

Project: Text Generation (Part Two)

Summary: We continue our first multi-day project by expanding our Scheme-based techniques for generating English text.

Contents:

- Introduction
- A Partial Solution: Structural Expressions
- Storing Structures in Files
- Detour: Refactoring
- Unifying Structures and Parts of Speech

Introduction

In the first reading and laboratory on text generation, you explored some basic strategies for generating text. In that work, you worked under a model in which we separated structural choice from word choice. In particular, you relied on the `generate-part-of-speech` procedure to generate a “random” word, but you relied on hand-coded selection process to choose a “random” sentence structure.

Since we’ve reviewed the structure of `generate-part-of-speech` in the first reading, let’s look for a bit at a typical sentence procedure.

```
(define sentence
  (lambda ()
    (let ((choice (random 100)))
      (cond
        ; 30% of the time, we use a simple subject/verb sentence.
        ((< choice 30)
         (string-append (capitalize (noun-phrase))
                        space
                        (generate-part-of-speech 'iverb)
                        period))
        ; 65% of the time, we use a subject/verb/object sentence.
        ((< choice 95)
         (string-append (capitalize (noun-phrase))
                        space
                        (generate-part-of-speech 'tverb)
                        space
                        (noun-phrase)
                        period))
        ; The remaining 5% of the time, we use an exclamation!
        (else (string-append (capitalize (generate-part-of-speech 'exclamation))
                              exclamation-point))))))
```

Is this a problem? It’s a bit of a problem. We designed the words files so that a non-expert could modify them and get more interesting sentences, but it is not possible to add new structures without some programming expertise. The code above is also fairly complex. For example, you need to make sure that

the choice of numbers for percentages include the previous percentages.

A Partial Solution: Structural Expressions

One thing we can do to simplify this code is to rely on a strategy we've used previously: We can write list expressions that represent the structure of the string we want and then use our own evaluation procedure to turn those lists into strings. For example, we might represent a very simple sentence with the following structure:

```
'(join "The" space noun space intransitive-verb period)
```

More generally, we might represent the first sentence from the reading as

```
'(join (capitalize noun-phrase) space transitive-verb noun-phase period)
```

How do we evaluate these structures? You may recall that we used a two-step process: First we evaluate all of the parameters to a procedure (e.g., the `noun-phrase` that is a parameter to `capitalize`) and then we apply the procedure to those parameters.

```
;;; Procedure:
;;; build
;;; Parameters:
;;; structure, the description of a text structure
;;; Purpose:
;;; Convert the structure into a "random" string.
;;; Produces:
;;; text, the constructed string
;;; Preconditions:
;;; structure is either:
;;; (1) a string;
;;; (2) a symbol; or
;;; (3) a list of structures.
;;; This form is not verified.
(define build
  (letrec ((build-all
            (lambda (structures)
              (if (null? structures)
                  null
                  (cons (build (car structures))
                        (build-all (cdr structures)))))))
    (lambda (structure)
      (cond
       ; Strings we accept verbatim
       ((string? structure) structure)
       ; Symbols we translate accordingly
       ((symbol? structure) (build-from-symbol structure))
       ; Lists require us to build the arguments recursively and then
       ; apply the procedure
       (else (apply-procedure (car structure)
                               (build-all (cdr structure)))))))
```

Note that we've added a new trick. In the evaluation strategy we used previously, we only accepted strings and lists. We now accept symbols, too, but then rely on a helper, `build-from-symbol` to do something with those symbols.

```
(define build-from-symbol
  (lambda (symbol)
    (cond
      ((eq? symbol 'adjective) (generate-part-of-speech 'adjective))
      ((eq? symbol 'article) (generate-part-of-speech 'article))
      ((eq? symbol 'exclamation) (generate-part-of-speech 'exclamation))
      ((eq? symbol 'intransitive-verb) (generate-part-of-speech 'iverb))
      ((eq? symbol 'name) (generate-part-of-speech 'name))
      ((eq? symbol 'noun) (generate-part-of-speech 'noun))
      ((eq? symbol 'transitive-verb) (generate-part-of-speech 'tverb))
      ((eq? symbol 'period) ".")
      ((eq? symbol 'space) " ")
      (else (symbol->string symbol))))))
```

We also need to write `apply-procedure`. It uses a similar technique to `build-from-symbol`. That is, it also checks the symbol (this time, the name of the procedure) and makes a decision as to what to do depending on the name of the procedure. It differs in that it potentially uses the parameters in doing whatever it is.

We'll start by supporting two basic procedures, `join`, which joins all the strings in a list, and `capitalize`, which capitalizes a string. (Note that since the second parameter of `apply-procedure` is a list, we'll need to grab the first element of the list.)

```
(define apply-procedure
  (lambda (proc params)
    (cond
      ((eq? proc 'join) (join params))
      ((eq? proc 'capitalize) (capitalize (car params)))
      (else (symbol->string proc))))))
```

You should have written `capitalize` in the first lab. If you did not, here's one version.

```
(define capitalize
  (lambda (str)
    (string-append (string (char-upcase (string-ref str 0)))
                   (substring str 1 (string-length str)))))
```

The `join` procedure simply recurses over the parameters, joining them together with `string-append`.

```
(define join
  (lambda (strings)
    (if (null? strings)
        ""
        (string-append (car strings) (join (cdr strings))))))
```

Now, we can build variants of the prototypical noun phrase using `build`,

```

> (build '(join article space adjective space noun))
"a great dog"
> (build '(join article space adjective space noun))
"a blue cat"
> (build '(join article space adjective space noun))
"a great dog"
> (build '(join article space adjective space noun))
"a enormous dog"
> (build '(join article space adjective space noun))
"the great cat"
> (build '(join article space adjective space noun))
"a blue llama"

```

That strategy can at least make the definition of noun-phrase a bit clearer.

```

(define noun-phrase
  (lambda ()
    (let ((choice (random 100)))
      (cond
        ; 50% of the time, we use the article adjective noun structure
        ((< choice 50)
         (build '(join article space adjective space noun)))
        ; 25% of the time, we use the article noun structure
        ((< choice 75)
         (build '(join article space noun)))
        ; 15% of the time, we use a name
        ((< choice 90)
         (build 'name))
        ; 10% of the time, use a possessive
        (else
         (build '(join name "'s" space noun)))))))

```

Storing Structures in Files

While this is an improvement, we still have a problem. In particular, we still have the complex conditional code (in both noun-phrase and sentence), and that code is difficult to generate and check.

How do we eliminate the complex conditionals? You may recall that we've already developed a technique for choosing randomly between different things with different probabilities. In particular, the random-word procedure uses the structure of the file to choose randomly. We could write something similar that chooses between structures. For example, we might create a file of the following form for noun phrases.

```

((join article space adjective space noun) 500)
((join article space noun) 250)
(name 150)
((join name "'s " noun) 99)
("an infrequently-occurring noun phrase" 1)

```

We then write a random-structure procedure that looks almost exactly like random-word, except that it processes structures, rather than words.

```
(define random-structure
  (lambda (fname)
    (let ((port (open-input-file fname))
          (rnd (random 1000)))
      (letrec ((kernel (lambda (num)
                        (let ((entry (read port)))
                          (if (< num (cadr entry))
                              (begin (close-input-port port)
                                      (build (car entry)))
                              (kernel (- num (cadr entry)))))))
              (kernel rnd))))))
```

Now, we can rewrite noun-phrase even more simply.

```
(define noun-phrase
  (lambda ()
    (random-structure (string-append root "noun-phrases"))))
```

Detour: Refactoring

You'll note that our code now has two very similar procedures, `random-word` and `random-structure`. Experienced programmers balk at seeing such similarities and strive to find a single procedure that accomplishes both tasks. Why? If we have two procedures, rather than one, then whenever we make a change in one, we'll probably need to make it in the other, and experience suggests that we'll sometimes forget. In addition, if we've found that a similar process works for two situations, it may work for more. We make it easier to support other situations by writing a more general common procedure.

The rewriting of duplicated code into a single procedure is one version of a process often called *refactoring*. We recommend that you get in a habit of looking for duplicated code and identifying ways to factor out the duplication.

In this case, we're lucky. Since every string is a structure, `random-structure` works just as well as `random-word` on a file that contains only string entries.

Hence, we can replace the call to `random-word` in `generate-part-of-speech` with a call to `random-structure`, and then remove `random-word` from our program.

Unifying Structures and Parts of Speech

Are we done yet? Not yet, but we're close. Unfortunately, we can't yet use the same strategy for simplifying sentence that we used for `noun-phrase` since `sentence` has a call to `noun-phrase`.

What do we do? We don't have to make much of a change. We simply add a line to `build-from-symbol`. For example, we might add the line

```
((eq? symbol 'noun-phrase) (noun-phrase))
```

However, since the body of `noun-phrase` is simply a call to `random-structure`, we might simply insert that call (and eliminate the `noun-phrase` procedure).

```
((eq? symbol 'noun-phrase)
 (random-structure (string-append root "noun-phrases")))
```

But the call to `(string-append root "noun-phrases")` is remarkably similar to the body of `generate-part-of-speech`. Hence, we can add `noun-phrase` to the valid parts of speech so that `generate-part-of-speech` will process it correctly. The line we add to `build-from-symbol` now closely resembles the other lines in that procedure.

```
((eq? symbol 'noun-phrase) (generate-part-of-speech 'noun-phrase))
```

After some updates to the related code, we can now write things like

```
> (build '(join (capitalize noun-phrase) space iverb))
```

Among other things, that means that we can now move the choice of sentence structure to a file, thereby eliminating the complex conditionals in that procedure. We'll work through the details in the lab.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.