

CHAPTER 3

The Craft and Craftiness of Hacking



I have nothing to declare but my genius.
—Oscar Wilde

I, for the first time, gave its proper place among the prime necessities of human well-being, to the internal culture of the individual.
—John Stuart Mill, *Autobiography*

Hackers value cleverness, ingenuity, and wit. These attributes arise not only when joking among friends or when hackers give talks but also during the process of making technology and writing smart pieces of code. Take, for example, this short snippet of what many hackers would consider exceptionally clever code written in the computer language Perl:

```
#count the number of stars in the sky
$cnt = $sky =~ tr/*/*/;
```

This line of Perl is a hacker homage to cleverness; it is a double entendre of semantic ingenuity and technical wittiness. To fully appreciate the semantic playfulness presented here, we must look at the finer points of a particular set of the developer population, the Perl hacker. Perl is a computer language in which terse but technically powerful expressions can be formed (in comparison to other programming languages). Many Perl coders take pride in condensing long segments of code into short and sometimes intentionally confusing (what coders often call “obfuscated”) one-liners (Monfort 2008). If this above line of code were to be “expanded” into something more traditional and accessible to Perl novices, it might read something like:

```
$cnt = 0;
$i = 0;
$skylen = length($sky)
while ($i < $skylen) {
```

```
$sky = substr($sky,0, $i) . '*' . substr($sky, $i+1,  
length($skylen));  
$i++;  
}  
$cnt = length($sky);
```

We see that the Perl programmer has taken six lines of code and reduced them to a single line by taking advantage of certain side effects found in the constructs of the Perl language, and the very act of exploiting these side effects is a great example of a hack. With this transformation of “prose” into terse “poetry,” the developer displays a mastery of the technical aspect of the language. This mastery is topped on the semantic level by a quip. The programmer has named the variable `$sky`, and the star is the asterisk (*) character.¹ The counting function in this program counts any appearance of the asterisk symbol—hence, “counting the number of stars in the sky.” This code has a technical function, but within a community of peers, its performance is also a declaration and demonstration of the author’s savvy.

Hackers will publicly acknowledge such acts of “genius” and are thus fiercely meritocratic—in ideology and practice. Yet given that so much of hacker production is collective, a fact increasingly acknowledged and even celebrated in the ethical philosophy of F/OSS, a commitment to individuality, meritocracy, and independence is potentially subverted by the reality of as well as desire to recognize their fundamental interdependence. The belief in the value of individuality coupled with the constant need for the help of other hackers points to a subtle paradox that textures their social world. The tension between individualism and collectivism, in particular, is negotiated through the extremely well-developed and common penchant that hackers have for performing cleverness, whether through technological production or humor. Hackers do not treat all forms of expression, technology, and production as original and worthy expressions of selfhood. Instead, one must constantly manifest, in the face of one’s peers, a discriminating and inventive mind by performing its existence through exceptionally ingenious and clever acts. By contributing a shining, awe-inspiring sliver of their creative self in a domain otherwise characterized by a common stock of knowledge and techniques, hacker utilize humor or clever code to perform their craftiness, and thus momentarily differentiate themselves from the greater collective of hackers.

While this chapter describes the ethnographic expression of humor and cleverness among hackers (which might be valuable and interesting in its own right), it does so at the service of other, analytic goals. Examining humor and cleverness will allow me to more richly demonstrate how tensions (say, between individualism and collectivism) arise through the course of technological practice, and how hackers partially resolve them. Taking a close look at these frictions takes us a long way toward understanding

the social context under which these hackers labor and why free speech ideals—in contrast to those of intellectual property instruments—resonate with their experiences. The friction between individualism and collectivism (and its articulation in meritocratic discussions) helps, for one, underwrite a dynamic social environment in which hackers labor. Second, this tension speaks directly to issues of authorship, selfhood, creativity, and intellectual property in a way that extends, contrasts, and critiques the dominant intellectual property regime.

The analysis opens by examining the pragmatics and aesthetics of hacking, by which I mean the constraints and properties of their technological activities, and contrasting the writings of two hackers, Espe and Da Mystik Homeboy (DMH). Understanding the pragmatics of hacking is necessary to grasp the contradictions/tensions that mark hacking along with what I call the poetics of hacking: the extreme value hackers place on ingenuity, craftiness, and cleverness. I will explore these largely through the angle of humor. The final section revisits the tension between individualism and collectivism. Hackers assert a form of individualism that valorizes self-expression and development among peers engaged in similar acts of technological production, while tightly entangled with each other through constant collaboration.

HACKER PRAGMATICS

PYTHON: REACHING A TRANSCENDENTAL SPACE

I remember when I found python, back in the 1.52 days [1.52 refers to a version number].² I was an unemployed slacker living in a student co-op. I'd sit in a (since disappeared) cafe in Berkeley and write reams of more or less useless code, simply for the joy of it. I'd reach some sort of transcendental state fueled by relevant whitespace, clear syntax, and pints of awfully strong, black coffee. In those days I first felt the pure abstract joy of programming in a powerful way—the ability to conjure these giant structures, manipulate them at will, have them contain and be contained by one another. I think I learned more in those couple of months, thanks to Google and a free ricochet connection, than in my previous years in CS [computer science].

Eventually, however, it became clear I had to get a real job. Flaky freelance contracts which never paid sucked so hard. So, I hemmed and hawed and was conflicted and finally got a job, and it involved perl. It was, perhaps, a worst-case perl scenario. A very rapidly growing website, a few developers with vastly different styles, a lack of real communication, and a pronounced lack of appreciation for namespaces. From my high tower of control and purity, I'd been thrown into a bubbling pool of vaguery and confusion. Cryptic variables would pop out of

the aether, make an appearance in a 2000 line CGI [Common Gateway Interface], and never be heard from again. Combating naming schemes would meet where different spheres of developer influence overlapped—\$postingTitle and \$PostingTitle doing battle in the same subroutine. Scripts almost—but not quite—deprecated. The situation is quite a bit more under control now, 3 years later.

—*Espe*

PERL: HACKING IN THE BIG BALL OF MUD

Perl has been derided by many people as an ugly, difficult to learn language that enforces bad habits. I generally do not advocate perl to people who are attempting to learn programming, or even mention it's existence. However, perl, for better or worse, is a culmination of decades of culture. Perl is a Unix Gematria—an arcane relation of symbols evolved in a manner similar to Jewish Qabbalistic numerology. Many other languages, such as python or Java, attempt to enforce a strict framework and rule set of contracts, interfaces, strong typing, and private methods to delineate functionality. While much of this stems from noble traditions of SmallTalk and ML [they are computer languages], much of it also fails to realize the point of these ancestral languages: categorization (such as through strict typing and object models) is itself a form of computation. When this fact is not respected, you wind up with a bastardized language that is [. . .] Anal.

Perl was designed by a linguist, and realizes that people have different things to say in different contexts, and your language is defined by the environment and not vice versa. As Paul Graham said, both the world and programming is a “Big Ball of Mud,” which perl has evolved around. The implicit variables, the open object model, the terse expressions all contribute to hacking on the Big Ball of Mud.

Finally, there is a very pragmatic reason to like perl: It will save your ass. Those who are fluent enough in the culture to realize that “this problem has been solved before,” will be able to invoke forces through perl. Again, similar to the numerologists, with a few arcane symbols that are undecipherable to the outside world, great acts of magik can be accomplished.

—*Da Mystik Homeboy*

Espe is a San Francisco hacker who is clearly fond of Python, an open-source computer language. Originally created by a Dutch programmer as a teaching language, Python is now a thriving open-source project. The language's distinguishing feature (both aesthetic and technical) is its strict technical parameters that require bold syntactic clarity. For example,

Python is unusual among programming languages in that the amount of space used to indent a line of code actually affects the code's meaning. On his blog (excerpted above), Espe explains how he was able to hack to his heart's delight for no other reason than to experience "the joy of programming." His stance toward Python is reverent, rooted in deep pleasure. He obviously adores both the formal structure—Python—and the substance—coffee—that have enabled him to hack for his own enjoyment and self-development. In this instance, Espe constructs programming as a pleasing, unencumbered exercise of ample creativity. He seeks in hacking to reach the elusive quality of perfection.

By the next paragraph, however, his register shifts to one of dismayed irreverence toward another programming language, Perl, considered by many to be the antithesis of Python, and therefore a source of antipathy for many Python fanatics. Eventually forced to hack for money (a problem itself for this programmer), he was handed "a worst-case scenario." Poorly coded Perl transformed programming from an activity of boundless satisfaction into a nightmarish ordeal. Espe describes this unfavorable turn of events as being plucked from his "high tower of control and purity," only to be "thrown into a bubbling pool of vaguery and confusion." In having to read and parse other people's codes, programmers routinely encounter what has been depicted aptly as a "twisting maze of corridors, a bottomless pit" (Ullman 2003, 262).

In the second extract, we have DMH, also a San Francisco hacker, but unlike Espe, a self-styled Perl alchemist. Perl's creator, a linguist and programmer named Larry Wall, intended the code to embody the flexible and often-irrational properties of a natural language. As noted by DMH, Perl's aesthetic and technical features are opaqueness, complexity, and flexibility. Also run as an open-source project, Perl is incorporated into the identity of many of its supporters, who call themselves Perl Monks, underscoring the single-minded dedication they have for what is considered a language that can produce poetic (or highly unreadable code) that is creatively displayed during obfuscated code contests, which are usually held for Perl, C, and C++.³

While DMH respects Perl for what it is most famous for—its cryptic nature and poetic elegance—he is drawn to Perl for pragmatic reasons. Its "implicit variables, the open object model, the terse expressions," DMH says, allow him to hack on the "Big Ball of Mud"—that is, the world of thick, unmanageable problems and constraints. For DMH, Perl's appeal lies in its extensive common stock of shared solutions and architectural flexibility, which he contrasts to Python, a language so "anal" it is unable to accomplish "great acts of magik." By this he means what is known among Perl geeks as the Perl's motto: "TIMTOWTDI" (There's more than one way to do it).

Digital computers allow for the creation and use of *mini-machines* (aka software) written by programmers using any number of computer languages.

Instead of having to build a piece of hardware for every type of desired function (like a calculator, music recorder, or word processor), the computer is a general-purpose machine that once animated by software programs, can potentially behave as all those functional objects. Espe captures the expansive technical capability of software when he defines coding as “the ability to conjure these giant structures, manipulate them at will, have them contain and be contained by one another.” This is computing in its dimension of unfettered freedom.

If at one level hackers adroitly exploit the expansive technical capabilities of the computer, they are also significantly limited by a powerful force field of constraint—the Big Ball of Mud that DMH refers to in his tract on Perl. Constraints are constant and of a nearly infinite variety, such as hardware specifications and failures, computer language syntax, “clueless” managers, inherited “crufty” or vague code, spam, incompatible file formats, “dumb” patent laws, misguided customers, technical specifications, and manager-dictated deadlines. Problems are so central to software that some have even portrayed “glitches” as the “manifestation of genuine software aesthetic” (Goriunova and Shulgin 2008, 111).

Programming thus entails an expansive form of exploration and production that unfolds into a labyrinthine landscape of intricate barriers and problems. Julian Dibbell (2006, 104; see also Ensmenger 2010, 3) depicts the nature of computing, quite poetically, as an “endlessly repeatable collusion of freedom and determinism—the warp and woof of fixed rules and free play, of running code and variable input.” Because of constraints and the complexity of coding, to hack up solutions effectively, as Michael Fischer (1999, 261) notes, requires “a constant need for translation, interfacing, sharing, and updating.”

As part of this practical capacity, the very nature of hacking—turning a system against itself—is the process of using existing code, comments, and technology for more than what their original authors intended. This is the paradox of constraint. Since many technical objects are simultaneously bound by certain limits yet exhibit potential excesses (Star and Griesemer 1998), during the course of their existence, they can be exploited and redirected toward new paths of functionality by acts of hacking. Hackers are thus attuned not simply to the workings of technology but also seek such an intimate understanding of technology’s capabilities and constraints that they are positioned to redirect it to some new, largely unforeseen plane. They collectively and individually derive pleasure in outwitting constraint. In essence, while hacking follows a craftlike practice, it is predicated on a stance of craftiness to move the craft forward. Hacking is where craft and craftiness converge.

Programming and similar technical activities require extremely rigorous logical skills, an unwavering sensitivity to detail (a single wrong character can render a program useless), and such an intimate command of a system that one can, if need be, exceed the conventional or intended constraints of

the system. It requires, in the words of programmer Ellen Ullman (2003, 177), a “relentless formalism.” Given the accelerated pace of technological change, hackers also have to perpetually learn new technologies as old ones are phased out due to obsolescence, in order to remain competitive in a marketplace.

Out of this routine form of technical activity hackers have constituted an expansive pragmatic practice of instrumental yet playful experimentation and production. In these activities the lines between play, exploration, pedagogy, and work are rarely rigidly drawn. Sometimes hackers will be motivated by a work-oriented goal, as is/was the case with DMH. At other times, they are motivated to hack for the sheer pleasure of doing so, as Espe emphasized. In either case, frustration *and* pleasure are fundamental to hacking.

A lifetime of creative and pleasurable technical production that often depends on computers also blurs the line between selves and objects. As famously phrased by Sherry Turkle (1984), computers are a hacker’s “second self.” The hacker relationship to computers and software, though, rarely exists in a steady state in which the self unproblematically melds with this object to catapult hackers into a posthuman, postmodern state of being. The hacker relationship with the computer is a far more finicky, prickly, and interesting affair in which computers themselves constantly misbehave and break down (as do the hackers, at times, when they burn out from such an intense and demanding craft). Hackers sometimes confront their computers as an unproblematic and beloved “object,” and at other times view them as an independent and recalcitrant “thing”—a differentiation posed by Heidegger ([1927] 2008) in his famous exploration of things and objects.

In Heidegger’s cartography, an object strikes its users as familiar and beyond the scope of critical awareness. Its social meaning is held in place through regular patterns of use and circulation. But when we misuse an object (a spoon used as a knife or a can opener utilized as a hammer) or when an object malfunctions, its thingness is laid bare in the sense that its material characteristic becomes evident. As noted by scholar of things and stuff Bill Brown (2001, 4), “the story of objects asserting themselves as things is the story of how the thing really names less an object than a particular subject-object relation.”

In order to appreciate the hacker relationship to computers, this subtle differentiation between an object and a thing is crucial. Hacker technical practices never enact a singular subject-object relation, but instead one that shifts depending on the context and activity. There are times when hackers *work with* computers, and in other cases they *work on* them. Much of hacker technical practice can be described as an attempt to contain the thingness of computers that arises through constant problems and constraints by transforming it back into a pacified, peaceful object that then becomes an ideal vehicle for technical production as well as creative expression. At times, their labor is characterized by grinding effort, and in other

instances, it involves far more pleasurable streams of seemingly friction-free work. The “Python versus Perl Wars” above articulates the metapragmatic understandings of hacker labor that makes it possible to enter into this relational oscillation in the first place.

HACKER CLEVERNESS

Humor can be dissected, as a frog can, but the thing dies
in the process and the innards are discouraging to any but
the pure scientific mind.

—E. B. White, *A Subtreasury of American Humor*

As the examples provided by Espe and DMH display, hacker technical practice is rooted in a playful, analytic, and especially reflective stance toward form that switches between reverence and irreverence depending on individual preferences as well as the context of activity. Hackers routinely engage in a lively oscillation of respect and disrespect for form, often expressed in arguments over the technical idiosyncrasies, strengths, and weaknesses of a programming language, OS, or text editor. These disagreements are the subject of a range of humorously formulated “holy wars,” such as Perl versus Python (which we just got a glimpse of), vi versus Emacs (text editors), and Berkeley Software Distribution versus Linux (different Unix-based OS). Despite this, hackers otherwise share an ideal about how labor and production should proceed: with remarkable craftiness and wit.

One important vehicle for expressing wit is humor. As Mary Douglas (1975, 96) famously theorized, joking brings together “disparate elements in such a way that one accepted pattern is challenged by the appearance of another,” and can be generally defined as “play upon form.” Before expanding on the role of humor among hackers, it is key to highlight that hackers are able to joke with such facility because of the habituated dispositions (Bourdieu 1977) of thought along with tacit knowledge (Polanyi 1966) acquired through a lifelong and routine practice of logic-oriented problem solving. Hackers liberally enjoy hacking almost anything, and because their cultivated technical practice requires an awareness and rearrangement of form, they are able to easily transfer embodied mental dispositions into other arenas. To put it bluntly, because hackers have spent years, possibly decades, working to outsmart various technical constraints, they are also good at joking. Humor requires a similarly irreverent, frequently ironic stance toward language, social conventions, and stereotypes (Douglas 1975).

The mastery and craft of hacking, however, do not fully account for the *craftiness of hackers*.⁴ Many of the engineering arts and sciences are guided by similar aesthetic-solving sensibilities, mandates, and preoccupations (Galison 1997; see also Jones and Galison 1998). Engineers and other

craftspeople, such as repairpersons, also deploy similar problem-solving skills rooted in tinkering: they must engage with the limits, possibilities, and constraints of various material objects, and fiddle around to find a nonobvious solution (Orr 1996; Sennett 2008).

Hacker aesthetics share these above-mentioned dispositions, but differ in that hackers see ingenuity and cleverness, often expressed through humor, as far more than a means to regiment and guide technological innovation.⁵ Among hackers, humor has a substantial life of its own. Hackers value craftiness and cleverness for their own sake. Whereas academic scientists tend to value referential cleverness as it concerns their work, hackers value cleverness as self-productive, and thus make it appropriate to nearly any context (mathematicians, though, are well known for their prolific humor that exceeds their discipline). Hackers idealize cleverness as a characteristic *par excellence* that transforms what they spend all of their time doing—creating technology and fixing problems in a great maelstrom of complexity and confusion—into an activity of shared and especially sensual pleasure.

Before extending my theoretical discussion on cleverness and humor, permit me to provide a few examples that are embedded in technical artifacts and one that arose during social interaction. Since much of hacker wit is so technically coded, it is difficult to translate it in any meaningful manner to a lay audience, and I am afraid it might not strike nongeek readers as all that humorous. Analyzing humor, after the fact, is also nearly never humorous, but hopefully it can still be analytically illuminating. I have chosen four examples that are more accessible to a nontechnical audience and supply at least a taste of the types of jokes common among hackers.

Peppering technical artifacts with clever quips occurs quite commonly in hacker technical naming conventions or documentation. For instance, most software applications also come with some sort of description of their purpose and functionality. Jaime Zawinski, the author of a software application called BBDB, portrays his creation via a smattering of jokes (most software applications include a description of their functionality):

BBDB is a rolodex-like database program for GNU Emacs. **BBDB** stands for **Insidious Big Brother Database**, and is not, repeat, *not* an obscure reference to the Buck Rogers TV series.

It provides the following features:

Integration with mail and news readers, with little or no interaction by the user:

easy (or automatic) display of the record corresponding to the sender of the current message; automatic creation of records based on the contents of the current message; [. . .]

While the “Insidious Big Brother Database” is an obvious and playful recognition of the common hacker mistrust of governmental authority, the Roger’s reference is more esoteric and thus only a small fraction of hackers will

be able to decipher it: those hackers who have watched the television series. With the cue offered in the documentation, those hackers will immediately catch the author's irony (that *this* is a reference to the show) and recognize that BBDB refers to the series' pint-size robot Twiki, whose preferred mode of communicating is a noise that sounds remarkably like "B-D-BBBB-D."

I am particularly fond of the next example contained in the manual (usually shortened to "man page") for Mutt, a popular email client among geeks. Man pages provide documentation and are included with almost all Unix systems. They typically follow a strict standard for conveying information about the program by designating a set of common categories under which programmers provide detailed information about the software, such as the name, synopsis, description, options, files, examples, and authors. One important category is bugs, where authors list the problems and glitches with the software. (Software can have a number of bugs and glitches yet still work. The bug category gives you a sense of what these glitches are and when they will emerge.) The Mutt man page exploits the fact that the word *mutt* can mean a mongrel dog. Notice the category of bugs:

NAME

mutt—The Mutt Mail User Agent

SYNOPSIS

mutt [-nRyzZ] [-e cmd] [-F file] [-m type] [-f file] [. . .]

DESCRIPTION

Mutt is a small but very powerful text based program for reading electronic mail under Unix operating systems, including support color terminals, MIME, and a threaded sorting mode.

OPTIONS

—A alias

An expanded version of the given alias is passed to stdout.

—a file

Attach a file to your message using MIME. [. . .]

BUGS

None. Mutts have fleas, not bugs.

FLEAS

Suspend/resume while editing a file with an external editor does not work under SunOS 4.x if you use the curses lib in /usr/5lib. It does work with the S-Lang library, however.

Resizing the screen while using an external pager causes

Mutt to go haywire on some systems. [. . .]

My last example of this subtle integration of wit in a technological artifact comes in the form of a warning message. Many software programs and related artifacts are accompanied by dramatic warnings that appear during configuration. These are intended to alert the user that its integration into some software systems may produce unanticipated, drastic, and completely undesirable results (like breaking multiple parts of your software system that took five weeks to get “just right”). Often this happens because a piece of software is still experimental and riddled with bugs. The following help message is available in the 2.6 branch of Linux kernel configuration and refers to the RAID-6 device driver, which at the time was still under development and hence buggy:

WARNING: RAID-6 is currently highly experimental. If you use it, there is no guarantee whatsoever that it won't destroy your data, eat your disk drives, insult your mother, or re-appoint George W. Bush

These three examples demonstrate that hackers value subtlety and irony of presentation. Hackers discretely embed nuanced, clever and frequently nonfunctional jokes within what are otherwise completely rational, conventional statements of function. Yet hackers never use jokes to undermine the functionality or trustworthiness of the code or documentation. These technical artifacts are judged seriously by geeks. The presence of wit only works to add to the value of the rational content by reminding the user that behind these highly systematized genres, there is a discriminating and creative individual.

Other instances of hacker wit occur in person and are less subtle. For example, at a security conference in 2001, Peiter Zatko, aka “Mudge,” a computer security researcher, professional, and hacker (once part of the famous hacker association L0pht Heavy Industries), arrived in a terrycloth bathrobe to present on a panel on PDAs. This bold sartorial statement distinguished him from his nonhacker colleagues, also security researchers, but scientists. It prioritized hacker over scientific identity. Mudge’s attire, however, performed a problematic public-private breach in the context of his talk, which focused on the changing use patterns of PDAs. “PDAs were designed for personal use, but are now being used more for business,” Zatko said. “There’s a security boundary that’s being crossed.”⁶ Zatko’s robe embodied his argument that the shift amounted to a breached security boundary: PDAs should not be used for sensitive, private data.

Though humor is found worldwide, instances like the ones just described are fruitful to the anthropologist because of their cultural particularity. As this playful practice usually induces laughter—a state of bodily affect that enraptures an audience—humor can potentially produce forms of collective awareness and shared sociality. Given these two properties, we can define humor, in the most general terms, as a play with form whose social force lies

in its ability to accentuate the performer, and which at times can work to delineate in-group membership.

Apart from this, the meaning of humor is otherwise quite culturally specific. The power to enrapture and entangle people can lead to entirely contrary social effects. In certain cases and types of groups, joking can establish and maintain hierarchies as well as social boundaries by, say, delineating social roles (Gusterson 1998; Mulkay 1988; Radcliffe-Brown 1952). In other cultural and historical contexts, humor pushes the envelope of conceptual boundaries in ways that may be fleeting and frivolous (Douglas 1975), or politically subversive (Bakhtin 1984; Critchley 2002). In other words, because the effect, purpose, and even form of humor are deeply context dependent, culturally inflected, and historically moored, it is a useful tool for analyzing broader forms of cultural meaning.

Among hackers, humor is a distilled and parsimonious instantiation of the adoration of cleverness. It is an especially effective way of enacting hackers' commitment to wittiness precisely because, unlike the objects of hacker technical production, joking has no strict functional utility, and speaks to the inherent appeal of creativity and cleverness for their own sake. Joking is a self-referential exercise that designates the joker as an intelligent person and cleverness as autonomously valuable.

It bears repetition that hackers draw on their pragmatic ability to manipulate form to engage in this type of joking. These two elements—being good at hacking and valuing cleverness for its own sake—exist in a tight and productive symbiosis, a mutually reinforcing relation that produces an abundance of humor among hackers. There is a close kinship between hacking and humor.

Insofar as humor is tethered to the moment of its utterance, it exudes an auric quality of spontaneous originality (Benjamin [1936] 2005), which among hackers authenticates the self as a distinctive and autonomous individual. Humor is one of the starkest expressions of the hacker "ideal self." By telling jokes, hackers externalize what they see as their intelligence and gain recognition from technically talented peers.

Like hacker technological production, humor also works to implicitly confirm the relational self who is joined to others by a shared domain of practice, and a common stock of implicit cultural and explicit technical knowledge. Recall that many jokes, such as technical Easter eggs, are received as pleasurable gifts. They not only break the monotony and grind of sitting at the computer, usually for hours a day as one churns out code or resolves problems, but also remind hackers of their shared experiences. "One might say that the simple telling of a joke," writes philosopher Simon Critchley (2002, 18), "recalls us to what is shared in our everyday practices. [. . .] So, humor reveals the depth of what we share." If humor creates fine distinctions, it also levels the ground, because in the very moments of laughter, hackers implicitly recognize and celebrate the shared world of meaning in which they work. After all, like many instances of joking, much of hacker

humor is so culturally coded (which here means technically inflected) that the only people who can routinely receive, and as such appreciate, their wit are other hackers. One must rely on the acknowledgment and judgment of those who can appreciate the performance of wit, because they share at least some of one's implicit values, explicit technical knowledge, and standards of creative evaluation.

To the extent that everyone enjoys laughter, humor functions much as a communal gift—the performance of which beckons others to follow suit. Indeed, once one hacker starts joking, many others will dive in. It also breaks the monotony and eases the strains of hacking, and so can also be seen as a mechanism to preserve hackers' humanity (and sanity) in the face of the merciless rationale of the machine they engage with everyday. When humor is woven into the actual code or technical artifacts animating the machine, it brings otherwise-mechanic language directly and unmistakably into the realm of human communication.⁷ Once part of the apparatus of human communication, humor powerfully confirms a shared mode of being in the world; in other words, it affirms a lifeworld. The very expression of humor is seen as proof that despite their physical dispersion and sense of independence, hackers nonetheless cohabit a shared social terrain built around a lifelong intimacy with technology and technical thinking—one they have come to celebrate.

Among hackers, humor functions in multiple capacities and undoubtedly reflects the value they place on productive autonomy as well as the drive to perform cleverness. Much of their humor is ironic—a play with form. Its purpose is to arrive on the scene of the joke (often a technical object) unexpectedly. This is also the ideal nature of a great hack, insofar as it should surprise other hackers into a stance of awe. Humor, as Douglas (1975, 96) reminds us, is “a play upon form that affords an opportunity for realizing that an accepted pattern has no necessity.” This definition bears a striking resemblance to the pragmatics of hacking; hackers are constantly playing on form, revealing that there is no single solution to a technical problem. And although hackers claim it is abominable to reinvent the wheel, in practice, they are constantly doing so as they follow their own creative instincts and visions.

In its ability to concurrently accentuate inclusiveness and exclusiveness, and make and level hierarchies, humor shapes conventions of sociality, ideals of creativity, and hackers' attitudes toward one another and outsiders. Now let's take a closer look at the tension between individuality and collectivism to which humor so delectably points us.

COMMUNAL POPULISM AND INDIVIDUAL ELITISM

If hacker pragmatics oscillate between a respect and disrespect for form, hacker sociality alternates between communal populism and individual elitism. Largely by way of F/OSS philosophy, hackers laud mutual aid and

cooperative reciprocity as vital features of technical collaboration. They spend an inordinate number of hours helping each other. But there is also an elitist stance that places an extremely high premium on self-reliance, individual achievement, and meritocracy.⁸ While the populist stance affirms the equal worth of everyone who contributes to an endeavor, the elitist one distributes credit, rewarding on the basis of superior accomplishment, technical prowess, and individual talent—all judged meticulously by other hackers. Hackers will spend hours helping each other, working closely together through some problem. Yet they also engage in agonistic practices of technical jousting and boasting with peers, and in turn, this works to create hierarchies of difference among this fraternal order of “elite wizards.” Ullman (1997, 101) condenses this tension into few words: “Humility is as mandatory as arrogance.” The line between elitism and populism is not simply an intellectual afterthought posed by me, the anthropologist, but also a living, relevant, affective reality discussed and dissected by hackers.

This duality arises during the course of their work, and is openly discussed in ethical and pragmatic terms. On the one hand, hackers speak of the importance of learning from others and construe knowledge production as a collective enterprise—and this rhetoric is frequently matched in practice by truly generous and copious acts of sharing. In any given minute of the day, I can log into one of the developers’ IRC channels, and there will be some developers asking a question, getting an answer, and giving thanks, as this example illustrates:

```
<zugschklus> does anybody know how to configure sound in KDE4? [KDE
    is a desktop environment.]
<pusling> Zugschklus: in systemsettings
<zugschklus> pusling: applications => settings?
<kibi> but AFAICT [“as far as I can tell”], what you have in svn helped me
    build various thingies against libqt4-dev and friends.
<pusling> Zugschklus: computer > s-ytemsettingsns
<pusling> KiBi: I think qt4 is now waiting in new.
<zugschklus> pusling: that part only has home, network, root and trash.
<kibi> pusling: oh, ok :(
<pusling> Zugschklus: do you have the package systemsettings installed?
<pusling> KiBi: so if you have special contacts to ftp team, feel free to use
    them.
<kibi> pusling: yep, seen it.
* kibi can try
<kibi> Ganneff: mhy: ^^^ if you want to help kfreebsd-* folks get more
    packages built, fast-tracking qt4-x11 would really be great. Thanks for
    considering. :)
<zugschklus> pusling: no, that was missing. thanks.
<pusling> Zugschklus: you then probably want to make sure you have
    kde-minimal installed.
```

Guiding this practice is the idea that the free software project represents an endeavor that far exceeds any single person's efforts, and thus everyone's contribution is highly regarded, whether it involves filing a bug report or offering a significant, large-scale innovation.

On the other hand, hackers often express a commitment to self-reliance, which can be at times displayed in a quite abrasive and elitist tone. The most famous token of this stance is the short quip "Read the Fucking Manual" (RTFM). It is worth noting that accusations or RTFM replies are rarer than instances of copious sharing. Let me provide two examples of RTFM in action. In the first, "Error" drops into a new channel after asking a question in the #perl channel, where he got a prompt RTFM, after which everyone else went back to discussing the band Metallica. In this channel, they did not offer an RTFM but instead suggested going to the #metallica channel, which in this case, is a joke [IRC channels are designated by #name-of-channel].

```
<813-error> i ask a question in #perl and get RTFM and they go back to
    talking about metallica [ . . . ]
<813-error> d Match digit character <that would be numbers right?
* C4 knows nothing of perl
<modem> same here :/
<modem> ask in #metallica
```

The second example does not contain a joke but rather only a rebuke in the form of RTFM:

```
<karsten> Ace2016: alsamixer / aumix are interactive ncurses programs
<ace2016> so?
<karsten> Ace2016: You may be able to steer 'em w/ stdin as well.
<ace2016> can't they accept a command like aumix—volume decrease
    10% or something like that?
<karsten> Ace2016: RTFM
<karsten> Ace2016: Which is to say, I don't know. Go look yourself.
```

These two poles of value reflect pervasive features of hacker social and technical production as it unfolds in everyday life. It only takes a few days of following hacker technical discussion to realize that many of their conversations, whether virtual or in person, are astonishingly long question-and-answer sessions. To manage the complexity of the technological landscape, hackers turn to fellow hackers (along with manuals, books, mailing lists, documentation, and search engines) for constant information, guidance, and help. Unlike academics—who at times religiously guard their data or findings until published, or only circulate them among a small group of trusted peers—hackers freely share their findings, insights, and solutions. More than ever, and especially in the context of free software projects, hackers see their productive mutual aid as the underlying living credo driving free software philosophy, and the methodology of collaboration and openness. Hackers

maintain that this mode of production is responsible for better hackers and better technology.

Alongside technical question-and-answer sessions, developers dissect the ethics of their labor. For example, on a Debian mentors' mailing list discussion, one aspiring hacker asked, "How did you get from the middle ground to guru-dom?? Or is the answer that if I need to ask, I will never be a hacker!???" A developer known for his humility and prolific contributions to the Debian project offered a lengthy response—a small section of which I quote below. In highlighting the importance of sharing, learning for others, and even coding for others, he affirms a populist stance, commonly expressed by many Debian developers:

One other inspiration for me has been helping people. Though this has been spottier than I could hope, I do from time to time end up doing some program entirely because I can see other people need it. This tends to broaden experience a lot. Things like writing programs for an unfamiliar platform (microsoft), in a unfamiliar language (spanish), and needing to work closely with the people who would use it, cannot help but change how you look at things. My most valuable experiences in this area have been when I had direct contact with the people who would be using the program, rather than just noticing a hole and deciding I would try to go fill it like you did.⁹

Here he accords weight to pedagogy and collective interdependence in which learning from and even coding for others is a crucial component of technical progress as well as self-development.

During this discussion, though, other developers stressed the importance of independence by urging the questioner to follow his own particular interests necessary to cultivate technical independence. For example, one developer offered the following advice:

I think you made two mistakes. [. . .] The first is looking to other people for problems to be solved. You'll never find the inspiration in solving problems that don't affect you. Since you don't feel the itch, you don't get much satisfaction from the scratch. Speaking for myself, I picked up a programming manual for my first computer and started reading; well before I was finished, I had two dozen ideas for programs to write. Those programs and their spinoffs kept me busy for a couple of years, and I loved it. Second, when an itch hits you, don't research to see if someone has already solved the problem. Solve it yourself. Mathematical texts aren't filled with answers right beside the problems; they teach you by making you work out the answers yourself.¹⁰

Simply in marking the question as misguided (because he looks to other people for problems to be solved), this developer asserts the value of self-determination. The original question violated what is the predominant

(though not unquestioned) norm of self-sufficiency among developers—a norm that captures the isolated and individualistic phenomenology of much of their labor, which for many hackers commenced in childhood.

One developer, in answering a question I had about the significance of free software, expressed this stance of technical self-determination and independence in the following terms: “If I am cut off from the world, then in theory then I can maintain my own domain over software. I don’t have to depend on anyone else; I can do it all myself. If my computing environment diverges from everyone else’s in the world, I can still keep on going.” This commitment to a fully autonomous, sovereign self who shuns any obvious signs of dependence on others is a common trait among developers. Given this mode of laboring, it is not surprising that hackers place so much emphasis on autonomy and self-sufficiency—qualities that are congenial to many hackers as they resonate so strongly with the very experience of intense periods of isolated labor.

Yet this statement of independence is based on a hypothetical scenario of being “cut off from the world”—something even this developer qualifies as unlikely.¹¹ In most practical instances, hackers are constantly plugged in, connected through various technical structures of communication. They work together as well as in complete isolation, for personal and joint public projects. Software theorist Matthew Fuller (2008, 5) describes how the freedom of coding gets subsumed by a host of conditions that always lay outside code proper: “Computation establishes a toy world in conformity with its axioms, but at the same time, when it becomes software, it must, by and large [. . .] come into combination with what lies outside of code.”

Generally, the need to both work alone and with others is experienced free of contradiction, because the two needs are complementary and readily recognized as such by most hackers. To take another example from the mailing list discussion on what transforms a mediocre hacker into a great one, a developer captured this duality by describing how hacking tacks between two productive extremes—the collaborative and individual—that are not mutually exclusive:

Creating a linux distribution is a group activity, but creating art is fundamentally a solitary, private experience. Turn off your internet connection; sit in a dark room, with nothing but the glow of a monitor, the warmth and hum of your computer, and the ideas will flow: Sometimes a trickle, sometimes a torrent.¹²

These two modes can clash, however. This is powerfully signaled through a form of stylized boasting that contrasts one’s intelligence with the idiocy of “mere users” of software. While users of free software are often lauded as essential participants in the broader project of technical development because they provide insightful queries and bug reports (and also are seen as possible future hackers), at other times they are deemed second-class

technical citizens.¹³ This designation is frequently accomplished through the only way in which socially uncomfortable topics can be routinely discussed: by joking. On developer IRC channels, hackers playfully mock users. By complaining about stupid questions and queries, hackers depict users as less worthy contributors for lack of technical proficiency, or may display their complaints elsewhere, such as including humorous email signatures that taunt the wider universe of (l)users.¹⁴ This condescending attitude is aptly and humorously conveyed in the following quote from a developers' email signature, originally formulated by Richard Cook: "Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

Users, though, are by no means the only type of persons subject to the humorous or more vitriolic accusation of technical incompetence. If a question is posed in the wrong register, is seen as uninteresting, or the answer can easily be found elsewhere, nearly anyone from a mere user to a "skilled" developer can receive the stylized and semihumorous RTFM rebuff. Stated on a hacker site with vivacious bite:

[RTFM] is a big chromatic dragon with bloodshot beady eyes and fangs the size of oars. RTFM is me screaming at you as fireballs come out of my mouth to get off your precious no-good tush, march down to the local bookstore or MAN page repository, and get the eff off my back because I'm trying very hard to get some freakin' work done. Jeez.¹⁵

If you are better informed with the knowledge that there is "NO MANUAL," you can quickly defend your honor (i.e., intelligence) by pointing this out and gain substantial respect if you take it on yourself to write documentation. Otherwise, you will have to swallow the rebuke, google for the information, and hope for a better response next time (or simply find another IRC channel and ask elsewhere).

A complicated set of norms and conventions surround asking for help. They depend on the social context of the query and who is asking the question. For example, once someone has garnered a certain amount of trust and respect, they can usually get away with asking what is seen as a nonchallenging, uninteresting question. Developers who have not yet established trust will frequently get immediate help if the question is seen to be a challenge, but a basic questions will raise immediate eyebrows, especially among strangers or members who are technically unvetted, and therefore must maneuver with more caution and tact.

RTFM is a comedic, though stern, form of social discipline. It pushes other hackers to learn and code for themselves as well as affirms that effort has been put into documentation—an accessible form of information that benefits the group—but in a way that still requires independent learning. Many users and developers complain of the lack of adequate documentation for

free software, faulting the tendency of some developers to exist in technical silos, “selfishly” coding only for themselves, and not attending to the needs of other users and developers by writing technically boring but necessary documentation. Many developers also note how the lack of extensive documentation can hinder collaborative technical work. Thus, when someone asks for information that in fact does exist in documentation, they often receive the RTFM rebuke, whose subtext says, “go learn for yourself, especially since others have already put in the work (i.e., documentation) to make this happen.” To give too much aid is to deny the conditions necessary for self-cultivation.

The use of RTFM is disputed as well. During the 2005 Debian project leader election, the issue of documentation erupted during a mailing list debate. The subject of RTFM rebukes was broached directly. One developer argued that RTFM is an inflammatory, unproductive response to newcomers who may find themselves confused and overwhelmed with Debian’s technical and procedural complexity. To make new users feel welcome, he believed that developers should refrain from replying with RTFM, and instead focus their efforts on achieving greater transparency and accessibility. While debating a Debian project leader candidate who had been with the project for years, he conveyed this commitment to corporate populism when he stated:

You know a lot about the project [and its project internals], so it’s all obvious to you. There are people among us who have not been part of Debian since 1.1, but who would like to know more about what’s happening behind the curtains. However, those people are often told to RTFM or go spend time in the code, or just not taken seriously.¹⁶

In response, the Debian project leader candidate defended the general use of RTFM, concisely enunciating the value of self-determination:

When the code is public, `rtfm` is the proper answer. One might add “document it properly afterwards” as well, though. When the data is available as well, that’s best. Some data cannot be made available for legal or other binding obligations (new queue, security archive). If you feel that some bits are missing and need to be documented better, point them out and get them documented better, maybe by doing it on your own. I know a lot about the project because I’ve been involved in many parts. Other developers are involved in many parts as well. Some other developers mostly whine about not being involved without trying to understand. *sigh*¹⁷

In other words, if the requested information is public, it is incumbent on the developer to seek it out, and if unsatisfied with the current state of accessibility, then the next logical step is to make it happen—by yourself. If one does, one can display self-determination and self-development, the vehicles by which to gain the respect of accomplished peers on a similarly paved technical path.

If the subject of elitism erupts on mailing list discussions over project organization, a form of stylized boasting, taunting, cajoling, and elitist disdain is also frequently performed through code. Here I provide two examples. And again note how humor is used in both, to some degree working to soften the abrasive tone of these messages.

The first one is written in the style of an “I-can’t-believe-how-idiotic-this-problem-I-have-to-solve-is rant” that disparages a bug in the Emacs email reader. Before addressing the significance of his code, permit me defer to the coder, Karl Fogel, to explain the context of the problem and the technical nature of his solution:

Basically, the mailreader insisted on colorizing my mail composition window, even though I tried every documented method available to ask it not to do that. In desperation, I finally wrote code to go “behind the back” of the mailreader, and fool it into thinking that it had already done the colorization when it actually hadn’t.¹⁸

The comments open with a statement of disbelief; take note of the naming of the variable, which I highlight in bold and italics:

```
;; I cannot believe what I have to do to turn off font locking in mail
;; and message buffers. Running “(font-lock-mode -1)” from every
;; possibly relevant gnus-*, mail-*, and message-* hook still left
my
;; reply buffers font-locked. Arrrgh.
;;
;; So the code below fools font-lock-mode into thinking the buffer
is
;; already fontified (so it will do nothing—see
;; font-lock.el:font-lock-mode for details), and then makes sure
that
;; the very last thing run when I hit reply to a message is to turn
;; off font-lock-mode in that buffer, from post-command-hook.
Then
;; that function removes itself from post-command-hook so it's
not run
;; with every command.
(defun kf-compensate-for-fucking-unbelievable-
emacs-lossage ()
(font-lock-mode -1)
(remove-hook
 'post-command-hook
 'kf-compensate-for-fucking-unbelievable-emacs-lossage))

(add-hook 'font-lock-mode-hook 'kf-font-lock-mode-hook)
(defun kf-font-lock-mode-hook ()
```

```
(if (or (eq major-mode 'message-mode)
      (eq major-mode 'mail-mode))
    (progn
      (make-local-variable 'font-lock-fontified)
      (setq font-lock-fontified t)
      (add-hook 'post-command-hook
'kf-compensate-for-fucking-unbelievable-emacs-
lossage)
    )))
```

By opening the comments with “I cannot believe what I have to do” and ending with “Arrrgh,” he signals the fact that this sort of trite problem is so idiotically banal, it should have never appeared in the *first place*. Fixing it is a waste of his superior mental resources. Lest there be any ambiguity as to what the author really thought about the code, he continues to drive the point home in his rant by naming the variable with an unmistakably deliberate insult: “compensate-for-fucking-unbelievable-emacs-lossage.”

During the course of my early research, I was shocked at the disjoint between the in-person real-world “codes of conduct” and the “codes of software conduct.” Nothing about this coder’s personality, who I got to know very well over the course of five years, would indicate such haughty declarations. There is no need for such an indication because these enunciations are rarely a matter of innate psychology. Instead, these are conventionalized statements by which hackers declare and demarcate their unique contribution to a collective endeavor. They also represent culturally sanctioned mechanisms for judgment.

Fogel’s code is an apt example of “face work” (Goffman 1967, 5)—when a hacker is sanctioned to perform a “line,” which is the “pattern of verbal and nonverbal acts by which he expresses his view of the situation and through this his evaluation of the participants, especially himself.” Within such a presentation, hackers can declare and demarcate their unique contribution to a piece of software while at the same time proferring technical judgment. One may even say that this taunting is their informal version of the academic peer-review process. In this particular case, Fogel is declaring the code he patched as an utter failure of the imagination.

Because these insults are critical evaluations of work, if hackers dare to make such pronouncements, they also have to make them technically clever enough to be accepted as accurate critiques. After a declaration is made, a hacker should be ready to enter the arena of competitive jousting. If one hacker judges some piece of code, it is almost guaranteed that another hacker may reply with chutzpah of their own, often in humorous guise.

The second example demonstrates this type of competitive play of technical volleyball, a form of “antiphony” of “call and response” common to jazz poetics (Gilroy 1993, 78). While jazz poetics may seem strange to apply

to hacking, I will expand on this connection later when addressing hacker notions of creativity. First, let's take a closer look at this portion of the code that shows the use of boasting to induce a response (I have highlighted the relevant section in *italics*):

```
/* Prime number generation
   Copyright (C) 1994 Free Software Foundation
```

```
This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2, or
   (at your option) any later version.
```

```
This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PUR-
   POSE. See the GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */
```

```
#include <stdlib.h>
#include <string.h>
```

```
/* Return the next prime greater than or equal to N. */
int
nextprime (int n)
{
    static int *q;
    static int k = 2;
    static int l = 2;
    int p;
    int *m;
    int i, j;

    /* You are not expected to understand this. */

    if (!q)
    {
        /* Init */
        q = malloc (sizeof (int) * 2);
        q[0] = 2;
        q[1] = 3;
    }
}
```

Derived from the FSF's Hurd development project, which is its kernel project, the code is a prime number generator. Programmers have told me that the technical details are fairly intricate, so I refrain here from providing an explanation of the actual mechanics of the code, and for the sake of analysis it is not necessary. The important element is the author's comment: *“/* You are not expected to understand this. */”* It reveals how boasting is an open invitation to engage in technical jousting—a playful taunt that explicitly encourages the technical comeback that proves the expectation wrong.

The author's intentions are pretty clear in the code, but here is his retroactive explanation: “At this point I offered the function as a challenge to Jim Blandy. [. . .] That the function was intended to produce prime numbers was never hidden; the challenge was to explain its technique.” Blandy took the call to technical arms and responded with his own exegesis of the algorithm. When the original author of the prime number function updated the code, he changed the taunt to *“/* See the comment at the end for an explanation of the algorithm used. */”* and at the end of the code, stated, “Jim produced the following brilliant explanation” and included it within the code (and again, I have indicated the relevant section in italics).

```
/* Prime number generation
C [ . . . ]
#include <stdlib.h>
#include <string.h>
/* Return the next prime greater than or equal to N. */ [ . . . ]
/* See the comment at the end for an explanation of the algo-
rithm
used. */
if (!q)
{
    /* Init */ [ . . . ]
    * [This code originally contained the comment “You are
not expected to understand this” (on the theory that ev-
ery Unix-like system should have such a comment some-
where, and now I have to find somewhere else to put it).
I then offered this function as a challenge to Jim Blandy.
At that time only the six comments in the function and
the description at the top were present.
Jim produced the following brilliant explanation.]
```

The static variable `q` points to a sorted array of the first `l` natural prime numbers. `k` is the number of elements which have been allocated to `q`, `l` $\leq$ `k`; we occasionally double `k` and realloc `q` accordingly to maintain this invariant.

The table is initialized to contain a few primes (lines 26, 27, 34-40). Subsequent code assumes the table isn't empty.

```

When passed a number n, we grow q until it contains a prime
>= n
(lines 45-70), do a binary search in q to find the least prime >=n
(lines 72-84), and return that. [ . . . ]

```

If some hackers are ready to pounce on what they deem as the idiocy of others, they are also as likely to dole out recognition where they see fit. Hence, even while hackers are on a path toward *self-development*, this self-fashioning is intimately bound to others, not simply because of a love of tinkering or the dependence derived from collaboration, but because any meritocratic order based on expertise fundamentally requires others for constant evaluation as well. Hackers use the path of humor, taunt, jousting, boasting, and argument for such expressions of technical taste and worthiness, and in the process, cultivate themselves as expert hackers.

JUST FREEDOM

Given hackers' proclivity for expressing cleverness, acknowledgment that they build on the shoulders of giants, need to garner recognition from others, and dual penchant for lauding populist collectivism and individual self-determination, what might these attributes reveal about hacker notions of personhood, creativity, and authorship?

It is not surprising that in so much of the literature, hackers are treated as quintessentially individualistic (Levy 1984; Turkle 1984). "The hacker," Turkle (1984, 229) writes, "is the defender of idiosyncrasy, individuality, genius and the cult of individual." Some authors argue that this individualism is a close variant of a politically suspicious libertarianism (Borsook 2000). Hackers are perpetually keen on asserting their individuality through acts of ingenuity, and thus these statements are unmistakably correct. In most accounts on hackers, however, the meaning of this individualism is treated as an ideological, unsavory cloak or is left underspecified. Why the pronounced performance of individualism? What does it say about how hackers conceptualize authorship? What tensions does it raise?

Because hackers do not automatically treat software as solely derivative of one laboring mind but instead see it as derivative of a collective effort, the constant drive to perform ingenuity reflects the formidable difficulty of claiming discrete inventiveness. After all, much of hacker production is based on a constant reworking of different technical assemblages directed toward new purposes and uses—a form of authorial recombination rarely acknowledged in traditional intellectual property law discourse.

Because of the tendency, especially now more than ever, for hackers to recognize the reality of collaboration, it may seem that they are moving toward the type of politics and ethics of authorship that flatly reject the ideal

of individualism altogether—a rejection famously explored in the works of Roland Barthes, Michel Foucault, and Dick Hebdige. In the F/OSS domain, hackers have not moved, even an inch, to decenter the persona of the author in the manner, say, most famously exemplified by Barthes, who in 1967 sought to dethrone the authority of an author: “To give a text an Author is to impose a limit on that text, to furnish it with a final signified, to close the writing.”¹⁹

Instead, among hackers the authorial figure seems to speak slightly louder, clamoring for and demanding credit and recognition, established through oral histories of software or etched into the infrastructure of production. Hackers record contributions and attributions in common files included with source code, such as the Authors and Contributors files (Yuill 2008). This archival drive helps partially explain why certain hackers can also receive the legendary status they do. This everyday discourse and inscription develops a shared historical awareness about who contributed what—one that brings attention to the conditions of production or the nature of the contribution. Furthermore, accountability and credit are built into many of the technical tools that facilitate collaboration, such as CVS and Subversion—software systems used to manage shared source code. These systems give developers the ability to track (and potentially revert to) incremental changes to files and report the changes to a mailing list as they are made, and are often used concurrently by many developers. Since developers all have accounts, these technologies not only *enable collaboration* but also provide *precise details of attribution*. Over time, this record accumulates into a richly documented palimpsest. Though individual attribution is certainly accorded, these technological palimpsests reflect unmistakably that complicated pieces of software are held in place by a grand collaborative effort that far exceeds any one person’s contribution.²⁰

In contrast to many accounts on authorship, I find that a short description about the aesthetics of jazz and its “cruel contradiction” is eerily evocative of the hacker creative predicament:

There is a cruel contradiction implicit in the art form itself. For true jazz is an art of individual assertion within and against the group. Each true jazz moment (as distinct from the uninspired commercial performance) springs from a context in which each artist challenges all the rest, each solo flight, or improvisation, represents (like the successive canvases of a painter) a definition of his identity: as individual, as member of the collectivity, and as link in the chain of tradition. Thus, because jazz finds its very life in an endless improvisation upon traditional materials, the jazzman must lose his identity even as he finds it. (Ellison 1964, 234; quoted in Gilroy 1993, 79)

Among hackers this cruelty, this difficulty in establishing discrete originality, is in reality not so cruel. It is treated like any interesting problem: an enticing

hurdle that invites rigorous intellectual intervention and a well-crafted solution within given constraints. Hackers clearly define the meaning of the free individual through this very persistent inclination to find solutions; they revel in directing their faculty for critical thought toward creating better technology or more sublime, beautiful code. The logic among hackers goes that if one can create beauty, originality, or solve a problem within the shackles of constraints, this must prove a *superior* form of creativity, intelligence, and individuality than the mere expression of some wholly original work.

Not every piece of technology made by hackers qualifies as a hack. The hack is particularly the “individual assertion within and against the group” (Ellison 1964, 234), which may be easily attached to an individual even though it is still indebted to a wider tradition and conversation. Hackers certainly engage in a creative, complex process partially separated from hierarchy, enfolding a mechanics of dissection, manipulation, and reassembly, in which various forms of collaboration are held in high esteem. Much of their labor is oriented toward finding a good enough solution so they can carry forth with their work. But their form of production is one that also generates a practice of cordial (and sometimes not-so-cordial) one-upping, which simultaneously acknowledges the hacker’s technical roots and yet at times strives to go beyond inherited forms in order to implement a better solution. If this solution is achieved, it will favorably reveal one’s capacity for original, critical thought—the core meaning of individuality among hackers.

Hackers recognize production as the extension or rearrangement of inherited formal traditions, which above all requires access to other people’s work. This precondition allows one to engage in constant acts of re-creation, expression, and circulation. Such an imperative goes against the grain of current intellectual property law rationalizations, which assume that the nature of selfhood and creativity is always a matter of novel creation or individualized inventive discovery.

Among F/OSS hackers, the moral economy of selfhood is not easily reducible to modern “possessive individualism” (Graeber 1997; Macpherson 1962). Nor does it entirely follow the craftsperson or the stand-alone romantic author figured by intellectual property jurisprudence but rather evinces other sensibilities that point to competing liberal concepts of individualism and freedom. While hackers envisage themselves as free and rational agents, in the context of free and open-source hacking, most hackers place less emphasis on the freedom to establish relations of property ownership and exchange. Instead, they formulate liberty as the condition necessary for individuals to develop the capacity for critical thought and self-development.²¹

While the hacker interpretation of labor, creativity, and individuality strays from influential liberal understandings of personhood—possessive individualism—it does not represent a wholly novel take on these themes.

It aligns with the type of person presupposed in free speech theory, perhaps most lucidly in Mill's writings, which influenced the shape, content, and philosophy of free speech jurisprudence as it now exists in the United States (Bollinger and Stone 2002; Passavant 2002). Mill, influenced by the Romantic tradition (Halliday 1976), defines a free individual as one who develops, determines, and changes their own desires, capacities, and interests autonomously through self-expression, debate, and reasoned deliberation (Donner 1991). It is a vision that fuses utilitarian and romantic commitments, and is built on the idea of human plasticity and development—the ability of the self to grow and develop through creative expression, mental activity, and deliberative discussion, usually by following one's own personally defined path. As Wendy Donner argues, this form of liberal self-cultivation also requires the establishment of standards by which to judge the development of the human faculties. Mill's "transformed conception of utility necessitates a new method of value measurement which relies heavily on the judgment of competent agents," writes Donner (1991, 142), "and thus essentially rests on a doctrine of human development and self-development." What is notable is how Mill ([1857] 1991, 93) contends in his famous *On Liberty* that an individual must follow their own path of development, because "persons [. . .] require different conditions for their spiritual development." Even if this Romantic inclination prioritizes the individual, one can only develop the critical faculties along with moral and aesthetic standards via a process of training and open-ended argumentation in debate with other similarly engaged individuals.

Much of free software legal philosophy and moral sensibilities bear remarkable similarities to this Millian (and thus also Romantically informed) vision of personhood, self-development, and liberty, although there are differences and specifications tied to hacking's unique relations between persons, labor, and technology. Hackers place tremendous faith in the necessity and power of expressive activity that springs from deep within the individual self—an expression that acts as the motor for positive technical change. Progress depends on the constant expression and reworking of already-existing technology. Thought, expression, and innovation should never be stifled, so long as, many developers told me during interviews, "no one else is hurt"—a sentiment that is part and parcel of Millian free speech theories.

Free software developers have come to treat the pursuit of knowledge and learning with inestimable high regard—as an almost sacred activity, vital for technical progress and essential for improving individual talents. As one software developer observed, "I can use the code for my own projects and I can improve the code of others. I can learn from the code so that I can become a better programmer myself, and then there is all my code out there so that you can use it. It is just freedom." The spirit of this statement is ubiquitous among F/OSS developers. A utilitarian ethic of freedom and

openness is increasingly seen as not only obvious but also indispensable in order to develop the “state of the art.”

For developers, technical expression should always be useful. If it isn’t, it denies the nature of software, which is to solve problems. Yet hackers also place tremendous value on the aesthetic pleasures of hacking, producing technology and software that may not have any immediate value but can be admired simply on its own elegant terms—as a conduit for personal self-expression.

Over years of coding software with other developers in free software projects where discourses about liberty run rampant, many developers come to view F/OSS as the apex of writing software, as we will see in the next chapter. It has, they say, the necessary legal and material features that can induce as well as fertilize creative production. In contrast to the corporate sphere, the F/OSS domain is seen as establishing the freedom necessary to pursue *personally* defined technical interests in a way that draws on the resources and skills of other individuals who are chasing down their own interests. In other words, the arena of F/OSS establishes all the necessary conditions (code, legal protection, technical tools, and peers) to cultivate the technical self and direct one’s abilities toward the utilitarian improvement of technology. While many developers enjoy working on their corporate projects, there is always a potential problem over the question of sovereignty. One developer told me during an interview that “managers [. . .] decide the shape of the project,” while the F/OSS arena allows either the individual or collective of hackers to make this decision instead. F/OSS allows for technical sovereignty.

The hacker formulation of individuality, as the pursuit of one’s interest for the mutual benefit of each other and society, is an apt example of the general characterization of modern individualism as defined, according to Taylor (2004, 20), by “relations of mutual service between equal individuals.” While much of liberal thought understands mutual service in terms of economic exchange, hackers relate to it through the very act of individual expression and technical creation—the only sound ways to truly animate the uniqueness of one’s being.

CONCLUSION

As noted in the previous section, even though hackers tend to approach other hackers as equals, they also construct themselves as high-tech cognoscenti creating the bleeding edge of technology. This elitism follows from their commitment to the organizational ideal of meritocracy, a performance-based system that applauds individual skill, encourages respectful competition between peers, and sanctions hierarchies between developers, especially in the F/OSS project to be discussed at length in the subsequent chapter.

The meritocratic ideal, ubiquitous in liberal thought, has particular resonance in the US popular imaginary. The United States is often thought of as a living embodiment of meritocracy: a nation where people are judged on their individual abilities alone. The system supposedly works so well because, as the media myth goes, the United States provides everyone with equal opportunity, usually through public education, to achieve their goals. As such, the hierarchies of difference that arise from one's ability (usually to achieve wealth) are sanctioned by this moral order as legitimate.

In many senses, hackers have drawn from what is still a prevalent trope of meritocracy to conceptualize how they treat one another and self-organize. In his classic account of hackers, Levy (1984, 43) includes this principle as one of the six elements that define the hacker ethic, noting that "hackers should be judged by their hacking, not bogus criteria such as degrees, age, race, or position," in which "people who trotted in with seemingly impressive credentials were not taken seriously until they proved themselves at the console of the computer."

Though written twenty years ago, this commitment to meritocracy still holds undeniable sway in the way F/OSS hackers construct norms of sociality and envision selfhood, not because it exists in the same exact way, but rather because hackers have given it new meaning by organizationally building the institution of the free software project guided by a dedication to meritocracies. Hackers who participate in free software projects routinely asserted that F/OSS projects are run as meritocracies. The doors are open to anyone, they insist; respect and authority are accorded along the lines of superior and frequently individual technological contribution. As we will see in the next chapters, F/OSS hackers may not build perfect meritocracies and yet they are certainly motivated to implement them.

For F/OSS hackers, it is imperative to constantly and recursively equalize the conditions by which other hackers can develop their skills and prove their worth to peers. As part of this equalization process, one must endow the community of hackers with resources like documentation and the fruits of one's labor: source code. The free software hacker does not privatize the source of value created, even those exceptional pieces of code that are undeniably one's own and seen to emerge from sheer technical ability. Within F/OSS, this value is fed back and circulated among peers, thereby contributing to an endowed and growing pool of resources through which other hackers can constantly engage in their asymptotic process of self-cultivation.

This constant recirculation of value is one way in which hackers can explicitly downplay their elitism and display their sound technical intentions to their peers. Their implementation of meritocracy contrasts markedly with the ideal of it in capitalist societies, where the privatization of value is legitimate as long as one generates wealth (or gains other forms of status) through one's personal ability. In fact, numerous issues over who and what are responsible for equalizing the terrain of competition plague

liberal democracies marked by a meritocratic ideal. This leveling is often seen as secured through such avenues as public education. That, in turn, raises questions like, Should capitalist philanthropists (such as John Rockefeller in the past and Gates in the present), individuals, governments, or property tax fund public education? With hackers, these sets of thorny issues are minimized, partially resolved by their constant recirculation of value, notably software and documentation, as well as debates and conflicts over mentorship and helping.

Still, the predominant sentiment is that once knowledge has been released to the collective of hackers, individuals must, on their own two feet, prove their worth by creating new forms of value that can be fed back recursively to the community. If one seeks too much help, this violates the hacker implementation of the proper meritocratic order, and one might be subjected to a stylized rebuff such as the common RTFM.

Among hackers, the commitment to elitism and meritocracy historically has run fairly strong. There is still an ambivalent relationship to elitism and this meritocratic ideal, however, as I will explore in more detail in the next chapter. I will show how those vested with authority on software projects, because of their success, are usually met with some degree of suspicion, and thus jokes and sometimes accusations of cabals run rampant among hackers. This requires them to constantly perform their trustworthiness and demonstrate their good technical intentions to the community at large. I now turn to the institution, the free software project, where technological production unfolds, and where commitments to free speech and meritocracy are further specified under the aegis of a tremendously varied set of ethical practices.