[Submitted to DAC 2005 — 15 August 2005]

A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics

Michael Mateas
Georgia Institute of Technology
School of Literature, Communication and Culture
686 Cherry Street, Atlanta, GA 30032 USA
michaelm@cc.gatech.edu

Nick Montfort
University of Pennsylvania
Department of Computer and Information Science
3330 Walnut Street, Philadelphia, PA 19104 USA
nickm@nickm.com

ABSTRACT

The standard idea of code aesthetics, when such an idea manifests itself at all, allows for programmers to have elegance and clarity as their standards. This paper explores programming practices in which other values are at work, showing that the aesthetics of code must be enlarged to accommodate them. The two practices considered are obfuscated programming and the creation of "weird languages" for coding. Connections between these two practices, and between these and other mechanical and literary aesthetic traditions, are discussed.

1. INTRODUCTION

Programmers write code in order to cause the computer to function in desired ways. But modern computer programs are written in a form, usually textual, that is also meant to be manipulable and understandable by human beings. For a programmer to understand what she herself is writing, and to incorporate code that others have written, and to simply learn how to program with greater facility and on a larger, more complex scale, code has been made legible to people. While a computer system may compile or interpret code, it is important to the nature of code that it is interpreted by people as well.

A typical perspective on code would be that clarity and elegance are the only possible values that programmers can have when writing it, although they may succeed to a greater or lesser extent at writing clear and elegant code. But if this were the case, how is it possible to explain the way that people sometimes intentionally obfuscate their code, making its functioning more or less impenetrable, even when there is no commercial or practical reason to do so?[1] The existence of obfuscated programming as a software development practice, and as an aesthetic practice, throws a wrench into the simplified theory of coding that would claim that coders must always strive for clarity. An additional complication is seen in programming languages that are themselves designed as jokes or parodies, sometimes called "weird programming languages" or "esoteric programming languages." Such languages are designed to make legibility of *any* program difficult. Obfuscated code and weird languages highlight how important the reading of code by humans is in software development. If some code is only to be read by a machine, it can be neither obfuscated nor clear: it can only function properly or not.

[1] Sometimes people might undertake to make their computer programs difficult to understand for commercial reasons — to thwart competitors and clients, for instance, and to increase others' dependence on them. This practice is entirely different from the obfuscated programming discussed in this paper.

This paper suggests some ways to enlarge the an aesthetics of code to account for the existence of obfuscated programming as a practice and for these unusual languages. Such consideration shows that a previously neglected layer of computing and new media is available for rich aesthetic understanding.

2. READING CODE

Version 2.1 of the online lexical reference system WordNet gives 11 senses for "read," including "look at, interpret, and say out loud something that is written or printed" and "interpret the significance of, as of palms, tea leaves, intestines, the sky, etc.; also of human behavior." [14] This discussion is about a fairly literal application of the most common sense, "interpret something that is written or printed," although of course code that appears on a screen (rather than being written or printed out) can be read.

The understanding of behaviors is certainly involved in reading code in the primary sense of "read," however. It is essential to any ordinary human reading of a computer program to develop an understanding of how the computer will behave, and what it will compute, when it runs the code that is being examined. In a popular book on the history of software, one of the developers of FORTRAN is characterized as "an extraordinary programmer who could 'execute' a program in his head, as a machine would, and then write error-free code with remarkable frequency." [7] Actually, all programmers must do this to some extent, using some internal model of what code will do. Just as understanding what a program does, and why, is critical on a practical level for the programmer, it is important to the aesthetics of code as well. Because code functions, "the aesthetic value of code lies in its execution, not simply its written form. To appreciate it fully we need to 'see' the code to fully grasp what it is we are experiencing and to build an understanding of the code's actions." [2]

The analysis of a computer program or system often involves examining how the program behaves and "reading" (in this other sense, "interpreting the significance of") the intention behind the program, the structure of the program, or the more fundamental reason for the outputs observed. This is very frequently done in reverse-engineering in "black-box" situations, where code and other internals are not available for inspection. A network administrator might also be able to "read" the behavior of a malfunctioning router and figure out the problem without looking at any code. But these types of analysis also apply to systems that are not governed by legible code at all, and are not, by themselves, examples of the phenomenon under consideration, the human reading and interpretation of particular texts, computer programs.

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Figure 1. An anonymous entry to the 1984 International Obfuscated C Code Contest that prints "hello, world!"

Reading in the main sense is about looking at something abstract. "Reading a photograph" sounds odd, perhaps because the photograph is not printed matter but also because it represents a framed perspective rather directly, with little abstraction. It is much more usual to read a diagram or map, because these are abstract representations. The development of software brought code into a legible condition. Cables patched into the ENIAC were not themselves legible, but assembly language for the stored-program EDSAC was. Human readability of programs was further enhanced as high-level programming languages, beginning with FORTRAN, were developed.

In the question and answer period after a lecture, Donald Knuth, the famous computer scientist who is author of *The Art of Computer Programming,* recalls reading the program SOAP from Stan Poley: "absolutely beautiful. Reading it was just like hearing a symphony, because every instruction was sort of doing two things and everything came together gracefully." He also remembers reading the code to a compiler written by Alan Perlis and others: "plodding and excruciating to read, because it just didn't possess any wit whatsoever. It got the job done, but its use of the computer was very disappointing." Knuth says of the aesthetics of reading programs and the reader's pleasure: "I do think issues of style do come through and make certain programs a genuine pleasure to read. Probably not, however, to the extent that they would give me any transcendental emotions." [6]

This discussion is not about any sentimental effects that code may have on the human reader, but does consider in detail the issues of programming style and the ways in which human readers read code. An aesthetic of code is suggested by Knuth's comments, one that is typified by beauty and grace and is clearly identified by Maurice Black in his dissertation, "The Art of Code":

> Computing culture ... has *adopted* a traditional model of literary aesthetics as a means of effecting change, finding political utility and social value in the well-crafted product that is at once entirely usable and wholly beautiful to contemplate. The distinctions are clearly evident in the respective disciplines' discourses: whereas terms such as "elegant" and "beautiful" circulate freely in computer culture to describe well-crafted code, elegance, beauty, and all their synonyms have been effectively exiled from the vocabulary of literary and cultural theory ... [1]

Black devotes a section to Knuth's aesthetic views and his concept of "literate programming," and another section to John Lions's book-length commentary on the beautiful, elegant Unix operating system. "The Art of Code" clearly establishes the classical aesthetic of programming as the dominant one in the discourse of software development. More recent articles, such as one entitled "Beautiful Code" that appeared in *Dr. Dobbs,* show that this aesthetic is still going strong: "Instead of searching for some automated measure ... perhaps we should be striving for beauty in our work because we believe that beautiful things are better." [3] It is fairly easy to find programmers extolling the beauty of programs and code snippets online, and also easy to find suggestions for writing elegant, clearly-written code in introductory programming textbooks.

There is a dark side to coding, however, one in which, even though a person can see into what would otherwise be the black box of the program, the source code itself is obscure, contrived to foil human legibility rather than enhance it.

3. HELLO, OBFUSCATION

In 1984 Landon Curt Noll and Larry Bassel held the first International Obfuscated C Code Contest. The contest was a success that has been repeated many times; judging of the 18th IOCCC was underway when this article was written. Only small, complete C programs can be entered in the contest, which rewards originality and the aesthetic abuse of the C language. The contest's stated goals include demonstrating the importance of programming style ("in an ironic way") and illustrating "some of the subtleties of the C language." [4]

An anonymous entry in the first IOCCC (Figure 1) accomplishes these goals in only two lines, and also plays on the conventional "hello, world!" program, a program which is typically used as a simple first example when a programming language. Brian Kernighan and Dennis Ritchie (the creator of C) begin their classic book *The C Programming Language* [5] with such a program:

```
#include <stdio.h>

main()

{

    printf("hello, world\n");

}
```

The obfuscated program prints "hello, world!" as it is supposed to, but in a very tortuous way. To see how this program comments on C programming style and the subtleties of C, it is necessary to discuss the program in detail, and to discuss the C programming language in detail. The explication that follows will be most easily followed by those who know how to program and will be best understood by those who have had some experience programming in C. However, the connection between the obfuscations seen in this code and the particular nature of C should be evident to some extent even to those who are not able, or do not wish, to follow all the details.

To begin, here is a clearer C program that prints "hello, world!":

```
main()
{
  write(0,"hello, world!\n",14);
}
```

Even this simple program comes with a bit more baggage than the BASIC equivalent, `10 PRINT "hello, world!"`, and it is more complex than the program Kernighan and Ritchie use

to introduce C. The system call `write` is used in this code with three arguments: `0` means the writing will be done to standard output; the second argument is the string to write, which includes a newline character encoded as `\n` at the end; and the third argument, `14`, is the length of the string, the number of characters in it. The following program adds one layer of obfuscation, by using a function to print out the `"hello, world!\n"` string one character at a time:

```
int i;

main()
{
  for(i=0 ; i<14 ; i++)
    {
    write_one_letter("hello, world!\n" + i);
    }
}

write_one_letter(letter)
{
  write(0,letter,1);
}
```

This makes it harder to see how the program works, but it makes visible some of the trickery that is possible, some would even say encouraged, in C. Notice that part of this program involves adding a string constant and a number, an operation which cannot be done in many strongly typed programming languages. In Java, where addition of String objects is defined as concatenation, evaluating the expression `("string" + 17)` involves constructing a String out of the number, then adding the two: the result is `"string17"`. A string constant in C is "really" a number, however, which means that adding a string and a number has an entirely different meaning. The string, seen as a number, is the address in memory where the first character resides. Add one to this number, and the result is the location of the second character. So this `for` loop, starting at position 0 and finishing at 13, has the effect of sending each character in the string to the `write_one_letter` function for printing.

To obfuscate the for loop a bit more, the `i<14` condition is written in a more elaborate way. Oddly enough, this condition could be written `"xxxxxxxxxxxxxx"[i]`, which has the effect of returning character number `i` from a string that has 14 characters in it. This yields a positive number (meaning TRUE) until `i` reaches 14, which corresponds to the end of the string; when the end of the string is reached it returns FALSE. This happens to be the case because strings in C are terminated with NULL, which, in C, means the same thing as FALSE. Now, to make things more puzzling, any array reference in C can either be written `a[b]` or `b[a]`. The values of `a` and `b` are added together and their sum is used to look up the array entry, so it doesn't matter which one is inside the brackets and which one comes before them. Thus, the condition can be written even more confusingly as `i["xxxxxxxxxxxxxx"]`. Also, any string that is 14 characters long can be used in this condition. To create additional confusion about the program's syntax, the fully-obfuscated program uses a different string to create the condition `i["]<i;++i){--i;}"]`. This makes it difficult to see where the data of the string ends and the code of the program begins.

The function `write_one_letter` is also given two additional, superfluous parameters and its name is changed to read. Redefining `read` to be a function that writes one letter is a particularly gruesome move, but this is allowed in C; `read` is a system call, not a keyword.

```
int i;

main()
{
  for(i=0 ; i["]<i;++i){--i;}"] ; i++)
    {
    read(0,"hello, world!\n" + i,1);
    }
}

read(j,letter,p)
{
  write(0,letter,1);
}
```

The meaningful name `letter` can be changed to `i` to make it seem as if this is the same `i` that was used previously — it is not. And, within the read function, `i` is written as `i--`, which suggests that the `i` up above might be getting decremented when this happens — it is not; this decrementing has no effect because *this* variable `i` "expires" immediately, at the end of the function. The call to `read` can be crammed into the increment part of the for statement, with the `++` operator is placed after `i`, to increment its value after the statement has been executed; then another `+` can be added to perform addition and make the puzzling-looking `+++`. The initialization of `i` to `0` can be left out. Integer variables in C are set to zero when they are defined, so the `i=0` in the program actually has no effect, except to make the program easier to understand. With these changes, the code looks like this:

```
int i;

main()
{
  for( ; i["]<i;++i){--i;}"] ;
    read(0,i+++"hello, world!\n",1));
}

read(j,i,p)
{
  write(0,i--,1);
}
```

There are only two differences between this code and the final obfuscated program: the formatting of the text and the use of some confusing ways to write zero and one. To turn to the second of these, one fancy way to write zero is `'-'-'-'`, that is, the numerical value of the `'-'` character subtracted from itself. Similarly, `'/'/'/'` divides the numerical value of the `'/'` character by itself, giving one. (Doing arithmetic with characters, like adding numbers and strings, is also not the most standard programming practice, although programmers are of course aware that characters have numerical representations.) The fancy zero and fancy one values that are obtained by doing this are passed to the read function as the variables `j` and `p`; that function then uses other elaborate ways to write zero and one.

`j/p+p` is always `0/2` in this code and thus always zero. `i/i` is always one. `i---j` is a way of writing `(i--)-j`, and, since `j` has the value zero, this does a meaningless subtraction and is the same as just writing `i--`. Adding in these elaborate ways of expressing zero and one, the code looks like this:

```
int i;

main()
{
  for( ; i["]<i;++i){--i;}"] ;
    read('-'-'-',i+++"hello, world!\n",'/'/'/'));
}

read(j,i,p)
{
  write(j/p+p,i---j,i/i);
}
```

The final program is the above code with all unnecessary whitespace removed and with the resulting line broken in two, using a backslash in the middle of the `"hello, world!\n"` string.

This example suffices to explain what obfuscations are and how they relate to the programming language in which they are written, although most IOCCC entries do far more elaborate things. Gavin Barraclough's 2004 entry, which won best of show, is exemplary. His program, less than 3600 characters in length, is actually formatted in a "friendly" way, but is cryptically scattered with one-letter variable names. The approximately two and a half pages of code provide, as the hint file explains,

> a 32-bit multitasking operating system for x86 computers, with GUI and filesystem, support for loading and executing user applications in elf binary format, with ps2 mouse and keyboard drivers, and vesa graphics. And a command shell. And an application - a simple text-file viewer. [4]

4. THE COMEDIAN AS THE LANGUAGE C

Some of the obfuscations that are seen in IOCCC, and some that can be seen in the "hello, world!" program, can be more or less universally applied by programmers, regardless of language. The use of meaningless variable names such as `j` and `p` is always possible. The deceptively-named variable `i` (which looks like an earlier variable `i`) and the misleadingly-named `read` function are other examples of a universal programming pitfall. Whenever variable and function names can be freely chosen, there is always the potential for the coder's choice to be uninformative or misleading. This can be intensified in C, where variable names are case sensitive; some programs take advantage of this to name variables `o` and `O`, for instance, inviting additional confusion with the number zero. This play, which can be called *naming obfuscation,* shows one very wide range of choices that programmers have. Such play refutes the idea that the programmer's task is automatic, value-neutral, and disconnected from the meanings of words in the world.

While these programs often critique or play with programming in general, the winning IOCCC programs also strongly assert their Cness. `a[b]` and `b[a]` do not mean the same thing in other languages, so a programmer could not choose the more confusing of the two. Other languages do not define the addition of strings and numbers, or they define it in a way that seems more intuitive, at least to beginning programmers. But C, by giving the programmer the power to use pointers into memory as numbers and to perform arithmetic with them, particularly enables this sort of *pointer confusion.* By showing how much room there is to program in perplexing ways — and yet accomplishing astounding results at the same time — obfuscated programs demonstrate that C is powerful, and also that clarity in C code is achieved only with effort.

The "fake ending" to the for loop in the hello world program, which is achieved by embedding a deceptive string `"]<i;++i){--i;}"`, is an example of *data/code confusion.* This is actually a mild example meant to fool a reader for a moment into thinking that this (meaningless) string is code; other obfuscated programs may transgress the code/data boundary in other ways, by consuming their source code as input, by generating their own code as output, or by modifying themselves as they run.

There is also an Obfuscated Perl contest, run annually by *The Perl Journal* since 1996. While Perl is quite unlike C, even beginning Perl programmers will be quick to realize the great potential for obfuscation that lies within the language. For one thing, Perl offers a dazzling variety of extremely useful special variables, represented with pairs of punctuation marks; this feature of the language nearly merits an obfuscation category of its own. Perl's powerful pattern-matching abilities also enable cryptic and deft string manipulations. Perl is sometimes de-acronymized as "Practical Extraction and Report Language," but has also been said to stand for "Pathologically Eclectic Rubbish Lister." The language is ideal for text processing, which means that printing "hello, world!" and other short messages can be done in even more interesting ways. Thus, the tradition of writing an obfuscated Perl program that prints "Just another Perl hacker," arose on USENET and became common enough that a program to do this is known simply as a JAPH. The popularity of these programs is attested to by the first section of the Perl FAQ, which answers the question "What is a JAPH?" [10]

More generally, Perl has as its mantra "there are many ways to do it." A half-dozen Perl programmers may easily know eight or ten different ways to code exactly the same thing. Because of this, obscure ways of doing fairly common tasks are lurking everywhere. A common, high-level obfuscation technique that is seen in obfuscated Perl and also in obfuscated C (however differently it may be expressed there) involves choosing the least likely way to do it. This could mean using a strange operator, a strange special variable, or an unusual function (or an ordinary function in an unusual way). It could also involve treating data that is typically seen as being one type as some other type, a view that is permissible according to the language but not intuitive.

Perl and C are distinguished by having obfuscated programming contests, but they are not widely despised languages — unlike, for instance, COBOL or Visual Basic. Why are these hateful programming languages not the targets of obfuscatory ridicule? The most obvious explanation is that the programmers who write obfuscated code are Perl and C hackers, often professional ones. They enjoy hacking in these languages, as do many free software developers and creative coders, and would not choose to program in COBOL or Visual Basic for fun. Their play with Perl and C is not pure pillory. In addition to making fun of some "misfeatures" or abusable features of the

languages, obfuscated code shows how powerful, flexible programming languages allow for creative coding, not only in terms of the output but in terms of the legibility and appearance of the source code.

What all obfuscations have in common — naming obfuscations and language-specific ones, such as choosing the least well-known language construct to accomplish something — is that they explore the *play* in a language, the free space that is available to programmers. If something can only be done one way, it cannot be obfuscated. The play in a programming language can also be used to make the program signify something else, besides being valid code that compiles or is interpreted to some running form.

5. MULTIPLE CODING

Recent IOCCC programs include a racing game in the style of *Pole Position,* a CGI-enabled Web server, and a maze displayer with code in the shape of a maze. It is common for obfuscated programs to be of unusual visual appearance. The code may spell out the name of the program, or the name of the contest, in large letters, or be in the form of some other ASCII art picture. This is a type of *double coding,* or, more generally, *multiple coding,* which can also be seen in Perl poetry and in "bilingual" programs.

The classic example of double coding in natural languages is the sentence "Jean put dire comment on tape," which is grammatical English and grammatical French, although each word has a different meaning in each language. (In French, the sentence means "Jean [male name] is able to say how one types.") Harry Mathews contributed to further French/English double coding by assembling the Mathews Corpus, a list of words which exist in both languages but have different meanings. In programming, an important first step was the 1968 *Algol* by Noël Arnaud, a book of poems composed from keywords in the Algol programming language. However, these poems are not executable programs; they are English poems that were assembled from a very restricted vocabulary. [8]

A notable modern ancestor of Arnaud's *Algol* is Perl poetry, in which text that can be read as poems are devised so as to also be valid Perl programs. As critics of code aesthetics have noted, even award-winning Perl poetry is often little more than an exercise of "porting" existing song lyrics into Perl, and the practice "does little to articulate the language of perl itself." [2] While it is possible to obfuscate a program, in the sense of the IOCCC or the Obfuscated Perl Contest, by fashioning it in the form of an English poem, the goals of competitive obfuscators and Perl poets appear to be quite different. Although a Perl poem must be a valid program, what the program actually does is often an afterthought in Perl poetry. For instance, the winning program in the first Perl Poetry Contest does nothing. The program's functioning is essential to obfuscated programming, though. So, while Perl poetry is an interesting phenomenon to many new media scholars, there are reasons, quite apart from any possible distaste for poetry, that this practice seems less interesting to programmers. The interesting phenomenon of multiple coding can be found in obfuscated programs, too, while these programs also feature impressive, intricate workings that are essential to their aesthetics.

Some other and quite extreme examples of multiple coding are also seen in programs that are "bilinguial" or "multilingual" and are analogous to "Jean put dire comment on tape" — they are valid computer programs in more than one computer language. These can be achieved by the re-use of keywords and operators or by using comments in one program to include code in another language.

6. HELLO, WEIRD

In the field of weird languages, also known as esoteric languages,[2] the programmer moves up a level to exploit not just the play of a particular language, but the play that is possible in programming language design itself. Weird programming languages are not designed for any real-world application or normal educational use; rather, they are intended to test the boundaries of programming language design. A quality they share with obfuscated code is that they often ironically comment on features of existing, traditional languages.

There are literally dozens of weird languages, commenting on many different aspects of language design, programming history and programming culture. A representative selection is considered here, with an eye towards understanding what these languages have to tell us about programming aesthetics.

Languages are considered in terms of four dimensions of analysis: 1) parody, spoof, or explicit commentary on language features, 2) a tendency to reduce the number of operations and strive toward computational minimalism, 3) to use of structured play to explicitly encourage and support double-coding, and 4) the goal of creating a puzzle, and of making programming difficult. These dimensions are not mutually exclusive categories, nor are they meant to be exhaustive. Any one weird language may be interesting in several of these ways, though one particular dimension will often be of special interest.

7. ABANDON ALL SANITY, YE WHO ENTER HERE: INTERCAL

INTERCAL is the canonical example of a language that parodies other programming languages. It is also the first weird language, and is highly respected in the weird language community. It was designed in 1972 at Princeton University by two students, Don Woods and James Lyon. (Later, while at Stanford, Woods was the co-author of the first interactive fiction, *Adventure*.) The explicit design goal of INTERCAL is

> …to have a compiler language which has nothing at all in common with any other major language. By 'major' we meant anything with which the author's were at all familiar, e.g., FORTRAN, BASIC, COBOL, ALGOL, SNOBOL, SPITBOL, FOCAL, SOLVE, TEACH, APL, LISP and PL/I." [13]

[2] "Esoteric" is a more common term for these languages, but it it is a term that could apply to programming languages overall (most people do not know how to program in any language) or to languages such as ML and Prolog, which are common in academia but infrequently used in industry. A better designation might be *art languages*. However, while such languages are undoubtedly a category of software art, developers of these languages do not use this term themselves, and it seems unfair to apply the term "art," with all of its connotations, to their work. While people might consider all sorts of languages to be "weird," that term's sense better captures the intention behind these languages, and it is used at times by language designers themselves.

INTERCAL borrows only variables, arrays, text input/output, and assignment from other languages. All other statements, operators and expressions are unique (and uniquely weird). INTERCAL has no simple `if` construction for doing conditional branching, no loop constructions, and no basic math operators — not even addition. Effects such as these must be achieved through composition of non-standard and counterintuitive constructs. In this sense INTERCAL also has puzzle aspects.

However, despite the claim that this language has "nothing at all in common with any other major language", INTERCAL clearly spoofs the features of contemporaneous languages, combining multiple language styles together to create an ungainly, unaesthetic style. From COBOL, INTERCAL borrows a verbose, English-like style, including optional syntax that increases the verbosity; all statements can be prepended with `PLEASE`. Sample INTERCAL statements in this COBOL style include `FORGET, REMEMBER, ABSTAIN` and `REINSTATE`. From FORTRAN, INTERCAL borrows the use of optional line numbers, which can appear in any order, to mark lines, and the `DO` construct, which in FORTRAN is used to initiate loops. In INTERCAL, however, every statement must begin with `DO`. Like APL, INTERCAL makes heavy use of single characters with special meaning, requiring even simple programs to be liberally sprinkled with non alphanumeric characters. In a sense, INTERCAL exaggerates the worst features of many languages and combines them together into a single language.

The compiler, appropriately called "ick," continues the parody. Anything the compiler can't understand, which in an normal language would result in a compilation error, is just skipped. This "forgiving" feature makes finding bugs very difficult; it also introduces a unique system for adding program comments. The programmer merely inserts non-compileable text anywhere in the program, being careful not to accidentally embed a bit of valid code in the middle of their comment.

The language manual hammers home the parody. After explaining that INTERCAL stands for "Compiler Language with No Pronounceable Acronym," the manual proceeds with a series of in jokes on language design. At one point the reader is presented with a logic diagram that claims to provide a simpler way of understanding the SELECT operation (SELECT being one of INTERCAL's two non-intuitive math operators): "The gates used are Warmenhovian logic gates, which means the outputs have four possible values: low, high, undefined …, and oscillating …" The reader is presented with a maze-like logic diagram in which lines needlessly zig-zag, sometimes dead-end, and all eventually connect at the system bus, the BUS LINE; of the many lines heading off diagram from the BUS LINE, all go "TO NEW YORK" except for the one "TO PHILIDELPHIA." All non-alphanumeric characters are given special names: tail (`,`), hybrid (`;`), mesh (`#`), worm (`-`) and so forth.

Thirty-three years later, INTERCAL still has a devoted following. Eric Raymond, the current maintainer of INTERCAL, revived the language in 1990 with his implementation C-INTERCAL, which added the `COME FROM` construct to the language — the inverse of the much-reviled `GO TO`.

8. MINIMALISM: BRAINFUCK

Languages that parody comment on other programming languages; languages in the minimalist vein, on the other hand, comment on the space of computation. Specifically, they call attention to the very small amount of structure needed to create a universal computational system. (A "system" in this sense can be as varied as a programming language, a formal mathematical system, or a physical processes, such as a machine.) A universal system can perform any computation that it is theoretically possible to perform; such a system can do anything that any other formal system is capable of doing, including emulating any other system. This property is what allows one to implement one language, such as Perl, in another language , such as C, or to implement an interpreter or compiler for a language directly in hardware (using logic gates), or to write a program that runs on some specific hardware to provide a platform for yet other programs (as the Java Virtual Machine does). Universality in a programming language is obviously a desired trait, since it means that the language places no limits on the processes that can be specified in the language. There are less powerful ways to compute, some of which are used often — for instance, regular expressions, of the sort found in the Find and Replace dialog of word processors, are powerful enough to tell whether a string has an even number of characters in it, but cannot determine whether the length of a string is a prime number, as a universal computer can.

Universal computation was discovered by Alan Turing and described in his 1937 investigation of the limits of computability, "On Computable Numbers." While his paper proved the counter-intuitive result that there exist formally specified problems for which there exists no computational process (that is, no program) for finding a solution, the important result for this paper was his definition of a notional machine, the Turing Machine, to specify what he meant by computation.

A Turing Machine consists of 1) an infinite tape, divided into cells (memory locations), along which a read/write head moves reading and writing symbols to and from the tape, and 2) a single state register that can store a symbol indicating the machine's current state. A Turing Machine is governed by a rule table which specifies, for each possible combination of state symbol and symbol read from the tape, what symbol the head will write to the tape, whether the head will move left or right, and what new symbol is stored in the state register. While it is easy to imagine that one could define a TM to compute a specific function, Turing proved that there exist TMs that can simulate the activity of any arbitrary TM; these are *universal Turing Machines*. The structure necessary to achieve universality is surprisingly small; for example, a universal TM can be defined using only 2 state symbols and 18 tape symbols (2x18).

Minimalist languages strive to achieve universality while providing the smallest number of language constructs possible. Such languages also often strive for syntactic minimalism, making the textual representation of programs minimal as well. Minimal languages are sometimes called Turing Tarpits, after epigram 54 in Alan Perlis' Epigrams of Programming: "54. Beware the Turing tar-pit in which everything is possible but nothing of interest is easy." [11].

Brainfuck is an archtypically minimalist language, providing merely seven commands, each represented by a single character. These commands operate on an array of 30,000 byte cells initialized to 0. The commands are:

> Increment the pointer (point to the memory cell to the right)

< Decrement the pointer (point to the memory cell to the left)

+ Increment the byte pointed to
- Decrement the byte pointed to
. Output the byte pointed to
, Accept a byte of input and write it into the byte pointed to
[Jump forward to the corresponding] if pointing to 0
] Jump back to the command after the corresponding [if pointing to a non-zero value.
A Brainfuck "hello, world" program follows:

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.++
+++++..+++.>++.<<+++++++++++++++.>.+++.------.---
-----.>+.>.
```

Minimalist languages also comment on computer architectures as well the nature of computation, and can have the flavor of a minimal assembly language. The language OISC explicitly parodies assembly language, for example. OISC stands for the "One Instruction Set Computer", referencing the standard acronyms RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer). OISC consists of a single instruction, subtract-and-branch-unless-positive. `subleq(a, b, c)` subtracts the contents of memory location a from the contents of memory location b, stores the result in b, and, if the result of the subtraction was 0 or negative, jumps to the address stored in memory location c. Assembly languages commonly contain separate arithmetic operations (add and subtract), as well as various branch operations that test a memory location and branch if the memory location is, for example, positive, or negative, or zero. OISC parodies assembly by combining an arithmetic and branch operation into a single instruction and providing that to the programmer as the *only* instruction.

9. STRUCTURED PLAY: SHAKESPEARE

Some weird languages encourage double coding by structuring the play within the language such that valid programs can also be read as a literary artifact. As was previously described, double-coding is certainly possible in languages such as C and Perl, and in fact is an important skill in the practice of obfuscated programming. But where C and Perl leave the space of play relatively unstructured, forcing the programmer to shoulder the burden of establishing a double coding, structured play languages, through their choice of keywords and their treatment of programmer defined names (e.g. variable names), support double coding within a specific genre of human-readable textual production. The language Shakespeare exemplifies this structured play aspect.

Here is a fragment of a Shakespeare program that reads input and prints it out in reverse order:

```
[Enter Othello and Lady Macbeth]

Othello:
You are nothing!

                Scene II: Pushing to the very end.

Lady Macbeth:
Open your mind! Remember yourself.

Othello:
You are as hard as the sum of yourself and a stone
wall. Am I as horrid as a flirt-gill?

Lady Macbeth:
If not, let us return to scene II. Recall your
imminent death!

Othello:
You are as small as the difference between
yourself and a hair!
```

Shakespeare structures the play of the language so as to double-code all programs as stage plays, specifically, as spoofs on Shakespearean plays. This is done primarily by structuring the play (that is, the free space) that standard languages provide in the naming of variables and constants. In standard languages, variable names are a free choice left to the programmer, while numeric constants (e.g. 1) are either specified by the textual representation of the number, or through a name the programmer has given to select constants. In contrast, Shakespeare Dramatis Personae (variables) must be the name of a character from some Shakespeare play, while constants are represented by nouns. The two fundamental constants in Shakespeare are -1 and 1. The dictionary of nouns recognized by the Shakespeare compiler have been divided into positive, negative, and neutral nouns. All positive (e.g. "lord", "angel", "joy") and neutral (e.g. "brother", "cow", "hair") nouns have the value 1. All negative nouns (e.g. "bastard", "begger", "codpiece") have the value -1.[3] Constants other than -1 and 1 are created by prefixing them with adjectives; each adjective multiplies the value by 2. So `sorry little codpiece` denotes the number -4.

The overall structure of Shakespeare follows that of a stageplay. Variables are declared in the Dramatis Personae section. Named acts and scenes become labeled locations for jumps; `let us return to scene II` is an example of a jump to a labeled location. Enter and exit (and exeunt) are used to declare which characters (variables) are active in a given scene; only two characters may be on stage at a time. Statements are accomplished through dialog. By talking to each other, characters set the values of their dialog partner and themselves, compare values, execute jumps, and so forth. Conditional jumps are accomplished by one character posing a true or false question, and the second character describing what action to take based on the truth value. Such a jump appears in the code previous code sample, where Othello asks Lady Macbeth `Am I as horrid as a flirt-gill?` (is the value of the variable Othello equal to -1), and Lady Macbeth responds `If not, let us return to scene II.`

In a programming language, keywords are words that have special meaning for the language, indicating commands or constructs, and thus can't be used as names by the programmer. An example from C is the keyword `for` used to perform iteration; `for` can not be used by the programmer as the name of a variable or function. In standard languages, keywords typically limit or bound play, as the keywords are generally not selected by language designers to facilitate double-coding. This is, in fact, what makes code poetry challenging; the code poet must hijack the language keywords in the service of a double-coding. In contrast, weird languages that structure play provide keywords to facilitate the double-coding that is generally encouraged by the language. Shakespeare keywords maintain a stylistic consistency with a melodramatic spoof of

[3]Interestingly, "Microsoft" is in the negative noun list.

Shakespearean plays. Output is accomplished via `Open your heart` (output value as number) and `Speak your mind` (output value as character), input by `Listen to your heart` (input value as number) and `Open your mind` (input value as character). A number of comparative synonyms are provided for accomplishing inequality tests. For example, `friendlier` and `jollier` perform the greater-than test, as in `are you friendlier than a fatherless bastard?`, while `punier` and `worse` perform the less-than test, as in `are you punier than a gentle king?`

Another language, Chef, illustrates different design decisions for structuring play. Chef facilities double-coding programs as recipes. Variables are declared in an ingredients list, with amounts indicating the initial value (e.g., `114 g of red salmon`). The type of measurement determines whether an ingredient is wet or dry; wet ingredients are output as characters, dry ingredients are output as numbers. Two types of memory are provided, mixing bowls and baking dishes. Mixing bowls hold ingredients which are still being manipulated, while baking dishes hold collections of ingredients to output. What makes Chef particularly interesting is that all operations have a sensible interpretation as a step in a food recipe. Where Shakespeare programs parody Shakespearean plays, and often contain dialog that doesn't work as dialog in a play ("you are as hard as the sum of yourself and a stone wall"), it is possible to write programs in Chef that might reasonably be carried out as a recipe. Chef recipes do have the unfortunate tendency to produce huge quantities of food, however, particularly because the sous-chef may be asked to produce sub-recipes, such as sauces, in a loop.

A number of languages structuring play have been based on other weird languages. Brainfuck is particularly popular in this regard, spawning languages such as FuckFuck (operators are replaced with curse words) and Cow (instructions are all the word "moo" with various capitalizations).

10. THE SUN THE SUN, HIS MIND PUZZLE: MALBOLGE

Languages that have a puzzle aspect explicitly seek to make programming difficult by providing unusual, counter-intuitive control constructs and operators. While INTERCAL certainly has puzzle aspects, its dominant feature is its parody of 1960s language design. Malbolge, named after the eighth circle of hell in Dante's *Inferno*, is a much more striking example of the puzzle language. Where INTERCAL sought to merely have no features in common with any other language, Malbolge had a different motivation, as author Ben Olmstead writes:

> It was noticed that, in the field of esoteric programming languages, there was a particular and surprising void: no programming language known to the author was specifically designed to be difficult to program in.
>
> Certainly, there were languages which were difficult to write in, and far more were difficult to read (see: Befunge, False, TWDL, RUBE...). But even INTERCAL and BrainF***, the two kings of mental torment, were designed with other goals …
>
> Hence the author created Malbolge. ... It was designed to be difficult to use, and so it is. It is designed to be incomprehensible, and so it is.
>
> So far, no Malbolge programs have been written. Thus, we cannot give an example. [9]

Malbolge was designed in 1998. It was not until 2000 that Andrew Cooke, using AI search techniques, succeeded in generating the first Malbolge program, the "hello, world!" program — actually, it prints `HEllO WORld` — that follows:

```
(=<`$9]7<5YXz7wT.3,+O/o'K%$H"'~D|#z@b=`{^Lx8%$Xmr
kpohm-kNi;gsedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA
@?>=<;:9876543s+O<oLm
```

Lou Scheffer performed a cryptanalysis of Malbolge and discovered "weaknesses" that the programmer can systematically exploit:

> The correct way to think about Malboge, I'm convinced, is as a cryptographer and not a programmer. Think of it as a complex code and/or algorithm that transforms input to output. Then study it to see if you can take advantage of its weaknesses to forge a message that produced the output you want. [12]

His analysis proved that the language allowed for universal computation. The "practical" result was the production of a Brainfuck to Malbolge compiler.

What makes Malbolge so difficult? Like many minimalist languages, Malbolge is a machine language written for a fictitious and feature-poor machine, and thus gains some difficulty of writing and significant difficulty of reading from the small amount of play provided to the programmer for expressing human, textual meanings. However, as Olmstead points out, the mere difficulty of machine language is not enough to produce a truly devilish language. The machine model upon which Malbolge runs has the following features which contribute to the difficulty of the language:

Trinary machine model. Programmers are used to all number representations bottoming out in binary representation at the machine-level. By making trits rather than bits the fundamental representation, this de-familiarizes the machine. This trinary orientation is borrowed from tri-INTERCAL, a trinary variant of INTERCAL.

Minimalism. Malbolge provides a minimal computational model. There are three registers, two of which are a data pointer and a code pointer, and seven instructions, represented by the ASCII characters (`j i * p < / v`). `j` and `i` manipulate the data and code pointer, `*` and `p` perform two trinary operations, `<` and `/` read and write characters from the A (accumulator) register, and `v` stops the machine.

Counterintuitive operations. Like INTERCAL, Malbolge does not provide standard constructs, such as conditional branching or arithmetic. Instead those operations must be built from two operations. `*` rotates the trinary cell pointed to be the D pointer 1 trit to the right. (Actually, bit-wise rotation is a standard operation on most computers — by providing this construct, Malbolge is being uncharacteristically forgiving.) `p` performs a tritwise operation on the contents of the A register and the number pointed to by D register. The p operation, often referred to as the *crazy* op, purposefully corresponds to no natural operation. In presenting the table that describes how trits are combined by the crazy op, Olmstead writes "don't look for a pattern, it's not there."

Indirect instruction decoding. In standard machine models of computation, the code that will be executed next is determined by a program counter. Usually, after executing one instruction, the program counter is simply incremented so that it points to the next one. The only other thing that can happen is a "branch," which corresponds, for instance, to `if` and `GOTO` statements. In this case, the execution of the current instruction causes the program counter's value to change, so that it points to some other location in memory. In either situation happens, the code that runs next is sitting somewhere in memory; it is directly fetched and run. In standard machine models, the instructions as laid out in memory are exactly the instructions the machine will execute.

Malbolge, in contrast, performs a complicated transformation on the instruction pointed at by the code pointer before executing it. As the manual states:

> When the interpreter tries to execute a program, it first checks to see if the current instruction is a graphical ASCII character (33 through 126). If it is, it subtracts 33 from it, adds C [the code pointer] to it, mods it by 94, then uses the result as an index into the following table of 94 characters:
>
> ```
> +b(29e*j1VMEKLyC})8&m#~W>qxdRp0wkrUo[D7,XTcA"lI
> .v%{gJh4G\-=O@5`_3i<?Z';FNQuY]szf$!BS/|t:Pn6^Ha
> ```

If the character indexed in the table is one of the seven characters corresponding Malbolge operations, the operation is executed. Otherwise the machine does nothing, except to increment both the code pointer and the data pointer (the constant incrementing of the data pointer provides another annoyance for the programmer). Note that the transformation depends on *where* the instruction resides in memory because C (the code pointer) is added as part of this step; the same value would execute as two different instructions at two different locations in memory. A Malbolge programmer cannot lay out the instructions she wants executed, but must lay out instructions so that after they have been taken through this complicated transformation, the eventual result will be the instructions that were supposed to be executed in the first place. To make matters more difficult, Malbolge programs can only consist of the seven characters that correspond to operations; the programmer can't simply write a program consisting of non-operation characters that will transform to operations.

Mandatory self-modifying code. In standard programming practice, code is treated as immutable. Though both code and data reside as patterns in memory, the block of memory patterns corresponding to code remains fixed, while the block of memory patterns corresponding to data is manipulated by the executing code. Self-modifying code treats its code block as mutable, literally changing its own operations as it runs. Self-modifying code is notoriously difficult to read and write; where the textual representation of the program is by necessity static, the structure of the process dynamically changes over time. In Malbolge, the programmer is forced to write self-modifying code, as code modification is built into the definition of code execution:

> After the instruction is executed, 33 is subtracted from the instruction at C, and the result is used as an index in the table below. The new character is then placed at C, and then C is incremented.
>
> ```
> 5z]&gqtyfr$(we4{WP)H-Zn,[%\3dL+Q;>U!pJS72FhOA1C
> B6v^=I_0/8|jsb9m<.TVac`uY*MK'X~xDl}REokN:#?G"i@
> ```

So, in addition to the complexities added by the indirect instruction decoding, the instructions are constantly changed by an arbitrary transformation. It is therefore impossible to write code in Malboge that does the same thing twice in a row. These factors account for the two years that passed before the first Malbolge "hello, world" program appeared.

Scheffer, in his cryptanalytic treatment of Malbolge, discovered a number of "weaknesses" that made it possible to write arbitrary programs in Malbolge — proving, therefore, that is is capable of universal computation. The most notable weaknesses are as follows: The permutation table used to modify code exhibits short cycles — that is, if one choses carefully, instructions can be selected that turn back into themselves before very long. Specifically, a permutation cycle is a sequence of code transformations that comes back to itself. The `p` instruction (the crazy op), when located at memory location 20, will turn into the `j` instruction (to store a value in memory) the first time it is executed, then into a "no op" (do nothing) once the `j` instruction is executed, then into another no op when the no op is executed, and finally, after this no op is executed, back to the `p` instruction. Another forgiving aspect of Malboge is that the branch instruction, i, is not modified, nor is its target. Exploiting these regularities allowed Scheffer to develop general Malbolge code constructs that, for example, allow one to create a block of code that performs a given function every other time it is executed, one that safely does nothing the alternate times. This discoveries paved the way for the creation of a BrainFuck to Malbolge compiler.

11. TOWARD A BROADER CODE AESTEHTICS

Programs in weird languages generally have the property of being difficult to read. This suggests that weird languages may be "auto-obfuscating," requiring obfuscation from programmers. But obfuscated code contests are not about merely producing code that is hard to read; they are about exploiting the syntax and semantics of the language to commenting on the language itself. Weird languages emphasizing minimalism and puzzles are "merely" hard to read in the same way that assembly language is hard to read; they provide so little play that it is virtually impossible to double-code interestingly. Languages structuring play, in contrast, are hard to read because of the insistence of the enforced double-coding. The textual meaning of the program is inevitably *not* about the procedural meaning of the program, but about some unrelated domain. Of the weird languages described here, it may be only INTERCAL that is truly auto-obfuscating. Since INTERCAL parodies several languages, resulting in a language in which nothing can be expressed cleanly or elegantly, the difficulty of reading INTERCAL programs is a result of such programs being about the parody languages, and thus in some sense about INTERCAL itself.

By commenting on the nature of programming itself, weird languages point the way towards a refined understanding of the nature of everyday coding practice. In their parody aspect, weird languages comment on how different language constructions influence programming style, as well as on the history of programming language design. In their minimalist aspect, weird languages comment on the nature of computation and the vast variety of structures capable of universal computation. In their puzzle aspect, weird languages comment on the inherent

cognitive difficulty of constructing effective programs. And in their structured play aspect, weird languages comment on the nature of double-coding, how it is the programs can simultaneously mean something for the machine and for human readers.

All of these aspects are seen in everyday programming practice. Programmers are extremely conscious of language style, of coding idioms that not only "get the job done", but do it in a way that is particularly appropriate for that language. Programmers actively structure the space of computation for solving specific problems, ranging from implementing sub-universal abstractions such as finite-state machines for solving problems such as string searching, up to writing interpreters and compilers for custom languages tailored to specific problem domains, such as Perl for string manipulation. All coding inevitably involves double-coding. "Good" code simultaneously specifies a mechanical process and *talks about* this mechanical process to a human reader. Finally, the puzzle-like nature of coding manifests not only because of the problem solving necessary to specify processes, but because code must additionally, and simultaneously, double-code, make appropriate use of language styles and idioms, and structure the space of computation. Weird languages thus tease apart phenomena present in all coding activity, phenomena that must be accounted for by any theory of code.

Programming has already been connected to literature in an interesting way, albeit without deep consideration of obfuscation and weird languages as programming practices.[1] Obfuscation and weird languages invite us to join programming contexts to the literary contexts that must obviously be considered when evaluating literary code. They also suggest that coding can resist clarity and elegance to strive instead for complexity, can make the familiar unfamiliar, and can wrestle with the language in which it is written, just as much contemporary literature does. When a program is double-coded to have some literary meaning, or indeed, any human meaning, this meaning can play with what programming language researchers call the semantics of the code: what the code actually does as it executes.[4] A very simple case of such play can even be seen in the obfuscated C "hello, world!" program, in which `read` is used to name a function that writes one letter. Such play, the levels of human meaning and machine meaning must both be considered.

As the name "Turing Machine" suggests, the computer is a machine. Whether it is realized as a physical device or imagined and abstract, it is made up of parts and performs tasks. A tradition of overcomplicated machinery has manifested itself in art in several ways, but perhaps most strikingly in Alfred Jarry's 'Pataphysics, "the science of imaginary solutions," which involves the design of complicated physical machinery and also the obfuscation of information and standards. As a joke, and as a parody of the complex French calendar, Jarry introduced a new calendar. It begins on his birthday and is divided into thirteen months, each of 29 days. Each day has an obscure name in the pataphysical calendar, and the last day of the month is, in all but two cases, an imaginary day. The second month, for instance, is "Haha," and the second day is "Dissolution of Edgar Allan Poe, dinomythurge." The Collège de 'pataphysique revises the calendar once in a while, changing the names of days.

[4] This is the view in operational semantics, at any rate; there are also other ways to consider program semantics.

An aesthetic of mechanical obfuscation is also seen in the kinetic installations of Peter Fischli and David Weiss and in their film "The Way Things Go" (1987-1988), as well as in the earlier visual art of Robert Storm Petersen, Heath Robinson, and Rube Goldberg. (The weird language RUBE was so named as a tribute to Goldberg.) These depictions and realizations of mechanical ecstasy comment on engineering practice and physical possibility, much as obfuscated coding and weird languages comment on programming and computation. These "art machines," like obfuscated programs, are interesting because they do something in a very complex way, but to be worth anyone's attention they must actually do *something* and have a machine meaning as well as a human one.

Perhaps most oddly, obfuscated programs and weird languages are inviting, asking for the full engagement of those who read them or program in them, offering to show how strangely things can be done. They invite theorists and critics of new media to look into the dark box of the machine and see how creativity is at work in there, too. To understand how programmer-artists, programmer-authors, game developers, and hackers of other stripes achieve what they do, it will be necessary to understand the full range of programming practices, to not just play with the finished, executable file, but to also consider the play that happens in programming it.

12. REFERENCES

[1] Black, M. J. The Art of Code. Ph.D. Dissertation, University of Pennsylvania. 2002.

[2] Cox, G., A. McLean, and A. Ward. The Aesthetics of Generative Code. http://www.generative.net/papers/aesthetics/ 2000.

[3] Heusser, M. Beautiful Code. *Dr. Dobb's*. www.ddj.com/documents/ddj1122411683430/ 2005.

[4] International Obfuscated C Code Contest. http://www.ioccc.org/

[5] Kernighan, B. W. and D. M. Ritchie. *The C Programming Language.* 2nd Ed. Prentice Hall, Englewood Cliffs, New Jersey. 1988.

[6] Knuth, D. E. *Things a Computer Scientist Rarely Talks About.* Center for the Study of Language and Information, Stanford, California. 2001.

[7] Lohr, Steve. *Go To.* Basic Books, New York. 2001.

[8] Mathews, H. and A. Brotchie, eds. *Oulipo Compendium.* Atlas Press, London. 1998.

[9] Olmstead, B. Malboge. http://www.antwon.com/other/malbolge/malbolge.txt 1998.

[10] Perl 5.6 FAQ. 23 May 1999. http://www.perldoc.com/perl5.6/pod/perlfaq1.html

[11] Perlis, A. Epigrams on Programming. SIGPLAN Notices, 17(9), September 1982. http://www.bio.cam.ac.uk/~mw263/Perlis_Epigrams.html

[12] Scheffer, L. http://www.lscheffer.com/malbolge.html

[13] Woods, D. and J. Lyon, *The INTERCAL Programming Language Revised Reference Manual*. 1st Ed. 1973, C-INTERCAL revisions, L. Howell and E. Raymond, 1996.

[14] WordNet 2.1. http://wordnet.princeton.edu/