CHAPTER P-3

Generation

In 2009, the year of Twine's debut, the poet and computational linguist Nick Montfort visited Taroko National Park in the Republic of China, the site of a famously splendid gorge on the Liwu River. Take a poet to the wilderness and the result is usually a nature poem. Crossing that exposure with computer science yields what is arguably a new kind of nature poem. Written (we could also say "coded") during the flight home, Montfort's *Taroko Gorge* is a poetry generator; you can read (watch?) the work at nickm.com/taroko_gorge (Montfort, *Taroko Gorge*). Its compact rules weave several sets of words into a richly impressionistic account of Montfort's hike through the gorge—which is to say, a poem—but the program runs on an endless loop, continually scrolling from bottom to top, adding lines that are apparently unique, made fresh every few seconds. This feature turns the nature poem into something like a simulation. *Taroko Gorge* captures in words the basis of natural beauty: an endlessly surprising permutation of given elements.

As we suggested in chapter P-2, electronic writing has a particular affinity for permutation, which brings us to a second important aspect of *Taroko Gorge*. Like the community of Twine developers, Montfort believes strongly in open-source software. He shares his poem-making code with anyone who wants to adapt it. Another poet, J. R. Carpenter,

has written a collection called *Generation(s)* consisting of her own reworkings of Montfort's earlier programs (Carpenter). Something similar, though less overtly organized, happened with *Taroko Gorge*. Friends and family of Montfort began swapping out the vocabulary in Montfort's program and restyling the poem in evocative and sometimes parodic ways (e.g., *Tokyo Garage* and *Takei, George*). The web page for the poem includes a table of intervening authors with links to their versions. Montfort ritually crosses out each name and reasserts his own at the bottom of the list—less an exercise of ego than a playful recognition that authorship isn't what it used to be.

Taroko Gorge is not a Twine work, but perhaps a first cousin. It is written in Python, a language popular with web coders. Montfort has written other works in JavaScript, the specialized programming language that provides an infrastructure for Twine. However, the idea of generated text, language programmatically assembled by combination or random selection, belongs to every branch of the software family. It is present in the naming ritual at the beginning of *With Those We Love Alive*, discussed in the last chapter and imitated in one of our practical exercises in this one. As another of our examples will show, text generation can be used to make static poems as well as endless simulations. Most important for our purposes, exploring this practice will show us more about the affordances and limitations of Twine, building on our encounter with textual variation in the previous practical chapter.

> ◊ As in other practical chapters, action items are boxed and set off with the symbol you see at left, in case you want to skip the contextual discussion. Examples in this chapter use Chapbook exclusively.

Supporting materials for this chapter can be found online at https://github.com/AMSUCF/Twining. See our discussion at the beginning of chapter P-1 about using the .html and .txt files to follow along or adapt our code to your own purposes.

Example 3.1: Mad Computer Libs

If we want to seem serious and dignified, we can describe the subject of this example as a *substitution grammar*, borrowing a scientific term from computational linguistics. However, like many serious and dignified things, our subject here is actually rooted in party games. In 1958, a pair of comedy writers, Leonard Stern and Roger Price, published *Mad Libs*, a book containing phrases, sentences, and paragraphs for which the player was meant to supply missing words, as outrageously as possible. In the "Mad-Mad" era of cocktails and party games, the book was a huge hit, even though it was hardly original. A generation earlier, the surrealists invented a practice called the exquisite corpse, in which standard patterns of language were intentionally disrupted by sharing a text with multiple authors, imposing new rules for each new writer, or allowing each writer to see only the most recent fragment of the text.

Though neither the surrealists nor the comedians knew it, they were working at the margins of computer science. In 1952, a British researcher named Christopher Strachey created a program to generate love letters (Wardrip-Fruin). Its sentence-template mechanism was closer to Mad Libs than the surrealist game, and the title of our first project reflects that fact (along with a glancing homage to Ted Nelson's *Computer Lib/Dream Machines* of 1974, the manifesto of our movement). Give your fingers a good flex before digging in. There are five passages, each with a fair number of words to type.

◊ Create a new story and call it anything you like. We'll be using Chapbook, so select that story format if it is not your default. Create a new passage and name it "Step 1." This name is referenced in a link, so change it with caution. In the new passage, enter the following text:

```
propNoun: 'Somebody'
--
Type a proper noun: the name of a real or
imaginary person, place or named thing, e.g.,
```

```
'Louise Pringle,' 'Jimmy One Nose,' 'H.M.S.
Winnebago.'

{text input for: 'propNoun'}

[[Next ->Step 2]]
```

We're working here with a Chapbook feature you haven't seen yet, the *text input* insert. As you may suspect, this insert creates a box into which the player is expected to type something at will. When the player leaves the "Step 1" passage, the contents of the input box are stored in the variable *propNoun*. The initial value we assign in the variables section shows up in the text-entry box and can be used as a default if the player declines to type anything.

◊ Create another new passage and name it "Step 2." It's very similar to "Step 1," except we're looking for an adverb this time.

```
adverb: 'furiously'
--
Type an adverb, e.g. 'triumphantly,' 'softly,'
'twice.'

{text input for: 'adverb'}

[[Next ->Step 3]]
```

◊ Create another new passage and name it "Step 3." Enter the following text:

```
verb: 'ignores'
--
Select a verb:
```

```
{dropdown menu for: 'verb', choices: ['avoids',
'wrangles', 'removes','finagles','blasts','enjoys
','terrifies','exhausts','tickles','amuses']}

[[Next ->Step 4]]
```

We could have kept on with the *text input* insert, but for the sake of exploration, we'll instead use the *dropdown menu* insert. It does pretty much what you'd expect, creating an expandable menu from which the player is expected to select. Obviously, it imposes more constraint than free input, a move you may want to make from time to time, even in such a minimally structured game. As with text input, we assign an initial value ("ignores") to the key variable. This word comes up as the default selection. Each time the player selects a word, it is assigned to the *verb* variable. The final selection (or default, if no selection is made) is passed on when the player clicks the "Next" link.

◊ Create another new passage and name it "Step 4." Enter the following text:

```
org: 'the Modern Language Association'
--
Select a civic organization:

{dropdown menu for: 'org', choices: ['Friends of
Linda','the Ancient Order of Voles', 'the Liars
League', 'International Mothers Helpers','Men
with Hats','the Committee of the Hole']}

[[Next -> Step 5]]
```

Again, this step follows the pattern of the one that preceded it: another dropdown menu, this time listing civic organizations. Feel free to shorten, expand, or modify this list. This project does not test for specific selections.

◊ We're almost done. Create another new passage and name it "Step 5." Enter the following text:

```
enders: ["in bed","for a limited time","in
stores everywhere","as seen on TV","in your
loudest dreams","where not prohibited by law"]

ender: enders[Math.floor(random.fraction*enders.
length)]
--
{propNoun} {adverb} {verb} {org} {ender}.

[[Play again? ->Step 1]]
```

The variable *ender* adds a final phrase to our basic noun-adverb-adjective sentence. Coming unforeseen, it's meant to add a punchline, like the phrase people insufferably recite after reading the contents of a fortune cookie—"in bed." (It's still technically a punchline even if it's dumb.) Joke-theory aside, have a look at the code with which we deliver this final phrase: it contains a trick you haven't seen before, which you will encounter again in the next project. The variable *enders* is an array, a list of values (in this case, strings) to which we can refer by number. The variable *ender* contains a selection from the array, using a complicated but powerful expression:

```
enders[Math.floor(random.fraction*enders.length)]
```

With a little variation, this is the same syntax used in JavaScript to make a random selection from an array. We're deliberately mixing JavaScript and Chapbook syntax.[1] In JavaScript, we indicate an item of

1 Twine is built on JavaScript, so its relationship to various Twine dialects, like Chapbook coding or Harlowe scripting, somewhat resembles that of an older language to English—JavaScript : Chapbook || Latin : English. This is an imperfect analogy, but it does help explain what's going on when we blend JavaScript and Chapbook syntax. It's like dropping a Latin phrase into an English sentence, exempli gratia.

an array by using the array name, followed by a number or expression in square brackets. The first element of the array is *enders[0]*, the second is *enders[1]*, and so on. The expression used here resolves to an integer value between 0 and its maximum range, inclusive of 0 but excluding the maximum. The *floor()* function of the Math object, which rounds a fractional value down to the next lowest whole number, comes from JavaScript. The *random.fraction* lookup is from Chapbook—the corresponding JavaScript would be *Math.random()*. Likewise, *enders.length* calls on the built-in *length* property of JavaScript arrays. When we wrote the first draft of this chapter, this technique was not documented; we discovered it by experiment. Since all Twine formats communicate with JavaScript, it's always worth probing for hidden connections—we'll see another in example 3.5. Sometimes the attempt is futile. It's not always possible to mix JavaScript and Chapbook features—some of our later examples in this chapter will explore the limits—but in this case, it works to welcome effect. We'll give more details of this technique in the next example.

First, though, we invite you to give our Mad Lib generator a few spins. You can of course break and abuse its grammar rules all you want. The next examples are more severe, if not more serious.

Example 3.2: Subject-Verb-Object Generation in Chapbook

◊ This example is quite compact, consisting of a single passage and one line of code in the text body. All the typing comes in the variables section. Create a new story and name it anything you like. (We suggest *S-V-O* as a handy nickname.) Start a new passage and name it "Sentence me." You can change this name if you want, provided you reflect the change in the final link. Here are the complete contents of the lone passage. Type away!

```
subjs: ["Edgar","The cat","Edgar
the cat","Gorgomon","Stephane
Grapelli's typewriter","An
```

```
astrolabe","Mrs. Macaleister","An implausible
gravy"]

theSubj: subjs[Math.floor(random.fraction*subjs.
length)]

verbs: ["eschews","thrashes","adores","invalidat
es","steals","withholds","accuses","dethrones"]

theVerb: verbs[Math.floor(random.fraction*verbs.
length)]

objs: ["Niall","the planet Mercury","Episode
Three","our better angels","Stephane Grapelli's
typewriter","furiously","space and time","to no
discernible purpose"]

theObj: objs[Math.floor(random.fraction*objs.
length)]
--
{theSubj} {theVerb} {theObj}.

[[Again! ->Sentence me]]
```

All the hard work comes at the top, in the variables section. We define three arrays and three string objects, each one containing a random selection from one of the arrays. There's one array/string pair each for subject, verb, and object. We've already discussed the hybrid Chapbook/JavaScript selection mechanism that made its first appearance at the end of example 3.1. There's no difference in that structure here. The Chapbook expression *random.fraction* resolves to a decimal between zero and one, which is exactly the same as the JavaScript *Math.random()* function. Multiplying by the length of the array gives us a fractional number between zero and the length of the array. So *random.fraction* might give us a value of 0.356792. Suppose our array has seven items. Multiplying

that number by a value between 0 and 7, say 4, gives us 1.427168. Only integers can be used as array selectors, so we need to do some rounding.

We use the JavaScript *Math.floor()* method to convert this decimal to the next lowest integer. That's because the numbering of arrays begins with zero and stops one short of the array's length value. If the array *Joey* has seven items, they'll be as follows:

```
Joey[0]
Joey[1]
Joey[2]
Joey[3]
Joey[4]
Joey[5]
Joey[6]
```

There's never a *Joey[7]*. Rounding down keeps us safely within the range.

With the array selections conveniently stored in our three respective string variables, all we need to do is deliver them, which we do with a line of three variable includes on the other side of the two dashes that close off the variables section. *Et voilà*.

Example 3.3: S-V-O in JavaScript

This example is something of a digression, so we won't go through the process of breaking it down for sequential construction. You can do the typing if you like. Even though you'll be typing in JavaScript, set your story up with Chapbook. That may seem odd, but it's time to reveal an important Chapbook affordance: *you can include extended bits of JavaScript code in Chapbook passages*. We'll take some first steps with this technique in the next few examples. If you're interested in going further, be sure to read the final section of this chapter, where we offer some important technical considerations.

It's been possible to include JavaScript in Twine projects from early on, but in version one of Twine, this code had to be entered in specially

marked passages. Twine 2 lets you put JavaScript directly into story passages. You do this with the *JavaScript* modifier, which is just what you see on the first line. You will use a *continue* modifier eventually to switch back to standard Chapbook mode.

◊ Here's the text to type, if you're inclined:

```
[JavaScript]
t=""
subjs = new Array("Edgar","The
cat","Edgar the cat","Gorgomon","Stephane
Grapelli's typewriter","An
astrolabe","Mrs. Macaleister","An implausible
gravy")
verbs = new Array("eschews","thrashes","adores"
,"invalidates","steals","withholds","accuses","
dethrones")
objs = new Array("Niall","the planet
Mercury","Episode Three","our
better angels","Stephane Grapelli's
typewriter","furiously","space and time","to no
discernible purpose")

t = subjs[Math.floor(Math.random()*subjs.length)]
+ " "
t += verbs[Math.floor(Math.random()*verbs.
length)] + " "
t += objs[Math.floor(Math.random()*objs.length)]
+ "."

write(t)

[continue]
[[More ->Tales from the Script]]
```

The basic architecture of this example is similar to that in 3.2: we set up three arrays, containing subject nouns, verbs, and object words or phrases. We make selections from the arrays using the three-step procedure explained earlier—generate a fraction, multiply by the array length, round downward.[2] The delivery mechanism is different. Variables defined in JavaScript can't be passed into Chapbook—this is the first of those functional limitations we'll need to explore. Chapbook allows only one thing to be done with a JavaScript variable, at least without some serious programming: you can pass it to a custom method called write(). This method, which should not be confused with the JavaScript / document object model (DOM) method called *document.write()*, does for the JavaScript variable what the Chapbook insert does for Chapbook variables. It writes the value into the visible text of the Twine passage. The value we write here comes from the variable *t*, which we use to build up our sentence one word at a time.

If you run this JavaScript-inflected example, you'll see the same output as in example 3.2: a subject-verb-object sentence. Since we can achieve the same end without wading into JavaScript, you may ask why we led you on this tour. There's a reason. In working with random selections, you'll sometimes want or need to do things that are not possible in Chapbook. We'll come to one of those cases in our next example.

Example 3.4: Nonrepeating Randoms and the Knuth Shuffle

Let's start by identifying a problem:

Tell us your story
I am a red wheelbarrow painted orange
Shake the boards and howl

2 For consistency, we are using the JavaScript generator for a random fraction, *Math.random()*. However, we have accidentally discovered that it is possible to substitute the Chapbook alternative, *random.fraction*, even inside a JavaScript modifier!

I am a red wheelbarrow painted orange
Shake the boards and howl
I am a red wheelbarrow painted orange
I am a red wheelbarrow painted orange
Struggle to define existence

Does this look like something someone has written? If so, we might wonder about all that repetition: two of the eight lines occur twice, and a third appears in quadruplicate. Maybe this is the work of a neo-minimalist poet who is really into repetition. Maybe these are lyrics from a song and the repeated lines are connected to something that makes sense musically. Or maybe this is just output from a bad text generator.

Let's suppose the generator in question works at the sentence level: our next example (3.5) will feature one of those. Perhaps this sentence-level generator just needs more sentences to draw from, though it's also possible its random-selection tool has a basic flaw.

The fundamental tools for generating random numbers in Chapbook are variations on an object named "d" for "die," the singular of "dice." There are variants for integer ranges of 4, 5, 6, 8, 10, 12, 20, 25, 50, and 100. We've already seen *random.fraction* in use, where it does the same thing as the JavaScript *Math.random()*. All these mechanisms have the same weakness: like physical dice, they can produce the same number twice (or more) in succession. In the Chapbook guide, Klimas alludes to Tom Stoppard's play *Rosencrantz and Guildenstern Are Dead*, which opens with a coin flip that stubbornly refuses to produce tails. A tenfold run of heads is vanishingly unlikely in both the real world and software (1024:1 against); but *d.8* returning back-to-back threes is much more probable. The odds may look long at 64:1, but software often involves repeated and rapid iteration—not to mention a thing called luck.

◊ In this example, we'll eliminate the possibility of repeating numbers for a defined range of random selections. This can't be done with Chapbook tools, so we'll turn to JavaScript. Create a new story in Twine using the Chapbook format. (We'll embed our JavaScript in a Chapbook story.) Create a single passage

and give it a useful name. We call our version "Loopy." Start with these lines:

```
[JavaScript]

sourceArray = new Array('Sunday','Monday','Tuesd
ay','Wednesday','Thursday','Friday','Saturday')
```

This bit should look familiar: it's a standard array declaration. We use the days of the week because they're a familiar sequence. The technique will work with a list of any kind.

◊ Next, we'll enter some more variables:

```
trackArray = new Array()
trackArray.push(99)
rNum = 99
```

The first of these lines creates an empty array called *trackArray*. The second line uses the *push()* method to place the number 99 into the first (and so far, only) position of *trackArray*. In the third line, we declare a variable called *rNum* and give it the value 99. The number 99 is essentially arbitrary: we need to use the same number both for the first array item and for *rNum*, but that number could be anything.

◊ Next, we'll write a JavaScript *function*:

```
function randy(){
   while(trackArray.includes(rNum)){
   rNum = Math.floor(Math.random()*sourceArray.
   length)
   }
trackArray.push(rNum)
return sourceArray[rNum]
}
```

A function (also referred to as a *custom method*) is a group of statements introduced by the keyword *function* and a set of parentheses. The *function body* is then defined within a set of curly braces. The statements within a function have a special status. They are not immediately put into effect (executed) but are held in reserve until the function is activated or *called*. Statements in a function can be called multiple times, often from diverse parts of a longer program.

This function contains a crucial piece: something called a *while loop*. We're using JavaScript in order to access this structure—*Chapbook does not include any kind of loop*. By contrast, there are two types of loops in JavaScript: limited loops, usually *for loops*, that run a specific number of times, and indefinite loops, which run as many times as needed until their stop conditions are met. The indefinite loop we're using here runs *while* we're waiting for a certain outcome. *While loops* are enormously powerful. They are, in fact, the only way to prevent repetition in a random-number sequence.

Programmers tend to be wary of indefinite loops because, in theory, they can turn into loops of a dangerous third kind: *infinite loops*. Unless you are trying to outwit a being of pure energy on *Star Trek*, infinite loops are bad.

Properly written, an indefinite loop is harmless. At the dawn of computing machinery, indefinite loops were avoided because they are inefficient, and computing cycles cost real money back then. In the not-too-distant future, as we recognize the energy impact of all our irresponsible computing, opinion may once more turn against these structures—though the worst offenders are cryptocurrencies and porn. For the moment, concern is muted. Use *while* while you can.

Our magic loop runs under one condition: the value of *rNum* occurs somewhere within the array *trackArray*. If this condition is true, we execute the line contained within the loop, which generates a random value for *rNum*. This is, by the way, the reason we set *rNum* initially to a number we also push into *trackArray*. We need a match in order to get our first generated random. Once we have that number, the loop then checks if this value is in *trackArray* already—in other words, if our number is used or unused. If the

number is fresh, the loop terminates. Outside of the loop, we push our guaranteed-unique number onto the tracking array and return the value in *sourceArray* (a day of the week) that corresponds to that number. When a function returns a value, it is fed back into other parts of the script or program.

Our little program eliminates repetition by sorting our original array into a nonsequential pattern. This is like shuffling a deck of cards. In fact, the scheme upon which this program is very loosely based is called the Knuth shuffle algorithm, named for the computer scientist Donald Knuth, author of the classic textbook *Literate Programming* (Knuth). Once we've established the basic principle of shuffling, we can proceed to action.

◊ Here's the last of the JavaScript:

```
for(var i=0; i<sourceArray.length; i++){
   write(randy()+'<br>')
   }
```

This is that other sort of loop, a *for loop*. It runs seven times (the length of our source array) and calls our unique-selection function (*randy()*) each time. Because we embed the call to *randy()* in a *write()* statement (which you'll remember from the previous example), the result of the selection is made part of the visible text.

◊ All that remains is the final Twine link, allowing us to replay the whole business:

```
[continue]
[[Again ->Loopy]]
```

If you haven't made any mistakes, this example should display the seven days of the week in a differently randomized order every time you reload its single passage. Why is this outcome significant? Well, each of these seven-day sequences is both random and nonrepeating;

in the second of our next pair of examples, you'll see why that matters. For the moment, though, let's try a fresh approach to text generation.

Example 3.5A: Situation Reports (Passages)

◊ Create a new story using Chapbook. Call it anything you like. This example is surpassed only by our too-many-links experiment (1.2) for number of passages. There are eleven in all, so the setup will involve a little tedium. You can spare yourself some repetitive strain by first creating ten new passages. Name each one numerically from one to ten—we won't count from zero this time. Open each passage in succession and enter the corresponding sentence from the following list—one sentence to a passage. Do not include the numbers in the passage text:

```
1. The specimen emits radiation in the X-band.
2. The density of the specimen appears to be
increasing.
3. The specimen does not respond to repeated
perturbation.
4. Whoooo da good specimen?!
5. The specimen may be entirely anechoic.
6. The density of the specimen has in fact
decreased.
7. Attempts to ascertain the origin of the
specimen are ongoing.
8. We were briefly unable to locate the specimen.
9. The specimen may have assimilated Technician
Anderson.
10. The specimen has no observable effulgence.
```

Remember when we promised an example of text generation at the sentence level? Well, here it is. More significantly, this generator also works at the *passage* level, which is an interesting way to operate in Twine.

◊ There's just one more passage to build now. Create a new passage and name it "Readout." Let's start with its variables section:

```
passages: ["1","2","3","4","5","6","7","8","9",
"10"]

passage1: passages[Math.floor(random.
fraction*passages.length)]
passage2: passages[Math.floor(random.
fraction*passages.length)]
passage3: passages[Math.floor(random.
fraction*passages.length)]
passage4: passages[Math.floor(random.
fraction*passages.length)]
passage5: passages[Math.floor(random.
fraction*passages.length)]
--
```

The first variable we create is an array called "passages." It simply stores the numerals from 1 to 10 as strings. Why not an array of numbers, you may ask? This is because passage names must be strings.

We've also set up five variables, each assigned a random selection from "passages"—in other words, a selection of five passages out of our set of ten. Yes, we can pick the same passage more than once. I am a red wheelbarrow painted orange. We'll fix this in the next example. You may wonder why we used a series of numbered variables instead of an array. While we could type out a five-item array in the same way we built our ten-piece array in the first line, all those randomizations would create typographic chaos, with their tricky embedding of elements. In JavaScript, we'd use a *for* loop to fill our array by repeating the random-number assignment five times. However, Chapbook has no loops at this writing. (Who knows if they'll be added later.) When you have a relatively small number of items, sometimes a simple series of variables will do.

```
◊ Now for the finishing touches, which follow
directly after the -- that closes the variables
section:


Situation Report {now.date}:
{embed passage: passage1}

{embed passage: passage2}

{embed passage: passage3}

{embed passage: passage4}

{embed passage: passage5}

[align right]
[[Update ->Readout]]
```

And there it is, a tiny bit of sci-fi horror in five sentences, making use of a very important Chapbook insert called *embed passage*. As the name suggests, this modifier copies the contents of the specified passage into the present passage at the point indicated. Using SugarCube and Harlowe, Twine writers have already developed similar techniques into a very fine art. Random selection among a range of passages—ideally a much larger set than the present ten—can be a powerful tool for building an unpredictable structure. It's interesting enough if all we're doing is scooping up one sentence at a time, but there's no reason the embedded passage can't contain Chapbook code like our S-V-O substitution grammar from example 3.1. We'll demonstrate that idea in example 3.6. Randomly choosing output *from a set of independent generators* could produce very surprising and potentially delightful results. Or maybe just the opposite. Such is the challenge of art.

Example 3.5B: The Horror . . . the Nonrepeating Horror

Meanwhile, let's take on a more approachable challenge: modifying our passage-based generator so it won't pick the same passage twice. We already know how to do this for selections from an internal array. Now we'll adapt our JavaScript code to integrate the Chapbook-based passage selector from the previous example.

◊ Set up a new story using Chapbook. We'll be using JavaScript within Chapbook—with an interesting twist or two.

Like 3.5A, this is a big one: eleven passages. There are two ways to go here. Since the ten embeddable passages are identical to those in 3.5A, you could start by duplicating and renaming your version of that story. If you take this work-saving option, open the passage called "Readout" and delete its contents. Then you're ready to go. If for some reason you enjoy tedious typing, then start a fresh story and repeat the procedure in the first step of example 3.5A: make ten new passages and enter a sentence in each. Whichever way you go, we'll assume you have eleven passages, each containing a sentence, and a passage called "Readout" with no contents. Ready?

◊ We'll start by entering a variables section in our new version of "Readout":

```
ep1: ''
ep2: ''
ep3: ''
ep4: ''
ep5: ''
--
```

You may ask yourself, *Hey, aren't we supposed to use JavaScript for this thing? So why do we have a Chapbook variables section?* It turns out that Chapbook has an originally undocumented quirk—if you declare

a variable first in Chapbook and *then declare it again in JavaScript*, you can do more than pass the value of the variable for screen display. Using this exploit, we can compute a value for a variable using JavaScript, then pass it to a Chapbook insert. (Klimas confirms this is permissible; he just overlooked the possibility when he wrote the first version of the Chapbook guide.) While we're asking skeptical questions, you might also ask why we're using five separate variables instead of an array. It turns out our JavaScript pass-through exploit doesn't work with arrays, at least at this writing. Oh well.

◊ With our obligatory Chapbook work out of the way, let's get started on the JavaScript. Enter the following:

```
[JavaScript]

sourceArray = new Array('01','02','03','04','05'
,'06','07','08','09','10')

trackArray = new Array()

trackArray.push(99)

rNum = 99
```

These lines should look familiar from example 3.4. They're the standard setup for our Knuth shuffle implementation. The source array contains the names you gave to the arrays that contain our sentences.

◊ Here's the next piece of our JavaScript:

```
for(i=0; i<5; i++){
   while(trackArray.includes(rNum)){
   rNum = Math.floor(Math.random()*sourceArray.
   length)
   }
```

```
trackArray.push(rNum)

if(i==0) ep1 = sourceArray[rNum]
if(i==1) ep2 = sourceArray[rNum]
if(i==2) ep3 = sourceArray[rNum]
if(i==3) ep4 = sourceArray[rNum]
if(i==4) ep5 = sourceArray[rNum]
```

This is our *while loop* again, this time wrapped not in a function but in a *for loop* with five iterations. As before, we push our guaranteed nonrepeating value onto *trackArray*. That's why we use a series of *if* tests to route our successive selections into five distinct variables whose values can be passed back to Chapbook.

◊ The remainder of the project is pure Chapbook. The structures should be familiar:

```
[continue]

{embed passage: ep1}

{embed passage: ep2}

{embed passage: ep3}

{embed passage: ep4}

{embed passage: ep5}

[[Again ->Readout]]
```

The *continue* modifier shifts us back to Chapbook mode. The embed inserts pull in the contents of the passages whose names were selected (without repetitions!) in our JavaScript shuffle maneuvers. That's the project. It has all the advantages of passage-based generation without

the flaw of inelegant repetition. The JavaScript-Chapbook trick play shows again why it's worth tinkering with code, especially when it's young. Software sometimes doesn't know its own strength.

Example 3.6: Free Verse, or You Get What You Pay For

Two examples back, we promised a project that uses embedded passages that generate text through local computation. Ideally, we'd try embedding a passage containing something like our S-V-O sentence generator. Though we have something else in mind for this example, let's think through the S-V-O experiment first.

◊ To try later: set up a Chapbook story, create passages named "bedfellow" and "testbed," and copy the Chapbook S-V-O code from example 3.2 into "bedfellow." In "testbed," type the following:

```
{embed passage: 'bedfellow'}
{embed passage: 'bedfellow'}
{embed passage: 'bedfellow'}
```

Lo and behold, you can embed the same passage more than once![3] Since "bedfellow" generates a plausibly fresh sentence each time it's accessed, you should end up with three unique pieces of nonsense. The world needs more nonsense.

With this conceptual exercise behind us, let's turn to something more concrete and hopefully less nonsensical: a genuine free-verse

3 At this point, a certain bad thought may come into your head: *What if I write a passage for embedding, which in turn embeds the passage that embeds it?* In other words,

> Passage A embeds passage B
>
> Passage B embeds passage A (which embeds B, which embeds A embedding B, and so forth)

The saints in heaven cry when you do this. Also, your browser and JavaScript make Twine knock it off after about a thousand iterations. Now you don't have to try this thing, right?

generator. *Free verse* is poetry without constrained rhyme or meter—low-hanging fruit for demonstration purposes, though you could build out the technique shown here to write more demanding forms, such as haiku, sonnets, or villanelles. As in example 3.5A, we'll work at the level of complete lines. That decision raised the specter of repetition in the earlier example, but as you'll see, we have no such worries here.

◊ Set up a Chapbook story called anything you like (ours is called *Free Verse*). Create seven passages in that story. Name the first one (the default starting passage) "Versify." Name the others "Opener," "Middle 1," "Middle 2," "Middle 3," "Middle 4," and "Finisher." Open "Versify" and enter the following:

```
{embed passage: 'Opener'}
{embed passage: 'Middle 1'}
{embed passage: 'Middle 2'}
{embed passage: 'Middle 3'}
{embed passage: 'Middle 4'}
{embed passage: 'Finisher'}

[[Again ->'Versify']]
```

No mysteries here: our main passage embeds all six subsequent passages in sequence, without any random choices. That business happens in the embedded passages themselves.

◊ Open "Opener" (that was awkward) and enter the following:

```
r: random.d6

open (r === 1): 'something I heard no one say'
open (r === 2): 'now this'
open (r === 3): 'between some dreams I thought I
heard'
```

```
open (r === 4): 'you might not believe this'
open (r === 5): 'this much the night allowed
me'
open (r === 6): 'a spider spun this for me once'
--
{open}
```

This is a reasonably simple structure. To the variable *r* we assign the results of the *random.d6* function, which is an integer between 1 and 6 inclusive. Next, we take a variable called *open* and assign it a value based on *r*. In Harlowe or JavaScript, we would use *if* statements or perhaps a *switch* construction, but we've learned to work differently in Chapbook (see example 2.4). We give the variable name, enclose the condition we want to match in parentheses, and after a colon, we give the value we want the variable to have if the condition is met. In example 2.4, we used this technique to create a scoring scale. Here it's a randomizer. (Notice we're using one of the whole-number random functions from Chapbook, this time without any funny business.)

After we close the variables section, we simply insert the variable. Chapbook automatically knows which permutation has been chosen. It's worth noting that when we perform the insertion here, the action effectively migrates to our main action passage, "Versify," where the present passage ("Opener") is embedded. Anything made visible in an embedded passage is made visible in the embedding passage.

◊ As you may guess, the remaining five passages are versions of "Opener" with different text. Here's "Middle 1":

```
r: random.d6

m_1 (r===1): 'water is a silence'
m_1 (r===2): 'a silence has come upon the
waters'
```

```
m_1 (r===3): 'we are the sum of waters'
m_1 (r===4): 'drink your water in silence'
m_1 (r===5): 'the silence of waters'
m_1 (r===6): 'water is never really silent'
--
{m_1}
```

◊ Enter the following in “Middle 2”:

```
r: random.d6

m_2 (r===1): 'imagine the invention of water'
m_2 (r===2): 'every moment is the beginning of
invention'
m_2 (r===3): 'the spark of nothing less is
this'
m_2 (r===4): 'for seeing the word so far'
m_2 (r===5): 'having carried words no further'
m_2 (r===6): 'water could not be silent'
--
{m_2}
```

◊ Enter the following in “Middle 3”:

```
r: random.d6

m_3 (r===1): 'quantify your blessings'
m_3 (r===2): 'render your account'
m_3 (r===3): 'spill out that bag of content'
m_3 (r===4): 'say what you contain'
m_3 (r===5): 'read the bill of particulars'
m_3 (r===6): 'gather up the washing'
--
{m_3}
```

◊ Enter the following in "Middle 4":

```
r: random.d6

m_4 (r===1): 'dreams beasts sex'
m_4 (r===2): 'monkeys jewels and fabulous
stories'
m_4 (r===3): 'a rock a mandarin a twisted pike'
m_4 (r===4): 'larks crows kingfishers calumets'
m_4 (r===5): 'gazettes and galley proofs'
m_4 (r===6): 'whelks whales and waterfowl'
--
{m_4}
```

◊ And finally, enter the following in "Finisher":

```
r: random.d6

finish (r===1): 'and that was something said'
finish (r===2): 'and nothing more of time'
finish (r===3): 'and then the rain came'
finish (r===4): 'until we end our song'
finish (r===5): 'or this and nothing more'
finish (r===6): 'the machine stops'
--
{finish}
```

We'll just say one more thing about this project: the writing doesn't aim at artistic merit, but it does stumble closer to seriousness, or at least coherence, than our other efforts. We do this to suggest the possibilities of this free-verse generator, or something like it, as a genuine literary device—or perhaps a gateway experience. Computational poetry is an established and flourishing field. Substitution grammars are just the beginning. More advanced work can involve N-gram text generation,

operations on large digital text bases, and various forms of machine learning. If you want to know more about the history of computational poetry, check out Chris Funkhouser's *Prehistoric Digital Poetry* (Funkhouser, *Prehistoric*). For recent trajectories, see his *New Directions in Digital Poetry* (Funkhouser, *New Directions*), as well as the website of the School for Poetic Computation: https://sfpc.io/.

Example 3.7: Game of Names (after Porpentine)

We'll finish with one from the heart. The naming ritual at the beginning of Porpentine's *With Those We Love Alive* creates a moment of high enchantment. We've discussed its mysteries in the previous critical chapter. Here we'll subject it to shameless imitation, partly for one more demonstration of the creative possibilities of Twine, and mainly because we think (well, one of us does) that making up character names is huge fun.

◊ Create a new Chapbook story and call it what you will. Make two passages. Name the starter passage "Choose Your Time." Name the other one "Your Name Will Be." Let's start with the variables section in the first passage. Open "Choose Your Time" and type the following:

```
theDays: ['Scum','Monster','Tomb','Weed','Thirst
','Fear','Scatter']

theSeasons: ['Waking','Making','Darkening','Ni
ght']
--
```

We make two arrays, one for the seven days of a fantastical week, the other for four eldritch seasons. This is less detail than Porpentine works with; we need to keep things manageable for demonstration purposes.

◊ After closing off the variables section with the required double dashes, add the following to "Choose Your Time":

```
Today is a {dropdown menu for: 'theDay',
choices: theDays}day in the season of {dropdown
menu for: 'theSeason', choices: theSeasons}.

[[So it is; who am I? ->Your Name Will Be]]
```

It's our old friend, the dropdown menu. We could have used cycling links as in the Porpentinian original, but no one likes a robotic imitator. Going with the *dropdown menu* insert also allows us to demonstrate a small but useful trick: you can define your menu options as an array, up in the variables section, then simply reference the array as the argument to "choices" within the insert. If for some reason you have a long menu of choices, this separation might be convenient.

◊ Open the second passage, "Your Name Will Be." This passage has a long variables section, which we'll break into pieces. Start with this:

```
firsts: ['Drag','Mars','Mol','Bren','Hal','Dom']

seconds: ['rak','ra','della','bim','bang','rica'
,'dottir','goth','gren','thing']

thirds: ['Hamble','Rumble','Storm','Mountain','R
iver','Valley','Moose','Squirrel']

fourths: ['hand','mind','foot','body','thumb','w
easel','love','song']
```

We have four arrays, each containing options for syllables in a four-syllable name. We'll have two kinds of names—some that are generated from these arrays, and another set arrived at differently.

◊ Here's the next piece of the variables section in "Your Name Will Be":

```
nameFirst: firsts[Math.floor(random.
fraction*firsts.length)]

nameSecond: seconds[Math.floor(random.
fraction*seconds.length)]

nameThird: thirds[Math.floor(random.
fraction*thirds.length)]

nameFourth: fourths[Math.floor(random.
fraction*fourths.length)]

id: nameFirst+nameSecond + " " +
nameThird+nameFourth
```

The four sequential variables receive random selections from the syllable arrays. The combination of these variables is assigned to the variable *id*. Note that the plus sign (+) works here as a *concatenation operator*, simply sticking together some strings. That's because JavaScript (and Chapbook) automatically changes the function of the plus sign when a string is involved. (If only numbers are involved, "+" signifies mathematical addition.)

◊ One more push to complete the variables section:

```
id (theDay==='Monster' && theMonth==='Waking'):
'Slam Danghandle'
id (theDay==='Weed' && theMonth==='Making'):
'Leah Romavant'
id (theDay==='Tomb' && theMonth==='Darkening'):
'Gnowth Marvydink'
id (theDay==='Thirst' && theMonth==='Night'):
'Crassa Foomstoffer'
```

```
id (theDay==='Fear' && theMonth==='Waking'):
'Blastgret Stimsocket'
id (theDay==='Scatter' && theMonth==='Making'):
'Meera Upfallen'
id (theDay==='Scum' && theMonth==='Darkening'):
'Kristel Vannafoy'
id (theDay==='Monster' && theMonth==='Night'):
'Markie Mistmother'
--
```

Don't forget the dashes closing the variables section. Here we have another instance of conditioned variables, as in examples 2.4 and 3.6. We single out eight combinations of day and season for special names not made from the syllable sets. We throw in this feature arbitrarily, but it does add some stakes to the naming game. There are twenty-eight permutations of days and seasons and eight special names, giving a 2:7 chance of obtaining one. If there were a story attached to these two passages, maybe a character with a special name would be treated differently than one with a generic, generated name. Or maybe, as in *With Those We Love Alive*, the name would essentially be a MacGuffin, elaborately generated but not otherwise consequential. Creativity is all about choices.

◊ The rest of the second passage is simple:

```
[align center]
Born on a {theDay}day of the {theMonth} month,
you are:

~~{id}~~

[continue]
[[Try again? ->Choose Your Time]]
```

Finish and test. Who will you turn out to be?

So ends our encounter with text generators. Like other applications of Twine, they may be silly, serious, or somewhere in between. They can incline toward the party-game fare of Mad Libs, to the revolutionary aesthetics of the exquisite corpse, to the endlessly iterative fascination of *Taroko Gorge*. Whichever way you turn, you'll be working with a system fundamentally dedicated to possibility, variation, contingency, and play. Chapter T-4 looks at the cultural implications of that iconoclastic, radically playful turn.

Technical Notes on JavaScript and Chapbook

This chapter has introduced the JavaScript modifier, opening the door to hybrid constructions. If you'd like to explore that path, you may want some experience with JavaScript first. Our bonus practical chapter presents a series of projects that work exclusively with JavaScript without touching Twine. They might be a good place to start, and of course, commercial guides to JavaScript authoring are abundant. An excellent resource for creative applications is Montfort's *Exploratory Programming for the Arts and Humanities* (Montfort, *Exploratory*). What follows here is fairly detailed and is meant primarily for those who have become interested in doing more ambitious things with JavaScript and Twine.

A subject of great concern to modern programmers is *the order of operations*—just when the computer processes your instructions. In Chapbook/JavaScript hybrids, you need to consider this issue in arranging your code. Let's say you have two custom methods or functions, A and B. Method A invokes method B. In browser-based JavaScript, you can place the definition of B after A in your code. Not so with Chapbook, which in our experience will throw a pink fit if you refer to a method you haven't previously defined.

The order of operations also has implications beyond the structure of code. If you've worked with JavaScript in web pages, you may have encountered a situation where a script's behavior depends on where it is placed within the page code. The "HEAD" division is usually safe, but not always—if, for instance, your script needs to interact with an

element that is dynamically added to the page by another script. The browser needs to load related elements before your script operates on them. A similar problem can arise with Chapbook if you write a script that changes a page element and do not either (a) make the script dependent on a user action, such as reloading the page, or (b) delay the first execution of the script by using *setTimeout()* or *setInterval()*. These are advanced topics that go beyond the purview of this book, but we thought we'd mention them in case Twine starts complaining that you've tried to modify a "null" object.

Finally, a general note about the Twine/JavaScript relationship. As we've noted, it's incestuous. Twine runs in JavaScript, so its hybridized structures are always a little . . . kinky. On occasion, you may find that JavaScript instructions, especially those with compound and complex syntaxes, don't behave as expected. JavaScript in Chapbook is not quite the same as JavaScript in a conventional web page. However, these instances are rare, and we can do a remarkable range of things without running into difficulties. As we've said, you can see more of those possibilities in the bonus chapter at the end of this book.

Works Cited

Carpenter, J. R. *Generation(s)*. Traumawien, 2010.

Funkhouser, Chris. *New Directions in Digital Poetry*. Bloomsbury, 2012.

———. *Prehistoric Digital Poetry*. University of Alabama Press, 2007.

Knuth, Donald. *Literate Programming*. Cambridge University Press, 1983.

Montfort, Nick. *Exploratory Programming for the Arts and Humanities*. MIT Press, 2016.

———. *Taroko Gorge*. 2009. https://nickm.com/taroko_gorge/.

Nelson, Theodor H. *Computer Lib/Dream Machines*. Mindful Press, 1987.

Wardrip-Fruin, Noah. "Christopher Strachey: The First Digital Artist?" Grand Text Auto, August 1, 2005. https://grandtextauto.soe.ucsc.edu/2005/08/01/christopher-strachey-first-digital-artist/.