7 Generative Code

Taroko Gorge

Author: Nick Montfort
Year: 2009
Languages: Python 2
File: taroko_gorge.py, an updated file available at https://nickm.com/code/taroko_gorge.py

Code

```
#!/usr/bin/env Python
#
# Taroko Gorge
#  A one-page Python program to generate an unbounded poem
#
# Nick Montfort
#  8 January 2009, Taroko Gorge National Park, Taiwan and Eva Air Flight 28
#
# x() splits a string into a list      c() is just random.choice()
# f() picks a fresh value from a list  p() prints a line and pauses
# cave() -- walking through the tunnels carved in the mountains
# path() -- walking along outdoors, seeing what is above (a) and below (b)
# site() -- stopping at a platform or viewing area

```

```
import time,random,sys
def x(s): return s.split(',')
def c(l): return random.choice(l)
a=x('brow,mist,shape,layer,the crag,stone,forest,height')
b=x('flow,basin,shape,vein,rippling,stone,cove,rock')

    def f(v):
        l=globals()[v]
        i=c(l[:-1])
        l.remove(i)
        globals()[v]=l+[i]
        return i

    def p(s=''):
        print s.capitalize()
        sys.stdout.flush()
        time.sleep(1.2)

    def cave():
        j=['encompassing',c(x('rough,fine'))]+\
        x('sinuous,straight,objective,arched,cool,clear,dim,driven')
        t=c([1,2,3,4])
        while len(j)>t:
            j.remove(c(j))
        v=' '+c(x('track,shade,translate,stamp,progress through,direct,run,enter'))
        return v+' the '+' '.join(j)

    def path():
        v=c(x('command,pace,roam,trail,frame,sweep,exercise,range'))
        u=f('a')
        if c([0,1]):
            if u[0]=='f':
                u=c([u,u,'monkey'])
            h=u+'s '+v
```

```
49.         else:
50.             h=u+' '+v+'s'
51.         return h+' the '+f('b')+c(x(',s'))
52.
53.     def site():
54.         return f(c(x('a,b')))+'s '+c(x('linger,dwell,rest,rela
x,hold,dream,hum'))
55.
56.     p()
57.     while True:
58.         p(path()+'.')
59.         m=c([0]*6+[1,2])
60.         for n in range(0,m):
61.             p(site()+'.')
62.         p(path()+'.')
63.         p()
64.         p(cave()+' --')
65.         p()
```

Notes

1: # precedes comments in Python.

3–4: Montfort gives the name (Taroko Gorge), constraint (one-page), and genre (“unbounded poem”).

6–7: Although putting his name in the code is somewhat unusual, putting the location where he wrote the code is highly unusual, suggesting not only that he expected a wider readership of this code beyond those merely looking to implement it, but also that he wanted people to consider its location of composition when perusing the code.

9–13: Montfort decodes all his variables and functions here, using one-letter variable names that give his code a minimalist feel.

16–17: Another feature that gives Montfort’s code its minimalist feel is that he reassigns the functions of `split` and `random` to single-character method function names (`x` and `s`, respectively). That also makes this code a bit harder to read.

18–19: These lists only contain eight words each, and two of the words appear on both lists, a simple set of inputs to create unbounded outputs.

34: Although some of his code draws from a longer list, this choice is merely between two adjectives.

46: Finding an f at the start of an "above" word (as in the case of "forest") triggers a sequence that includes the word *monkey*.

47: This monkey feature will become a staple of Taroko Gorge variations.

48–50: For greater variation from a small swap, the program cleverly adds the s to either the subject or the verb.

58–65: Start here. This section presents the main function for producing the stanzas. I recommend beginning to read the code here and then traversing the waterfall upward to find the definitions of the methods and variables.

Functionality

Taroko Gorge produces an endless stream of poetry, following a consistent set of randomized patterns. The basic pattern offers a path (noun + verb + object), followed by zero-to-two sites (noun + verb), another path, and a cave (verb + the + noun + adjective + object). The pattern is roughly ABBA-C, with some additional Bs on occasion. The lines of poetry continue to scroll until the program is stopped.

Code and Poetry

To the relief of many programmers, critical code studies does not read all computer code like poetry, but code that has been written as poetry invites just such exploration and exegesis. Much of the code analyzed so far in this book was not designed explicitly for wider audiences beyond its developers (and maintainers), although most of the examples have turned out to have wider audiences with more varied backgrounds and reading agendas and interests than the authors anticipated. The notion of artful code has been around since Knuth's Turing Award Lecture, at least. The O'Reilly collection *Beautiful Code* (Oram and Wilson 2007) celebrates artful code found in the wild, examples of artful code for coders. However, this computational age has introduced a new figure, the artist-programmer, artists who write code, who write *with* code, who bear a sense that when they are creating code, they are creating aesthetic art objects, either poetic in themselves or in combination with what they produce. Geoff Cox and Alex McLean (2012) offer many examples of this conscious, artful communication in *Speaking Code*. Perhaps a bridgework between code written aesthetically and the aesthetics of code can be found in Angus Croll's (2015) ingenious *If Hemingway Wrote JavaScript*, which offers creative reflections on how literary artists would write source code. Some beautiful code presents its signs with a visual aesthetics; the code looks like other poetry. Other beautiful code appeals to valences of concision or clever devices, such as a particularly novel use of recursion. Of course, beauty is only one aesthetic category and is obviously subjective. What seems to differentiate artful code and code art is the intentional development for human readers as a primary audience.

Code artists not only write to wider audiences but are creating code with their own poetic sensibilities. The realm of code and code-like texts (used here loosely to mean literary art objects) includes works that compile and those that do not, such as the hybrid of computer-code-like elements and natural language found in the *mezangelle* of Mez Breeze. It includes programmers hiding concrete poetry in code, as in bpNichol's First Screening written in BASIC and published in 1984 (see Huth 2007).[1] It also includes Zach Blas's queer programming language Transcoder (2017), which mixes queer theory and code-like methods and which has been used by Julie Levin Russo to create a kind of speculative fiction program (Marino 2012). Similarly, Winnie Soon (2017) created Vocable Code, inspired by *Speaking Code* (Cox and McLean 2012) and Arielle Schlessinger's speculative feminist programming language, known as C+= (Soon 2018). It also includes poetic works that look like programs but that are not written in existing programming languages, such as the poetry of Margaret Rhee in *Love, Robot* (2017), a collection that includes poems as encoded algorithms. In the book *Moonbit*, James Dobson and Rena Mosteirin offer "erasure poetry," created by selecting words from the code used in the Apollo 11 moon landing.[2] This category also includes the creative works of Alan Sondheim (2005) and John Cayley and the disruptive projects of Ben Grosser, who sees his programming as a form of critical code studies in executable form.[3] Other code, like that of the Transborder Immigrant Tool, offers poetry through its functioning, the resonance of its processes, following the traditions of conceptual writing, whether following the Latin American, North American, or other variety of this experimental tradition, placing process and premise above what we once called poems but here call "output."

To write code *for* and *as* poetry is very different than writing code to solve a utilitarian problem, although again I have been working in this book to unsettle the notion that code ever exists in a purely utilitarian space. Perhaps Heidegger offers a way out of this binary in his mediation on the root of technology, *techne*, which in ancient Greek names both craft and art. Combine that with the interventions of conceptual artists such as Duchamp, who famously framed a factory-made snow shovel as a work of art, and the distinction between "code for making art" and "code as art" becomes even more porous. If "craft" becomes art just by means of its recontextualization by an artist, then can even very ordinary, mundane code participate in an aesthetics of immanence? If Sol LeWitt could make the instructions to create art into the work of art itself, is not code that makes art also art? I don't mean to return to the threatening prospect of reading all code as poetry. I only wish to question a notion of codework that limits code art to those works that have aesthetic sensibilities in their code.[4] Before we get caught in an endless loop, let us bracket it or, in the parlance of coding, comment it out for now.

In this chapter, I take up a more clear instance of code as poetry as I explore code written and remixed by poets.[5] The object of study is Taroko Gorge, code that generates a work of electronic literature, or rather code that *is* a work of electronic literature, demonstrating how poets can explore the aesthetics of code through their programming practices. I will be reading this code as an aesthetic object and more specifically as a work of poetry.

The story of this code begins with a grand challenge issued in 2008 at Visionary Landscapes, a conference of the Electronic Literature Organization (ELO). The Electronic Literature Organization is a community of scholars and artists focused on the potential for digitally born literary works. At this conference dedicated to innovation and experimentation in the digital literary arts, there were at least four presentations that discussed the program ELIZA, at least one of which cited Janet Murray, author of the influential *Hamlet on the Holodeck* (1998), as she situates the program as one of the first works of electronic literature. ELIZA is the well-known program by Joseph Weizenbaum that, when it follows the DOCTOR script, plays the role of a Rogerian psychotherapist, asking questions of the interactor and following those questions with further questions. In one particular talk, blogging partners Andrew Stern and Nick Montfort offered a challenge, which on its surface seemed fairly simple but at its core stood as a kind of grand challenge to creators of electronic literature gathered for that congress.

By Montfort and Stern's reading, ELIZA's success was not that it deceived people into thinking it was human, achieving what Noah Wardrip-Fruin (2009) would later call "the ELIZA effect." These scholars were interested not so much in ELIZA the running software but ELIZA the artifact of code. In this presentation before a community of avant-garde digital artists, Montfort and Stern proposed that ELIZA's power was its ability to inspire so much creativity in those who would encounter it, whether drawing them in to play with it, inciting their critical readings, or inspiring their own adaptations, of which there are legion, more if you count sophisticated great-great-granddaughters such as Amazon Alexa and Apple's Siri. In their words, ELIZA is "a rather small amount of code that lacked multimedia elements, contained very little pre-written text, and was developed by a single person, Joseph Weizenbaum" (Stern and Montfort 2008). The challenge they placed before the community of electronic artists was to create code objects such as ELIZA that were simple, yet elegant enough to instigate this flurry of activity, both creative and critical.

ELIZA, according to Montfort and Stern, was a model for future works of electronic literature because of its key features:

Engaging deeply with language.

Dealing with a fundamental issue, concern, anxiety, or question about computing and technology.

Being interactive and immediate—impressing the interactor in an instant.

Being understandable after more effort is applied and the program is explored further.

Being general to different computer platforms and easily ported.

Being process-intensive—driven by computation rather than data. (Montfort and Stern 2008)

Critical to their framing of ELIZA was a sense that even though the program did not rely on computational networks or whiz-bang graphics, its relatively simple (and published) code offered rewards to those who interacted with it, either when processed to execute or as code to be read, analyzed, and adapted.

At the time, I heard in the keynotes of this speech the outlines for a project, as if Stern and Montfort were describing their work in progress. As it turns out, Montfort was offering a kind of manifesto for his own life's work. In this chapter, I look at one of his works that in partially meeting ELIZA's challenge offers a few insights into the generative potential of code art.

A year earlier, Montfort had presented another aspect of the ELIZA challenge, as I am calling it, in a presentation on the BASIC game *Hammurabi*, which he discussed in a talk with another *Grand Text Auto* collaborator, Michael Mateas (Montfort and Mateas 2007). In the talk, the two creator-theorists[6] again offer a reading of a fairly easy-to-follow program. *Hammurabi* is a BASIC game, "the first popular simulation game," which "was popularized in David H. Like, the German magazine *c't*, Ahl's 1978 *BASIC Computer Games* and went on to be often ported, rewritten, and adapted by computer hobbyists" (Montfort and Mateas 2007). Ahl's book offered owners of the newly available home computers code they could enter right into their machines and watch it execute, just as they had with 10 PRINT. The through-line from *Hammurabi* to ELIZA is the notion of relatively legible code or scripts that can be easily read and modified. The title of the talk plays on the fact that the code for the game takes the laws governing transactions from ancient Babylon (Hammurabi's code) and places it in the hands of programmers who can be little Hammurabis creating code of their own, which creates simulations of the lands that they virtually rule.

Such tantalizing fun certainly evokes the magic implied by the magicians on the cover of Abelson and Sussman's *The Structure and Interpretation of Computer Programs*. Quoting John Barth's short story "Chimera," those authors note: "It's in the words that the magic is—Abracadabra, Open Sesame, and the rest—but the magic words in one story aren't magical in the next" (1996, 487). This magical mastery is also the "sourcery" that Wendy Chun cautions against when programmers imagine that their words are making things happen through source code (2011). Montfort in his 2007 and 2008 talks is imagining not magic in code, but the magic of a teacher who presents those he encounters with objects that they can then learn from and then reimagine. As he and Stern presented, "We are interested in imagining a system that would introduce a new form, like that of the chatterbot, and that would inspire reworking and reimagining by artists" (Stern and Montfort 2008). As it turned out, less than a year later, Montfort would compose just such a piece.

Generous Poetry Generators

Before exploring code that forms (and is) poetry, let us reflect on pervasive poetic forms, such as the sonnet. The sonnet has been around for centuries. Petrarch wrote sonnets. Shakespeare wrote sonnets. Different forms, of course, but sonnets nonetheless. By now there are probably enough sonnets to wallpaper Westminster Abbey several times over. Haiku is an even older and (deceptively) simpler form. To number haiku would be to sit on James Joyce's immortalized Fontana beach, counting grains of sand. These poetic forms are merely a set of formal constraints and conventions of content, yet when attempted by generations of artistic minds of varying linguistic and cultural backgrounds, those restrictions, those boundaries, prove to be highly generative.

Of course, the quest to create new poetic forms has likewise produced its own vast bestiary. In fact, the challenge to create a new form has been so attractive that poetry collectives like the Ouvoir de Literature Potential (Oulipo) have made the creation of new forms, or Synthouliposm, their primary raison d'être. As Oulipian Raymond Queneau explained, "We call potential literature the search for new forms and structures that may be used by writers in any way they see fit" (quoted in Wolff 2007). Obviously creating a new form is one task, but convincing other writers to use that form is another. Enter onto that pitch digital computers, engines of procedural creation, and the potential for the creation of new poems has increased beyond measure, for a computer program can create a new poetic form and then iterate that form ad infinitum.

One branch of the Oulipo, the Atelier de Littérature Assistée par la Mathématique et les Ordinateurs (ALAMO), took the primarily paper- and print-based approaches of the group into the realm of algorithms and digital computers. This spinoff cadre attempted to implement and extend some of these procedures. As the Oulipans declared, "This is a new era in the history of literature: 'Thus, the time of created creations, which was that of the literary works we know, should cede to the era of *creating creations*, capable of developing from themselves and beyond themselves, in a manner at once predictable and inexhaustibly unforeseen'" (Wolff 2007).

Poetry generators have been around arguably since the first computers. Christopher Strachey, who worked with Alan Turing on the Manchester Mark I, developed a program to generate love letters (Wardrip-Fruin 2005). Although these love letters were not specifically poetry per se, or not framed as poetry, this early linguistic generator did point the way for countless generators to come. A poetry generator works with the sublime potential of combinatorics, an artistic constraint Bill Seaman (2001) has identified, which relishes the infinite potential of new combinations. With the advent of the personal computer and the rapid development of creative networks across the World Wide Web, the number of computer-based poetry generators has multiplied like our lists of sonnets and poetic forms. Over the next few pages, I would like to consider not the poems generated by these programs, but poetry generators as forms of poetry, focusing on one particular case study, Taroko Gorge, which has generated not only poems but a legacy of other generators.

Taroko Gorge first appeared in January 2009 as a one-page Python poetry generator on MIT professor and poet Nick Montfort's web page, later republished in his collection *#!* (which he pronounces "she-bang"; Montfort 2014). The program is an elegant piece of code that builds on Montfort's previous experiments with generators. *Elegance* refers to an aesthetic aspect of its code, its beauty, the way that it reads. Elegance is a kind of x factor, a je ne sais quoi for code, no more an objective measure of the code than elegance is in the grace of a stride or in the fall of a hem. Elegance is in the eye of the person reading or writing the code; computational devices, as far as we know, are largely indifferent to such aesthetics. Perhaps a provisional definition of *elegance* is concision mixed with cleverness without obfuscation.

It was in his Turing Award Lecture that computer science pioneer Donald Knuth (1974) argued for computer programming as an art. In his essay, Knuth argues that programming should be elegant, where elegance is not so much about adornment as a kind of Strunk and White, highly clear prose: simple, straightforward, legible, easy to adapt and reuse. It is this last property that Taroko Gorge demonstrates so well. But its elegance may not be readily apparent.

Montfort's own brand of elegance grows out of his love of concision. One of his prior creations, ppg256 (a 256-character Perl poetry generator), exemplifies this aesthetic perfectly. The number 256 is the number of characters (letters, numbers, and punctuation marks) that constitute the software, written in the language Perl. This generator creates poems from what is essentially one line of code. Here's an example of a poem it generated:

```
the nunelf and one hip gungod hit it.
```

The generator works by drawing from sets of syllables and combining them in a poetic structure. Although this poem may not read like contemporary lyrical poetry, such as the work of Poet Laureate Natasha Trethewey, it does resemble sound or abstract poetry. Nonetheless, the continuous generation of similar lines is a feat for a program that looks like this:

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

A nonprogrammer, or even just a newcomer to this approach, might wonder where the words are that the generator uses to create these poems, for they rely on no external texts or grammars or dictionaries. Even without knowing Perl, you can look at the first string of letters, `apebotboyelfgodmannunorcgunhateel`, and see the little units (trigrams, three-letter combinations) that would become nunelf and gungod. The program creates lines by drawing out individual trigrams and assembling them into words and phrases. No single poem produced by these generators can truly sum them up. For that, one needs to have the code. At that point, the algorithm becomes the poetry. Nonetheless, ppg256's works did not become a genre that others took up and adapted the way they would Montfort's later piece. Perhaps, notes Jeremy Douglass, that has something to do with the nature of Perl, as a language that is famously "write once, read never" (pers. interview, March 5, 2019). What these generators gain in concision, they lose in clarity, along with some of that simplicity and accessibility Montfort saw in ELIZA and *Hammurabi*.

It is worthwhile at this point to characterize Nick Montfort, who has been an active participant in the Critical Code Studies Working Groups and helped spearhead *10*

PRINT, the first book on the topic (Montfort et al. 2013). Even before critical code studies, Montfort was working away at digital art through code, although primarily in the realm of interactive fiction. However, he does not contribute his code solely as an artist. As one of the coeditors of the Platform Studies series at MIT, Montfort has been fostering critical studies in these areas. More importantly, he has been one of the few code artists to publish and present explications of his own code. He has even published a discussion of his code in the comments of his code, in an essay he wrote with collaborator poet Stephanie Strickland about their piece *Sea and Spar Between*.

As I have written elsewhere, Montfort has a programmer-poet's obsession with concision and elegance. Consider, for example, his collaborative book project (on which I was a coauthor) that focused on the exegesis of a one-line BASIC program for the Commodore 64 (Montfort et al. 2013). His interest in the line of code was not necessarily its output (which appears to be a continuously scrolling random maze formation) but rather the way in which that one line of code could inspire novice programmers to experiment. For his part, Montfort remembered that line of code from his first encounter with it, decades earlier, in the programming manual for the Commodore 64. That encounter, I would argue, instilled in the poet a sense of the way simple, concise, and elegant code could generate not only endless varieties of patterns that aggregate into a pleasing whole but also, like all great art, variations, adaptations, and reappropriations. This one-line program taught him not only the power of simple, pseudorandom pattern generation but also the generative power of simple and clever prose. Like the young poet who first encounters Basho's haiku, Montfort had encountered a kind of program that for him would become a genre that he would further explore and adapt.

To understand how this code becomes an aesthetic object, one has to stop thinking about code as something purely functional (such as the plumbing in your house) and instead as something both functional and aesthetic (like the bright pink and blue pipes used in construction projects in Berlin). Or perhaps a better example would be a beautiful stretch of road that is easy to drive on, well maintained, and lined with lovely elms. Code is written not merely for machines to process but as a form of communication between programmers, especially those who must later maintain and develop the code. Montfort values concision highly in his coding aesthetic. When discussing ppg256, he recounts his informal study of *Perl golfers*, programmers who attempt to reduce their lines of code like Tiger Woods, chipping away at their own stroke counts. Fitting his poetry generator into 256 characters puts him on par with some of the very best in the field. But, as the Perl Golf contest demonstrates, it is also an arena in which programmers can demonstrate the grace of their algorithms and the efficiency of their thought embodied in the code. Code is also an expression of thought. A cleverly designed algorithm has the force of a novel poetic conceit. Though some lines of code can be as functionally alike as two nails, they are not necessarily formally or aesthetically equivalent.

It is also important to consider ppg256 against the backdrop of Montfort's other projects. For over a decade, Montfort has worked to create tools to inspire other authors to engage in the computational production of literature. Not only in his work with ELO, having even served as its president, and in his teaching, but Montfort has also developed a platform for writing interactive fiction called Curveship. As an authorware platform, it is considerably more powerful and extensive than his microscopic generators. However, none of Montfort's previous attempts to invite other writers to a computational pickup poetry game has been as productive as his work Taroko Gorge. In this poetry generator's relatively short existence, it has spawned more than a dozen variations—each drawing upon Montfort's code and developing it in a unique way.

Taroko Gorge

Taroko Gorge is a poem generator that produces stanzas on the topic of the beautiful Taroko Gorge National Park in Taiwan. Like an electronic-age Emerson, Montfort composed the program mostly in the natural setting of the park, finishing it up on the plane afterward. As he is fond of constraints, Montfort applied some to this generator. For example, the code was not to exceed a page. Montfort explains, "I defined this 'single page' very traditionally, in terms of line printer output: The text was not to exceed 66 80-column lines" (2012). This attention to the form of the code presents a visual aesthetic more akin to concrete poetry. Like his poetry generators, Montfort's code has a minimalist character, harmonizing with the simple natural imagery his generator produces.

Unlike the versions of ppg256, the words in this generator come from a very traditional set of images. Rather than long chains of trigrams, Montfort gives himself room here to create lists of readily recognizable words that can be used interchangeably within their particular position in the lines of the poem. Any given word list, or array,[7] offers options of words from the same part of speech—for example:

- Brow, mist, shape, layer, the crag, stone, forest, height
- Command, pace, roam, trail, frame, sweep, exercise, range
- Flow, basin, shape, vein, rippling, stone, cove, rock

- Track, shade, translate, stamp, progress through, direct, run, enter
- Sinuous straight, objective, arched, cool, clear, dim, driven
- Linger, dwell, rest, relax, hold, dream, hum
- Rough, fine

From that selection, the generator produces verse, such as this:

```
Brow ranges the coves.
Forests dwell.
Forests hum.
Brows trail the cove.

  progress through the encompassing cool—

The crags sweep the flows.
Forests relax.
Heights command the shapes.

  enter the sinuous—

The crag ranges the veins.
Forests exercise the veins.
  track the straight objective arched clear—

Monkeys frame the stones.
Shape commands the cove.
  direct the straight objective driven—
```

Much more sophisticated than ppg256, the lines of poetry generated have a sparse quality apropos of their object. The alternating use of the article *the* gives weight to the objects and actors, who seem both specific and timeless. They are both concrete (rocks, stones) and abstract (shapes, flows), metaphorical (brows, veins) and material (mist, forest). Montfort's simple constructs of path, site, and cave reveal an artist creating little units of phrase as meditative spaces.

Montfort has called Taroko Gorge an “unbounded” nature poem, but it is important to realize that he is not referring to any poem generated by the code but to the code itself as the poem. This takes some readjustment. What makes the poem limitless is that the program, once executed, continues to iterate. Boundlessness thus is a characteristic not of any one set out of output, but of the capacity of the program to develop poetry without limit.

Furthermore, the Python code of Montfort's generator presents a kind of elegance that gets lost in the HTML/JavaScript version. In fact, Montfort has written, "Python is a programming language I prefer for when I'm thinking" (2010). Consider a brief comparison. Here is the code for making a line of Python:

```
p()
while True:
    p(path()+'.')
    m=c([0]*6+[1,2])
    for n in range(0,m):
        p(site()+'.')
    p(path()+'.')
    p()
    p(cave()+' --')
    p()
```

The variable names are all defined in the comment at the head of this code. The pattern of the poetry is this:

```
          Path.
             0, 1, or 2 sites.
          path
          Cave --

Path = Noun + verb + object.
Site =   Noun verb
Cave = verb + the + noun + adjective + object
```

The word lists are made up of two primary groups, *a* and *b*, which the HTML/JavaScript reveals to be *above* and *below*. From these simple structures, drawing upon relatively brief lists, the generator produces multitudes.

By contrast, here is the same function in HTML/JavaScript:

```
function do_line() {
var main=document.getElementById('main');
if (t<=25) {
t+=1;
} else {
main.removeChild(document.getElementById('main').firstChild);
}
if (n===0) {
text=' ';
} else if (n==1) {
paths=2+rand_range(2);
text=path();
} else if (n<paths) {
text=site();
} else if (n==paths) {
text=path();
} else if (n==paths+1) {
text=' ';
} else if (n==paths+2) {
text=cave();
} else {
text=' ';
n=0;
}
n+=1;p
text=text.substring(0,1).toUpperCase()+text.substring(1,text.length);
last=document.createElement('div');
last.appendChild(document.createTextNode(text));
main.appendChild(last);
}
```

The HTML/JavaScript does in thirty lines what the Python does in ten. The Python has an elegance that the translated version can't quite match.

As a poet and programmer, Montfort is well aware of the challenges of translation. When he published the code on the site *Media Commons* (2012), Montfort printed a JPG image of the printout, treating the printout of the code like a poet's manuscript.

This same code is published in *#!* (Montfort 2014), leading John Cayley (2015) to argue that it is the code that is the poem because the code can be read by humans, whereas the infinitely generated content cannot, a state that again points more toward the regenerating landscape of the national park that inspired the work: its fruits can be experienced only ever as the blossom of a moment. On the other hand, Aden Evens (2018) suggests that the infinite set of all possible conditions is what makes this piece poetry. Regardless of where one sees the poetry, the procedure or the possibility, both exist, in hibernation or as seeds, in the code.

This code has a type of elegance that echoes Montfort's other work. To read the code is like tracing the waterfalls beneath the Eternal Spring Shrine back to their origin. The reader can begin at the bottom and then, in graceful loops, work their way up into the code and back again. After learning the shorthand for the methods, primarily the methods to split, get a random number, and pick an item from a list, the reader can hike the code by beginning with the last section. The code at the end offers the form of each stanza:

```
p(path()+'.')
m=c([0]*6+[1,2])
for n in range(0,m):
   p(site()+'.')
p(path()+'.')
p()
p(cave()+' --')
p()
```

The first encounter is with the `p` method, which affects capitalization and timing. Those effects are applied to `path`. As readers encounter the `path` method, they can then go and read the definition of the path. `Path` offers one list of words but also a selection from list a, sending the reader up to read that line of words. Descending back to the stanza code, the reader encounters `site`, which sends her back up to the `site` method. Returning again to the stanza, the reader encounters `path` again and finally `cave`, which sends the reader back to that method.

As one might take any number of paths through a natural park, sometimes on a marked path, sometimes off, my portrait of reading through graceful loops offers one encounter with the code out of a near infinite array of approaches. However, I would argue that this Python code of Taroko Gorge offers a clear and concise organization that is refreshing to read, especially in contrast to reading some of the other, more elaborate case studies in this book and the larger software from which they were excerpted. Remember, Montfort is writing this code so that, like a snapshot of the falls, the code can be read on one page. He has made specific choices—namely, assigning variables and methods to single-character names—to give his code an austerity akin to his ppg256 works. However, in this case the code's simplicity and elegance are complemented by the content of the code and the continuous spring of simple yet stirring nature imagery that it engenders.

But because the HTML/JavaScript version can be rendered in a web-browser without downloading Python, that is the version that was adapted by so many and that raised Montfort's poem to the level of ELIZA's challenge, at least in part. Not only is the JavaScript more readily accessible to writers and readers on the internet, but Montfort's JavaScript is easier to read because the methods appear in more verbose, rather than abbreviated, form. For example, in place of the method `c` in Python, the JavaScript uses `rand_range`. Consequently, although there are not prominent Python variations of the generator, there are many JavaScript ones created by programmers of all levels, who could more easily see how the piece functioned. Returning to the discussion of readability of code in chapter 4, this analysis of code offers an example of two versions of the same code, one that is concise and elegant to the writer (the first reader) and one that is more accessible for modification by those who would adapt it.

The Descendants

Although Taroko Gorge would become a subject of many adaptations, the artists who would remake the project mostly changed the data, rather than the rules of the code. On the one hand, such adaptations would seem to follow the model of the sonnet or haiku, in which those who take up the form supply new words to fit into its constraints but for the most part do not change the constraints themselves. To write a sonnet is not to change the rules of a sonnet.

However, because Montfort is a poet of code, one who offers his code itself up as the poetry (Marino 2010b), and because we are viewing this work in the context of ELIZA's grand challenge, the adaptations seem on the surface to miss the mark by engaging with the data and not the rules. Such a reading misses the creative intervention of those who followed. They were not forking and adapting the code. Instead their work reimagines the poem, coopts its form, through a process more akin to remix.

To call these adaptations *remixes* is not to denigrate them but instead to locate them in a rich cultural practice of recycling and reinvention. In *Remix Theory*, Eduardo Navas (2012) theorizes the remix as a "cultural glue" that is "always unoriginal" but at the same time a potential "tool of autonomy" (4). The remix artist takes the existing material and transforms it through the method of cut/copy rather than creating something new from scratch. In like fashion, the remixers of Taroko Gorge did not rewrite the program, but copying the code, changed the data. (Technically, the program itself is the remixer par excellence as it is the one reshuffling all the data.) The remixing of Taroko Gorge has acted as a cultural glue that binds together disparate artists with widely varied aesthetic priorities and poetic interests.

The first of the adaptations, or remixes, was written by another of Montfort's blogging collaborators, Scott Rettberg. Rettberg is a writer of digital literature perhaps best known as the cofounder of the Electronic Literature Organization, whose work is known for a postmodern playfulness and ironic tendencies. Rettberg describes his reworking of Montfort's poem as a "hack," originally sending the link to his remix with a note that he had "made a few improvements" (Rettberg 2019, 47–48). At the time, neither Rettberg nor Montfort expected others to follow suit.

As the first of the adaptations, Scott Rettberg's Tokyo Garage (2009) could have been a one-off work of e-lit, or more specifically a one-off adaptation of a one-off poetry generator. Consider the style of its poetry:

> Scholars hate the dog.
> Undercover cops explode.
> Mystics perspire.
> Drummers subdue the Roppongi drunks.
>   imagine the lithe uptight digital blinking—
>
> Spokesmodels proselytize the nose rings.
> Hallucinations transport the subways.
>   digest the scattered—

Rettberg uses a larger set of words than Montfort, but rather than drawing from nature, they arise from a particularly Japanese style of commercial culture (Godzilla, kabuki dancer, Speed Racer) set in a noisy and sullied streetscene (technicolor nightmare, prostitute, pickpocket, bribe, hassle, grope). In his opening comment in the code of his adaptation, Rettberg writes, "This here is a total remix of the classic and elegant generated nature poem Tokoro [*sic*] Gorge by Nick Montfort. He wrote the code here. I hacked the words to make it more about urbanity, modernity, and my idea of Tokyo, a city I have never been to." If Montfort's poem is an ode to the simple, boundless natural beauty of a park, Rettberg's poem offers a noisy homage to popular culture with a Japanese pop–inspired aesthetic. Although Rettberg claims to have "hacked the words," it is hard to consider this reworking to be mischief. As John Cayley (2015) reminds us, “most, if not all, of his work is published with actual or implicit license allowing copying, reuse, and modification so long as attribution and the same licensing terms are maintained.” Rettberg’s self-incrimination is a bit of poetic performance of his own.

His gesture might have been the end of the matter, except something about Rettberg’s adaptation caught the attention of future developers, as his adaptation developed some of the conventions of this poetic form or perhaps new poetic genre: remixes of Taroko Gorge. One was that the titles would be a play on Montfort’s original title, with many of the listed variations starting with the letters T and G, including Takei, George (2011) and Toy Garbage (2011), as well as other wordplay, such as Yoko Engorged (2011), Fred and George (2012), Alone Engaged (2011), and Gorge (2010).

Gorge by JR Carpenter

The next major adaptation of Taroko Gorge was electronic poet J. R. Carpenter’s Gorge, which, following Rettberg, she calls a remix. (*Remix* is an interesting term to use because it originally referred to a reworking of a recording of a song.) Carpenter does not rearrange the code or the content of Taroko Gorge, but instead, like Rettberg, changes the data. By the time she worked on her remix, Carpenter was already an established print and electronic poet, known for her own word play and technical innovation, as well as a media art historian and theorist.

Rather than taking a location, such as the park in Montfort’s poem or the urban space in Rettberg’s, Carpenter takes for her focus an act, that of gorging oneself. Carpenter writes: “A gorge is a steep-sided canyon, a passage, a gullet. To gorge is to stuff with food, to devour greedily. GORGE is a new poetry generator by J. R. Carpenter. This never-ending tract spews verse approximations, poetic paroxysms on food, consumption, decadence and desire.” *Gorge* the noun becomes the verb *to gorge*.

Her variables for above and below give a sense of the piece:

```
var above='appetite, brain, craving, desire, digestive juice,
digestive tract, enzyme, gaze, glaze, gorge, gullet, head,
incisor, intellect, jaw,knowledge, language, maw, mandible, mind,
molar, muscle, mouth, nose, passion, sight, smell, spit, sweat,
spirit, thirst, throat'.split(', ');

var below=', bladder, blood vessel, bowl, bowel, crust, dip,
dressing,film, gut, lip, lower lip, proffered finger, finger
```

```
tip, flared nostril, flushed cheek, meal, membrane, morsel,most
intimate odour, palm, passage, persistent scent, pore, sauce, soft
pocket, slightest sliver, stomach, surface, thick spread, tongue,
taste bud, vein, vinaigrette'.split(', ');
```

Montfort's initial eight words have quadrupled in the `above` variable, and `below` has thirty-four words. In the place of simple nature images are words drawn from the realm of eating (glaze, gullet, molar), but also more broadly sensual appetites (proffered finger, palm, flushed cheek). They are the signs and substance of gorging oneself, and that gorging is hardly limited to food: she also adds *film* and *gaze*, words that together conjure a feminist critique of male visual desire (as in the "male gaze"). Both the above and the below groups feature body parts that perform the consumption. The above list includes maw, jaw, mandible, molar, mouth, and throat, the below list lip, lower lip, and tongue. However, the above and below groups do offer a conceptual divide. The seat of this desire in the above section prioritizes the mind (brain, head, intellect, knowledge), whereas the below section emphasizes the digestive (stomach, bladder, bowel, gut). Another group of words in the below list emphasizes body parts that might register arousal (flared nostril, flushed cheek). These words suggest that gorging is a sensual activity. And another list (most intimate odor and proffered finger) suggest a sexual dimension, although that "most intimate odour" could very well refer to flatulence, a mere effect of gorging. Nonetheless, casting that odor as intimate returns us from the itemizing of mere body parts and physiological effects to the above list, with its *craving* and *desire*.

The moments when the list breaks out of an expected pattern change the meaning of the other words on the list because they suggest another dimension of meaning. In a mere list of foods, "thick spread" sounds like mayonnaise, but in the list with "flushed cheek" it evokes a sensual encounter, making this gorge far more than a shopping list and changing the way its other words (blood vessel, vein, aroma) resonate. That is to say, Carpenter has done far more than swap in new data; she has remixed the poem with a poet's methods of connotation and allusion. To read these observations out of the code is not to "cheat" because, as I mentioned, she has published the code for our consumption so that we might savor the lists even in their potential state, even just as ingredients in this poetic pantry. To read these poems is to peruse their code.

Significantly, Carpenter has published the code in a poetry collection with instructions on how to adapt the generator. The generosity of this move is itself generative. Carpenter's publication of the code and its output is also instructive. She presents the contents of the prior authors' code as variables—that is, the Montfort variables and the

Rettberg variables. Her presentation suggests that the code is the same and that only the data changes—similar to presenting the form of a Petrarchan sonnet separate from the various versions.[8]

Even some of Montfort's original words remain in Carpenter's piece. Note that in Rettberg's adaptation, he had changed Montfort's monkey line to the following:

```
if ((words=='pachinko parlor')&&(rand_range(3)==1)) {
  words='mobile phone '+choose(trans)
```

However, in Carpenter's code, the `forest` trigger for the output `monkeys` remains:

```
if ((words=='forest')&&(rand_range(3)==1)) {
words='monkeys '+choose(trans);
```

This section of code produces only the lines relating to monkeys. But because Carpenter does not use *forest*, this particular piece of code remains like a memento tucked away in a trunk someone has purchased at a flea market. It may never be used or noticed by the reader yet is a tie to the original owner.

A closer examination of Carpenter's and Rettberg's code reveals a sense of what they consider the data and what they consider the form. On the one hand, outside of `monkeys`, Carpenter, like Rettberg, replaces the data for all of the main variables (the arrays: `above`, `below`, `trans`, `intrans`, `texture`). On the other hand, neither Carpenter nor Rettberg rename these variables themselves (`above`, `below`, `path`, `cave`). Taroko Gorge persists in these images. Carpenter also does not include these variable names either in the Montfort variables or the Rettberg variables. So whereas Montfort's poem about a trip to a national park follows paths with caves, both Carpenter's Gorge and Rettberg's Tokyo Garage also include paths and caves. The distinction may be that as the author of the initial algorithm, Montfort was thinking more consciously about his code as the thing being read, whereas Carpenter and Rettberg saw their remixing work as something that happens at the level of the data. Nonetheless, by publishing her code alongside sample output, Carpenter surely invited such a reading. Regardless of what they changed, by adapting the poem generator for their own purposes, Carpenter and Rettberg led the way for many subsequent poets to continue the regeneration of this generator.

After Carpenter's adaptation, the descendants multiply. Among the other variations, consider Talan Memmott's Toy Garbage, Kathi Inman Berens's Tournedo Gorge (2012), and Mark Sample's Takei, George. Other adapters include Eric Snodgrass, Maria

Engberg, Flourish Klink, Andrew Plotkin, Brendan Howell, Adam Sylvain, Leonardo Flores, Alireza Mahzoon, Sonny Rae Tempest, Helen Burgess, Judy Malloy, Bob Bonsall, and Chuck Rybak. All these creators' poetry generators share basic underlying programming; however, with the differences in their themes—such as toy nostalgia, culinary quests, and *Star Trek* culture—the generators can hardly be called versions of Montfort's original. They are unique poems in their own right. Or, to return to the beginning of this chapter, they are generators in the new poetic form of Taroko Gorge generators. Each continues to play on Montfort's original title, mostly keeping to the original verse structure but at times varying, and mostly using his code. What Montfort has created, then, is not so much an infinite poem as a genesis of a chain of poetic action, of engagement with poetic elements mostly as data. His generators are generative, and his poetic meditation on nature leads to electronic reflections on pop culture, human vice, and other nuances of being, which we might sum up as the stuff of poetry.

However, the variations on his code in the earlier Taroko Gorge remixes have not attended to the instructions, a central object of Montfort's aesthetic project, but instead to the data. In the hierarchical world of code art, data can take second-class status for it seems like mere content to be shuffled around. It is the content that the computer does not "care" about. Anastasia Salter (2017) warns against such a hierarchical emphasis on code in an age of the "leaky pipeline," which loses (i.e., leaks) its bright and creative female programmers due to unsupportive or hostile learning and work environments.[9] Because she offers Taroko Gorge and its variants as an access point to coding, as a way in for those without formal training, I would be remiss to reinsert the division by celebrating adaptations of instructions over adaptations of data. In fact, I suspect that binary is, like so many used to enforce a hierarchy, false.

Moreover, in this instance, in which the remixers have changed the values of the variables but not the names of the methods, they have, in a sense, changed content that the computer does care about for it will need to store those strings as values—albeit as ones and zeroes and ultimately, as Kittler has acknowledged, as electrical signals. The method names by contrast will disappear at the level of the assembly code. Thus, they exist in the JavaScript but disappear in assembly. Such distinctions, however, are immaterial because JavaScript is the language that the reader encounters, and these method names, though arbitrary, are symbols in that system. Ultimately, the distinction between adaptations that change the instructions of the code and those that change the data can make value claims only in the context of human aesthetics. The computer cares little about art, as far as we know. At this point, it is useful to return to the example that inspired the challenge: ELIZA.

ELIZA is a conversational system built by Joseph Weizenbaum on the MAD-SLIP (Symetric List Processor) system. SLIP is an application programming interface written in FORTRAN II (Weizenbaum 1963). What we generally call ELIZA is actually a script called DOCTOR that Weizenbaum wrote for that system, a script he published in 1966, and it is possible from reading through the keywords and the templates for the responses—or, to use the language of bots, the grammars—to develop a sense of how the program operated. Although we do not have the ELIZA system itself, between transcripts and accounts of interactions with the program and this script, we can get a sense of how it worked. The code begins with a greeting statement:

```
(HOW DO YOU DO. PLEASE TELL ME YOUR PROBLEM)
START
```

Start begins a communication exchange. If the person says the keyword "sorry," the system recognizes it in the following template:

```
(SORRY ((0).
```

It reacts by drawing from a series of responses:

```
(PLEASE DON'T APOLIGIZE)
(APOLOGIES ARE NOT NECESSARY) (WHAT FEELINGS DO YOU HAVE WHEN YOU
APOLOGIZE) (I'VE TOLD YOU THAT APOLOGIES ARE NOT REQUIRED)))
```

The system seems to store values in variables represented by numerals. For example, the keyword `REMEMBER` in the following statement:

```
(REMEMBER 5
```

That triggers the following result templates, which make use of the value stored in 4 and 5:

```
((0 YOU REMEMBER 0) (DO YOU OFTEN THINK OF 4)
(DOES THINKING OF 4 BRING ANYTHING ELSE TO MIND)
(WHAT ELSE DO YOU REMEMBER)
(WHY DO YOU REMEMBER 4 JUST NOW)
(WHAT IN THE PRESENT SITUATION REMINDS YOU OF 4)
(WHAT IS THE CONNECTION BETWEEN ME AND 4)) (0 DO I REMEMBER 0)
(DID YOU THINK I WOULD FORGET 5) (WHY DO YOU THINK I SHOULD RECALL
5 NOW) (WHAT ABOUT 5) (=WHAT) (YOU MENTIONED 5)) ((0) (NEWKEY)))
```

That said, because we cannot currently access a functioning version of ELIZA, the software implementations of ELIZA are functional equivalents. Because the code of ELIZA is unavailable for reading, those who adapt it are not interacting with its code but often customizing the content (the keywords and responses) of a similar system—for example, one of the many JavaScript versions that circulate on the internet. Like the remixers of Taroko Gorge, they can change the data without changing the instructions since we do not have the original instructions of ELIZA system itself.

Since Weizenbaum's publication of ELIZA, chatbots have proliferated (Marino 2006b). Not only do people remix the original ELIZA, but they build their own systems, varying the character, the persona, the nature of the call and response. ELIZA has inspired automated telephone agents, novel generators, and art other projects, such as Peggy Weil's Mr. Mind in *The Blurring Test*, which challenges the player to prove to the computer they are human. They have built platforms for chatbot authoring, such as Richard Wallace's Artificial Intelligence Markup Language (AIML), which powers his ALICE bot. Furthermore, programmers have been inspired to wrestle with more complex systems for character-based human-computer interaction, leading all the way to the Siris and Alexas of today—and who knows what else by the time you are reading this.

Argot, Ogre Ok!

One of the first Taroko Gorge descendants to meaningfully play with the instructions of Nick's code was Andrew Plotkin's variant, which is a remix of the remixes. Plotkin is primarily an interactive fiction writer, well-known to Montfort and the electronic literature community. By calling attention to their interactions with Montfort's code and foregrounding code on the display level of the generator, Andrew Plotkin's Argot, Ogre, OK! (2011) performs its own critical code studies on Montfort's poem and its variants.

In Plotkin's version, named for an anagram of Taroko Gorge,[10] he mashes together pairs of versions of Taroko Gorge, specifically Rettberg's Tokyo Garage; Carpenter's Gorge; Eric Snodgrass's Yoko Engorged; Mark Sample's Takei, George; Talan Memmott's Toy Garbage; Maria Engberg's Alone Engaged; and Flourish Klink's Fred and George.

Plotkin's presentation comments on the Taroko Gorge lineage by displaying the code that generates the poems, one line at a time, beside boxes (iframes) that show the poems they generate. At the same time, Plotkin varies the stanza structure and mixes the word pools from the various projects two at a time. For example, his code creates a new mashup, "Alone and George," which combines Klink's and Engberg's variations. Plotkin notes that his own code lacks a bit of the sparse simplicity of Montfort's, but he feels he has remained true to the aesthetic of the original.

Plotkin documents his process in an elaborate comment at the beginning of his code. In a piece that displays parts of its own code on the screen, it is interesting to see what Plotkin does not display. First, he is quick to assure the readers that "Yes, this page really does execute the code that's displayed in the left column, and it really does generate the text in the right column." Plotkin is addressing readers of this code, whether they are looking to remix Taroko Gorge further or approaching the piece as readers of literature. Just to highlight the point, this is a writer/remixer of code writing a comment in the code to an audience whom he anticipates interpreting the code. We have reached a moment in which poet programmers are addressing outside audiences, inviting audiences to explore, to interpret, and to engage with their code.

Through his code, Plotkin comments on the remix tradition of Taroko Gorge, as emblematized in his treatment of the monkey feature. First, he offers a quote in his epigraph to his opening comment in his code:

"Does it have a monkey?"

"Yes, the monkey is 'taboo.'"

—Nick Montfort and Flourish Klink, Sept 26, at dinner

If you were not yet convinced of the heteroglossia of code or that it it is more than merely instructions and their documentation, here is a passage that would seem more at home in a story by Dorothy Parker. This passage suggests that the monkey of Montfort's original poem, and its presence or omission in its remixes, had become a point of humor. The taboo Klink mentions is her response to the monkey. In her code:

```
if ((words=='wizard')&&(rand_range(3)==1)) {
  words='taboos '+choose(trans);
```

Klink's trigger word, her *forest*, is the word *wizard*, and the word that she adds as a consequence, her monkey, is *taboos*. By including the exchange between Montfort and Klink, Plotkin is identifying the ways in which Taroko Gorge has become a poetic form with its own consistent features. Plotkin's homage to this part of the poetic form is as follows:

```
if ((words==src.monkeysee)&&(rand_range(3)==1)) {words=src.
monkeydo+' '+choose(src.trans);
```

Each poem object has a designated `monkeysee` and `monkeydo`, which correspond to the triggers and consequences respectively, including the following:

Taroko Gorge: forest, monkeys

Tokyo Garage: pachinko parlor, mobile phone

Gorge: forest, monkeys

Toy Garbage: BABY ALIVE, MOTHER MAY I

Yoko Engorged: [one space], null

Takei, George: Sulu's smile, Kirk smolders and Sulu vows to

Alone Engaged: old lover, mobile phone

Taroko Gorge: wizard, taboos

Plotkin has made the monkey into the `monkeysee` that triggers the `monkeydo`, now an attribute of his object that comically calls attention to this mischief-maker in the code of Taroko Gorge and its remixes. The monkey feature of the code is also now a sign of the degree to which the programming remixer engaged with the code.

By mixing the remixes, Plotkin's code identifies the similarities in the instruction layers while creating a melange of the data. At the same time, Plotkin's own code seems to take up the challenge of Taroko Gorge, reworking at the layer of instructions with a sense of Montfort's original provocation, with some ironic metamixing of his own. Plotkin is hardly alone in playing with the instructions. Other variants of Taroko Gorge have taken them up, adding their own layers of instructions, as in the case of J. R. Carpenter's Along the Briny Beach (2012), which includes a generated graphical coastline among other features. What's notable about Plotkin's intervention is that it uses code to comment on Montfort's code, even foregrounding the code by bringing it to the presentation layer.

So does Taroko Gorge answer the ELIZA Challenge sufficiently? It is certainly too early to tell. On one level, Taroko Gorge is like ELIZA, many of the descendants of which merely follow the same functional structure as that original chatbot but the authors of which have changed the content of the keywords and triggered responses. On the other hand, the true descendants of Taroko Gorge may look nothing like it.

They could generate endless lists of beer names or tiny protests. They could be used to generate whole novels. It all depends on what one counts as Taroko Gorge. What are its specifications? Is it a generator of infinite variations of the ABBA-C pattern? Or is it one in a long line of text generators, the grandchildren of which, like Siri and Alexa to ELIZA, may be unrecognizable (and abhorrent) to the designer of their grandmother.

For the purposes of this book, Taroko Gorge stands as an example of this new literary moment, one in which poet programmers engage with language in and through their code, generating poetry while producing works of art in code that invite exploration whether through reading, execution, or remixing. Regardless of where it stands in the ELIZA challenge, Taroko Gorge has succeeded in drawing poet-programmers in to play with its code as a possibility space of its own. Critical code studies invites readers and writers to reflect on this code, exploring not just what it does, but what it means.