A SHORT INTRODUCTION TO STEP

PART ONE: GRAMMARS

WHAT IS STEP?

Step is a programming language tailored for generating text. At its most basic, it lets you specify text templates with blanks to fill in, then fills them based on the options and procedures you specify.

Step is also what AI researchers call a "planner", meaning that it can in some sense decide what to do now by thinking ahead to what each of its available choices would lead to. We'll talk more about that later.

GENERATING TEXT

The simplest step program is:

```
        Hello: Hello world!
```

The portion in boldface tells us that we're defining a little program called "Hello" and when we run it, it should print the text, "Hello world!" If we put this in a file, load the file into Step, and then type "Hello" at Step to tell it to run Hello, it will respond with:

```
        Hello world!
```

ALTERNATIVES AND VARIATION

We can define several different ways of saying hello:

```
        Hello: Hello world!
        Hello: Hola!
        Hello: Hi there!
        Hello: Bon jour!
```

Each of these lines specifies a different way of saying hello. However, if you put this in your file, Step, will still always use the first line it finds that "works" (more will be said of working and failing later). To make it choose randomly, we add the annotation "`[randomly]`" at the beginning:

```
        [randomly]
        Hello: Hello world!
        Hello: Hola!
        Hello: Hi there!
        Hello: Bon jour!
```

FOR PROGRAMMERS

If you haven't programmed before, you should ignore these side-bars.

Step (Simple TExt Planner) is a domain-specific language for text generation that uses a lot of ideas from the languages PLANNER and PROLOG from the early 1970s.

Methods for Step tasks **don't have to work all the time**. You can specify many different possible methods to try. Step will systematically try them until it finds one that works.

Each of those methods might call other tasks with other fallible methods that call other tasks, and so on.

Step automatically searches the tree of possible method choices. When the program completes, it's as if Step has magically guessed all the right methods to use.

This feature is sometimes called **non-determinism**. Step is a non-deterministic language.

This tells Step to choose randomly. Each time you run **Hello**, it will make a new random choice of which line to print.

TASKS AND METHODS

We've implicitly introduced a few different ideas here. One is the idea you can write "little programs" like Hello. The official term for these is *tasks*. A task is something you can ask the system to *do*. Tasks are named. The task we've been talking about is called `Hello`. Task names have to start with a capital letter.

We've also seen that you can specify how to do a task by giving the name of the task, a colon, and then some text to print. But you can specify many different ways to carry out a task. Each of these lines that explain how to perform a task is called a *method*. So a method is a line that looks like:

TaskName: *stuff to print*

However, it's too limiting to require a method to fit on one line, so you can split it across many lines by ending the line at the colon and then putting in as many lines of text as you like, ending with the magic keyword, `[end]`. So we can have a long method for saying Hello by saying:

```
Hello:
Uh, yes.  Hello.  My name is, uh, Roylance.  Uh, Richard.  Richard Roylance.

Yes.

So, um, I'm here for the, um…

Yes, that's right.  The public speaking course.
[end]
```

The last thing we've seen so far are little annotations that can be attached to methods to change how they behave. In particular, we've seen [randomly], which states that the methods for the task being defined should be tried in random order. We'll see a bunch of other annotations shortly.

TASKS AS TEMPLATES

Suppose we had a task like this:

```
[randomly]
Sentence: the cat ate the dog
Sentence: the cat chased the dog
Sentence: the cat plotted world domination with the dog
```

Notice that all these methods have the same text, with just the verb changed. That means that every time we want to add a new sentence, we have to retype the cat and dog parts. Rather than doing that, we can define the verb part as its own task:

```
[randomly]
Verb: ate
Verb: chased
Verb: plotted world domination with
```

Then we can incorporate the Verb task inside the Sentence task, by enclosing it in brackets:

```
Sentence: the cat [Verb] the dog
```

Now we only need one method for the sentence. Moreover, we can be egalitarian and allow the dog to be the actor sometimes and not just the cat:

```
[randomly]
Sentence: the cat [Verb] the dog
Sentence: the dog [Verb] the cat
```

GRAMMARS

Another way to write this is to say:

```
Sentence: [Noun] [Verb] [Noun]

[randomly]
Noun: the cat
Noun: the dog

[randomly]
Verb: ate
Verb: chased
Verb: plotted world domination with
```

In linguistics and computer science, this structure of templates inside templates inside other templates is called a *grammar*. The grammars we've written so far where there are no restrictions on how choices are made between different templates are called context-free grammars. Context-free grammars are popular ways of generating text.

One issue with context-free grammars is that they often generate odd-sounding sentences. For example, this one generates "the cat ate the cat", which leave the reader wondering if the cat ate itself or another cat. In either scenario, a fluent English speaker wouldn't use the phrase "the cat ate the cat" to describe it. Issues like this are one of the reasons it's difficult to generate good text. We'll talk about ways of dealing with some of these issues later.

ADAPTING TEXT

The Situationist International's 1960 manifesto begins:

> The existing framework cannot subdue the new human force that is increasing day by day alongside the irresistible development of technology and the dissatisfaction of its possible uses in our senseless social life.

We can appropriate this text and use it to make a manifesto generator more adapted to the present day by "bracketing" parts of it so that they are generated randomly. For example:

```
Manifesto:
The existing framework cannot subdue the new human force that is increasing
day by day alongside the irresistible development of [Technology] and the
```

```
dissatisfaction of its possible uses in [Our] [Senseless] [Social] life.
[end]

[randomly]
Our: our
Our: your
Our: my

[randomly]
Technology: Facebook
Technology: Twitter
Technology: Zoom
Technology: email

[randomly]
Senseless: senseless
Senseless: worthless
Senseless: hopeless
Senseless: lame

[randomly]
Social: social
Social: work
Social: love
Social: school
```

This version can then generate dumb little tweets such as:

> The existing framework cannot subdue the new human force that is increasing day by day alongside the irresistible development of **Zoom** and the dissatisfaction of its possible uses in **my lame social** life.

APPENDIX: HOW TO RUN CODE IN STEP

All of the aforementioned is about how to write code, but we haven't said much about the physical mechanics of running a step program. Here are the basics.

WHERE TO PUT YOUR CODE

Step will look for your code in the `Step` directory of your `Documents` folder. So make a directory called `Step` inside `Documents`.

Step also lets you work on several projects at once, with the code in different subfolders of Documents/Step. So make a new subfolder of `Documents/Step` to hold your code. For example, `Documents/Step/FirstProject`.

Now make a file inside `Documents/Step/FirstProject` called "`first step.step`" (or anything ending in `.step`) and put the following in it:

```
Hello: Hello world.
```

Make sure you've saved the file.

Now start Step and type the following in the green box and hit return:

```
project FirstProject
```

It should print out that it's loaded FirstProject.

RUNNING A COMMAND

Now type:

```
Hello
```

in the green box and hit return. Hopefully, it will print "Hello World."

REVISING YOUR CODE (EDITING AND RELOADING)

Now change the file so it says something like:

```
Hello: this is some different text.
```

Save the file, go back to the Step window and type Control-R (for reload). It should say something like "project reloaded". Now run Hello again. You should see the revised text.

Those are the basic mechanics of entering and running code. Now try making a grammar to generate silly phrases!