solfish: Generating Text from an Existing Grammar

Bob DuCharme ID#041-56-0107 May 10, 1996
Final Project for Natural Language Processing G22.2590 Professor Grishman

"Truth rests on the mathematical reeds of the infinite and everything moves forward by order of the eagle riding pillion, while the genius of vegetable flotillas claps and the oracle is pronounced by fluid electric fish."

– *Andre Breton, "Soluble Fish," tr. Richard Seaver and Helen R. Lane*

"Her hazy ghosts only eat with fish. Ghosts fish often with moons. Fish mean to think. Zombie eggs generally sleep."

– *solfish*

Purpose

`solfish` is a Common Lisp program that generates sentences by randomly picking vocabulary and grammar productions from an existing dictionary and a grammar implementing several grammatical production restrictions. It has two purposes: first, to call attention to weaknesses in a grammar by generating sentences considered correct by that grammar but which alert a human proofreader as improperly constructed. Secondly, and less important (although a boon to the patience of the aforementioned proofreader) the whimsical nature of the generated sentences can be entertaining, recalling William Chamberlain's 1983 program Racter[1] and `solfish`'s namesake, surrealism founder Andre Breton's 1924 short story "Soluble Fish."

While the entertainment value is enhanced with the addition of appropriately poetic vocabulary (e.g. "dream," "moons," "ghost," "forgotten," etc.) it is fortuitous that the word "fish," a classic image in surrealist literature, was already one of the twelve nouns in the original `rl-dictionary` used by `solfish`.[2]

Writing a program from scratch that generates grammatically correct sentences is not difficult. The point of this exercise was not purely to generate sentences, as with Racter, but as a tool to focus the refinement of existing grammars. Originally, I had thought it would be most useful in grammars generated using statistical analysis of large bodies of text, but as we'll see, it found several errors in a hand-crafted grammar as well. While I had planned to add new restrictions to

[1] William Chamberlain, *The Policeman's Beard is Half Constructed: Computer prose and poetry by Racter* (New York: Warner Books, 1984).

[2] On the long-running television show "Cheers," the overly-intellectual waitress Dianne once told the joke, "How many surrealists does it take to change a lightbulb? A fish!" The bar's patrons, unfamiliar with Breton's work, did not see the humor.

extend its capabilities beyond those of the simple grammar that I began with, some of the errors it found led me to realize that machine-generated sentences can require much further refinement of even a simple grammar than I had originally expected.

Using It

`solfish.lisp` was written and tested using Allegro Common Lisp 4.2 running on a Sun workstation running SunOS Release 4.1.4. Loading it into the Lisp interpreter first loads the NYU natural language processing class Lisp parser based on the restriction language grammar described in Grishman[3]. Next, it executes the nlp-load-rl function that loads the default `rl-grammar` and `rl-dictionary` files, and then copies the dictionary entries into a new hash table keyed on word categories (noun, verb, transitive verb, etc.) as opposed to the vocabulary word strings so that random selection of words based on their category is possible.

The `(expand lhs)` function returns a tree structure (as defined by the `defstruct` function `in cf-parsers.lisp`) based on a randomly-chosen right-hand side production from the loaded grammar for the `lhs` parameter passed to the function. Any members of the chosen production are also expanded into trees, making a complete tree to return. Typically, this is called with the command

```
(expand 'sentence)
```

although it was called with many other grammar production `lhs` values in the development and testing process.

The `(words tree)` function takes one of these tree structures and writes it out as a string of words delimited by spaces. In theory, anything returned by `(words (expand 'sentence))` should be accepted and parsed by the `(analyze-sentence)` function in `cf-parsers.lisp`.

The `(print-sentence)` function, which takes no arguments, calls `(words (expand 'sentence))` and another function that formats the output a bit more nicely.

The `print-tree` function, which prints a tree as a list whose nested structure shows the tree's structure, often comes in handy when generating sentences with `solfish`. When `print-sentence` outputs something odd-looking `print-tree` is usually the first thing called (with an argument of *tree*) in order to see how the sentence's structure was composed from the given grammar.

Typical use of `solfish` involves the repeated calling of `(print-sentence)` as we look for interesting sentences. The (100-sentences) function, which also takes no arguments, creates these sentences in bulk by writing 100 of them to the text file `solfish.out`. In addition to speeding the proofreading throughput, this function also allows a rough benchmark of the program's efficiency.

[3] Ralph Grishman, *Computational Linguistics* (Cambridge: Cambridge University Press, 1986) pp. 57-63.

How it Works

Before beginning this program, the extent of my Lisp programming experience was a cursory exposure to Scheme in a programming languages survey course, so I really learned it as I went alone. This accounts for some rather tortured constructs found in the code.

When the expansion of a phrase label has a choice of productions to make, giving each production an equal chance of selection isn't necessarily the best way to proceed. To counteract this to some extent, I added code to the `(pickrhs)` function to increase the chance of `null` being picked by 75% when it was one of the choices. I arrived at this figure by trial and error.

Each time a given phrase is created, any listed restrictions are run on it. If it fails any, it is thrown out and a new phrase is generated for the same phrase label. With the class `rl-grammar`, this discarding and recreation did not create a significant computational expense; 10 executions of the `solfish` `100-sentences` function, measured with the Common Lisp `time` function, show an average of 98.5 milliseconds of total system CPU time per `100-sentences` execution and an average of 37,626 milliseconds of elapsed real time.

This average wait by the user of 38 seconds for the generation of 100 sentences shows that the computational expense of this implementation of restrictions, which is the biggest potential bottleneck in execution time, is not a significant factor. (This was run with the rl-grammar restrictions `agree-restr`, `subcat-restr`, and `count-restr`; additional restrictions added to the `sf-grammar` augmentation of `rl-grammar` caused problems described below that prevented `100-sentences` from completing.)

Weaknesses Found in rl-dictionary

While the main purpose of `solfish` is to help locate problems with a defined grammar, we can attribute certain sentence problems to the specification of certain dictionary entries. For example, the sentence "zombie fish eats" (which used an augmented version of `rl-dictionary` described below, showing just how much more interesting some well-chosen vocabulary can make these sentences) shows that the `rl-dictionary` entry for "fish" as a noun in

```
(word "fish"  ((cat n))
              ((cat tv) (number plural  (objlist (null)))
              ((cat v) (objlist (null))))
```

should specify not only that it is a noun, but also that it is a countable singular noun (to reject "zombie fish eats" but accept "his zombie fish eats") or that it is a plural noun, so that the original sentence would be rejected but "zombie fish eat" would be accepted. A direct object would also be interesting, to see exactly what the zombie fish find appetizing.

Because the correction of such problems requires so little effort, and because the use of pre-existing dictionaries to test evolving grammars is more common than the reverse situation, I shall limit the remaining discussion to grammar problems revealed by `solfish` output. All cited sentences, despite their grammatical problems, are parsed by `cf-parser`'s top-down parser as acceptable.

`solfish` includes `sf-dictionary`, a copy of `rl-dictionary` augmented to expand the subject matter of the `solfish` output and reduce the repetitive nature of the sentences' topics. It also incorporates changes like the one described above for "fish" and other refinements of `rl-dictionary` definitions.

To test the problem described above, note that it also parses "happy fish eats" with no problem. "zombie fish eats" will not work because the adjective "zombie" is in `sf-dictionary` with the described corrections for the `ncount` and `number` parameters, and it is not in `rl-dictionary`, so substitute an `rl-dictionary` adjective such as "happy" to see a demonstration of the problem.

Weaknesses Found in rl-grammar

I learned that, in creating a grammar that accepts human-created input, a certain amount of deep structure can be taken for granted, but with machine-generated input, nothing can. So, while the author of a grammar doesn't worry about dealing with the possibility of the sentence "what what cats eat eat" (which `rl-grammar` finds no problem with), machine-generated sentences can come up with many strange things. This section describes two such problems that were corrected by adding restrictions to `sf-grammar`.

The "what what what" Problem

The following productions

```
nrep = "what" subject sa verb sa
subject = nstg
nstg = lnr | nrep
```

allow an nrep's subject to begin with the word "what," causing a repetition of the word so that `rl-grammar` will parse the sentence "what what cats eat eat" as an allowable sentence. The following restriction on an nrep, added to `sf-grammar`, prevents this:

```
;;;
;;;  NO-WHAT-WHAT checks whether first term after nrep's "what" is another
;;;                one, which would mean "what what" string.
(defun no-what-what ()
   (not (equal (words (core (element 'subject))) "what "))
)
```

This works, but it occasionally causes a function call stack overflow, for the following reason: because most restriction functions require information from the tree in question's internal structure, all of it's subtrees must be created before we call the functions that verify it soundness. This can be a problem in a sentence like the following Gertrude Stein-like sentence, which was created without the `no-what-what` restriction when the odds of `null` being picked were increased by 50% instead of the current setting of 75%. It shows that four or five “what” strings in a row are not unusual:

> "Only what what what what a egg to what what his fish eats with happy mean to what cats eats eat generally generally mean with his happy cat only mean to what what what what what my cats only fish generally eat mean generally to what what what what what happy eggs generally generally sleep with fish mean to mean fish to what what what what my mean sleep with mean eats with what mean eat sleep with cats only eats generally with happy eggs fish eat only only fish with eggs mean with her cat only sleep only sleep to her egg only sleep to my fish."

The levels of recursion and nesting necessary before the interpreter finds that the second and third words of the sentence should cause the restriction to reject the clause enclosing so much of the sentence cause the following error message with every fifth or sixth sentence generated:

```
Error: Received signal number 1000 (function call stack overflow)
  [condition type: SYNCHRONOUS-OPERATING-SYSTEM-SIGNAL]
```

Resetting compiler settings with `(proclaim ‘(optimize speed n))` doesn’t help; rearranging the grammar may help, but this brings up other issues addressed in the conclusion.

This issue did highlight the importance of making `solfish` ignore a particular restriction when given the same instruction as `cf-parser`:

```
(setf (get ‘no-what-what ‘ignore) t)
```

This will work with `solfish.` In general, I found that if the stack was piling toward an overflow when this restriction was on and I was running Lisp from within Emacs, two Ctrl-C keystrokes returned me to the Lisp command line.

Consecutive Sentence Adjuncts

Using the following productions, rl-grammar has no problem with the sentence “cats eat generally generally,” because allowing `object` to be null allows two sentence adjuncts (`sa`) in a row:

```
assertion = sa subject sa verb sa object sa
object    = nstg | tovo | null
```

While it’s tempting to write a restriction that just prevents the same word twice in a row anywhere in the sentence (which would also take care of the “what what what” problem) the sentence “What eggs fish fish sometimes” generated by solfish is actually correct, and should be allowed. One approach to fixing the problem would be to rearrange the productions, but instead I wrote the `no-consec-sec` restriction for the `assertion` production to reject an `assertion` in which two non-null sentence adjuncts were separated by a null subject, verb, or object. (I now see that no productions will allow a null subject or verb, so the `no-consec-sec` code could be simpler.)

Conclusion, Further Work

The “what what” problem, although resolved here to some degree, highlights the key problem with using a sentence generator to detect problems with a grammar developed for parsing: What are the chances that even a beginner at creating English sentences would create a sentence with the word “what” repeated two or more times in a row? Adding restrictions or rewriting productions to account for problems with such a minuscule chance of happening could be a waste of programming and computing time.

On the other hand, I originally saw the potential use of such a tool in checking grammars derived from the automated analysis of a large body of text because I thought that a grammar with little human judgment involved in its development would need more checking. I now see that a grammar developed in this fashion would offer the opportunity for a more sophisticated and therefore potentially more useful sentence generator because of a key element of these grammars that could provide a way out of the “what what what” problem: stored statistics that can provide guidance in the probability of a given production being selected at any given time. As a similar but very primitive means towards keeping generated sentence lengths reasonable, `solfish` adds 75% to the chance of `null` being picked if it’s a possibility, but percentages based on real data to guide the selection of every production of the grammar could result in far more natural sentences—depending on the corpus used!

"A eggs generally sleep to mean."

```
(SENTENCE
 (CENTER
  (ASSERTION (SA (NULL))
   (SUBJECT
    (NSTG
     (LNR (LN (TPOS (T "a")) (APOS (NULL))) (N "eggs") (RN (NULL)))))
   (SA (NULL))
   (VERB (LTVR (LV (D "generally")) (TV "sleep") (RV (NULL))))
   (SA
    (PN (P "to")
     (NSTG
      (LNR (LN (TPOS (NULL)) (APOS (NULL))) (N "mean") (RN (NULL))))))
   (OBJECT (NULL)) (SA (NULL)))))
```

The lack of distinction between the article "a" as a determiner and possessive adjectives allows the LNR "a eggs." The "sleep to mean" is more of a problem with rl-dictionary than with rl-grammar, because "sleep" is considered an acceptable transitive verb and "mean" has an entry as a plural noun.