

Only you can see this message



This story's distribution setting is off. [Learn more](#)

A Minimal Syntax For Quantum Text



Aaron A. Reed

Oct 10, 2019 · 8 min read

Subcutaneous, my upcoming novel that changes each time it's printed, works like this: there's a master text with the whole story, occasionally splitting into alternate versions and variants at the level of words, sentences, or even whole scenes. Each time the book is "rendered" for a new reader, a single option from each set of variants is randomly chosen, resulting in one particular version of the story. I wrote earlier about the aesthetics of why you'd do this, but the *how* is interesting too.

I call this kind of writing "quantum authoring," because the author must hold all the possible versions in their head at once and keep each one interesting and consistent. Unfortunately, this kind of writing is often intertwined with programming or other mentally exhausting tasks, like operating a complex tool or remembering a finicky syntax. For this project, I wanted to write in a format that was as lightweight and unobtrusive as possible, so I could keep my brain entirely in "writing" mode while working on content. What I came up with was a minimal format called *.quant*, and I want to talk a bit about why I made it and what it's good for.

The *.quant* format had a couple fairly simple requirements. First, I didn't want it to distract from my creative process. Having written a lot of procedural text over the years (usually for games) there's nothing worse than trying to be creative when you're trying to remember syntax, mistyping special characters, or fighting with compiler errors.

```
"friendsAlongTheWay": {  
  conditions: "theme_processOverProduct && tag_bonding && !  
  tag_threat"
```

```

tag_threat",
cast: "/tags_bonding/anyone/",
name: "_name/a/ realizes life's not about endings.",
text: "\"What about you?\" _name/b/ said, shoving a
      crumpled sweater into _their/b/ duffel bag. _ifTheme/
      fantastical/\"After everything that's happened here,
      what's/\"What's/ next for the great _name/a/~, famous
      polar explorer?\"<br><br>_They/a/ smiled, distracted b
      _ifPersonalSymbol/a/itemNoun/the&nbsp;/(&nbsp;in
      _their/a/ hands.)/(the book about Carina Station _they
      a/~'d brought with _them~/) _ifTheme/selfRealization/
      Then, realizing something profound,/After a moment's
      consideration,/ _they/a/ {put it in the suitcase, too|
      decided to leave it behind}.<br><br>\"{I'm not sure.
      Maybe|Definitely} {another adventure. Somewhere
      tropical, I think|something more normal. A nice desk
      job, maybe|some follow-up work. There are still
      questions to be answered}.\"<br><br>_name/b/ laughed.
      _timeGate/\"You said it/1950/\"You got that right
      /1980/\"Damn straight/~,\" _they/b/ said. _ifTheme/
      horror/\"Somewhere far away from this awful place, tha
      's for sure. Well,/\"Well/ whatever you do, stay in
      touch, okay? _timeGate/Postcards are cheap/1997/E-mail
      s free/2007/I've got unlimited texts, you know/~.\"<br>
      <br>_ifTheme/goItAlone/\"Maybe in a couple years,\"/\"
      I will,\"/ _name/a/ said, _gender/a/(_gender/b/punchin
      his arm fondly/rolling his eyes fondly/)(_gender/b/
      putting a hand on his shoulder fondly/putting a hand o
      her shoulder fondly/)/~. \"{You won't escape from me
      that easily|You can count on that|I will|I won't forge
      you|The world hasn't seen the last of us}.\"",
themes: ["processOverProduct", "selfRealization"]
}

```

A particularly egregious example of dynamic text authoring for The Ice-Bound Concordance, involving escaped quote marks, spacing codes and hacks, variable character genders, nested conditionals, and two different systems for variable text. This was the kind of thing I was hoping to avoid this time around.

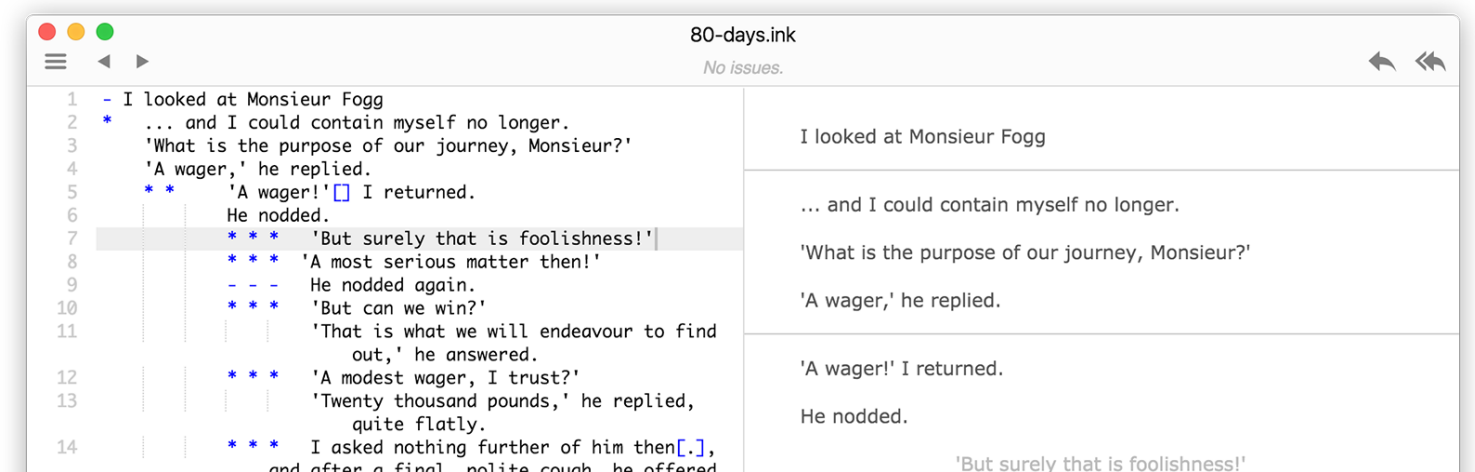
By contrast, I wanted .quant writing to be as simple to type as possible. I also wanted it to be as close to idiot-proof as possible, for a very important reason: since the *Subcutanean* generator automatically exports a new print-ready PDF from the master file each time a copy of the book is ordered, which is then uploaded to a print-on-demand service,

turned into a physical book, and shipped off — all without manual human intervention — any errors would be much more galling. Normally the worst-case scenario for some broken procedural text is that a player sees a mistake in a momentary message. Here, that error would be preserved forever in the pages of the book: or in the worst case, cause its entire text to be corrupted, resulting in an unhappy buyer and an expensive replacement.

These two requirements — a minimalist syntax and trying to reduce as much as possible the chance of error — led to some early initial constraints. First, I decided there would be no routines or GOTOs in the format. This was an easier decision than it would have been for a game, because *Subcutaneous* has no branching or interaction: the plot proceeds chapter to chapter in the same way for each reader, varying not in the overall structure but in the way individual scenes play out, in which particular details are revealed or omitted. I also didn't need any reusable pieces of text that might have to show up in different contexts or situations, as one would generally need for an interactive work, so that also made this simplification more possible.

Second, I realized I needed two major kinds of variation: simple alternatives that didn't need to be reasoned over or remembered, and choices that would impact text in multiple places. The latter implies variables, which can be set and later checked, so I needed to account for that.

I considered a number of existing solutions for procedural text authoring, but in part because my use case is so particular, none of them quite met my needs. Languages like Ink designed for games with explicit choice points weren't really appropriate, as these are essentially centered entirely around GOTOs as a paradigm.



```

15      * *      'Ah[.'],' I replied, uncertain what I thought.
16      - -      After that,
17      *      ... but I said nothing and
18      - we passed the day in silence.
19      - -> END

```

'A most serious matter then!'

Ink (running in IDE Inky) is a nice minimalist interactive text language, but designed for branching stories with explicit choices. (Screenshot courtesy Inkle.)

Tracery is a popular language for procedural text, but is optimized for writing long chains of nested expansions. This means most of what you're doing is defining keys meant to be expanded elsewhere, which was more heavyweight than I needed: a major desiderata was being able to read through the dynamic text along with the static text, composing and editing both together within the same flow.

```

1  var tracery = require("tracery-grammar");
2
3  var symbols = {
4    "color": ["orange","blue","white","black","grey","purple",
5             "indigo","turquoise"]
6    , "animal": ["unicorn","raven","sparrow","scorpion","coyote",
7                "eagle","owl","lizard","zebra","duck","kitten"]
8    , "natureNoun": ["ocean","mountain","forest","cloud","river",
9                    "tree","sky","sea","desert"]
10   , "name": ["Arjun","Yuuma","Darcy","Mia","Chiaki","Izzi",
11              "Azra","Lina"]
12 }
13
14 var grammar = tracery.createGrammar(symbols);
15
16 var textExpansion = grammar.flatten("The #color# #animal# of
17 the #natureNoun# is called #name#");
18
19 console.log(textExpansion);
20
21

```

Using Tracery by Kate Compton in Javascript; example courtesy Dan Cox.

I'm also very familiar with Inform 7's way of handling variant texts, and thought about basing my compiler around its syntax. But it's also a bit heavyweight for my use case, in

part because of its natural language paradigm, and in part because of the more powerful control it offers over different things to do with textual variants.

"You dip into the chapter on [one of]freshwater fish[or]hairless mammals[or]extinct birds[or]amphibians such as the black salamander[in random order]."

Text substitutions in Inform 7.

Other tools were also unsuitable for various reasons, such as requiring IDEs rather than support for text files. I did in fact find things quite similar to what I was looking for, such as the Javascript library Bracery (which starts off with a very similar syntax to the one I ended up using, before getting more complex). But ultimately I decided to roll my own Python tool that would work with the rest of the tech stack I needed to make this project happen.

In the .quant format as it ended up, the most common use case is the simple inline variant. These are indicated like this:

There are [two|three|four] lights!

Square brackets were chosen (as were all control symbols) for the unlikelihood that they'll appear in regular prose. They save a shift keystroke compared to curly braces. Pipes are better than slashes (which do sometimes appear in prose) and in most fonts stand out a bit above and below the line, making them more visually obvious. Note that I also made a syntax highlighter for Sublime Text, seen in these screenshots, as the first line of defense against obvious syntax errors like forgetting a bracket.

A single bracketed text will either be printed or not, at random: this is the same as [text|] but slightly more elegant. For instance, the below might result in "...I almost forgave him" or "...I forgave him."

In the end, I [almost] forgave him.

(Technically, written this way the null option above would have two spaces between *I* and *forgave*. Because I knew my output was LaTeX code which ignores extra whitespace, I knew I could likewise ignore this issue: the same would also have been true if my output was HTML. In other contexts (like Inform 7, for instance) one generally needs to spend more time getting the exact position of the brackets right because spacing is preserved in the output. I did still have to worry about punctuation joins and so on — in a later post I'll talk about a separate tool I built to help catch those errors.)

Sometimes you want certain alternatives to appear more or less often. I thought about whether I needed this for *Subcutanean* — it felt conceptually purer in some ways to keep the selection entirely random — but I decided I did want to allow for the possibility of some texts that were rare or even very rare, appearing only in one or two books out of a hundred. It would be easy to get bogged down with possibilities here: Inform, for instance, supports various kinds of randomness like “with decreasingly likely outcomes” that makes each option less likely be selected than the one before it. But I decided I wanted to deploy this in specific places with tight control over distributions, so ultimately I just went with a simple method of directly specifying numeric probabilities for each option, in the situations where I needed to do so.

```
"Hey, [50>Charles|45>Chuck|5>C-Dog], how's  
it going?" I said.
```

Not actually a line from *Subcutanean*.

The numbers must always sum to 100, except that the final number can be omitted to assume the remaining distribution space: in this example leaving off the 5> would still have the effect of a 5% chance of choosing “C-Dog”. The parser can then complain if the numbers don’t add up. This exactness gets annoying for very long lists of variants but I didn’t anticipate needing very many of those. In fact the current draft has only one instance of a very long list that specifies probabilities.

*

The last major part of the syntax was to control random selections that would affect multiple pieces of text: setting variables.

```
[DEFINE BreakupSubplot]
```

```
[DEFINE verbose|taciturn]
```

These are explicitly defined so the compiler can catch typos or mismatches when they're used, and they're not case sensitive because case sensitivity is dumb. For my particular use case I didn't need anything more complex than booleans or enums, which the two examples above demonstrate: for each rendering, *BreakupSubplot* will be randomly true or false, and either *verbose* or *taciturn* will be true; if there were more options here, only one of them would be true on any given run. (You can also assign probabilities to variable assignments: `[DEFINE 25>verbose].`)

Text can then be gated based on whether a variable is true by starting with the variable reference:

```
I was feeling [verbose>rather ebullient and  
in tip-top shape|fine] that day.
```

To reduce the possibility of a word to be printed getting confused for a variable reference (by either the parser or the writer), a variable is allowed to appear in exactly two places: immediately after a **DEFINE**, or immediately before a **>**. The distinct separator character provides precise control over leading spacing (compared to a potentially ambiguous syntax, something like `[@verbose rather ebullient...]`) and makes it easier to catch any spurious uses: **>** it turn must be preceded by a number or a recognized variable.

I thought long and hard about whether I should add conditional logic to this syntax, for instance to have text only printed when both *BreakupSubplot* and *verbose* are true. I

finally decided not to allow this, in part because of the much greater likelihood of introducing authoring errors this way, but mostly out of a sense of aesthetic purity. After many of my past projects with exceedingly complicated procedural text, I thought it would be a nice exercise to keep all the randomness at a single hierarchical level. No spending time writing complex nested prose that might only be seen by a tiny percentage of readers; no compounded branches multiplying the amount of possibility states I needed to cover for a single sentence. If I wrote five variants (hand-selected usages of probability aside), there'd be a one in five chance that any of those pieces of text would be seen. Simple and clean.

I felt very smug about this until I immediately hit on a situation that required compound decisions after all, and had to go back to the drawing board. More about that in my next post.

Subcutaneous is an upcoming horror novel that changes with each new copy. Find out how to pre-order it now, or follow the project on Twitter, Facebook, or Goodreads.

