"gv!crswmf. 10.02247 ekt
sjes og!bdes qm↔vhf zaom0♀10.22528 cnutlew!ni!afeq qp▼sie w`lk+↔10.25399 amttmfv
▼qg!bhfrUakd okc!dpwn-!patv"gv!crswmf. 9.279136 ektsjes og!bdes qm↔vhf zaom0♀9.2
52805 cnutlew!ni afeq▲qp▼sie w`lk+↔9.248098 amttofv▼qg!bhfrUakd okc down-!patv"g
v!crswmf. 8.268296 ektsjes og!bdes qm↔vhf zbom0♀8.319245 cnutlew!ni!afeq▲qp sie
w`lk+↔8.237695 amttofv▼qg!bh
```

Fig. 3. *99 Bottles* completes. (Screen capture by Daniel Temkin)

Entropy draws from languages with self-modifying code, such as Malbolge, designed to be the most difficult language to program in. In Malbolge, each program runs as a large loop, with every command firing many times. When a command executes, it alters its behavior, keeping the source code in constant flux, delivering different behavior with every iteration [6]. From a more practical standpoint, the OISC (One Instruction Set Computer) languages are also self-modifying, where the source code and the data the program operates on are one and the same, in the interest of a minimalist physical architecture. As challenging and nonintuitive as these languages might be, they are still Turing Complete, meaning capable of representing any algorithm runnable on a conventional machine. Entropy is not Turing Complete, nor does it belong to any of the more limited categories of computational complexity, because the language is probabilistic: we can't describe a set of algorithms it can perform, even in an idealized, virtual space, since the same code will produce inconsistent results from one run to the next [7].

In that respect, it is informed by glitch art practices that undermine the objectivity of the machine. Entropy was inspired in part by the work of the Dutch glitch duo JODI and was initially presented at the GLI.TC/H 2010 Conference in Chicago. Curt Cloninger's essay written for that conference says that "The glitch foregrounds and problematizes this myth of pure transcendental data, of pure and perfect signal. The glitch is a perpetual reminder of the immanent, real-time embodiment of executed code" [8].

Drunk Eliza

As Entropy was inspired by Joseph Weizenbaum's writing, it seemed natural to see how his most famous program, the chatbot Eliza, would function rewritten in Entropy. Eliza is an extremely simple program which, regardless, is seen as having a personality, something which Weizenbaum found disturbing. I wondered what personality Eliza would have, written in code where her databank decays over time. Would it be like HAL's slow degeneration in 2001, intelligence reduced to a sad shadow of itself? Or something entirely different?

Given the simplicity of the program, with no iterators to corrupt, the program behavior remained the same, apart from an occasional random crash. Much more prominent was the steady decay of her responses, resembling the slurring of words, which inspired the name change to "Drunk Eliza." Once it was put online, the response to the program was fascinating to see. I did not record people's conversations with Drunk Eliza, so my evidence of the conversations they had were from public sources revealed by the users themselves, including a Metafilter thread where many posted their conversations with the chatbot [9].

The labels used in conversation, "YOU" and "DRUNK ELIZA," are themselves Entropy variables, and, as the most commonly used words in the program, their decay becomes a good marker of how long the program has run, and advanced Eliza's banks have corroded. From the Metafilter thread, here is an excerpt by a user with the handle 0xFCAF:

```
YMU:          aw geez you should drink some water
ERTNK ELIZA:  #PH, I SHOULD DRYOUNK SOYOU WATER
YMU:          yup that
ERTNK ELIZA:  #CAN YOU ELABORATE ON THAT?
YMU:          hey you're sounding better, let's do shots
ERTNL ELIZA:  #DOET IT*QLEASE YOU TO BELICVDI AM SOUNDING BETTER, LET'S DO SHOTS
YMU:          WHOOOOO
ERTNL ELIZA:  #COME COME!ELUCSDATE cOUR THOUGHTS.
```

This style of response was the most common; as Eliza's databanks devolved, people would respond in a similarly "drunken" manner, often by mashing keys. As the Metafilter discussion progressed, however, it took only a few entries before guys began hitting on Drunk Eliza, starting with a user who quickly followed up with:

> i'm spending my saturday night trying to seduce a drunk adaptation of a chat-bot coded in 1985. what has become of my life.

What originally seemed good-natured became less so as the tendency to hit on the chatbot became more pronounced. This was remarked on by user Blasdelb:

> There is something pretty profoundly disturbing to me that, upon the creation of a drunk program with a female name, our first impulse is to either attempt to "seduce" her or coerce her into other sexual acts.

These two inclinations; the one toward writing in a drunken way, and that of hitting on the chatbot, sum up nearly all of the interactions I've seen posted online. The fact these are all shared publicly (again, I don't track conversations on the site) makes it more disturbing that this is the type of conversation people decided to share.

Drunk Eliza was designed to bring the experience of the Entropy programming language to non-programmers; but putting a work into the public space sometimes means the public will bring the work to a very different place than the artist had intended.

Entropy.JS

Programming languages are particularly open to reinterpretation by others. A programming language is nothing more than a lexicon paired with syntactic and semantic rules. Esoteric languages are sometimes released simply as the ruleset, with no specific implementation as a working compiler or interpreter. Once a language is released, it is hoped that others will write code in the language to explore it, and sometimes create their own implementations. As an esolanger (a creator of esoteric programming languages), this is what one hopes will happen, that the language will be used by others. It is a true collaborative form.

Andrew Hoyer, a programmer and designer I was not familiar with before this project, created his own partial implementation of Entropy, using Javascript. Rather than writing a full interpreter in JS, where the entire Entropy language could be recreated in style and syntax, Hoyer chose to make a JS library, so that the code is written in Javascript, and selectively enforced, by adding an "Entropy watcher" to specific variables.

It is worth noting how different this approach to adaptation is from how digital art is appropriated within the art community. Hoyer decided what was important about Entropy: its behavior, not the aesthetic of the code itself. While there were aesthetic considerations to the code style—the banal and awkward Pascal/C syntax I selected was a considered design choice—I agree with Hoyer's assessment that this is not the essential aspect of the work, and that the trade-off in allowing it to function within Javascript is to the language's advantage. Working with Javascript not only makes the language more widely available, it also opens up the language to visual content, which would be a struggle for the original Entropy compiler, written for the .NET compiler.

On Hoyer's site, the Entropy.JS language is used to animate a set of variables, expressing visually the randomized numeric changes. He has an image of the Mona Lisa (Fig. 4), which is activated as one mouses over it, becoming blurrier and more approximated. Even the title of the page, "Entropy," slowly decays, updating its random changes immediately to the screen. This is all possible only by picking and choosing which aspects of the page are Entropic and which are controlled by conventional Javascript. It nicely brings the experience of the Entropy language to nonprogrammers.

FatFinger

Human readers have a high tolerance for elision, for swapping of letters, or missing symbols, as natural text obfuscators like SpellFucker [10] have shown. Programming languages generally have no tolerance for this. The history of obfuscated code and of esoteric programming languages are breaks from clarity of presentation; however, they are still written in languages like C; the code below might be inscrutable, but it is still exacting; it looks random, but if the wrong "magic number" is used, it will not run as intended [11,12].

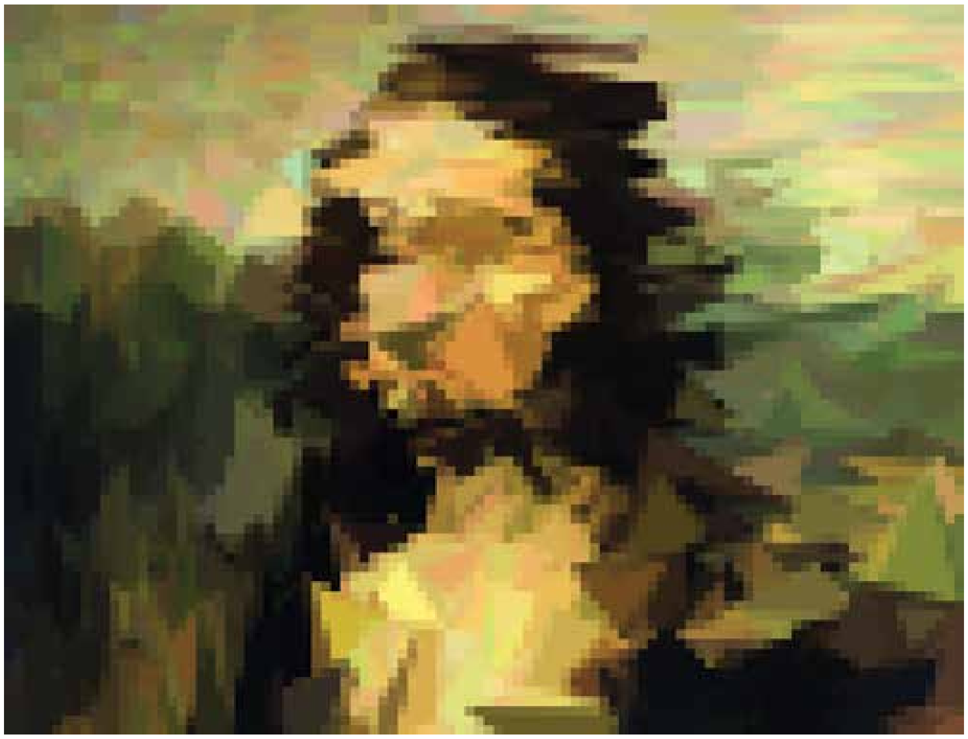


Fig. 4. *The Mona Lisa in Entropy.JS*, after several minutes of mouse-over action. (© Andrew Hoyer. Screen capture by Daniel Temkin.)

```
main(a,b)char**b;{int c=1,d=c,e=a-d;for(;e;e--)_(e)<_(c)?c=e:_(e)>_(d)?d=e:7;
while(++e<a)printf("\xe2\x96%c",129+(**b=8*(_(e)-_(c))/(_(d)-_(c))));}
```

—A tiny implementation of command-line "sparkline" data visualization;
an IOCCC winner, 2013. David Lowe [13]

With common debates such as tabs vs spaces, programmers have long brought their compulsive habits in trying to placate the compiler to bear in attempts to organize their thoughts into logical style. Javascript has been a constant headache for programmers who want to impose such order. It is an unruly language, with dynamic typing and implicit declarations. Mistype a variable name when assigning a value, and a new variable with the typo'd name is created. Compare an integer with a string and you'll get nonintuitive results. Language supersets like TypeScript and frameworks like JQuery and Angular try to reign in JavaScript's unmanageability, as do modern integrated development environments (such as Visual Studio Code), which helpfully indicate possibly unintended statements with IntelliSense squigglies.

On the opposite site, we have projects like Martin Kleppe's JSFuck project, a coding style for Javascript that illustrates how just a few punctuation signs are themselves a Turing Complete dialect, due to quirks in the JS language; something certainly unintended by Javascript's designer. The similarly critical piece, FuckIt.JS, "corrects" Javascript code by steamrolling it—any line that can't be interpreted as JS is commented out, until something fully runnable emerges. While FuckIt is primarily a joke about frustrations of the language, JSFuck is a simple and beautiful observation about the language, which is only successful because of a host of nonintuitive rules in the JS language [14].

In 2017, I created FatFinger, a Javascript dialect which allows for scrambled and seemingly nonsensical text to function as JS. The interpreter analyzes each line of code and finds the closest word for each unroecognized token, translating it behind the scenes into runnable JS. This allows the programmer to write in a sloppy, but human, style, full of typos. FatFinger guesses at the programmer's intentions.

The sample Hello World program for FatFingerJS looks like this:

```
<script type="text/javascript" src="fatfinger.js"></script>
<script type="text/javoscript"> // any misspelling of javascript works here
  vart x = "herrrllo werld"
  dokkkkumint.rit3(xx)
</script>
```

At the top is the inclusion of the fatfinger.js library; this needs to be typed correctly (or cut and pasted from another source) in order to include and invoke the library. The second script block works with any misspelling of Javascript—be careful not to spell it correctly, or the rest of the code will be read as regular Javascript, not FatFingered JS. From there, the variable x is declared, with "vart," a misspelling of "var," and its value is printed to the page with a variation of "document.write." Semicolons are not included at the end of each line.

FatFinger uses Damerau-Levenshtein Distance, a string metric for measuring the edit distance between two sequences. It is an indicator of how many changes are needed for one word to become another: insertions, deletions, transpositions, or substitutions of a character within a word. Since it has a limited vocabulary, it allows for changes that fall very far from the original word (such as "dokkkkkkkkkkkkkkkkkkkumint" for "document") that are connected by a phonetic relationship (to us) but succeed within FatFinger because it happens that there's no other word similar enough to "document" (that has, say, a big long line of k's) to match against.

This *99 Bottles of Beer* program has typos of the words *if*, *bottles*, *console* and *counter*:

```
var bottles;
for (var counter = 99; contr >= 1; counter = counter—1)
{
        if (counter == 1) {
                botles = ‘bottle’;
        } else {
                bottles = ‘bottles’;
        }
        constole.log(counter+“ ”+ bottless +“ of ber on the wall.”);
        if (countr < 99) {
                conssole.lg(“”);
                consoles.logg(counter+“ ”+ botttles+“ o beer on the wall.”);
        }
        conable.log(counter+“ ”+botttttles+“ of beer.”);
        console.lo(“Take one down.”);
        console.log(“Pass it arund.”);
        ift (ount == 1) {
                console.log(“No botles of beer on the wall.”);
        }
}
```

Lacking true reflection, there’s no way in Javascript to find everything in scope at the moment. Instead, FatFinger’s interpreter looks to the Window object, which contains every object in scope, and uses this to get a list of eligible variables. In addition, it tracks each variable as it’s declared and keeps that in its list of eligible words. While FatFinger attempts to track scope, it is not great at scope, which further encourages developers to declare everything at the highest possible level (essentially globally). This is considered a poor programming practice, but is right in line with FatFinger’s approach to code.

Part of the inspiration for FatFinger was seeing how art students new to Javascript actually use the language. At a class visit with Paul Hertz’s Artware class at School of the Art Institute of Chicago in 2015, I observed student projects written in Javascript by undergraduates with varying levels of skill with the language. Since the assignment allowed for incorporating existing JS programs found online, many beginning students chose to cut and paste found scripts in a patchwork that did more or less what they wanted their code to do and adjusted them to work together and better fit their design. This led to code that was not organized well, full of declared variables never used, no logical structure—a mess of code, but an empowering one, allowing nonprogrammers to shape code before developing the skills necessary to completely rework it.

Where Entropy has to be written correctly, to run erratically, FatFinger is written erratically, to execute exactly as it would with well-written JavaScript. As it is still a new language, the way it will be used and abused by programmers has yet to become apparent. The hope is that its critique of code will resonate with programmers and others ready to reevaluate their relationship with the machine.

Conclusion

No discipline brings the inherent irrationality of human thought to the forefront more directly than programming. The style of most programming languages is aspirational; they connote orderliness and structure, in the face of heaps of evidence that bugs are endemic to code. This creates the perfect structure for human pseudo-logic to run wild.

Both these projects speak to the actual experience of coding, which is fraught with error. Entropy makes error inevitable, while FatFinger tolerates a sloppiness of text (and of the thought behind it) that ordinarily would never pass muster with the interpreter. They work against the compulsiveness of programming. They encourage a style that is more accepting of the inevitable presence of error and of the limited capacity of the programmer to control the machine.

References and Notes

1. J. Weizenbaum, *Computer Power and Human Reason: From Judgment to Calculation* (London: WH Freeman & Co., 1976) pp. 111–131.
2. Wendy Hui Kyong Chun, *Programmed Visions* (Cambridge, MA: MIT Press, 2011) pp. 19–54.
3. reddit.com, "What's Your Most Controversial Technical Opinion?," posted 8 December 2017: <www.reddit.com/r/programming/comments/7ifinq/whats_your_most_controversial_technical_opinion/> (accessed 10 January 2018).
4. Scott Reynolds, "Well Architected != Over-Architected": <https://lostechies.com/scottreynolds/2009/10/01/well-constructed-over-architected/>.
5. Jimmy Soni and Rob Goodman, *A Mind at Play* (2017) pp. 161–162.
6. Ben Olmstead and Daniel Temkin, "Interview with Ben Olmstead—esoteric.codes. Retrieved 28 September 2017," esoteric.codes (2014): <http://esoteric.codes/post/101675489813/interview-with-ben-olmstead>.
7. John E. Hopcraft, Rajeev Motwani and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (2008).
8. "Drunk Eliza," Metafilter (2012): <www.metafilter.com/113232/Drunk-Eliza>.
9. Curt Cloninger, "GltchLnguistx: The Machine in the Ghost / Static Trapped in Mouths" (2010): <www.lab404.com/glitch/>.
10. Igor Pavlov, "SpellFucker" (2017): <https://spellfucker.com/>.
11. Daniel Temkin, "Three Obfuscators for Natural Language" (2017): <http://esoteric.codes/post/168502942367/three-obfuscators-for-natural-language>.
12. Michael Matteas and Nick Montfort, "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics," in *Proceedings of the 6th Digital Arts and Culture Conference* (2005) pp. 144–153.
13. David Lowe, "Sparkl: A Tiny Implementation of Command-Line 'Sparkline' Data Visualization," International Obfuscated C Code Contest, 2013 (2013): <www.ioccc.org/2013/dlowe/hint.html>.
14. Martin Kleppe and Daniel Temkin, "Interview with Martin Kleppe," esoteric.codes (2017): <http://esoteric.codes/post/157780744195/interview-with-martin-kleppe>.