The ppg256 Series of Minimal Poetry Generators

Nick Montfort
Massachusetts Institute of Technology
77 Massachusetts Ave, 14N–233
Cambridge, MA 02139 USA
nickm@nickm.com

ABSTRACT
I discuss the four Perl poetry generators I have developed in the ppg256 series. My discussion of each program begins with the entire 256 characters of code and continues with an explication of this code, a description of aspects of my development process, and a discussion of how my thinking about computation and poetry developed during that process. In writing these programs, I came to understand more about the importance of framing to the reception of texts as poems, about how computational poetic concepts of part of speech might differ from established linguistic ones, about morphological and syntactical variability, and about how to usefully think about possible texts as being drawn from a probability distribution.

Keywords
form, minimal systems, poetics, poetry generation, programming, text generation, writing under constraint

1. THE ppg256 SERIES
Since 2007, I have been working on a series of very short Perl poetry generators. This ppg256 series [1] is comprised of systems that are simply 256 characters of code. They run in a standard Perl interpreter, using no external data sources, online or local, making use of no special libraries and invoking no other programs. These tiny programs are investigations into language, poetics, and computation.

My work has been assisted by the constrained form I set for myself, which has helped me to focus on the computational manipulation of strings to generate poetic language. This has kept me from delving into complicated (and even rather simple) statistical models, crawling the Web or other large-scale data sources to seek patterns, and implementing elaborate AI systems employing planning or search. In part, I undertook this project as an alternative to a large-scale creative text-generating system that I was working on and have continued to work on [2]. The ppg256 series, which is also an ongoing project, currently consists of four programs. The fourth of these was written to output all the control characters needed to drive an LED sign, so that the 256-character program by itself is sufficient to control this display.

The way the ppg256 programs actually operate is represented very poorly, if not completely misrepresented, by single poems shown as sample output. Such output suggests that an excerpted product, perhaps ready for submission to a literary magazine, is the ultimate goal of this project. This is not the case. This perspective can lead one to overlook the importance of the code (as human-readable and as machine-interpretable) and the texture of the continually-generated language that is produced by a running program. Ideally, the programs in this project should be obtained by the reader as source code (all of it is provided in this paper, although within each program, the lines will need to be joined to make one long line) and should be run. The output should then be read for a while. By way of introducing these programs in a paper, however, and to show that these programs can at least be imagined as having different styles and voices, here are four example poems, one generated by each of the four programs:

```
the rank

 pots at rats
 rand to pang
 dink no mash

       ◊

the pans

 sin
 the shin skits skit
 pit & chill
 skill
 shit
 grin & chill
 of fill
 kit twits to chin
 twin

       ◊

the__boyman
and
one__godape
top_it

       ◊

misflip on flowlon, guy
```

In the discussion that follows, I describe what some of my goals were as I started on each program. I then explicate each program and describe the development process and how this constrained programming practice is an investigation into poetics, computation, and language. Specifically, as I describe the four programs, I address why programming is part of my practice, why I use Perl rather than some other language, why I have chosen to constrain the programs to be only 256 characters in length, why I believe that drawing from a distribution of possible texts is an interesting way to engage with and conceptualize the production

Digital Arts and Culture, December 12–15, 2009, Irvine, California, USA.

of language, and why I selected a special type of display for use with the fourth program in the series. I characterize this project as it relates to the work of other poetry generator developers. I also provide specific examples of things I have learned about poetry and language through this practice, findings I do not believe I would have reached by simply writing poems or by using standard computer science and computational linguistics methods.

While I am concerned with how a computer system can be read and understood as creative, my project serves to invite and provoke readers. It is not an attempt to contribute to the theory of creativity. My project deals with the poet's cognition in composition, the programmer's cognition in programming, the reader's cognition in understanding how a program works, and the reader's cognition in reading poems. These are matters of poetics (of both human language and code) and aesthetics (of both code and human language). ppg256 does not attempt to contribute to cognitive science; Furthermore, I have not drawn on any insights from cognitive science in the work I have undertaken so far. Rather, this project is an attempt to inquire about how we think as we undertake these types of production and reception, and about the nature of computation and the English language, from a standpoint that is a complement to scientific perspectives: that of a creative digital media practice. This paper offers no theoretical conclusions. Instead, it tries to document my practice and to serve as a trace of my journey so far, mentioning some ways in which my thinking has developed as I have developed these programs.

2. ppg256-1

```
perl -le 'sub b{@_=unpack"(A2)*",pop;$_[rand
@_]}sub w{" ".b("cococacamamadebapabohamolab
uratamihopodito").b("estsnslldsckregspsstedb
snelengkemsattewsntarshnknd")}{$_="\n\nthe".
w."\n";$_=w." ".b("attoonnoof").w if$l;s/[au
][ae]/a/;print;$l=0if$l++>rand 9;sleep 1;red
o}'
```

The same code, with added whitespace and comments:

```
sub b{ # Return a bigram stored in a string
@_ = unpack"(A2)*",pop;
$_[rand@_]}

sub w{ # Return a word: space+bigram+bigram
" ".b("cococacamamadebapabohamolaburatamihop
odito").b("estsnslldsckregspsstedbsnelengkem
sattewsntarshnknd")}

{ # Main loop
$_ = "\n\nthe".w."\n";
$_ = w." ".b("attoonnoof").w if $l;
s/[au][ae]/a/;
print;
$l = 0 if $l++ > rand 9;
sleep 1;
redo}
```

2.1 Goals

As I wrote in discussing an earlier version of this program [3], my new year's poem for 2008, written at the end of 2007, was a computer program. The poem I distributed on New Year's was an earlier version of the code that was just quoted. Developing this program involved attempts to drive process intensity up, keep program size down, and uncover what significant and yet very simple moves could be made by a poetry generator.

Specifically, I was interested in creating a tiny program that would generate texts that people would recognize as poems. I wanted the program to generate a wide variety of words — to have a large vocabulary — but I was willing to have syntactical regularity, words of similar or the same length, and words that did not appear in a dictionary, as long as most words were dictionary words and as long as the others looked somewhat like English-like.

2.2 Explication of Code

A subroutine, b(), is defined first; then another subroutine, w(). At a high level, b() picks a bigram (a two-letter sequence) from a string, and w() calls b() twice to generate a four-letter word with a space in front of it. The main loop of the program follows these subroutines and is surrounded by braces. In that loop, it is the "redo" at the very end that causes the program to loop forever — or at least until the program is interrupted by the user pressing ctrl-C, by a power outage, or by something else outside the program itself.

The main loop begins by assigning a value to "$_" — the Perl special variable representing what I'll call the default string. (The full, proper name is "the default input and pattern-searching space.") What gets assigned to this string is two newlines, the word "the", a four-letter generated word with a space in front of it, and another newline. This value is a title, something like "the rank" with two newlines before it and one after it.

The next part of the main loop obliterates the title, replacing it with a line if the program isn't on "line zero." That is, if $l has a non-zero value, it overwrites the default string with a space plus a four-letter word, a short closed-class word ("at", "to", "on", "no", or "of"), and another space plus four-letter word. The result is something like "pots at rats". The next part of the main loop cleans up a bit by replacing "aa", "ae", "ua", and "ue" with the letter a. This improves the line in some cases, changing some four-letter strings (most of which don't correspond to English words) to three-letter words (most of which do). The "print;" then prints out whatever is in the default string; it will have a newline appended because the "-l" option to Perl is used. The next bit of code increments $l (which has the value 0 to begin with), checks to see if it is greater than a random number between 0 and 9, and sets $l back to 0 (to start generating a new poem) if it is. Except for a pause of a second, that's all there is before the "redo" at the end.

The b() subroutine unpacks a string (such as "attoonnoof") into two-letter sections and picks one of those (such as "at") uniformly at random. This bigram is then returned.

The w() subroutine joins together a space, two letters from the first long string, and two letters from the second long string. The first string,

"cococacamamadebapabohamolaburatamihopodito"

holds 21 bigrams, the first of which is "co" and the last of which is "to". The second one,

"estsnslldsckregspsstedbsnelengkemsattewsntarshnknd"

holds 25 bigrams. Each of the 25 bigrams in the second string are chosen uniformly at random by the program, but the bigrams in the first string are not equiprobable because “co”, “ca”, and “ma” are repeated; there are only 18 unique bigrams represented. This is a reasonably inexpensive way to define a non-uniform probability distribution, one in which certain choices are more likely. I selected these two sets of bigrams by considering the most frequent two-letter beginnings to four-letter words and their most frequent two-letter endings. The words generated by w() can be found in a dictionary about 60% of the time, and even when they are not, they often still seem to be plausible as English words or names. The substitution of “aa”, “ae”, “ua”, and “ue”, done in the main loop, improves this percentage.

2.3 Development Process

I began, inspired by Hugh Kenner and Joseph O'Rourke’s Travesty [4] and Charles O. Hartman’s work with that system, by looking at how I might compactly and interestingly encode the distribution of English letters — the unigram distribution — to generate strings that looked English-like. (I have also had Hartman’s engaging discussion of his poetry generation development [5] in mind as I wrote this paper.)

Although my finished program does not use external data sources, and was never intended to, I did make use of such sources to determine properties of English and of the sort of language I wanted to generate. I took a text file edition of *Moby Dick* and used the frequency of letters to determine a unigram distribution. I wrote several true one-liner Perl programs (not having settled on the 256-character constraint yet) to print letters and spaces, approximating this probability distribution. I figured out how to do this somewhat compactly and cleverly. But as Kenner and Hartman, and Claude Shannon before them, found, this method produces at best a distant shadow of English, very seldom resulting in a word and certainly not in anything with more structure. The process was like dumping a bag of Scrabble tiles on the table. For instance, this encoding of an approximate English unigram frequency distribution in a 65-character Perl program only produces English words about 3% of time, and these are almost all one- and two-letter words:

```
perl -e'{print substr("we cleft mud"." in ea
rshot "x3,rand 54,1);redo;}'
```

A decent model for language of course does not generate each letter independently, as the line of code above does. It represents the conditional probability of letters as they appear in a sequence. For instance, “u” is extremely likely as a next letter if the current letter is “q”. But building a conditional probability model into a very tiny program, a one-line (or even slightly longer) Perl program, seems impossible. There is too much information — 26×26 probabilities — to pack into a few bytes.

One alternative to this limited technique would be to find extremely representative data to put into the program itself, something that was a distillation of English. I looked into whether I could find any kind of encodings of English which were themselves English — for instance, words or sentences whose substrings were all, or almost all, also English. I also sought words that could be beheaded (that is, their first letters could be removed) multiple times to create new words. An advantage of this approach, seen also in the one-liner above, was that the data contained in my tiny program would end up being legible itself. It was a nice idea, but getting a tiny program to generate language even without using a legible encoding of data was hard enough. Also, my work on this first program was constrained by time as well as space; I needed to finish some version by January 1.

As I worked further, I began looking more deeply into the accomplishments of Perl golfers, who strive to write Perl programs that are as compact as possible [6]. They start with a completely specified task, which is not what I was doing, but in trying to compress a reasonably complex program I was attempting something similar. I approached the problem more in the manner of a writer of obfuscated code [7], choosing something interesting to do in an unusual way. However, I was not trying to make my program intentionally difficult to understand, only to provide myself with a useful constraint on program size that would lead me to focus on important techniques. Realizing that 80 characters would probably be too few for this first effort, I settled on a limit for program size in bytes that was a fairly small power of two. Keeping the program to 256 characters meant that it would be small enough to be copied and pasted easily by others; it was also a small enough target that I was able to do much of my work on the command line itself, without recourse to a text editor.

Finally, I discovered a word generation method that was compact but which relied on the structure and position of bigrams (pairs of letters) within words. I decided to generate only four-letter words, and to see how well the initial and final bigrams (the only parts of these words) would match up if the most frequent ones were joined at random. My work with non-conditional unigram generation, and some other not very effective attempts, hadn’t managed to even reach 10% in terms of generating “real” (dictionary) words. My first, rough attempt to join pairs of bigrams, on the other hand, produced dictionary words 40–50% of the time. Of course, getting a high accuracy with word generation, by itself, isn’t a challenge. A program that prints “Hi” forever produces English words 100% of the time. A suitable generator of this sort needs a balance between the high quality of English-like output (many words being recognized as English or appearing in a dictionary) and diversity of words it can produce. The four-letter word generation technique, although it could only produce four-letter words and only a subset of them were in a large English lexicon, was remarkably diverse in its output.

By this point, observing screenfuls of vaguely English-like words had brought to my attention that a stream of words is not easily recognized as a poem. This can be addressed in framing the program itself — text linking to the program or a placard on an exhibit can assert that the program is a poetry generator — but I hoped that the program itself could output text that would be seen as poem-like, even without such cues. I began working to have the generator create lines. As I did this, I developed a clever generator of short words that used the string “atonof” and compacted five two-letter words into a six-letter string. As it happened, the less clever generator using the string “attoonnoof” occupied fewer characters overall. It is the one used in the version shown here.

Even with lineation, the system didn’t seem done. Printing an endless stream of lines also didn’t seem to be proper poetry generation, since this output isn’t easily recognized as a poem. So, compacting what I had done even further, I added the highest-level, outer loop to title the poems and determine a number of lines for each. The addition of titles and an overall stanza/strophe shape to the poems was, I believe, a very important step. The title provided something for the poem to be read against, opening the

lines to meaning. I have heard poets claim that titles have the opposite effect, which they may, in particular cases. This experience with adding titles to a poem showed me that titles are not always limiting and can invite deeper reading and more engagement and interest by providing an additional, sometimes powerful, juxtaposition.

There were other ways of potentially augmenting ppg256-1, some of which I might have been able to fit into even my first version of the program. A sort of schematic rhyme, for instance, can be accomplished fairly easily by just holding the last bigram in memory and re-using it. The results, however, read like doggerel. A program that does this seems to be "cheating" by making up words to rhyme with earlier ones, making the effect of the invented words plodding, even though these same words could be appreciated as interesting in the version without schematic rhyme. I also looked into varying the length of words so that every line did not have four letters, two letters, and four letters. In the first version of the program, I rejected this because the fixed pattern 4-2-4 pleased me and I saw no easy way to pleasingly vary the length of the longer words. When I was later able to change certain four-letter strings into three-letter words and improve the system's diction, I included this capability, introducing a slight variation into the lines.

2.4 Programming and the Use of Perl

Given the compelling presentation and ongoing discussion of concepts such as expressive AI [8] and expressive processing [9], it seems that it should not be necessary to justify the writing of a computer program as part of a literary or artistic practice. Regardless: I have been developing text-based programs as part of my writing practice because my interests are in exploring computation, language, and their relationship. I am not investigating database or hypertext structures; rather, I am considering what computation can do to produce language. A poetry generating program seems appropriate for this investigation. Among other things, as the story of these first programs shows, this form compels me to trade off data for code (or vice versa) and to make each of these suited to the other.

Perl, the Practical Extraction and Report Language (also sometimes called the Pathologically Eclectic Rubbish Lister), was created for text processing and is amenable to being used for offhand tasks. It is possible to write "one liners" in Perl, at the shell prompt, and to use these to solve many text-processing problems, avoiding the need to even save and separately run a program. There are also existing traditions of creative and compressed programming in Perl, namely, the obfuscated programming tradition of creating "JAPHs" [7] and the competitive program compression of Perl golf [6]. These rich practices (although they are young, compared to the traditions of poetry) and my study of particular short Perl programs have helped me to think in new ways about this project, just as poetics conintues to contribute to my design goals and directions and just as I have considered particular techniques and poems in iterating each program to produce new sorts of output.

3. ppg256-2

```
perl -le 'sub p{split/,/,pop;$_[rand@_]}{$_=
p("sw,-aw,&w,saw".", "x$l);s//p("aw,w")/e;s/
/ /g;$_="\n\nthe s\n"if!$l;s/s/ws/;s/a/p("a,
the,to,of")/e;s/w/p("b,ch,f,gr,k,p,sh,s,sk,s
p,tw")."i".p("ll,n,t")/eg;s/(b|p|f)i/$1.p("a
,i")/e;print;$l=0if$l++>6+rand 9;sleep 1;red
o}'
```

The same code, with added whitespace and comments:

```
sub p{ # Pick from a comma-delimited string
split/,/,pop;
$_[rand@_]}

{ # Main loop
$_ = p("sw,-aw,&w,saw".", "x$l);
s//p("aw,w")/e;
s// /g;
$_ = "\n\nthe s\n"if!$l;s/s/ws/;
s/a/p("a,the,to,of")/e;
s/w/p("b,ch,f,gr,k,p,sh,s,sk,sp,tw")."i".p("
ll,n,t")/eg;
s/(b|p|f)i/$1.p("a,i")/e;
print;
$l=0 if $l++ > 6+rand 9;
sleep 1;
redo}
```

3.1 Goals

Having found a way to generate a large number of different words, many of them English and almost all English-like, I became interested in generating poems that were less regular in several ways: in the shapes of their strophes, in the syntax of their lines, and in the length of words that they generated. I also wanted to bring in additional connections between the sounds of words, schematically or otherwise. I wondered if an interesting program could be developed that would only output dictionary words rather than "making up" words. If I could develop a program that did some of these things, I was willing to end up with one that had a much smaller vocabulary than did ppg256-1, as long as variation of other sorts made the system interesting enough.

3.2 Explication of Code

ppg256-2 uses a different technique for splitting apart strings and choosing one section of them. Because variation in word length was important in this program, it was not appropriate to build words out of two two-letter components. Instead, parts of words are selected from a comma-delimited list, represented as a string, such as "b,ch,f,gr,k,p,sh,s,sk,sp,tw". The short words are selected from "a,the,to,of". Each long word is made from a one- or two-letter beginning, the vowel "i" (which is then replaced by "a" once in a while), and a one-or-two letter ending. Finally, a long word may have an "s" at the end, which has the effect of pluralizing the word (if it is read as a noun) or conjugating it (if it is read as a verb). Thus, long words can be three, four, five, or six letters in length.

The code first defines the subroutine p(), which is used to pick a section of a comma-delimited string. The rest of the code is the main loop, which ends in "redo", as the main loop of ppg256-1 also does. The first statement assigns to the default string ($_) either "sw", "-aw", "&w", "saw", or " " (a space). As lines are generated and the value of $l increases, more and more spaces are added to the distribution, and it becomes more and more likely that a space will be chosen instead of one of the first four options. Next, a substitution on the empty string adds either "aw" or "w" to the beginning of the default string. If the default string

contained “sw” before, it holds either “awsw” or “wsw” after this substitution. If it held a space beforehand, it holds either “aw ” or “w ” afterwards.

At this point the syntax of the line has been determined; the short string will be expanded and words will be put in place. The next statement adds a space between each character in the default string, making, for instance, “awsw” into “a w s w”. The next statement obliterates all of this work, replacing the default string with the syntax used for a title, if $l is 0 and the process is therefore on line zero. Whether that happened or not, the expansion of the syntax continues in the next statement, which replaces “s” with “ws”, so that “a w s w” becomes “a w ws w”. Then, “a” is replaced by one of four short words. If the one chosen is “of”, the new default string would be “of w ws w”. Next, a substitution statement replaces each “w” in the string with a long word generator (pick a prefix, add “i”, pick a suffix) and runs the generator, so that each “w” ends up replaced by a long word. The result might transform “of w ws w” into “of grin pits chip”. The last statement before the print statement sometimes (with 50% probability) changes the vowel “i” to “a”, only if it occurs after a p, b, or f. The substitution would happen to the word “pits” half the time; the “i” would remain, otherwise. If the substitution does happen, the line becomes “of grin pats chip”.

After these manipulations, the default string (containing either the line or the title) is printed, $l is reset to zero if it exceeds a random number that is between six and fifteen, the program pauses for a second, and the loop is repeated.

3.3 Development Process

In working on this second program, I developed a word generator that used a small set of consonant prefixes and suffixes and, initially, just the vowel “i”. The productivity of this generator seemed sure to disappoint. Without the “a” rewrite, it could generate only 12×3 = 36 words, or 72 words if both the base forms and the forms ending in “s” are counted. But even a set of 36 base forms proved remarkable in certain ways. All sorts of alliteration and rhyme arose naturally when words were drawn from this distribution. When drawing from just the base forms, one rhyme is guaranteed in every set of four words, since there are only three possible word endings. The lack of regularity and the presence of more than a handful of words meant that the result was lively in some ways. With the “a” rewrite added, the words were no longer monotonously univocalic, and it was possible to hear the vowels as more interestingly assonant. New half-rhymes appeared, improving the texture of sound.

Two approaches to providing “global” features in poetry generation — coherence, adherence to a theme, the return to an earlier statement in closing — are exemplified by Jim Carpenter’s ETC [10] and Eric Elshtain and Jon Trowbridge’s Gnoetry [11]. ETC’s architecture influenced ppg256-1 and is to some extent reproduced in miniature in these programs [12]. But the ppg256 programs do not follow ETC in providing high-level, global rules for composition that can direct the program to “wrap up” a poem with reference to what has been written earlier. The global features of ppg256-2 poems come together more in the way that Gnoetry, using statistical methods, accomplishes a consistent texture in poems. That system, trained on a corpus of writing (usually from a well-known fiction writer), produces language that recalls earlier writings and that coheres because traces of earlier topics, themes, and styles persist though the process of computer composition using those texts.

In the ppg256 series, my engagement is not, for instance, with Joseph Conrad via his statistically modeled *Heart of Darkness*, but with the properties of the ordinary English lexicon: the most common word beginnings and endings for words of a certain type, for instance. Although I did use a particular text file with inflected English words, my questions and techniques are directed more at English than at any one dictionary or lexicon. The very common word beginnings and endings that I selected would be very similar, if not identical, if I had used a different file with a slightly different set of English words.

In adding the occasional “a” rewrite, which happens only half the time and only when a word contains “bi”, “pi”, or “fi,” I made the distribution of possible long words non-uniform. That is, some long words are more likely than others. This is the case for a different reason in ppg256-1, where three bigrams are repeated. While having a non-uniform distribution is not necessary for an aesthetic result, and is not required by any poetics of probability, it seems to reflect certain things about our experience. It particularly seems interesting, and in some ways naturalistic, to have a program select from some equiprobable choices and some that are more rare, as ppg256-2 does.

The length of lines in a poem is determined in ppg256-1 by a conditional probability distribution, the sort that is useful for modeling language but difficult to encode in a short space. As noted in the explication, each possible syntax is not selected with the same probability; as the poems grows, short lines become more likely. This very simple conditional probability implementation gives some not entirely regular shape to each poem’s strophe, providing each with a tendency to taper off.

The distinctions between the output of ppg256-1 and that of ppg256-2 are certainly due to differences in the data they work upon (stored as strings), and do not result from computational differences alone. To emphasize process intensity and to investigate the importance of computation is not to dismiss the need for the selection of good source text for a program to work upon. One of my insights into my own practice that I have developed further in the ppg256 series is the importance of jointly developing data and code, or, at least, of defining the data with particular sensitivity for the ways in which the code works. The sounds of the limited set of words produced by ppg256-2 work well in the context of a program that generates a variety of lines and strophe shapes. The lexical variety of ppg256-1 can be effectively framed by more regular lines.

3.4 Short Words and Long Words

The ppg256 programs are written from a perspective that is poetic, and that comes from programming and writing practice, rather than being linguistic in any ordinary, scientific sense. They encode this perspective in the categories of words that they define and the range of words they generate.

Educated speakers of English are well aware of the categories adjective, noun, verb, preposition, and article, which are almost always employed in computational linguistics systems. Distinctions between these are certainly necessary for determining the grammatical structure of sentences, but systems which do not do full parsing often still tag words with their parts of speech, a process which can be done very accurately in most cases. An

industrial-strength language generator will also need to make these part-of-speech distinctions. Such distinctions are usually not made in 256-character programs, however.

Because the chosen length constraint makes it impractical to have separate generators for each part of speech, my programs use more offhand (but still linguistically relevant) categories. The "short words" that they generate are not restricted to articles or prepositions — pronouns are included in the "short words" of ppg256-3 — but they are all what are known as closed-class words. In a closed-class category of words, new words cannot easily be coined by speakers. Nouns and verbs are open-class words, allowing "blog," "staycation," and other terms to be invented as the need arises. A new preposition or article cannot be added as easily, though.

ppg256-2 even more clearly takes advantage of the capability of English nouns to almost always be used as verbs. This program, like ppg256-1, collapses nouns and verbs into "long words," but adds syntax and inflection that works well in this new category. Long words generated by this program can be read as nouns or verbs depending upon their situation in a line and whether they are inflected with "s".

To keep the short word generation code compact, ppg256-2 generates only "a", never "an". (A good bit of additional code would be required to determine if the following long word began with a vowel and to make the adjustment.) Instead of doing this more elaborate form of short word generation, ppg256-2 simply generates only long words that begin with a consonant so that "an" is never needed.

3.5 The 256-Character Limit

Italo Calvino said "an Oulipian writer ... runs faster when there are hurdles on the track." There is a strong current of constrained writing practice represented most prominently by those in the Oulipo, including Calvino, Harry Mathews, George Perec, and Jacques Roubaud, but also developed in the work of Walter Abish, Christian Bök, William Gillespie, Mary Godolphin, Doug Nufer, Jackson Mac Low, George Starbuck, and Michel Thaler. And, of course, there are more traditional limits placed by particular poetic forms (such as the sonnet) and by poetic meters. Creating a poem of a certain length (and width) is hardly unusual. Enforcing a compositional constraint at the level of character or letter is, if somewhat less traditional, not a novelty from the standpoint of writing practice.

There are also many precedents for this sort of constraint in digital media practice, however. Formats and protocols impose their own sometimes austere limits. The 140-character limit of Twitter is even more restrictive than the 160-character maximum of the SMS message. Even when the technology does not demand it, digital media contests of all sort limit the size of entries to provide challenge and to focus contestants on the task at hand. There are size limits on one-line and longer programs in the International Obfuscated C Code Contest [7], for instance. This sort of limitation is particularly in play in the demoscene, where coders work to develop process-intensive audiovisual programs that will dazzle viewers and amaze fellow programmers. On demoscene site pouet.net, numerous length-constrained demos are available, in 32b (bytes), 64b, 128b, 256b, 512b, 1k, 4k, 8k, 16k, 32k, 40k, 64k, 80k, 96k, 100k, 128k, and 256k sizes.

After seeing what could be done with shorter programs, and aware of the advances that larger-scale systems have made, I determined that in this series, programs would be 256b (256 characters) in length. This limit is a power of two, as "natural" for the computer as is a multiple of ten for the digital human. This size allows for variety in syntax, vocabulary, and strophe shape, but compels me to determine where computation will be spent and requires that effort be expended for a variety of words to be produced. As ppg256-2 showed, 256 characters is enough room for different sorts of generators to be fashioned. Programs of this size, however, are short enough to type in if one starts from a printout or wants to get a program running on a non-networked computer without removable media. And, they are short enough to seem comprehensibile, based on their length. Whether or not a reader knows Perl and wants to trace through the code, he or she can at least immediately believe that the code *can* be understood instead of imagining the generator as an imposing black box full of tremendous, complicated machinery.

4. ppg256-3

```
perl -le 'sub p{(unpack"(A3)*",pop)[rand 18]
}sub w{p("apebotboyelfgodmannunorcgunhateel"
x2)}sub n{p("theone"x8)._.p("bigdimdunfathip
litredwanwax")._.w.w."\n"}{print"\n".n."and\
n".n.p("cutgothitjammetputransettop"x2)._.p(
"herhimin it offon outup us "x2);sleep 4;red
o}'
```

The same code, with added whitespace and comments:

```
sub p{ # Pick a 3-letter substring from a
       # string 54 (18*3) characters long
(unpack"(A3)*",pop)[rand 18]}

sub w{ # Return a word: 3-letter + 3-letter
p("apebotboyelfgodmannunorcgunhateel"x2)}

sub n{ # Return a name: article (optional) +
       # adjective (optional) + word
p("theone"x8)._.p("bigdimdunfathiplitredwanw
ax")._.w.w."\n"}

{ # Main loop
print"\n".n."and\n".n.p("cutgothitjammetputr
ansettop"x2)._.p("herhimin it offon outup us
 "x2);
sleep 4;
redo}
```

4.1 Goals

In writing this program, I hoped to at least strongly suggest a narrative, if not directly tell one, by portraying an action involving characters. My interest was still in poetry generation rather than story generation, and in the sounds of the language, and how memorable that language is, rather than in creating a full fictional world with psychologically authentic characters. I was also particularly interested in exploring the use and generation of conjunction, juxtaposition, compound words, and kennings.

4.2 Explication of Code

The p() subroutine that begins the program does the sort of picking that b() in ppg256-1 does: It unpacks a string into an array of three-character elements, then selects one of eighteen elements

at random. The subroutine after this, w(), returns a three-letter word by picking from the string

"apebotboyelfgodmannunorcgunhateel"x2

That is, two copies of that string connected one after the other. Then, the subroutine n() produces one type of line. It begins with either the word "the", the word "one", or nothing; an underscore ("_") follows; a three-letter adjective from the string "bigdimdunfathiplitredwanwax" or nothing is after that; and then an underscore, two three-letter words (without a space between them), and a newline are added. For instance, "the_fat_boyman" and "__elfgod" can both be generated by this subroutine.

The main loop prints one of the n() lines, then a line with just the word "and", and then another n() line. This is followed by a line with a verb phrase such as "cut_it " or "jam_out", made by selecting a three-letter section from each of two strings. The program then pauses for four seconds and repeats the main loop.

The use of underscores instead of spaces provides some typographical variety and might signal to some that this work is meant to be poetry rather than fiction. It leaves the text legible, however, and saves a few characters of code. Each time the underscore appears in the code, the replacement of that character with a space would require a double quote on each side and would add two characters to the program.

Because the p() subroutine always picks one element numbered 0 through 17, it can be used to always pick one of nine elements (by putting a "x2" at the end of a string with nine choices in it) or it can be used to return a blank half the time and to pick one of nine elements the other half of the time (just pass in the string with nine choices *without* "x2" at the end). It can also be used to print "the" or "one" almost all of time but to print nothing occasionally, as when "theone"x8 is passed in.

4.3 Development Process

I had been doing some non-digital writing with three-letter words and was intrigued by the possibilities of a three-letter lexicon. At the same time, I was interested in juxtaposing whole words (not bigrams or consonants) to create new ones. I was thinking of the venerable English poetic element called the kenning, a condensed metaphor such as *guthwine* (warfriend), which indicates a sword. In contemporary writing, striking compound words have been used by Cormac McCarthy, particularly in his *Blood Meridian*.

ppg256-1 assembles its long words out of bigrams whose semantics are not evident. ppg256-2 uses consonants which are also not usually thought of as meaningful in and of themselves. Both character names and verb phrases are assembled in ppg256-3 differently, using short words. I found to my surprise that a very limited set of three-letter words (nine words that could be anthropomorphic, including "eel", "man", "nun", and "ape") seemed much less repetitive when one selection from the list was concatenated with another (to make, for instance, "eelman", "manape", "nunman", or "eelape"). Of course, there are 81 possibilities for such words rather than nine, but these possibilities come about by picking twice from the same small set of options. Despite the lexical limitation, the meaningfulness of the three-letter words let them combine into unusual and provocative longer words. A "manape" is not an "apeman" and even characters such as a "manman" and an "apeape" invite further thought, perhaps because of their odd insistence. There are also what seemed at first to be contradictions in coinages such as "nunman". These can be resolved a few different ways by a reader.

Similarly, different closed-class words placed after three-letter verbs form very different verb phrases, creating much more variety than one might expect from a list of nine verbs. To put up is not the same as to put in, or to put her or him or it, or to put off, or, of course, to put out. The poem sometimes lacks a direct object when it seems that it should have one, but if the reader's mind is active, imagining who the characters could be, why they have come together, and what they are starting to do, this is hardly a problem.

4.4 Randomness

Few in digital media have had much to say about randomness; authors, artists, and critics alike seem to find its occurrence in work, and certainly the term itself, distasteful. Scott Rettberg, however, has described how the use of randomness relates to Dada techniques and can be explored and discussed rather than avoided [13]. Essentially, a program that does something at random — or, more correctly, that approximates this using a pseudorandom process — chooses an element from a distribution. If it picks one element out of a set such that every element is equally likely, it is choosing uniformly at random. If some elements are more likely that others, the probabilities for each choice are non-uniform.

As an alternative to choosing an element at random, a program can output every combination of elements one after the other. This is what Brion Gysin and Ian Sommerville did in their permutation poems, one of which included every permutation of I AM THAT I AM [14]. Another exhaustive program of this sort is John F. Simon, Jr.'s *Every Icon*, which will, if it continues running, eventually display every 32×32 black-and-white icon [15]. Clearly the exhaustive approach has its particular rhetoric, but it is hard to see how programs like these should, in every case, be privileged over ones that sample repeatedly from a distribution and offer something different to the reader or viewer. The exhaustive program shows that every alternative either has been or potentially can be computed. The random program gives a different, more individuated sense of what a distribution is like.

Nanette Wylde's *Storyland* is a simple and amusing program to randomly generate very short stories [16]. Talan Memmott's *Self Portrait(s) [as Other(s)]* assembles images and somewhat authoritative-sounding curatorial texts from fragments, also at random [17]. The effect of these two pieces would be entirely different and significantly reduced if they were converted into exhaustive programs that generated every possible combination one after the other, making a slight change each time. The effect of a random program can be like overhearing bits of a conversation, perhaps a conversation that is most interesting when only partially overheard. It can be more along the lines of meeting a few people from a particular country and less like having everyone from that country arrayed in an enormous gymnasium.

In terms of their poetics, random programs demand that an author define interesting distributions over texts rather than simply writing a single text that is appropriate.

5. ppg256-4

```
perl -e 'sub c{$_=pop;$_[rand split]}sub w{c
("b br d f fl l m p s tr w").c"ad ag ap at a
```

```
y ip on ot ow"}{$|=print"\0\0\0\0\0\1Z00\2AA
\33 b".c("be de mis re pre ").w." ".c("a on
the that")." ".w.w.", ".c("boss bro buddy do
gg dude guy man pal vato")."\4";sleep 4;redo
}' > /dev/alpha

perl -le 'sub c{$_=pop;$_[rand split]}sub w{
c("b br d f fl l m p s tr w").c"ad ag ap at
ay ip on ot ow"}{$|=print "\n".c("be de mis
re pre ").w." ".c("a on the that")." ".w.w."
, ".c("boss bro buddy dogg dude guy man pal
vato")."\4";sleep 4;redo} #No LED sign versi
on'
```

The same code, with added whitespace and comments:

```
sub c{ # Choose from space-delimited string
$_=pop;$_[rand split]}

sub w{ # Return a word: beginning + ending
c("b br d f fl l m p s tr w").c"ad ag ap at
ay ip on ot ow"}

{ # Main loop
$|=print "\n".c("be de mis re pre ").w."
".c("a on the that")." ".w.w.", ".c("boss
bro buddy dogg dude guy man pal vato")."\4";
sleep 4;
redo} #No LED sign version
```

5.1 Goals

I hoped that it would be possible to generate interesting text given the further constraint of a very small display and the need to write special code to drive this display. The use of such a display would make this program particularly amenable to gallery presentation, allowing it to reach a different group of viewers. I also wanted to see if I could develop a voice that was gendered and addressing someone of a particular gender. Finally, I wanted to continue to explore how the combination of different syllables into words could work to create a variety of English-like sounds.

5.2 Explication of Code

The main unusual feature of this program should be the series of control characters that, along with the redirection of the output to a special device, are needed to drive the LED display. (These begin with a series of five null characters, each indicated by "\0".) In the main loop, the return value of the print statement is assigned to $|, forcing the output buffer to be flushed each time. This does not need to be done in every loop, but it does not hurt anything to assign to $| repeatedly and it saves space to do so. To save a few characters, split is used without any arguments here and the strings that it splits are delimited by spaces. Otherwise, this is a program like the others that has two subroutines and a main loop.

The c() subroutine picks an element at random from a space-delimited list. The w() subroutine uses c() to build a three or four letter "word," which is perhaps better called a syllable, since it is actually used as a component of a word. The main loop then emits the necessary control characters and prints the following: an etymological prefix ("be", "de", "mis", "re", or "pe") most of the time (the space at the end means that the prefix will sometimes be omitted); a syllable; a space; either "a", "on", "the", or "that"; a space; two syllables; a comma and a space; and a term from this list, stored as a space-delimited string:

"boss bro buddy dogg dude guy man pal vato"

The whole poem is output at once, as with ppg256-3, and there is a four-second pause before control returns to the beginning of the main loop.

5.3 Development Process

As noted earlier, this is the first ppg256 to draw a poem uniformly at random from a set of poems. That is, of the 174,653,820 possible poems, each one is equally likely to appear. The need to include LED sign control codes, which made the available space for poetry generation even tighter, discouraged me from creating a more varied distribution. Also, I noted that only one poem would be seen at once on the LED display, and that a poem would be gone forever after four seconds, so a gallery visitor would probably not even have time to share one with a friend before the next poem superseded it. A rare treat of some sort would present itself in temporal sequence, but not as part of a spatial display.

For a text generator or other aesthetic program to draw from a uniform distribution is not itself a flaw, just as a non-uniform distribution does not by itself make for a wonderfully aesthetic program. The question is whether the distribution is appropriate to the goals of the project. In this case, each text is some sort of command or request that is nonsensical but interesting-sounding, and definitely English, and that closes with a familiar term of address. The texts combine etymological prefixes with less sensible syllables, leaving the reader to imagine what is being requested by this strange speaker and why. The lack of some occasional rare surprise in terms of form does not seem to me to be a failing.

The set of terms used to conclude the utterance were chosen to be terms of address that are actually used by people, but also to suggest a racial and gendered identity. Racially marked terms such as "dogg" and "vato" can be used by people of any race in addressing people of any race, but however they are used, they cannot help but remind a reader or listener of race. All of the terms of address are gendered male; even "buddy," which seems to have come from "brother," "pal," which etymologically can be traced to the Sanskrit "bhrātṛ" (brother), and "boss," from the Dutch "baas" (manager, foreman), the diminutive of which is used to address very young boys. They are "real words" in the sense that they appear in English dictionaries, as will seldom be the case with the two other long words in the poem. But they are also real in that they more or less unambiguously situate the addressee as male, strongly suggest that the speaker is male, and remind us that words have other important contexts and have their heritage within particular social and cultural communities. In all of these ways, they contrast with the more playful and exploratory constructions that occur earlier in the poem.

I was originally quite interested in having one of the terms of address — perhaps one that occurs only very rarely — be "motherfucker". This ending to the short poem would have been both provocative and, I thought, appropriate to the type of voice that I was trying to shape. But the 256-character constraint and the size of the display made it implausible for me to include a very long term along with code to have it appear infrequently. The term "mofo" sounded comedic in comparison and didn't seem to fit. However, I am pleased that the voice of this generator is at least

slightly homophobic and can generate the syllable "fag," exactly the sort of syllable a brusque, masculine voice, issuing a command and speaking to another male, might say. This unpleasant component of utterance, occurring almost as a side effect of a generation process that produces a variety of English syllables, will perhaps invite similar reflections as would an occasional "motherfucker," without seeming to be a surface gimmick appended as an afterthought.

5.4 LED Sign Display

Whether the area of practice is called new media, digital media, computational art, digital writing, electronic literature, or something similar, it is not essentially about the screen. Early games and literary projects on the computer were apprehended and interacted with via Teletypes and other print terminals. Sound works and text-based works accessed through text readers show that screens are not required for digital art overall or for interactive electronic writing specifically. While an LED sign is a matrix of points, each of which can be illuminated, the difference between this type of display and the usual high-resolution projected image or backlit screen helps to de-emphasize the visual display as a component and to suggest that the focus of the project is elsewhere, on the computation. Having the system print its output would do something of the same thing, but at the cost of suggesting that ink and paper are the privileged channel for the transmission of poems and that the system was put together to use this venerable means of publication.

Computational art of all sorts cannot help but repurpose instruments and systems, since computers are manufactured to turn the gears of commerce, industry, and the military. There is special pleasure, however, in obtaining a used LED sign that has a lottery advertisement still in flash ROM and transforming it into a window into language, poetry, and computation.

6. ASSESSING MINIMALITY

The programs in the ppg256 series are not minimal in the mathematical sense: it is possible to write poetry generators in even fewer characters. But they are very concise, and the size limit adhered to in creating them has helped me to avoid certain pitfalls. If these poetry generators produce aesthetically pleasing and interesting output, it is not because they have a large store of data and are simply shoveling lines onto the screen from that store. If they build a discernible voice, it is not from the statistical properties of existing texts or from the accumulation of dozens and dozens of different rules. Any success these programs have must come from a few instructions which can be read in a moment and can be completely discussed in the space of a page.

There are, certainly, some questions that minimal generators are better at answering than others. They show how a sort of "naïve linguistics" can be developed, a useful and telling simplification from a poetic perspective. But just as they deal with and help to reveal some aspects of English (such as *nouns can almost always be used as verbs*), they are probably not well-suited to working deeply with other features of the language (such as *English has a large number of loan words from a variety of other languages*). They seem to be poorly suited to subtly deploying allusions and references or to certain types of etymological play. And they are not very good at unrolling either a powerful progression of symbols or a complex narrative.

Minimal poetry generators, however, constitute important probes and round out our tray of poetic instruments. While larger systems have their place, programmer/poets, particularly those working on larger-scale poetry generators, can clearly benefit from examining language by using tiny, complete generators.

7. REFERENCES

[1] Montfort, N. 2008–2009. ppg256 (Perl Poetry Generator in 256 characters). On nickm.com. http://nickm.com/poems/ppg256.html

[2] Montfort, N. 2009. Curveship: An Interactive Fiction System for Interactive Narrating. In Proceedings of the NAACL HLT Workshop on Computational Approaches to Linguistic Creativity (Boulder, Colorado, June 4 2009) CALC-09, 55–62. http://www.aclweb.org/anthology/W/W09/W09-2008.pdf

[3] Montfort, N. 2008. ppg256-1 Discussion. On nickm.com. http://nickm.com/poems/ppg256-1_writeup.html

[4] Kenner, H. and O'Rourke, J. 1984. A Travesty Generator for Micros. In BYTE 9 (12): 129–131, 449–469.

[5] Hartman, C. O. 1996. Virtual Muse: Experiments in Computer Poetry. Wesleyan Univ. Press., Middletown, CT.

[6] splinky. 2000. Announcing the First Annual Perl Golf Apocalypse. On PerlMonks. http://www.perlmonks.org/index.pl?node_id=21442

[7] Montfort, N. 2008. Obfuscated Code. In Software Studies: A Lexicon, ed. M. Fuller. MIT Press, Cambridge, MA, 193–199.

[8] Mateas, M. 2001. Expressive AI: A Hybrid Art and Science Practice. In Leonardo 34 (2): 147–153.

[9] Wardrip-Fruin, N. 2009. Expressive Processing: Digital Fictions, Computer Games, and Software Studies. MIT Press, Cambridge, MA.

[10] Carpenter, J. 2007. Erica T Carter 3 Beta. http://etc.wharton.upenn.edu:8080/Etc3beta/ Offline as of 2008.

[11] Elshtain, E. and J. Trowbridge. 2007–2009. Markovian Parallax Generate. http://mchainpoetics.wordpress.com/

[12] Carpenter, J. 2008. Does size really matter? On The Prosthetic Imagination. http://theprostheticimagination.blogspot.com/2008/01/does-size-really-matter.html

[13] Rettberg, S. 2008. Dada Redux: Elements of Dadaist Practice in Contemporary Electronic Literature. In Fibreculture Journal 11. http://journal.fibreculture.org/issue11/issue11_rettberg.html

[14] Gysin, B., ed. J. Weiss. 2001. Back in No Time: The Brion Gyson Reader. Wesleyan Univ. Press., Middletown, CT.

[15] Simon, J. F. Jr. 1997. Every Icon. http://www.numeral.com/eicon.html

[16] Wylde, N. 2006. Storyland. In The Electronic Literature Collection, Vol. 1. http://collection.eliterature.org/1/works/wylde__storyland.html

[17] Memmott, T. 2006. Self Portrait(s) [as Other(s)]. In The Electronic Literature Collection, Vol. 1. http://collection.eliterature.org/1/works/memmott__self_portraits_as_others.html