CAMELS AND NEEDLES: COMPUTER POETRY MEETS THE PERL PROGRAMMING LANGUAGE

Sharon Hopkins, Telos Corporation

ABSTRACT

Although various forms of literature have been created with the assistance of a computer, and even been generated by computer programs, it is only very recently that literary works have actually been written in a computer language. A computer-language poem need not necessarily produce any output: it may succeed merely by fooling the parser into thinking it is an ordinary program. The Perl programming language has proved well-suited to the creation of computer-language poetry.

INTRODUCTION

Over the past few decades, considerable work has been done in the area of computer-generated writing. Projects have ranged from programs designed to find simple anagrams, to dedicated pangram [1] generators, to entire poems or stories (usually nonsensical) produced by stringing together chains of words that occur next to each other in human-generated texts. One French group [2] went so far as to experiment with computer-generated writings that could be read in both French and English. A number of science fiction and fantasy stories have been written in the form of flow charts, or designed to appear as if they had been written in some programming language yet to be invented. But until very recently, little or no attempt has been made to develop human-readable creative writings *in an existing computer language*. The Perl programming language has proved to be well suited to the creation of poetry that not only has meaning in itself, but can also be successfully executed by a computer.

Why Computer Poetry?

The question naturally arises: Why write poetry in a computer language at all? Computer-language poetry is essentially no different from any other formal poetry, except that the rules of the computer language dictate the form the poem must take, and provide a mechanism for measuring success. If the poem executes without error, it has succeeded. The great advantage formal poetry has over free verse is the balance provided between familiarity and strangeness, stasis and innovation. When reading rhymed poetry, for example, the reader develops an expectation of how each line will end. The poet's task is to satisfy that expectation, by providing a rhyme, while surprising the reader with a word or meaning outside of his expectations. With computer-language poetry, the surprise is enhanced: the reader develops expectations as to what is going to come next based on familiarity with the standard uses of that language's vocabulary. In Perl, if the word read() appears, the person reading that Perl script will naturally expect the items in parentheses to be a filehandle, a scalar variable, a length value, and maybe an offset. The poet, finding read() in the Perl manual or reference guide, is more likely to write "read (books, poems, stories)". In the same way, sin() (of sine, cosine, and tangent) becomes sin, the downfall of man. Another advantage to writing computer-language poetry, in addition to the challenge provided by the constraints of the form, is that you can get some creative writing done and still look like you're working when the boss glares over your shoulder.

Why Perl?

The Perl programming language provides several features that make poetry writing easier than it might be in other languages. Cobol poetry, for example, would always have to start with IDENTIFICATION DIVISION. While that would make an excellent opening for one poem, it's hard to see what could be done for the next one. Besides, getting a volunteer to write poetry in Cobol is likely to be impossible. We do have one example of a poem in FORTRAN (Fig. 1), but FORTRAN, too, is tightly constrained in that most of the interesting things have to be said up front. Layout of FORTRAN poems is also tricky, given the limitations on what can safely be put in the first few columns. One of the reasons for Perl's popularity is that you don't have to say what you are going to say before you say it. In addition, Perl has in its vocabulary something on the order of 250 words, many of which don't complain when abused. Shell poetry is an attractive possibility, with plenty of good words available. Unfortunately, most of the words available in a shell script are likely to return non-poetic messages at artistically inappropriate moments. In Perl, there is a decent supply of usable words, white space is rarely critical, and there is almost always "More Than One Way To Do It". And since Perl is interpreted rather than compiled, you never have to keep ugly binaries around pretending to be your poem; in addition, poem fragments can be tested quickly by running perl interactively, or by feeding lines to perl on the command line via the perl -e switch.

A Little History

Randal Schwartz probably deserves credit for inspiring the first-ever Perl poems. Randal's "Just another Perl hacker," (JAPH) signature programs prompted me to suggest to Larry Wall that he write a JAPH that would also be a haiku. By lunchtime that day (mid-March of 1990), Larry had produced the world's first Perl poem (Fig. 2):

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

In this poem, the q operator causes the next character (in this case a newline) to be taken as a single quote, with the next occurrence of that delimiter taken as the closing quote. Thus, the single-quoted line 'Just another Perl hacker,' is printed to STDOUT. In Perl, the "unless $spring" line is mostly filler, since $spring is undefined. In poetical terms, however, "$spring" is very important: haiku poetry is supposed to specify (directly or indirectly) the season of the year. As for the q operator, that reads in English as the word "queue", which makes perfect sense in the context of the poem.

A FEW WAYS TO DO IT

The above poem demonstrates a particular difficulty with Perl poetry. Since Perl variables begin with $, @ , or % , the aspiring Perl poet must decide whether or not these type markers should be pronounced. In the "Just another Perl hacker," poem, pronouncing $ as "dollar" is necessary, as haiku by definition consist of one 5 syllable line, one 7 syllable line, and a final 5 syllable line. Notice that STDOUT must also be pronounced "Standard Out" for the poem to work. Most Perl poetry that has been written so far has tried to work Perl's special characters into the poems, reading $ as "dollar", and @ as "at". Similarly, punctuation marks (semicolons at ends of lines, parentheses, etc.) are most often worked into the text of a Perl poem. An alternative approach to Perl poetry punctuation has been to line all extraneous markings (quotes, semicolons, etc.) along the right hand margin, and to ignore dollar signs and the like when they occur within the body of the poem. This can help make Perl poetry look more like English, but it tends to be messy, with random bits of unmatched punctuation remaining scattered through the poem.

Working Perl Poems

There are basically two types of Perl poetry: those poems that produce output, and those that do not. The former, not surprisingly, are much harder to write. After the first Perl haiku, the only examples of "useful" Perl poetry are Craig Counterman's "Ode to My Thesis" (Fig. 3) and "time to party" (Fig. 4). "Ode to My Thesis" is readable in English, parses in Perl, and produces text output summarizing and concluding the theme developed within the poem. In order to hide items like the >&2 construct, and other unpronounceables, most of the punctuation has been relegated to the far right of the screen or page. Disposing of inelegant punctuation has the added advantage of obscuring the intent of the program, making it hard for the casual observer to guess what output the poem will produce. The theme of "Ode" interacts nicely with the Perl vocabulary, as words like "write", "study", and "sleep" seem natural in a poem dedicated to the joys of completing a dissertation.

In "time to party", Craig uses Perl's file-handling mechanism to make the poem produce output: in this case, creating an output file called 'a happy greeting' which contains the signature line for the poem. Other methods for producing output from a Perl poem might include system calls with useful return messages, or output from the die() and warn() commands. For input obfuscation, various tricks can be played on the Perl copyright notice, which is accessed through the $] special variable. This technique has been used in at least one JAPH script, and could be made to work in a Perl poem using Craig's format, with non-poetic constructs moved out of the line of vision.

"A poem should not mean but be." [3]

Non-useful Perl poetry, my own specialty, has two basic visual formats: the first type looks more like standard English free-verse poetry, and tends to be built on individual functions and operators that are selected to create a particular mood or image; poems in the second format tend to be more stanza-oriented, and to take a sort of word-salad approach, using any and all Perl reserved words that can be made to fit. Most poems of the first type, those with a more free-form linear organization, tend to be what I refer to as "keyword" poems. The poems are built around one or more Perl reserved words that also carry a weight of meaning in English. Examples of this type of poem are my "listen" and "reverse" poems (Figs. 5 & 6).

In "listen", a second-person poem speaking directly to the reader, most of the weight of the poem is carried by Perl reserved words: listen(), open(), join(), connect(), sort(), select(). Since Perl parses bare words as quoted strings, extra words can be included in each expression to provide added meaning in English without causing syntax errors. Thus, join() becomes "join (you, me)", and connect() becomes "connect (us, together)". Because Perl rarely worries about white space, the poem can be arranged on the page for a stronger visual impact. For example, the statement "do not die (like this) if sin abounds;" can be broken with a newline (making the phrase read as separate lines in English) without confusing Perl. In addition, the comma operator can be used to stretch out Perl statements, and relieve the monotony of a poem filled with semicolons.

The poem "reverse" makes use of Perl's conditional (if-then-else) operator, ?: , to provide variation in tone, so that the poem reads more as a conversation, not merely as a set of commands or instructions. In this poem, words like "alarm", "warn", "sin", "die", and "kill" are used to create tension, and to increase the poem's emotional impact. The fact that all of these are perfectly ordinary and mundane functions in Perl heightens their surprise value when used in a poem. At the same time, the oddity and unfamiliarity of Perl syntax, when read in English, gives the poem a sinister and mysterious flavor. At the end of the poem, the keyword "reverse" (used as "reverse after") is intended to give the poem an ironic twist, making it clear that nothing previous in the poem can be trusted.

The critical process for the keyword type of perl poem is weeding out words and phrases that do not enhance the meaning of that particular poem. I have one Perl poem, called "limits", that is composed almost entirely of lines that occurred to me while I was writing "listen", but which did not seem to fit the "listen" poem thematically. It can be difficult, however, to strike a balance between meaning and mystery; it's hard to determine when you have said enough to make the poem coherent as poetry without sacrificing the integrity of the program. Because it's easy to include quoted material in Perl programs, one is often tempted to simply wrap a pair of quotes around everything that won't pass the Perl parser. The typical end-result is a poem that is neither surprising enough to be interesting in English nor Perlish enough to be interesting as a Perl program. My poem "rush" (Fig. 7) unfortunately falls into this category.

The word-salad type Perl poetry is often more interesting than the keyword-based Perl poetry, both from a programming and from a literary standpoint. Two example poems of this type are Larry Wall's "Black Perl" poem and my "shopping" poem (Figs. 7 & 8). Unlike the more loosely organized Perl poems, these stanza-based poems are generally created using the "kitchen sink" approach. Working from a list of all of Perl's reserved words (and pieces of words), the goal is to fit as many as possible into the Perl poem while maintaining a consistent theme or image. This type of poem is likely to be more humorous than those with a more restricted vocabulary, and the end results are surprisingly readable. Producing a "kitchen sink" Perl poem can require considerable ingenuity on the part of the programmer-poet, since many Perl words do not fit easily into English-language poetry. For example, Perl's system() function becomes "system-atically" in the poem "shopping", and unlink() is used in "unlink arms" in "Black Perl". "Black Perl" is particularly remarkable since it was written before poetry optimizations (allowing bare words to be interpreted as quoted strings) had been added to Perl.

The visual impact of the "kitchen sink" type of Perl poem also differs from that of the keyword-based poems. In both "Black Perl" and "shopping", the action of the poem is divided into separate visual blocks, each with an identifying label. In Perl, statement labels can be any single word (preferably upper case), followed immediately by a colon. Statement labels provide an excellent method for inserting extra English words into a Perl poem while providing the poem with additional structure and cohesiveness. While this technique is also used in the keyword type poems, it is not as evident as in the more stanza-oriented Perl poetry.

"All poems are language problems." [4]

When writing poetry that is meant to run without producing any output, one useful trick is to make sure the program exits without executing all of its statements. In the "Black Perl" poem, the program actually exits about a third of the way through the first line. The rest of the poem will still have been checked for syntax errors, but the various remaining "die", "warn", "kill", and "exit" functions are never actually called. Likewise, in the "shopping" poem, the goto statement at the end of the first stanza causes Perl to skip past the second stanza, which would otherwise provide an ugly warning message. This poem also exits before its end, thereby avoiding an error message in the "later:" stanza, at "goto car" (since subroutine "car" does not exist). Judicious use of the . (concatenation) , (comma) and ?: (conditional) operators can help with semicolon avoidance. Similarly, playing with white space can help make a Perl poem more readable.

PROS AND CONS OF PERL POETRY

Writing poetry in Perl provides a number of literary "pluses". The nature of computer languages, where most of the available words are commands, forces the computer-language poet to write in the imperative voice, a technique that is otherwise usually avoided. But the use of short, imperative words in a poem can actually serve to heighten the drama, especially with meaning-laden words such as "listen", "open", "wait", or "kill". In Perl, all but the last statement in a program require a concluding semicolon; leaving the final statement of a Perl poem bare of punctuation can help the poem's meaning seep into the reader's subconscious (this technique is used in all of the non-working perl poems included with this paper).

I had hoped that learning to write poetry in Perl would be of assistance in learning more conventional Perl programming, but this has not proved true in practice. I have absorbed a fair amount of Perl syntax, and a little bit of Perl "style", but in general Perl programs that read as poetry tend not to be in idiomatic Perl. For example, constructs that are very important in Perl, such as the $_ special variable, and associative arrays, are difficult to work into Perl poetry. Perl semantics in particular get short-changed, as a main goal of Perl poetry is to use reserved words for unexpected purposes. One learns how many arguments each function takes, but not what a given function is supposed to *do*. In addition, use of all-lower-case unquoted words is a bad habit to get into, as new reserved words might be added to Perl at any point. A sloppy "if it parses, do it" mentality is not generally a useful programming style to adopt.

CONCLUSION

With only three or four practicing Perl poets, the field is still too new, and too small, for a thorough study of Perl poetry. Other computer languages boast an even smaller set of practicing poets, making cross-language poetry comparisons impossible to undertake. Perl is currently the language of choice for writing computer-language poetry, but this may be more by accident than by design. Clearly, Perl is suitable for writing poetry, but not ideal. As Perl continues to spread, and more people begin to express themselves in Perl, it is hoped that more poetry will be written in Perl by people of various backgrounds and writing (or hacking) styles. Poetry in other programming languages (even Cobol) would also be welcome; it may be that some other language, already in existence, would be even better suited to poetic flights than Perl is. It is hard to imagine, however, another programming language sufficiently quirky to attract the sort of hacker who would willingly program in poetry.

REFERENCES

[1] A pangram is a sentence that contains each letter of the alphabet, a definite number of times. See Dewdney, A.K. "Computer Recreations: A computational garden sprouting anagrams, pangrams and few weeds." *Scientific American.* Vol. 251, Num. 4, pp. 20-27, October 1984.

[2] Motte, Warren F., ed. *Oulipo: A Primer of Potential Literature.* University of Nebraska Press, 1986.

[3] From "Ars Poetica", by Archibald MacLeish. This statement is frequently used as a justification for writing poetry completely lacking in communication value.

[4] From "When the Light Blinks On", an article by Eliot T. Jacobson in rec.arts.poems, message-ID <4437@oucsace.cs.OHIOU.EDU>. Original author unknown.

NOTE ON TITLE

The first part of the title for this paper, "Camels and Needles", is taken from a passage in the Bible where the statement is made that "it is easier for a camel to go through the eye of a needle than for a rich man to enter the kingdom of God." (Matt. 19:24) Writing perl poetry is kind of like shoving camels through needles...

AVAILABILITY

For more information regarding perl poetry, or for copies of perl poems written by Sharon Hopkins, e-mail sharon@jpl-devvax.jpl.nasa.gov.

Other authors whose works are discussed here must be contacted individually regarding further reproduction of their poems.

APPENDIX

Figure 1: "program life"

```
      program life

      implicit none
      real people
      real problems
      complex relationships
      volatile people
      common problems
      character friendship
      logical nothing
      external influences

      open (1,file=friendship,status='new')
2     format (a,'letter')
      write (1,2) 'her'
c     what happens
      if (nothing) write (1,2) 'her again'
      continue
      if (nothing) then
        close (1,status='delete')
      else
        open (2,file='reply',status='old',readonly)
        read (2,2) friendship
        close (2,status='keep')
        close (1,status='save')
      end if

      end
```

David Mar, 18-Mar-1991.
mar@astrop.physics.su.OZ.AU

[Used by permission.]

Figure 2: perl haiku (untitled)

```
               print STDOUT q
               Just another Perl hacker,
               unless $spring

# Larry Wall
# lwall@jpl-devvax.jpl.nasa.gov
```

[Used by permission.]

Figure 3: "Ode to My Thesis"

```
# Ode to My Thesis, a Perl Poem
# (must be run on Perl 4.0 or higher)

<<birth                                                                    ;
G
   r
      o
         w
            t
               h
re-
birth

seek                                                                       (
        enlightenment, knowledge, experience                               );

goto MIT;

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

leave.  then, return to                                                    ;
              MIT                                                          :
now,
        $done = 'a Ph.D.                                                   ">&2';

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

study, study, do study;

push                                                                       (
        myself, computers, experiments                                     ),

read                                                                       (
        data, references, books                                            ),

study,
        write,
              write,
                      write,

do more if time                                                            ;
redo if $errors                                                            ;

do more_work if questions_remain                                           ;

$all_are_answered?  yes.

now :
        write,
        chop if length $too_great                                          ;

format                                                                     =
          Thesis
.
                        tell all,
                        done, finally                                      .
                        now, do rest                                       .

shout.
```

```
        and                                                                   .
                hear                                                          .
                        it                                                    .
                                `echo                                         "

                Now I am $done`

# Craig Counterman, April 27, 1991
# ccount@athena.mit.edu

[Used by permission.]
```

Figure 3.1: "Ode" output

```
[Output from "Ode to My Thesis"]

    I'll be here a while
          Thesis
          Thesis
          Thesis


                Now I am a Ph.D.
```

Figure 4: "time to party"

```
# time to party
# run using perl 4.003 or higher

<<;
done with my thesis

shift @gears;
study($no_more);
send(email, to, join(@the, @party, system));

open(with, ">a happy greeting");
time, to, join (@the, flock(with, $relaxed), values %and_have_fun);
connect(with_old, $friends);
rename($myself, $name = "ilyn");
$attend, local($parties);

pack(food, and, games);
wait; for (it) {;};

goto party;
open Door;
send(greetings, to, hosts, guests);
party:

tell stories;
listen(to_stories, at . length);
read(comics, $philosophy, $games);

seek(partners, $for, $fun);
select(with), caution;
each %seeks, %joy;

$consume, pop @and_food;
print $name .$on .$glass;

$lasers, $shine; while ($batteries) { last;};

time; $to, sleep
sin, perhaps;

$rest,
$till .$next .$weekend;


# Craig Counterman, April 27, 1991
# ccount@athena.mit.edu
```

[Used by permission.]

Figure 5: "listen"

```
APPEAL:

listen (please, please);

    open yourself, wide,
     join (you, me),
    connect (us,together),

tell me.

do something if distressed;

     @dawn, dance;
     @evening, sing;
     read (books,poems,stories) until peaceful;
     study if able;

     write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

         do not die (like this)
         if sin abounds;


keys (hidden), open locks, doors, tell secrets;
    do not, I-beg-you, close them, yet.

                    accept (yourself, changes),
                    bind (grief, despair);

    require truth, goodness if-you-will, each moment;


select (always), length-of-days



# Sharon Hopkins, Feb. 21, 1991
# listen (a perl poem)
```

Figure 6: "reverse"

```
first:

tempted?  values lie:

    do not, friend, alarm, "the flock";
     wait until later;
         warn "someone else"

& die quietly if-you-must;


then:

sin? seek absolution?
if so, do-not-tell-us, "on the sly";

    print it "in letters of fire",
    write it, "across the sky";

kill yourself slowly while each observes;
 do it (proudly, publicly) if you-repent;

     tell us,
     all,
     everything;


reverse after



# Sharon Hopkins, Feb.27, 1991
# reverse (a perl poem)
```

Figure 7: "rush"

```
'love was'


&& 'love will be' if
               (I, ever-faithful),
do wait, patiently;

"negative", "worldly", values disappear,

@last, 'love triumphs';

    join (hands, checkbooks),
    pop champagne-corks,

"live happily-ever-after".

     "not so" ?
      tell me:  "I listen",
                                   (do-not-hear);

push (rush, hurry) && die lonely if not-careful;

"I will wait."


&wait



# Sharon Hopkins, June 26, 1991
# rush (a perl poem)
```

Figure 8: "Black Perl"

```
BEFOREHAND: close door, each window & exit;  wait until time.
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
     kill spiders, pop them, chop, split, kill them.
        unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
     values aside, each one;
         die sheep! die to reverse the system
          you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
     wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
     select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
     wait until next year, next decade;
         sleep, sleep, die yourself,
          die at last


# Larry Wall
# lwall@jpl-devvax.jpl.nasa.gov
```

[Used by permission.]

Figure 9: "shopping"

```
# (must be run under Perl 4.003 or higher)

home: read (ads, 20, 30); sort drawers; getsockname; package returns;
      write note, tell husband; pack (purse, full, $bills);
      unlink frontgate; split, goto mall;

parking: shift gears; splice (truck, van, "crunch");
      reverse; truncate (cars, 3); require ids, insurance;
      tell bozos, wait; warn them if alarm;
      keys pocketed, open mall-doors;

mall: join (crowds, frenzy); return old-stuff, goto sales if `time-is-right`;
      listen (muzak, children); seek (bargains, deals, gadgets, goodies),
      system-atically; select (carefully), "what you want"; kill no time;
      "need more credit cards" ? open accounts: "buy now, pay later!";
      push (people, "out of my way!"); values everywhere; "Buy Everything!!!";

later: reset stopwatch; sort purchases; log $$, checks, etc.
       exit; goto car, home; join (husband, kids);
       tell children, each (twice), goto bed;
       tell spouse, "Goodnight";

#################################################################

husband: pipe (lit, glowing), close drapes;
      accept (resigned, accustomed), defeat


# Sharon Hopkins, July 18, 1991
# shopping (a perl poem)
```

BIOGRAPHY

Sharon Hopkins is a Software Test Engineer with Telos Corporation, working at the Jet Propulsion Laboratory in Pasadena, California. She received her Bachelor of Arts in History from Pomona College, where she was also the Dole/Kinney Creative Writing Prize winner for 1989.