Fundamentals of Computer Science I (CS151.01 2006F)

Project: Text Generation (Part One)

Summary: We begin our first multi-day project by exploring some Scheme-based techniques for generating English text.

Contents:

Introduction

One of the first great goals of Artificial Intelligence was to make computers understand natural language (e.g., English). Unfortunately, while great progress has been made in simulating understanding (e.g., by parroting words back at the writer or speaker), less progress has been made in actual understanding. Recall, for example, the famous example of what techniques are needed to distinguish how the same words are used differently in the phrases *time flies like an arrow* and *fruit flies like a banana*.

On the other hand, it can be relatively straightforward to produce correct English sentences. We simply choose appropriate sentence structures and then fill in the parts of speech in those structures. You have explored a more restricted version of this strategy in the Mad-Libs game (and discovered, of course, that it's not so easy as we suggest here).

In this project, we will explore some simple, Scheme-based, techniques for generating English sentences.

Design Decisions

We will begin approaching the problem using two techniques common to Scheme programming: (1) We will represent each part of the text we are generating as a procedure and (2) We will use a file of Scheme values to store our data.

In particular, we will write procedures for the main structures of our writings. These procedures will include `sentence` and `noun-phrase`. (You may develop others as time goes on.)

For the more basic parts of speech (nouns, adjectives, verbs, articles, etc.), we will use a common procedure, `generate-part-of-speech`, that takes a symbol representing the part of speech as a parameter and returns a difficult-to-predict word that fits that part of speech. Why do we not write a separate procedure for each of these basic parts of speech? Because their implementations are so similar.

To support the easy generation of these "random" words, we will create a file for each part of speech listing words that are classified as that part of speech. For each word, we will store a list in the file of the form

`(word frequency other-info)`

Initially, we care only about the word, which is a string, and the frequency, which is an integer. The *other-info* is left as an optional part of the list for use in more advanced applications. For example, we might represent the number of syllables or the ending sound.

Each frequency indicates the number of times the word should appear, on average, in a list of 1000 words that are classified as that part of speech. For example, if the only adjectives we use are `red`, `black`, and `colorless` and `red` appears 60% of the time, `black` 30% of the time, and `colorless` 10% of the time, our file might resemble the following.

```
("red" 600 1)
("black" 300 1)
("colorless" 100 3)
```

Structural Procedures

As you should recall from the previous strategy on section, we rely on procedures to provide the structural information. For example, if we decide that we will make all sentences take the form *noun-phrase transitive-verb noun-phrase*, we might write the following definition:

```
(define sentence
  (lambda ()
    (string-append (noun-phrase)
                   space
                   (generate-part-of-speech 'tverb)
                   space
                   (noun-phrase)
                   period)))
```

The `noun-phrase` procedure is defined similarly.

```
(define noun-phrase
  (lambda ()
    (string-append
     (generate-part-of-speech 'article)
     space
     (generate-part-of-speech 'adjective)
     space
     (generate-part-of-speech 'noun))))
```

The observant reader may have noted a few potential issues with these definitions. For example, no provision is made for capitalizing the initial letter in sentences and only one structure of sentence is available. You will correct both of these deficiencies (and others) in the corresponding laboratory.

Generating Parts of Speech

The structural procedures above give the general shape of the sentence. However, the require another procedure, which we call `generate-part-of-speech` to obtain the actual words to fill in the structure. You can think of `generate-part-of-speech` as acting like a player in a strange game of Mad Libs®. We will make the `generate-part-of-speech` a type of husk: Its roles wll be to (1)

make sure that the part of speech is valid, (2) translate the part of speech to a file name, and (3) call the kernel (which we call `random-word`) using that file name.

```
(define generate-part-of-speech
  (lambda (part)
    (if (member part (list 'adjective 'article 'noun 'tverb))
        (random-word (part->filename pos))
        (string-append "ERROR[" (value->string part) "]"))))
```

The real work goes on in `random-word`. As you might expect, `random-word` begins by opening the file and generating a random value between 0 and 999.

```
(define random-word
  (lambda (fname)
    (let ((port (open-input-file fname))
          (rnd (random 1000)))
```

We will then recurse through the file, checking whether the random number we've generated is less than the frequency associated with the word. If so, we choose the word. Recall that the frequency is the second element of the entry and the word is the first.

```
(let ((entry (read port)))
  (if (< wordnum (cadr entry))
      (begin (close-input-port port) (car entry))
      ...))
```

If not, we need to move on to the next word, which we do with a recursive call. There is a subtlety to that recursive call - we need to update something to ensure that a word is eventually chosen in a recursive. For example, suppose we have five words, each appearing 200 times in 1000 words. Suppose also that our random number is 732 (the number of times that Sam has written a `length` procedure, for those of you paying close attention). We don't want to choose the first word because 732 is greater than or equal to 200. However, if we simply recurse onto the next line of the file without doing anything to the 732, we'll make little progress. (In this example, we'll compare 732 to the next 200, then to the next, then to the next, and then we'll run out of words.) So, what should we do? We could change the file format, but that's inconvenient. Instead, we'll subtract the words seen from the random number. Hence, we'll compare 732 to the 200 for the first word, then 532 to the 200 for the second word, then 332 to the 200 for the third word, then 132 to the 200 for the last word. In that case, we finally have a number less than the number of appearances of that word, so we return that word.

We fill in that update and the first call to `kernel` and we're done. Putting it all together, we get the following.

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