Nick Montfort
#!
Counterpath Press, 2014.

REVIEWED BY AFTON WILKY

: ! :: Program : Poem

Nick Montfort's recent book, *#!* (pronounced *Shebang*), explores representation through language as an iterative system. Comprised of a series of programs, which can be run in a computer console, and poems produced by the programs, *#!* is as much a script or documentation of a performance piece as a book of poetry. As both script and document, process and poetry, *#!* models a duality that can be mapped onto writing, language, poetry, representation and metaphor in ways that carve out a future of writing informed by computational processes. In *#!*, humanistic and programming languages collide, generating poems that highlight the iterative nature of language and programs that draw connections between language and computable forms and functions.

Including both program and poem, the structure of *#!* invites comparison between elements of the literal language of the program and the humanistic language they output. For example, in "Round," readers are presented with the code for a program that computes pi indefinitely and uses digits of pi (0-9) to select a word or element from a ten-item list:

```
word = ['\n', 'in', 'crease', 'form', 'tends', 'tense', 'to', 'tone', 'vent', 'verse']
line = ""
pi = compute_pi()
print
```

One of the most striking elements of the program, however, is the number of variables and the number of transformations they undergo within the function that computes pi:

```
def compute_pi():
  q, r, t, k, m, x = long(1), long(0), long(1), long(1), long(3), long(3)
  while True:
      if 4 * q + r - t < m * t:
          yield m
        q, r, t, k, m, x = 10 * q, 10 * (r - m * t), t, k, (10 * (3 * q + r)) // t - 10 * m, x
      else:
        q, r, t, k, m, x = q * k, (2 * q + r) * x, t * x, k + 1, (q * (7 * k + 2) + r * x) // (t *
x), x + 2
```

This function is quite difficult to read, even if you're familiar with Python (the programming language it's written in).

The difficulty of reading the above function models, in a sense, some of the difficulty of reading a poem. In the function, instead of writing q = long(1), r = long(0), and so on, the function uses an abbreviated form in which commas indicate the end of one assignment and beginning of the next. To read the line of code requires reading across six elements on either side of the equals sign. Thus, reading requires either keeping track of which element in the list you're on or retaining the values for each. In a loose sense, this is similar to the way a poem is read. Considering the first stanza of the poem version of "Round,"

> form intends intense verse crease to tense form tense vent verse tone verse form crease form vent tends to crease to tends form form vent form crease tone verse tense

we have a series of phrases compressed that could be more readable (and heard more musically) as a series of short lines (e.g. form intends intense

verse / crease to tense form / tense vent verse tone / verse form, crease form / vent tends to crease / to tends form / form vent form / crease tone, verse tense). Beyond the compression into a single, more paragraph-like, stanza, the syntax here is missing connective language that would clarify the relationships between actions and objects. For example, while understanding the verb-object relationships in "form intends intense verse," the same is not true of "to tends form" or "form vent form." While it is possible and enjoyable to imagine these relationships, to allow a single reading these phrases would require connective language (e.g. "a tendency is form" or "the form vents the form"). Thus, through parallel structures, poem and program model the tendencies and characteristics of one another with difference that increases their visibility.

Further, and perhaps, more importantly, the difficulty of reading this function is because there are multiple variables and because the value of a variable is usually the result of computations on other variables (e.g. r = 10 * (r - m * t)). What begins to be apparent is the extent to which the role of the variables in this function can be mapped onto the individual words of a poem or even metaphor itself—variables here are constantly in flux and their value (read: signification) is only in relation to the variables (read: words) around them. Behind the text that looks like a poem:

tone vent into tends

to crease vent to crease

vent verse verse vent to crease vent

form tends vent crease tense form tends crease inintone

you have a complex system of selection operating invisibly. Its result is

the poem. In "Round," pi functions in place of semantic meaning and/or grammar. As in both, the interrelatedness and interdependence of each element is the cohesion that makes a poem (a single unit). The series of choices that shape the poem perform their role from outside the poem.

Similar to language, one of the defining features of programs may in fact be their capacity to represent elements in their abstract form. Like language, the program "Round" can point to concepts that fail in practice; for instance, infinity in the form of non-terminating loops. Specifically, "Round" is a program-poem that computes indefinitely and *is* round to the degree that it remains an infinite computation of itself; pi can only be fully circular in its abstracted form, *pi*. In practice, the program slows down as more and more steps are required to compute the next digit. Likewise, within the context of a linear space like a book or console, the repetition of computation remains distinct from repetition within the poem—even as the poem has the capacity to repeat itself, the result of repetition is always new. Thus, there is no infinite poem, only the concept of a poem that continues to perpetuate itself outside the scope of an individual poem and a program that points to the possibility of an infinite poem that either gets interrupted or exhausts itself in practice.

In such ways, *#!* is a context within which program, poem, book, console and performance space inform and shape one another. In book form, *#!* follows a program-poem sequence in which program and poem have the same title and in which each program is followed by the poem it wrote. For example, the 32-character Perl program, "Alphabet Expanding":

```
#!/usr/bin/perl

{print$,=$''x($.+=.01),a..z;redo}
```

and the poem "Alphabet Expanding,":

abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz
... abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz ... a b c d e f g h i j k l m n
o p q r s t u v w x y z a b c d e f g h i j k l m n o
p q r s t u v w x y z a b c d e f g h i j k l m n o p
q r s t u v w x y z a b c d e f g h i j k l m n o p q
r s t u v w x y z
...

This program-poem, consists of a loop that prints the alphabet one hundred times before adding an increment of space between letters and at the beginning and end of the alphabet as a whole. With each repetition of the loop, more space is added until eventually letters become a texture. By the poem's last loops, diagonal lines made up of letters become visible and appear to rain down the page, undulating delicately. When this program-poem is run in a computer console, the computer returns letters and spaces in such a way that they fill the console with an animated static and movement that is more like a film than a poem.

By adding space, "Alphabet Expanding," makes visible a collapsing of unit boundaries: the alphabet is expanded to the point where letters stand alone and continues to expand further until letters are encompassed again, by diagonal lines. Considering these three units—alphabet, letter and line—and their implementation in the poem, line seems particularly important. By moving diagonally, away from the left to right, top to bottom directionality of reading in English, line is the element that begins to push the poem further into the context of visual texts and out of language-based texts. Likewise, in the console version of the poem, the film-like motion and speed prevents any left to right, top to bottom reading. In this way, line and motion collapse even a fourth unit: text to be read.

What is and isn't "text to be read" seems, in fact, to be a pivotal question of *#!* and one that is complicated by the book's program-poem structure,

as well as the iterative nature of writing and the poems themselves. Within the context of the book, readers are presented with two versions of each piece and, thus to what extent *#!*—and indeed any book or language in general—can be a singular representation is called into question. This idea of the multiplicity of any representation or expression, which permeates both the content and structure of *#!*, is particularly evident in the poem, "ASCII Hegemony."

Another of the 32-character Perl programs in the book, "ASCII Hegemony" prints the ninety-five printing characters (0-9, a-z, A-Z, plus symbols) of the ASCII alphabet indefinitely, until the program is interrupted or crashes:

"ASCII Hegemony"

```
#!/usr/bin/perl

{print" ".chr for 32..126;redo}
```

"ASCII Hegemony"

```
   ! " # $ %   ' ( ) * + , - . / 0 1 2 3 4 5 6 7
8 9 : ; < = > ? @ A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b
c d e f g h i j k l m n o p q r s t u v w x y
z { | } ~
```

Because computers are an inherently number-based system, computing with binary numbers (where one is true and zero is false), in order to compute text computers require a set of codes that correspond to and represent text. One of the oldest and most common of these character-encoding systems is the American Standard Code for Information Interchange (ASCII). Thus, for a computer, any text is already a representation and additionally, in ASCII, characters might be represented by any one of several codes.

By titling the piece "ASCII Hegemony," attention is focused on the representation already embedded in any alphabetic system (for both computers and people), while the letters themselves embody the program-poem / invisible-visible split *#!* occupies. Further, the program's infinite loop suggests that any alphabetic system will continue to be reproduced without change until there is interruption, corruption of the program, or some other kind of failure—the stopping point is always outside the code itself.

Returning to the question of what makes this a "text to be read," "ASCII Hegemony" presents readers with a program that iterates all the visible (printable) characters in the computer's alphabet in a grid. In doing so, the program and poem deny the reader any semantic content beyond association between ASCII and the alphabet. The grid form too, pulls the piece away from sentences and lines, syntax and grammar. What's left is the raw material that is the potentiality and past of writing—these are the characters from which writing can and has been done. Additionally, to present the material of writing as writing, "ASCII Hegemony," employs context and genre. By being a part of a book that calls itself poetry, "ASCII Hegemony" presents readers with a narrative in the form of a dichotomy—this is part of a series of texts categorized as poetry and yet the program looks more like a poem than the "poem" does.

Thus, by placing the piece within the context of "text to be read" (labeling *#!* "poetry") and presenting only the material of writing (stripping the program-poem of semantic content), "ASCII Hegemony" broadens the scope of what is "text to be read" to ASCII characters as a system and as material. As with other programs in *#!,* applying the logic of this program to writing or poetry is suggestive. In this case, the program suggests that transformation of poetry and writing will result from forces outside the poem itself. For the program these forces are limits to computational endurance and interruption, but I'd say that what these might be for poetry are both the writing taking place right now and the writing yet to be done. In such ways, it is within the gaps between the program-piece duality

presented in *#!* where questions regarding poetry, computer generated poetry, and representation itself can begin. The nearest text-based parallel to Montfort's *#!* would perhaps be a process statement and piece, but the more useful comparison might be with script and performance, a comparison complicated by the fact that performance versions of *#!* also exist. In *#!*, Montfort engages both human and computer-based thought in order to carve out the future of both poetry and language.