Weird Languages (1)
For Software Studies
Michael Mateas, michaelm@cc.gatech.edu

Programming languages are often seen as given, an immutable logic within which everyday coding practice takes place. Viewed in this light, a programming language becomes a tool to be mastered, a means to an end. The practice of writing obfuscated code (see Montfort, this volume) exploits the syntactic and semantic play of a language to create code that, often humorously, comments on the constructs provided by a specific language. But the constructs and logics of languages are themselves contingent, abstractions pulled into being out of the space of computational possibility, and enforced and maintained by nothing more than programs, specifically the interpreters and compilers that implement the language.

In the field of weird languages, also known as esoteric languages (2), programmers explore and exploit the play that is possible in programming language design. 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 itself. A quality they share with obfuscated code is that they often ironically comment on features of existing, traditional languages.

There are literally dozens, if not hundreds 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) the 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.

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." [Woods & Lyon 1973]

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 verbose, English-like constructs, 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. INTERCAL exaggerates the worst features of many languages and combines them together into a single language.

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`.

While parody languages comment on other programming languages, languages in the minimalist vein 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. Universal computation was discovered by Alan Turing and described in his 1937 investigation of the limits of computability, "On Computable Numbers." [Turing 1936] 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 provides a virtual hardware platform for 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.

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.” [Perlis 1982].

Brainfuck is an archetypically minimalist language, providing merely eight 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 program which prints out the string “Hello World”, follows.

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

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. 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 specific constants. In contrast, Shakespeare Dramatis Personae (variables) must be the name of a character from a Shakespeare play, while constants are represented by nouns. The two fundamental constants in Shakespeare are -1 and 1. The 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”, “beggar”, “codpiece”) have the value -1. 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.

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.

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. Thus, in some sense, Chef structures play to establish a *triple-coding*: the executable machine meaning of the code, the human meaning of the code as a literary artifact, and the executable human meaning of the code as steps that can be carried out to produce food.

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 (all instructions are the word "moo" with various capitalizations).

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 a 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…
> 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." [Olmstead 1998]

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
```

The writing of more complex Malbolge programs was enabled by Lou Scheffer's cryptanalysis of Malbolge in which he 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." [Scheffer] 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: a trinary, rather than binary, machine model, minimalism, counterintuitive operations, indirect instruction coding (the meaning of a program symbol depends on where it sits in memory), and mandatory self-modifying code (code mutates as it executes, so it never does the same thing twice). These factors account for the two years that passed before the first Malbolge "hello, world" program appeared.

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.

(1) Parts of this article are based on a paper (Mateas and Montfort 2005) that Nick Montfort and I presented at Digital Arts and Culture 2005.

(2) "Esoteric" is a more common term for these languages, but 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. The term “weird” better captures the intention behind these languages, and is used at times by the language designers themselves.

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

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

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

Turing, A. M. On Computable Numbers, With an Application to the Entscheidungsproblem. In *Proc. London Math. Soc.*, 2(42) (1936), 230-265; A correction’ *ibid*, 43, 544-546.

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