01010000 01001100 01000001 01011001
Play Elements in Computer Programming

Samantha Breslin

This article explores the role of play in human interaction with computers in the context of computer programming. The author considers many facets of programming including the literary practice of coding, the abstract design of programs, and more mundane activities such as testing, debugging, and hacking. She discusses how these incorporate the aesthetics, creative imagination, and game play of programmers. She suggests that the seemingly intractable and unplayful elements of computers, in fact, invite playful responses and actions by programmers and that programmers use play to understand, engage with, and creatively imagine and reconfigure the complexity of computer systems. She concludes that human-machine relationships and computer programming constitute fruitful areas for further play research. **Key words:** computer programming and poetry; debugging; hacking; human and computer relations; play and computers; testing

[01001001:I]ntroduction

In several episodes of *Star Trek: The Next Generation,* the crew of the U.S.S. *Enterprise* encounter the Borg. Members of this cybernetic species live together as a hive mind and attempt to assimilate all others in the galaxy into the "perfection" that is their collective. The Borg represent an ongoing academic and popular discussion about the relationship between humans and cybernetic and computing machines that began around the 1940s (e.g. Bowker 1993; Downey 1998; Haraway 1991; Hayles 1999, 2005; Turkle 2005). One facet of this discussion entails the belief or fear that intelligent machines will replace the human species as the dominant form of life on Earth or, in the case of the Borg, that cybernetic machines will replace all biological life in the galaxy. This worry also appears in other science-fiction works—in the Cybermen in the British series *Doctor Who*, the Cylons in the television series *Battlestar Galactica,* and the Terminator in the *Terminator* movies, to name but a few. The potential "extinction-level risks

American Journal of Play, volume 5, number 3 © The Strong
Contact Samantha Breslin at sdbreslin@mun.ca

to our species as a whole" by developments in fields like Artificial Intelligence and biotechnologies has even been proposed as the focus of a multidisciplinary research center at the University of Cambridge (Price, Rees, and Tallinn 2012). The relationship between humans and machines represents a struggle for dominance and control (as the Borg state to all those they meet: "You will be assimilated, resistance is futile"), though themes of human resistance against the onslaught of emotionless and tireless machines also abound.

Embedded in these nightmares of machine domination and assimilation rests some intriguing ideas about play. If we believe foundational play scholar Johan Huizinga, play suffuses human life. We are *Homo ludens* in the sense that human play "adorns life, amplifies it and is to that extent a necessity both for the individual—as a life function—and for society by reason of the meaning it contains" (Huizinga 1980, 9). On the other hand, most scholars see machines as incapable of play, as lacking imagination and human emotion. The Borg embody the threat we perceive in humans' encounters with machines that individuality, emotion, and play will be overtaken by the interface with technology. In the spirit of Donna Haraway's call to take "*pleasure* in the confusion of boundaries" among humans and machines, I wish to complicate this ominous polarity between playful humanity and calculating machines by exploring human relationships with machines as we experience them today (Haraway 1991). I consider specifically the role of play in the interaction between humans and computers as it relates to computer programming.

We have seen growing interest among those in anthropology and other social-science disciplines in the use of computers for play and gaming (Malaby 2009), and the growth appears especially in online virtual worlds such as *World of Warcraft* (Bainbridge 2010; Boellstorff 2008; Corneliussen and Rettberg 2008; Dibbell 2006; Nardi 2009) and other digital games such as online chess and poker (Desjarlais 2011; Consalvo 2007; Schull 2005). As I show, however, it is not merely the products of computer programming—the programs—that enable forms of play. The process of producing programs itself also often operates through forms of play. Some science and technology scholars mention in passing the playful aspects of programming (e.g. Turkle 2005; Downey 1998), and computer scientists and other programmers often debate the role of aesthetics and creativity in programming practices, both of which I discuss in this article. I propose, however, computer programming as a potential area of investigation for play research on a variety of topics and issues such as the boundaries between work and play, literary forms of play, the creation of imaginary worlds,

gender and programming play, and virtual game play and risk. More generally, the pervasiveness of play in computer programming also suggests that playfulness, as opposed to polarities centering on control and dominance, may provide an alternative and useful way of approaching and exploring human-machine relationships.

I begin with a brief introduction to computer programming and emphasize the ways that the digital nature of computation, as well as the design and structure of computing languages and environments, significantly restrict the interactions between programmers and computers. I then outline my theoretical and methodological approach to understanding the relationship of humans and computers. I consider several ways that computer programmers play with computational restrictions and use computer code as an expressive and creative medium. I start with expressions and discussions of aesthetics related to computer code and follow that with a consideration how the practice of coding itself involves an imaginative process of abstraction. I consider how the daily practices of programmers, such as hacking, testing, and debugging, intertwine with competition, creative exploration, and game play. I argue that play and creativity are key to the human-computer interface. Rather than struggling for dominance and control over digital computers or feeling assimilated into their computational logic, computer programmers use play to understand, engage with, and imagine and reconfigure the complexity of computer systems. The seemingly intractable and distinctly nonplayful mechanistic qualities of computers in fact invite playful responses and actions.

[01000010:B]ackground Information

Computer programming is the practice of creating code, which consists of sequences of instructions that operate computers. In other words, programming is the practice of telling computers what to do. To provide these instructions, programmers use a wide variety of computer languages. Many such languages are considered "high-level," meaning they can be read and understood by humans with relative ease. Figure 1 shows a simple example of a program written in a high-level language known as C. This program displays on the computer screen "You will be assimilated." This program is a variation of a very common program known as "Hello World!" Most programmers use Hello World when first learning programming and code their computers to display the text "Hello World!"

```
#include <stdio.h>

int main() {
        printf("You will be assimilated.");
        return 0;
}
```

Figure 1. "You will be assimilated" program written in C

For a program to execute (i.e. run) on a computer, it must be translated to machine-readable code. Machine-readable code consists of a series of instructions in binary—consisting solely of *1*s and *0*s—which translate into particular voltages transmitted across the computer hardware and are stored in various formats and media. Figure 2, for example, presents the program from figure 1 expressed in machine code (displayed in hexadecimal—base *16*—for the sake of readability). The significance of this conversion is that all computer programming is strictly regulated by the capabilities of the computer hardware that applies these binary instructions. The hardware can calculate only a limited set of instructions; these instructions must appear in particular formats; all instructions written in nonbinary formats must be translated to a set of binary instructions; and programs must be logically coherent to execute properly (Berry 2011). The title of this article "01010000 01001100 01000001 01011001," which translates to "PLAY" in American Standard Code for Information Interchange (ASCII) characters, emphasizes this binary nature of computation. Every single character, every instruction, and every idea written on a computer must be translated to binary. There are no exceptions.

In addition to the limitations imposed by computer hardware, the languages and environments in which programmers choose to create code also restrict their programming because these languages and environments come with specific rules for instruction sets and formatting. Instruction in computer-programming methods and a variety of social interactions among programmers also instill a variety of conventions regarding proper ways of writing and structuring programs (Berry 2011). Human interaction with computers is thus strictly regulated by computer hardware, software, and human social conventions. Such constraints seem ominous for human play, suggesting that they

```
0000000 014c 0004 0000 0000 0138 0000 000f 0000
0000020 0000 0104 742e 7865 0074 0000 0000 0000
0000040 0000 0000 0040 0000 00b4 0000 0110 0000
0000060 0000 0000 0004 0000 0020 6030 642e 7461
0000100 0061 0000 0000 0000 0000 0000 0000 0000
0000120 0000 0000 0000 0000 0000 0000 0000 0000
0000140 0040 c030 622e 7373 0000 0000 0000 0000
0000160 0000 0000 0000 0000 0000 0000 0000 0000
0000200 0000 0000 0000 0000 0080 c030 722e 6164
0000220 6174 0000 0000 0000 0000 0000 001c 0000
0000240 00f4 0000 0000 0000 0000 0000 0000 0000
0000260 0040 4030 8955 83e5 08ec e483 b8f0 0000
0000300 0000 c083 830f 0fc0 e8c1 c104 04e0 4589
0000320 8bfc fc45 00e8 0000 e800 0000 0000 04c7
0000340 0024 0000 e800 0000 0000 00b8 0000 c900
0000360 90c3 9090 6f59 2075 6977 6c6c 6220 2065
0000400 7361 6973 696d 616c 6574 2e64 0000 0000
0000420 0021 0000 000d 0000 0014 0026 0000 000b
0000440 0000 0014 002d 0000 0009 0000 0006 0032
0000460 0000 000e 0000 0014 662e 6c69 0065 0000
0000500 0000 0000 fffe 0000 0167 7361 6973 696d
0000520 616c 6974 6e6f 2e32 0063 0000 6d5f 6961
0000540 006e 0000 0000 0000 0001 0020 0002 742e
0000560 7865 0074 0000 0000 0000 0001 0000 0103
0000600 003d 0000 0004 0000 0000 0000 0000 0000
0000620 0000 642e 7461 0061 0000 0000 0000 0002
0000640 0000 0103 0000 0000 0000 0000 0000 0000
0000660 0000 0000 0000 622e 7373 0000 0000 0000
0000700 0000 0003 0000 0103 0000 0000 0000 0000
0000720 0000 0000 0000 0000 0000 722e 6164 6174
0000740 0000 0000 0000 0004 0000 0103 0019 0000
0000760 0000 0000 0000 0000 0000 0000 0000 5f5f
0001000 6d5f 6961 006e 0000 0000 0000 0020 0102
0001020 0000 0000 0000 0000 0000 0000 0000 0000
0001040 0000 5f5f 6c61 6f6c 6163 0000 0000 0000
0001060 0000 0002 705f 6972 746e 0066 0000 0000
0001100 0000 0020 0002 0004 0000
0001112
```

Figure 2. Program from figure 1 in hexadecimal

overpower any attempt at play within the lifelessness of machines. But I explore throughout this article how various types of play are made possible by strictly regulated relationships and how programmers may adopt, resist, and reconfigure these restrictions.

[01010100:T]heoretical and Methodological Approach

To explore the different forms of play possible within this restricted relationship, this article draws extensively on the introductions to coding as social practice by Adrian Mackenzie (2005) and David M. Berry (2011) and on hacking and other programming practices by Gabriella Coleman (2004, 2009) and her research with Alex Golub (2008). These works explore many examples of coding as aesthetic and creative practice. I also consider discussions by computer scientists about their own discipline and examine discussions on Internet forums, web pages, and other online sources. In addition, I rely on my own experiences as an undergraduate and co-op student in computer science. Given the broad subject, I intend merely to introduce the many ways in which play intertwines with computer programming. I suggest that coding constitutes an arena with many possibilities for play research and offer an alternative to the image of humans dominated by the actions and restrictions of machines.

An idea espoused by Bruno Latour underlies this article: although computers, software, and code are all human-made objects, these machines and technical devices, in fact, act in the material and social world and interact with human and nonhuman actors (Latour 1993, 1996). More specifically, such objects are tied into a variety of social and material relationships and thereby are "not a human thing, nor is it an inhuman thing. It offers, rather, a continuous passage, a commerce, an *interchange*, between what humans inscribe in it and what it prescribes to humans. A thing that possesses body and soul indissolubly. The soul of machines constitutes the social element. The body of the social element is constituted by machines" (Latour 1996, 213).

Along with the necessity of binary encoding of computing instructions, computers also insist, for example, that programmers write programs in linear, sequential logic (Eck 2011; Berry 2011). The binary base of digital computers also requires that programmers parcel up the world into discrete pieces (Berry 2011). Through the interchange of these prescriptions by computers, code, and software with their use and configuration by human programmers, technologies

affect the social and material world. For example, they can reconfigure social relations and insist on certain forms and types of interactions (Mackenzie 2005).

Code, I should emphasize, has a dual existence that significantly affects how it acts in this world. Code can exist simultaneously as both text and execution, distinguished by Berry as code and software, respectively (Berry 2011). These existences denote one another and are deeply implicated in the constitution of programmers versus users (which are not necessarily distinct). Latour discusses how all human-technology relations involve a redistribution of abilities and responsibilities (Latour 1996). Yet, code and software taken together are particularly exceptional in that they constitute a "highly involuted, historically media-specific distribution of agency" (Mackenzie 2006, 19). As Berry sees it, these involutions and the many relations that software has with other computers, code, humans, and programs, have significantly affected how we experience the world today (Berry 2011). It is this ability for code and software to act pervasively and affect human social life that make dystopian visions of machines taking over the world seem possible, not some distant and fantastical science fiction. Instead, let me turn to a consideration of playful forms of interaction that occur between programmers and computers.

[01000001:A]esthetics and code

Computer languages come in many shapes and forms. While they are functionally written to be read and executed by computers, they are also languages designed and used by humans. As Huizinga tells us of human language, every expression and metaphor is a play on words in which humans construct a "second, poetic world alongside the world of nature" (Huizinga 1980, 4). This section explores how programmers use programming languages in creative, expressive, and playful ways. They see beauty in the writing of code, they use it as a poetic form of language, and they play with programming aesthetics, reconfiguring the restrictions of computers in the way they express themselves through computer languages. Programmers thereby insist on the human and machine expressiveness of computer languages.

When digital computers were first produced and developed throughout the 1950s and 1960s, computer programmers emerged as the experts of these new technologies. Programmers were seen as masters of a "black art," "uniquely creative" in their ability to develop programs and solve computing problems in

a manner distinct from what many considered the more rigorous and systematic approaches of science and engineering. Computing has since grown more legitimate and professional, boasting academic computer science curricula and programs and professional organizations (Ensmenger 2003). Nevertheless, computer science continues to struggle academically and professionally in recognizing the artistic nature of computer programming (i.e. Knuth 1973).

Good coding "style"—a particular aesthetic for writing code—is often taught to students as essential to producing clearly written code, code both easy to read and easy to adapt into larger program structures. Yet, good coding style also suggests a form of artistry (Knuth 1973; Black 2002). In his lecture for the Turing Award, for example, David Knuth declares that "when we prepare a program, it can be like poetry or music . . . programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and to establish a system of consistent rules." He goes on to claim that programs can be "elegant," "sparkling," "noble," and "truly magnificent" (Knuth 1974, 670). Significantly, for programmers and for my purposes, such statements emphasize the similarities between computer languages and human languages and so provide a medium for play on words for programmers. Free and open source software (F/OSS) developers find this argument particularly significant. Many of them even argue for code to be protected as a form of speech under the First Amendment of the U.S. Constitution (Mackenzie 2005).

A haiku written by Seth Schoen exemplifies how programmers emphasize and make use of the interconnections between these human and computing languages (Mackenzie 2005, 30; Coleman 2009). A portion of this haiku appears in figure 3. The arrest of Jon Lech Johansen, who had created and distributed a software program known as DeCSS that many claimed violated copyright laws, inspired Schoen to write his poetry. Within its 456 stanzas, it actually contained the algorithm for the DeCSS program (Coleman 2009). An algorithm consists of a series of instructions, like do *a*, followed by *b*, then *c*, then repeat. Programmers often begin by writing algorithms in "pseudocode" prior to writing in actual code. Pseudocode is not a particular programming language but substitutes as an informal general form to describe the process of what programmers want to achieve. Thus, programmers would appreciate the algorithm contained in the haiku as a very creative and playful expression of pseudocode.

Court testamony following Johansens's arrest explains that the haiku demonstrates how "the path from idea to human language to source code to object code is a continuum. As one moves from one to the other, the levels of precision

```
So this number is
once again, the player key:
(trade secret haiku?)

Eighty-one; and then
one hundred three – two times; then
two hundred (less three)

Two hundred and twenty
Four and last (of course not least)
The humble zero
…
We write precisely
since such is our habit in
talking to machines;

we say exactly
how to do a thing or how
every detail works.

The poet has choice
of words and order, symbols,
imagery, and use

of metaphor. She
can allude, suggest, permit
ambiguities.
```

Figure 3. Portion of haiku by Seth Schoen (Coleman 2009, 4431/N44)

and, arguably, abstraction increase. . . . But each form expresses the same idea, albeit in different ways" (Mackenzie 2005, 30). The haiku critiques and mocks laws that aim to make particular forms of code illegal while also demonstrating the creativity of programmers. The code for the deCSS program also appeared in other forms of artwork. It was, for example, printed on t-shirts in the image of a DVD logo (Mackenzie 2005).

The haiku speaks directly to the topic of this article, expressing the technical constraints of talking to machines, but also the metaphor, ambiguity, and poetry of writing algorithms and code. As Coleman observes, "programmers conceive

of their craft as technically precise (thus functional) yet fundamentally expressive" (Coleman 2009, 443). Thus, we find in computer languages the play that Huizinga also sees in human language. Some might even argue that there are more aspects of play in computer languages, because they are contained within the "magic circle" of a computer, must adhere to a "system of consistent rules," and are distinct in many ways from "ordinary" or "real" life (Huizinga 1980, 8–11). These characteristics of play described by Huizinga are less obvious in human languages, languages inherently part of our everyday life.

The particularities of the play with language possible through code appear in poems that are valid code or programs themselves. These poems play directly with the constraints of computer languages and address the implied dual audiences of code: "programs can simultaneously mean something for the machine and for human readers" (Mateas and Montfort 2005, 152). As a rule, code poems must be parsable, meaning they must translate to a valid set of instructions in machine-code (Hopkins n.d.). Huizinga argues that poetry extends far beyond the realm of aesthetics and involves competition, sociality, and ritual. Yet, whether you produce poetry for the sake of beauty, status, or the "divine," he concludes that "to call poetry . . . a playing with words and language is no metaphor: it is the precise and literal truth" (Huizinga 1980, 132). Code poems, because of their dual audiences—they speak to both humans and machines—involve a double play not often found in poems directed solely at humans. In addition, some of these code poems also create productive output, offering a third level of expression. An example of such a poem is seen in figure 4.

This poem takes advantage of several features of the Perl language in which it is written, including its variety of key words and its lack of restrictions in the use of tabs and new lines, which have specific meanings in some languages. Hopkins (n.d.) argues that Perl suits the creation of code poems in particular because of its inherent flexibility. Other languages, however, are more restrictive in their structure and provide fewer meaningful key words for the poet to use. Perl's design is "practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal)" (Robert n.d.). In the design of Perl, we also see an acknowledgement of classical coding aesthetics and style, which I have discussed. In its flexibility, Perl surpasses these ideals, however, emphasizing practicality but also offering programmers a wider range of expressive possibilities (Mateas and Montfort 2005). Figure 5 illustrates this feature of Perl.

It is interesting to note, however, that many consider Perl poetry as "kitsch art" (Mackenzie 2005, 27). I have also heard that because of its flexibility, Perl

```
# Ode to My Thesis, a Perl Poem
# (must be run on Perl 4.0 or higher)
<<birth                                                 ;
G
   r
      o
         w
            t
               h
re-
birth

seek                                                    (
        enlightenment, knowledge, experience            );

goto MIT;

sleep "too little", study $a_lot,
wait, then                                              .
        "B.S.",

leave. then, return to                                  ;
        MIT                                             :

now,
        $done = 'a Ph.D.                                ">&2';

warn pop @mom, "I'll be here a while                    \n";

study, study, do study;

push                                                    (
       myself, computers, experiments                   ),

read                                                    (
       data, references, books                          ),

study
       write,
               write,
                       write,

do more if time                                         ;
redo if $errors                                         ;

do more_work if questions_remain                        ;

$all_are_answered? yes.

now :
       write,
       chop if length $too_great                        ;

format                                                  =
       Thesis

.
                       tell all,                        .
                       done, finally                    .
                       now, do rest
shout.
       and                                              .
               hear                                     .
                       it                               .
                             `echo                      "

             Now I am $done`
# Craig Counterman, April 27, 1991
# ccount@athena.mit.edu
```

Figure 4. “Ode to My Thesis” a Perl Poem written by Craig Counterman (Hopkins n.d., figure 3)

```
Output:
I'll be here a while
        Thesis
        Thesis
        Thesis

                Now I am a Ph.D.
```

Figure 5. Output from "Ode to My Thesis" (Hopkins n.d., figure 4)

is not a "real" programming language when compared to more structured and restrictive ones. While the poem in figure 4 produces output, it is not a practical program in the sense that it does not do anything useful. Aside from creating output through poetry, it is also not a technically difficult program and thus "less interesting to [many] programmers" (Mateas and Montfort 2005, 148). Still, by the very nature of introducing poetry and aesthetics to programming, programmers assert the expressiveness and playfulness of computer languages. Yet, by contesting the usefulness of Perl poetry, programmers also insist on technicality as an important measure of computer languages. They do not see the duality of human and machine audiences as a constraint but as part of the expressiveness of the language. A program should be creative, beautiful, and technical, but it should also do something useful (ideally in a creative and beautiful way).

We also find this emphasis on technicality as part of creativity in the International Obfuscated C Code Contest (IOCCC). The contest goal is to write the most obscure/obfuscated C program according to the following rules: (1) "To show the importance of programming style, in an ironic way"; (2) "To stress C compilers with unusual code"; (3) "To illustrate some of the subtleties of the C language"; and (4) "To provide a safe forum for poor C code. :-)" (Broukhis, Cooper, and Noll 2012). Figure 6 provides an example of one winning entry from 1998.

The contest illustrates the significance of technical proficiency in writing code, in playing with the restrictions imposed by particular languages and computers in general by seeking to "stress" the compilers as well as playing with conventional ideas of aesthetics. As Mateas and Montfort comment, "Such play refutes the idea that the programmer's task is automatic, value neutral, and disconnected from the meaning of words in the world" in "classical" aesthetics

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l              ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=  20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

Figure 6. Winning entry by Raymond Cheong from the 2001 IOCCC (Berry 2011,89)

as in much obfuscated code (Mateas and Montfort 2005, 147). As the DeCSS haiku also espoused, programmers have restrictions, but they also have choice in areas such as the program structure, the use of particular key words, and variable names. The winning entry displayed in figure 6, for example, creatively uses white space to suggest the function of the program, which takes in a number and returns its square root (Berry 2011). Such contests have additional functions for programmers. Through them programmers demonstrate their technical prowess and compete with one another in technical ability and programming creativity. They also demonstrate their knowledge of good code by deliberately creating bad or obfuscated code. The contests also serve as performatives of programmers' identities and a sharing of particular values of technical creativity (Case and Piñeiro 2006).

Computer languages thus provide an expressive medium for programmers. Similar to human languages, they enable technical, poetic, competitive, and aesthetic play with the words, syntax, grammar, and visual arrangement of the language. The limitations of computing machinery, languages, and social practices in these contexts are not seen as barriers to the capacity of programmers to express themselves; programmers are not assimilated within the linear and binary nature of computation. Rather, the limitations are a basis for play in themselves. Philosopher and programmer Ian Bogost has also played with producing machine-written haikus in his video game *A Slow Year*, seeing it

as the bridge between "machine-as-poem (the games) and poetry as meaning-machines (poetry in the ordinary sense)" (McCollough 2011). I discuss "machine-as-poem" in the next section. Machine-generated haikus, however, emphasize the machinal qualities of human language and human play.

For many, machine poetry could never be authentically playful. As linguist and computer scientist Chrysanne DiMarco states, a computer "could fake it, but poetry comes out of a person's emotions, feelings, sensibilities, experiences. A self-awareness. A machine might be able to generate a facsimile of poetry, but it's hollow" (Choudhuri 2011, 105). For DiMarco, poetry is thereby a human form of expression. Nevertheless as Bogost's machine poems suggest, there is systematicity, regularity, rule, and constraint in human play and poetry, and there is unpredictability and surprise in the operation (or expression) of machines (McCullough 2011), an issue I also discuss in this article. Encounters with computers and their languages thus invite us to consider the ways we are similar and different from machines and play with the possibilities provided by such encounters (see Turkle 2005). The restrictions of human-computer interaction challenge and invite programmers to stress the limits of computing languages, obfuscate classical aesthetics, or poeticize. Machine-generated poems point to the playful possibilities that computers call out and enable. Case and Piñeiro (2006) discuss how programmers instead more often felt constrained or restricted in their abilities to create beautiful code by the limitations of deadlines and coding practices for commercial output rather than by the limitations found in the human-computer interface itself.

[01000011:C]omputing Imagination

The use of high-level languages involves not only a means of expression but also offers a source of abstraction, imagination, and creation. Huizinga's idea of a "second, poetic world" created through language finds concrete reflection in reality with computing languages. In this section, I present high-level languages as abstractions of machine operations. Ideally, when we create and run programs, the translation between the abstract language and its concrete implementation in machine language should be exact. Practically, however, because of the complexity of the computer software controlling these programs and the translation among various levels of abstraction, the computing itself contains significant levels of indeterminacy (Berry 2011). Thus, instead of imagining the design of

programming languages and environments as a one-to-one translation from high-level code to machine code where the output precisely and exactly matches the input, we should regard them as a form of creative imagination. When programmers create a programming language, they rely on conceptual models of how both computers and live reality operate. They also sometimes play with these concepts, poking fun at the functional application of much computing and emphasizing the expressive human facet of computer programming.

To explore how programming languages represent the abstract imaginary worlds of computer operation, I use the example of Java (Mackenzie 2006). First created in 1995, Java Virtual Machine (JVM) and Java Programming Language have become popular among programmers (Mackenzie 2006). Java serves our purposes because the Java Virtual Machine introduces a layer of abstraction between machine operation and computer languages that other programming languages do not have. This layer exemplifies the creation of a second poetic world through computing language. In Java, programming language transforms into Java Byte Code, which consists of the machine code for an imaginary machine implemented by the Java Virtual Machine (Mackenzie 2006). This imaginary machine language is then translated by the JVM to the actual machine language of the computer on which it is running. Machine code is platform dependent, meaning it is understandable only by specific types of computers. Just as English speakers do not understand French, an Apple computer does not understand PC machine-code. The creation of an abstract generic machine—the Java Virtual Machine—allows programs to run on any computer that has this software-based machine installed.

The Java Virtual Machine thus represents a creative solution to a concrete problem and plays with the realities of machine hardware. By imagining how they wish computers operated—independent of hardware—and expressing that wish in code, programmers have literally created a new computing world where the wish holds true. In other words, the Java Virtual Machine was produced as an imaginary machine through a play with programming language to create a second poetic world where all Java programs can run free of hardware specificities. As Mackenzie points out, the abstraction of the JVM is never quite as complete as designers claim. Nevertheless, the act of coding and the software is a creative and a "collective imagining" of the world of computing possibilities (Mackenzie 2006).

The creation of new computing worlds through particular languages grows more evident when we consider the Java Programming Language itself. Java is

what those in computing call an object-oriented programming language. It allows a programmer to create a world of objects, each with particular properties and each functioning according to specific pieces of code. Creating this world also involves imagining the "actual" world as constituted by a discrete and bounded collection of objects. A scheduling program for a college registrar's office, for example, would contain objects for student, classroom, course, and section. Of course, in terms of machine code, the computer knows nothing of objects as concepts; it does not know what a student or a classroom is. Object-oriented programming, as well as programming in every other high-level computing language, is an abstraction of the human world and of the computer world that renders each understandable to the other. This process involves a play with language and a play with reality to construct such mutually intelligible worlds.

Programmers build one level of abstraction upon another, beginning with a virtual machine abstracted from any concrete computer hardware, on top of which they lay a language-world of objects, through which they create a virtual registrar's office complete with students, classrooms, and courses. In relation to the design and creation of technical projects, "the circumscribing, the coding, and the visualization of the division of tasks allows a piling-up of Russian dolls that increases the complication of the whole" (Latour 1996, 217). The multiple play on reality and language "for programmers, computing in a dual sense, as a technology and as an activity, becomes a total realm for the freedom of creation and expression" (Coleman 2004, 512). The Java Virtual Machine and Java Programming Language are not so far removed from the creation of virtual worlds such as *Second Life* and *World of Warcraft* (e.g. Boellstorff 2008; Malaby 2009; Bainbridge 2010). The former are simply lacking the graphical displays and user interface of such virtual worlds. As Bogost explains in relation to his game poetry, "When one writes software, one builds a machine for producing a variety of results, rather than a single result fashioned and polished by hand. The outcomes are unpredictable, surprising, and sometimes even broken. That's what it feels like to make a game, too" (McCullough 2011 n.p.).

I discuss further the significance of these indeterminacies in relation to possibilities for game play in the next section, but as Mackenzie comments, "Imagining generates relations over time" (Mackenzie 2005, 138). The construction of imaginary computing worlds is associated with the concomitant realization of those computing worlds. As I mentioned, a play with reality underlies the practice of computational design. In many ways, the realization of Huizinga's second (poetic) world through play with language becomes instantiated out-

side the magic circle of the computer. This construction of reality both reflects the vast magnitude of possibilities that computers seemingly offer to humans (Coleman 2004) and how human life seems increasingly lived through and in computing worlds (Berry 2011; Malaby 2009; Mackenzie 2005; Bainbridge 2010). The writing of code, the proliferation of software, and its application to our daily lives from personal computers to cell phones, all create a world organized and mediated through digital computers as we delegate various forms of agency to these machines (Latour 1996). In this context, a world dominated by computers seems an impending reality. These imaginings and realizations require acknowledging that language plays in the form of code and its execution in software is a very serious form of play.

Returning to the magic circle of the computer, however, I touch on a final example of "weird languages" to emphasize the creativity involved in processes of imagination. Similar to the IOCCC, these languages parody "normal" computing languages, play with ideas of both human and machine audiences, and often offer programmers a challenging puzzle to decipher. There are a variety of examples: INTERCAL consolidates the worst features of existing functional languages; Brainfuck reduces the number of commands and space for computation to a minimum, playing with the readability of computing languages; Chef insists that all code look like a recipe, albeit one that creates inordinate amounts of food; Shakespeare insists that all code look like a Shakespearean play; and Malbolge seeks to make programming as difficult as possible (Mateas and Montfort 2005).

Let me briefly address this urge toward the difficult here. Like the IOCCC, Malbolge presents a challenge to programmers and plays with the idea that computing code must be functional or aesthetic in the "classical" sense. It also emphasizes the comparative simplicity of machine code. While programmers are capable of writing in binary machine code, although it is a tedious process, Malbolge required programmers to use cryptographic deciphering and Artificial Intelligence search techniques to produce a workable program. On the other hand, Malbolge is so difficult because it plays with programmers' standard assumptions about computers. For example, Malbolge machine code is based in trinary (also referred to as ternary), meaning all instructions and data are represented by *0*, *1*, or *2*. Malbolge runs on a virtual machine similar to the Java Virtual Machine, which mediates between this trinary language and binary digital computer hardware. In terms of using the language, trinary makes it more complicated for programmers to translate or understand numbers as they are now represented in base *3*. In addition, Malbolge also offers no debugger to help

determine the cause of malfunctioning code, includes minimal and unexpected constructs for computation, and modifies the code itself as the program executes (Mateas and Montfort 2005).

Designing and programming in Malbolge, therefore, represents a display of technical ability but also emphasizes the relative lack of restrictions imposed by standard computers and programming languages. Malbolge also contests the notion that computers and programs must operate in a particular way, such as through sequential logic, by creating a language through which the program modifies itself. Thus, the world does not necessarily operate according to digital logic, and humans are not necessarily assimilated within a digital universe. Instead, humans have created the machines and a world that operates in this way, so humans are in control. Note that here I have returned to the language of control and resistance, insisting that humans will not be assimilated and resistance is not futile. However, I want to emphasize an important difference in my treatment of Malbolge; Malbolge shows how both the possibilities and limitations of machines are invitations for play among willing programmers. Binary computation, for some, may represent the monotony of encountering machines. Rather than becoming assimilated with the emotionless machine, however, Malbolge proposes a reconfigured relationship. It plays with the nature of the human-computer encounter.

As with the creativity applied by programmers in writing computer code, the limitations enforced by the computational environment do not have to be considered restrictions; we can see them as rules, rules to be mocked, played with, and sometimes broken. We adhere to particular forms of languages from programming habit or from economic restrictions related to commercial software production (Fishwick 2002). In contrast, the human-computer interface seems a realm of possibilities where programmers can create new computational worlds. Each of these worlds reflect a "double-coding" with reference to the human reality in which they function and the machine reality that operates them (Mateas and Montfort 2005). More specifically, these worlds are poetic plays on both human and machine realities. The abstraction and imagining of computing and human worlds also allows each to understand the other. Yet, even if human play does not assimilate to the digital world of computation, code, software, and computers act in this world. Thus, computers also capture and modify the human world through the distributed agency of code (Mackenzie 2005). Many forms of computational imaginaries should then be acknowledged as a serious form of play with real-world effects.

[01000001:A] World of Play

Having discussed how computing languages offer programmers a means of creative expression and their design offers them a means of creating imaginary computing worlds, I now address the concrete practices of programmers—writing, debugging, and testing code—and consider how programmers interact with the imaginary computing worlds they create. I also look at how the "numerous layers of software [that] serve to create inner unstable universes within which further abstraction takes place" provide a game-like environment for programmers (Berry 2011, 140) and let programming itself function as an extreme sport, one entailing risky encounters, dangerous situations, and engaging and fun explorations, as programmers compete with the computers, themselves, and each other.

Two very common and interrelated practices occupy much of programmers' work: testing and debugging. As an introductory Java programming textbook explains, good programs should be "correct" and "robust"—they do what they should do and they can handle "illegal" behavior or unexpected situations (Eck 2011). Testing determines whether a program is correct and robust; debugging fixes problems when a program fails a test. Errors come in the form of improper syntax, more elusive problems of incorrect logic, unexpected behavior of the underlying programming environment or the programming language, or a simple mistake made by the programmer. Theoretically, a program can be perfect; practically, most programs contain bugs. There is much indeterminacy involved in the programming and use of code that become more and more complex with each level of abstraction (Berry 2011).

Testing, in essence, attempts to break the program. Testing virtual game design or computer chess may itself involve actually playing games. Yet even if programmers are not testing a game, the indeterminacies contained within the constructed order of computational space present them with a game-like environment. In addition, while testing requires a systematic approach to ascertain the overall correctness and robustness of the program, it also requires creativity to think of unusual ways that something can go wrong. We can thus consider the process of testing and debugging as a competition against—and a creative exploration of—the computer. In discussing the game-like quality of the virtual world *Second Life*, Malaby defines a game as "a semibounded and socially legitimate domain of contrived contingency that generates unpredictable outcomes" (Malaby 2009, 84). This aspect of contingency involves a mix of constraint and

possibility or, in other words, of controlled indeterminacy.

The difference between a computer bug and indeterminacy in a game lies with its legitimacy: a computer bug is not supposed to be there. Bugs can also have significant real-world consequences. There have been examples of failed space missions, the loss of millions of dollars, injury, or even death due to errors in software (Eck 2011). Both serious and mundane, the work of finding and eliminating a bug nevertheless produces the game-like atmosphere described by Malaby because, although the bug itself is not legitimate, testing and debugging are legitimate encounters with indeterminacy. As I have mentioned, the contained world of the computer and the inherent constraints and order imposed by computer hardware satisfy many of the criteria of play and games set by Huizinga. Ross Smith, director of test for Microsoft Corporation, also argues that, in a work environment, the use of "productivity games" surrounding testing—competitions among testers, for example—provides an arena for creativity, risk taking, trust, and fun that supports involvement, productivity, and innovation (*American Journal of Play* 2011).

We might also frame "hacking" as an exploration of—and competition within—game-like computing worlds. Hacking has developed various meanings over time. Gabriella Coleman and Alex Golub (2008) think of it as a multiplicity of practices based on distinct ideals of liberalism and freedom. Yet, across all such distinctions, hackers themselves seem to value tinkering and learning through programming practice (Coleman and Golub 2008). For example, Mackenzie observes that Linux creator Linus Torvalds and other programmers consider Linux "above all a program by men for men who like to play with computing hardware" (Mackenzie 2006, 89). Hackers use coding to play with the capabilities of computer software and hardware. As Turkle claims, the exploration of hacking involves seeing "complex computer systems as places where you can let things get more and more complicated . . . playing with the issue of control by living on the narrow line between having it and losing it" (Turkle 1988, 36). In some senses, we might consider hacking a form of extreme sport or a personal contest against the hackers' own bodies, against particular programs, and against hardware systems. Hackers often challenge what they see as the more institutionalized programming and programmers embodied in computer science disciplines, administrative systems, research laboratories, and industry (Turkle 2005).

Hacking resembles what Thomas Henricks calls "disorderly play" (Henricks 2009). Henricks points out that theories of play often involve orderly social,

cultural, and psychological constructs but that play also creatively challenges norms, organizations, and values. He suggests that "play features a dialogue—a give-and-take of well-matched participants—and rebellion—the thwarting of more powerful others—as well as attempts at control of and letting go of restraints" (Henricks 2009, 29). Hacking plays with the rules of computers and with the rules of legal systems and capitalist institutions, or, in a more mundane context, with coding institutions. The hacker underground, for example, "envisages hacking as a constant arms race between those with the knowledge and power to erect barriers and those with the equal power, knowledge and especially desire, to disarm them" (Coleman and Golub 2008, 263). More generally, hackers tend to push their bodies to the limit, working for hours on end with no food or sleep. Hacking involves game-like encounters with indeterminacy by raising the issue of maintaining or losing control, by exploiting possible weaknesses in software programs, and by grappling with the complexity of computing systems. Thus, hacking presents the converse to the orderly play of software poetry and the construction of computing worlds.

Mackenzie's discussion of Linux also reveals the strongly gendered aspect of hacking. Computer science as a discipline in Western countries, and hacking in particular, is dominated by men (Turkle 1988; Adam 2003; Margolis and Fisher 2002). According to Sherry Turkle, men see the in-depth exploration of computing worlds as a safe form of play where they have control of the arena, unlike the lack of control proffered by the complexity of human relationships (Turkle 1988, 2005). Applying this to games, Malaby suggests that male gamers tend to prefer the individual performances, for example, required by massive-multiplayer-online games or first-person shooters, as opposed to women who tend to prefer the social interaction found in games like online *Scrabble* (Malaby 2009). While this contrast may be overly simplistic, both in terms of gender divisions and the social play involved in different types of games, the demonstration of technical prowess in hacking, playing particular games or discussed in relation to obfuscated code contests do appear to be performances of masculinity based on the competitive exploration of computing worlds.

For hackers, this form of risky play and competition also serves as a strategy for learning about computers, their limitations, and how those limits can be transgressed (Turkle 1988). Returning to the practices of testing and debugging, we can see them as a slightly more mundane form of hacking-like play with the capabilities of computer software and hardware. As one computer scientist complained in response to his attempts to introduce formal testing methods,

programmers thought that "debugging was fun! It turned out that they derived the major part of their professional excitement from *not quite understanding what they were doing* and from chasing the bugs that should not have been introduced in the first place" (Dijkstra 1993, 3). Malaby points out that learning in itself functions like a game, as individuals encounter new ideas and circumstances that must be incorporated into their understanding of the world and how they function in it (Malaby 2009). Henricks suggests that this is, more generally, an important function of disorderly play as a means of encountering and learning about the world, the way it operates, and the possibilities it holds (Henricks 2009).

The interaction between humans and computers in testing, debugging, and hacking can thus be characterized as a "risky encounter" (Berry 2011, 140). Programmers compete against their own bodies, against computer hardware and software, and against the elusiveness of the bug. Ultimately, however, humans remain in control of this process, and in this sense the risky encounter "is safe risk" (Turkle 1988, 37). While the potential for computers to dominate human life is tangible given their ubiquity and pervasiveness, the play of testing, hacking, and debugging introduces human creativity. If all else fails, one can simply pull the plug and regain mastery over the life of the computer (Turkle 2005), although fears of domination and control stem from the vision that one day pulling the plug will not be possible. Nevertheless, as I mentioned earlier, programmers do not see the limitations imposed by computer hardware and software on programming as restrictions. Rather, the computing environment, in all its levels of abstraction, offers hackers, debuggers, and testers a world of possibilities (and indeterminacies) to explore and challenge. Hackers in particular demonstrate "an absorption, a devotion that passes into rapture" in the computing world (Huizinga 1980, 8). In attempting to break the computing rules (social, material, technical), the hackers are the "cheaters" of the computing world who take the game so seriously they need to break or remake the rules to win. In such an approach, programming practice is not simply the systematic development of useful software but a fun endeavor ripe with possibilities.

[01000011:C]onclusions

I began this essay with the dystopian nightmare of the Borg. This cybernetic species suggests the possibility that computers and machines might one day

dominate and assimilate humanity into the seeming lifeless and homogenous world of *1*s and *0*s. Resistance is futile. The human-computer interaction of computer programming is certainly regulated by a variety of rules and restrictions. Digital computers understand only binary code, in particular formats and orders. High-level computing languages also have their own rules about structure, format, and keywords. In addition, social expectations insist on certain styles of programming. In concretely considering human interaction with computers in the context of programming, however, I have complicated this polarity and shown how human play is an integral part of our interface with machines. It is in fact the many restrictions on computer programming that make various forms of play possible.

I have briefly outlined three aspects of programming, pervaded by play, that constitute fruitful areas for future play research. In particular, code serves as an expressive medium programmers use to create beauty, poeticize, and obfuscate, producing both human and computer meaning. In designing computer languages and environments, programmers engage in a form of collective imagining that creates new "poetic" computing worlds. These imaginings have real-world effects, but they also consist of a creative process full of possibilities. The rules and restrictions imposed by computers, code, and software construct a "magic-circle" of play for programmers (Huizinga 1980, 8). The worlds these programmers create can be literary, obscure, incomprehensible, or practical. In testing, debugging, and hacking, programmers explore the worlds they have created in a game-like environment. They track down bugs, compete and reconfigure the possibilities of software and hardware, and sometimes even break the rules.

These many forms of play are a key component to the human-computer interface. Processes of creativity and expression have allowed programmers to imagine, create, and reconfigure the computer world and the real world in astonishing ways. The process of abstraction involved also provides a key for programmers to understand the complexity of computer operations. Perhaps most significantly, however, the strict world of computing, far from being dominated or assimilated into a lifeless world of digital computation, enables and invites programmers to play as part of the relationship with their computers. It is both a human response to the restrictions of computation—an insistence that they are not the mechanical thinking machines—and a summoning by computers as a realm of possibilities for human creation and exploration. Thus, programmers write poetry, they create incomprehensible languages, and they break the computing rules. Play thus suggests an alternative or reconfigured envisioning

of human-machine relationships, one full of possibilities for interaction and engagement, rather than destruction and domination (Haraway 1991).

References

Association for Computing Machinery. n.d. "A.M. Turing Award." http://amturing.acm.org/byyear.cfm.

Adam, Alison E. 2003. "Hacking into Hacking: Gender and the Hacker Phenomenon." *ACM SIGCAS Computers and Society* 33:3.

American Journal of Play. 2012. "How Play and Games Transform the Culture of Work: An Interview with Ross Smith." *American Journal of Play* 5:1–21.

Bainbridge, William Sims. 2010. *The Warcraft Civilization: Social Science in a Virtual World.*

Berry, David M. 2011. *The Philosophy of Software*: *Code and Mediation in the Digital Age*.

Black, Maurice J. 2002. "The Art of Code." PhD diss, University of Pennsylvania.

Boellstorff, Tom. 2008. *Coming of Age in Second Life: An Anthropologist Explores the Virtually Human*.

Bowker, Geoffrey. 1993. "How to Be Universal: Some Cybernetic Strategies, 1943–1970." *Social Studies of Science* 23:107–27.

Broukhis, Leo, Simon Cooper, and Landon Curt Noll. 2012. "Goals of the Contest." *The International Obfuscated C Code Contest*. http://www.ioccc.org/.

Case, Peter, and Erik Piñeiro. 2006. "Aesthetics, Performativity and Resistance in the Narratives of a Computer Programming Community." *Human Relations* 59:753–82.

Choudhuri, Aradhana. 2011. "Code as Poetry." *Arc Poetry Magazine.* 66: 97–108.

Coleman, Gabriella. 2004. "The Political Agnosticism of Free and Open Source Software and the Inadvertent Politics of Contrast." *Anthropological Quarterly* 77:507–19.

———. 2009. "Code is Speech: Legal Tinkering, Expertise, and Protest among Free and Open Source Software Developers." *Cultural Anthropology* 24:420–54.

Coleman, Gabriella, and Alex Golub. 2008. "Hacker Practice: Moral Genres and Cultural Articulation of Liberalism." *Anthropological Theory* 8:255–77.

Consalvo, Mia. 2007. *Cheating: Gaining Advantage in Videogames*.

Corneliussen, Hilde, and Jill Walker Rettberg, eds. 2008. *Digital Culture, Play, and Identity: A World of Warcraft Reader*

Downey, Gary Lee. 1998. *The Machine in Me: An Anthropologist Sits among Computer Engineers*.
Eck, David J. 2011. "Introduction to Programming Using Java." http://math.hws.edu/eck/cs124/downloads/javanotes6-linked.pdf.
Ensmenger, Nathan L. 2003. "Letting the 'Computer Boys' Take Over: Technology and the Politics of Organizational Transformation." *International Review of Social History* 48:153–80.
Fishwick, Paul A. 2002. "Aesthetic Programming: Crafting Personalized Software." *Leonardo* 35:383–90.
Haraway, Donna J. 1991. "A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century." In *Simians, Cyborgs, and Women: The Reinvention of Nature*, 149–82.
Hayles, N. Katherine. 1999. *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature, and Informatics*.
———. 2005. *My Mother was a Computer: Digital Subjects and Literary Texts*.
Helmreich, Stefan. 1998. *Silicon Second Nature: Culturing Artificial Life in a Digital World*.
Henricks, Thomas S. 2009. "Orderly and Disorderly Play: A Comparison." *American Journal of Play* 2:12–40.
Hopkins, Sharon. n.d. "Camels and Needles: Computer Poetry Meets the Perl Programming Language." http://budi.insan.co.id/courses/el2001/plpaper.pdf.
Huizinga, Johan. 1980. *Homo Ludens: A Study of the Play-Element in Culture*.
Knuth, Donald E. 1973. *The Art of Computer Programming*.
———. 1974. "Computer Programming as an Art." *Communications of the ACM* 17:667–73.
Latour, Bruno. 1993. *We Have Never Been Modern*. Translated by Catherine Porter.
———. 1996. *Aramis or the Love of Technology*. Translated by Catherine Porter.
Leventhal, Laura Marie. 1988. "Experience of Programming Beauty: Some Patterns of Programming Aesthetics." *International Journal of Man-Machine Studies* 28:525–50.
Levy, Steven. 1984. *Hackers: Heroes of the Computer Revolution*.
Mackenzie, Adrian. 2006. *Cutting Code: Software and Sociality*.
Malaby, Thomas M. 2009. *Making Virtual Worlds: Linden Lab and "Second Life."*
———. 2009. "Anthropology and Play: The Contours of Playful Experience." *New Literary History* 40:205–18.
Margolis, Jane, and Allan Fisher. 2002. *Unlocking the Clubhouse: Women in Computing*.
Mateas, Michael, and Nick Montfort. 2005. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics." In *Proceedings of the 6th Digital Arts and Culture Conference,* 144–53.
McCollough, Aaron. 2011. "4K Formalism: An Interview with Ian Bogost." *The Journal of Electronic Publishing* 14. http://dx.doi.org/10.3998/3336451.0014.205.
Nardi, Bonnie A. 2010. *My Life as a Night Elf Priest: An Anthropological Account of "World of Warcraft."*
Price, Huw, Martin Rees and Jaan Tallinn. 2012. "The Cambridge Project for Existential

Risk." University of Cambridge. http://cser.org/.
Roberts, Kirrily "Skud". n.d. "What is Perl?" perldoc.perl.org: Perl Programming Documentation. http://perldoc.perl.org/perlintro.html.
Schull, Natasha. 2005. "Digital Gambling: The Coincidence of Desire and Design." *The Annals of the American Academy of Political and Social Science* 597:65–81.
Turkle, Sherry. 1988. "Computational Reticence: Why Women Fear the Intimate Machine." In *Technology and Women's Voices: Keeping in Touch*, edited by Cheris Kramarae, 41–61.
———. 2005. *The Second Self: Computers and the Human Spirit*.