by Jeffreys Copeland and Haemer

CHET PHILLIPS

'Twas brillig, and
the slithy toves
Did gyre and
gimble in the wabe;
All mimsy were
the borogroves,
And the mome
raths outgrabe.
– Lewis Carroll
The Hunting
of the Snark

Nonsense

We have spent a lot of effort over the years building text-processing programs. We've built tools for formatting text, reformatting text, examining text, extracting text, compressing text, encrypting text, and washing text in the kitchen sink. Most recently we've even told you how to read text on your Palm handheld–see our January and April columns (`http://swexpert.com/C9/SE.C9.JAN.01.pdf` and `http://swexpert.com/C9/SE.C9.APR.01.pdf`). We've written TeX device drivers and invented pre- and post-processors for *troff*. We've ported *troff* enough times and in enough different countries that we can probably do it in our sleep.

We have also spent a lot of time explaining how important testing is. How do we get test data for our test programs? Sometimes we get the test text from the same source as our live data. Sometimes we grab random test text off the Web–the RISKS digest from either Usenet newsgroup `comp.risks` or `ftp://ftp.sri.com/risks/` is a great source. Sometimes we generate the test text ourselves–and that's our exercise for this month. (And as an exercise, try typing "test text" three times fast.)

Why do we want random text rather than purpose-build data? Because it is often more likely to find bugs in the corners of our code. Sometimes we find we generate test text like the *expected* input, and not enough like the *likely* input.

How to Do It

One of the current favorite methods of generating random text is the Markov-chain algorithm. Markov chains gather groups of words that occur in some domain space of input text, and then randomly re-generates chains of words, so that the randomly-generated text looks a lot like the source text. For example, if we're building three-word chains and the word "chips" only appears after the words "fish and" in our input text, "chips" will always be preceeded by "fish and" in our output. If "fish" is only followed by "and chips" in the input data, anytime when we randomly generate "fish," we will (as night follows day) follow it with "and chips."

Markov chains have been used for such disparate purposes as simulating Web page navigation and generating a completely fake Usenet persona. There is an excellent discussion of Markov algorithms in Chapter 3 of Kernighan and Pike's *The Practice of Programming* (Addison-Wesley, 1999, ISBN 0-201-61586-X).

(We will confess that since this column was actually written on a warm, beautiful summer day, we were sorely tempted to actually generate the column from Markov chains of previous columns. But we thought better of it.)

It's also worth noting that we vaguely remember an episode of *The Avengers*, long ago, when we were wee lads and televisions only had two colors (black and white), which centered on a computer (with a console disguised as a grand piano) that generated random

romance novels. We've been unable to place the episode, but would be delighted to hear which one it was.

There are other possible approaches to the problem, however. We could just produce random letters, in random length "words," randomly punctuated. But that wouldn't look very much like real input data. One consequence of that would be that our hyphenation algorithms wouldn't get a realistic workout.

We could improve on that method by choosing letters in the same frequencies as they actually appear in English, except that we still would be generating unreadable "words." Alternately, we could use a Markov chain on the letters to generate our "words"–rather the inverse of `typo`, an early UNIX spell checking program that used letter frequencies to find unlikely words, rather than using a dictionary lookup. (See "Statistical Text Processing" in the UNIX issue of *The Bell Systems Technical Journal*, July-August 1978, Vol. 57, No. 6, Part 2, pp 2137-2154.)

What if we bumped up a level and generated strings of randomly chosen words? We'd still have some of the same problems. It would be much more salutary to generate real sentences out of randomly chosen words. We could then actually read the input text, which would make it much easier to correlate the input text with the bugs in the output. How to do it, though?

We could use the refrigerator magnets someone gave us with words for generating tabloid headlines, and write down every sentence some visitor to our kitchen generates, for example, "Michael's satanic glove father of Elvis space alien baby." Then again, maybe not.

There's another way.

Grammar in Reverse

Normally, as programmers, when we look at grammar diagrams, it's because we're planning to use them to analyze rather than synthesize. We usually want to parse a sentence–usually a statement in a computer language. However, now we want to think about what constitutes a natural language sentence so we can generate one. Put another way, rather than diagramming sentences, like we used to do in elementary school, we're using the grammar rules to fill in a sketch of a sentence.

The first useful question to ask is "what's the grammar for a sentence?" Generally, we'll want to generate a subject followed by a verb, followed by an object. Put into Backus-Naur Form, that's

```
<sentence> := <subject-phrase> <verb-phrase>
   <object-phrase>
```

By specifying the grammar for each token, we would eventually end up with rules like

```
<object-phrase> := <verb> <adverb>?
<verb> := come | give | go | get
<adverb> := well | darkly
```

(Our original implemation of this program had a frighteningly short list of available words. For explanatory purposes, we'll use that same short list here. The sentences sometimes sound alike, which caused longtime friend and collegue Chris Kostanick to dub it the "darkly cheese program.")

At this point, we've got enough bits to write code for sentence generation. Let's have at it:

```
#! /usr/local/bin/perl -w
#  generate junk sentences
# $Id: junk,v 1.3 2001/08/06 00:23:01 jeff Exp $
```

We start with the usual boilerplate, then proceed into some POD (that's Perl-speak for plain old documentation). POD allows us to include documentation directly in the body of the program. In this case, we show the generating grammar we're going to use.

```
=pod
What's a sentence?  We construct sentences
from the following simple grammar.

   <sentence> :== <subj-phrase> <verb-phrase>
      <object-phrase>
   <subj-phrase> :== <subject>
      {<CONJUNCTION> <subject>}
   <subject> :== <ADJECTIVE>? <SUBJ_NOUN>
   <object-phrase> :== <object>
      {<CONJUNCTION> <object>}
   <object> :== <ADJECTIVE>? <OBJ_NOUN>
   <verb-phrase> :== <VERB> <ADVERB>?

Each of the base parts of speech (in caps)
is generated randomly from an internal list.
=cut
```

Let's do a brief review of the grammar for the grammar, that is, what the symbols in Backus-Naur Form mean. Things contained in < . . . > are symbols. We've used capitals to be tokens. (In the UNIX model, typically the lowest-level tokens are the ones returned to *yacc* from *lex*.) For our purposes, the word tokens will be chosen from lists. Like in Perl regular expressions, `?` is an element that appears zero or one times; here an expression in curly braces also indicates an element that appears zero or more times.

Now we need the actual vocabulary. We provide an array of words for each of the lowest level tokens.

```
@VERB = ( "come from", "get", "give", "go to",
   "keep", "let", "make", "put",
   "seem", "take", "be", "do" );
@OBJ_NOUN = ( "him", "her", "it", "them",
   "Bob", "Carol", "Ted", "Alice",
   "Holmes", "Watson", "cheese", "lunch" );
@SUBJ_NOUN = ( "he", "she", "it", "they",
   "Bob", "Carol", "Ted", "Alice",
   "Holmes", "Watson", "cheese", "lunch" );
@ADJECTIVE = ( "blue", "green", "awful",
   "tasty" );
@ADVERB = ( "badly", "well", "darkly" );
```

```
@CONJUNCTION = ( "and", "or" );
```

We've kept this list short, since the last thing you need is to look at all the words in the dictionary. You can get a longer version at the usual Web sites, or add more words on your own.

We'll also need routines to randomly grab words out of each of those lists:

```
my @words;

# generate the parts of speech
sub verb() {
   push @words, $VERB[rand @VERB]; }
sub obj_noun()  {
   push @words, $OBJ_NOUN[rand @OBJ_NOUN]; }
sub subj_noun()     {
   push @words, $SUBJ_NOUN[rand @SUBJ_NOUN]; }
sub adjective() {
   push @words, $ADJECTIVE[rand @ADJECTIVE]; }
sub adverb()  {
   push @words, $ADVERB[rand @ADVERB]; }
sub conjunction()  {
   push@words,$CONJUNCTION[rand@CONJUNCTION];}
```

We're grabbing a random element out of each array, based on its size, and pushing it onto our array of words.

One other thing we need to be able to do is generate an optional element. For example, in the case of a noun phrase, we want to be able to generate a conjunction and another noun with a specified probability.

```
# probability check
sub probably {
   my $x = shift;
   rand() < $x;
}
```

This routine allows us to say things like

```
adjective() if( probably(.3) );
```

If we were writing in C, we'd define macros so we could write the more natural

```
maybe(.3) adjective();
```

Alternately, we could define a `maybe()` function in Perl that took a probability and a pointer to a function.

Clearly we'll need a routine to produce a sentence.

```
sub sentence() {
   @words = ();
   subj_phrase();
   verb_phrase();
   obj_phrase();
   print ucfirst(join(" ",@words)), ".\n";
}
```

We begin by clearing the array words, and then adding subject, verb, and object phrases. We finish off by printing the words, using `ucfirst` to ensure that the first letter of the sentence is capitalized.

Subject and object phrases are parallel constructions. We optionally can use multiple subjects (or objects) connected by conjunctions. Given the probability we've chosen, one in 25 of the phrases will have more than one conjunction involved.

```
sub subj_phrase() {
   subject();
   while( probably(.2) ) {
      conjunction();
      subject();
   }
}

sub obj_phrase() {
   object();
   while( probably(.2) ) {
      conjunction();
      object();
   }
}
```

Of course, we then need to generate the subjects and objects. These are optional adjectives with a noun. For example, "dead Bob" or "old Jake."

```
sub subject() {
   adjective() if( probably(.3) );
   subj_noun();
}

sub object() {
   adjective() if( probably(.3) );
   obj_noun();
}
```

Remember that we have already looked at the routines to generate object and subject nouns and adjectives for their respective lists.

Verb phrases are remarkably similar to what we have done already.

```
sub verb_phrase() {
   verb();
   adverb() if( probably(.25) );
}
```

Those are the parts we needed. Now we can construct our main program. We begin by defaulting to 100 sentences if we haven't asked for a different number on the command line.

```
my $count = @ARGV ? $ARGV[0] : 100;
```

We generate the requested number of sentences, and add a blank line to begin a new paragraph after about one in 10 sentences.

```
while( $count-- > 0 ) {
   sentence();
   print "\n"  if( probably(.1) );
}
```

What sort of text does this generate? Given the list of words we've provided, we get output such as:

```
Awful Ted go to darkly green lunch.
Green Carol take Alice.
She let Bob.
It come from badly green Carol.
Awful they go to Carol.
Tasty Bob come from tasty lunch.
```

A Random Summary

Clearly there are improvements that could be made. We should be using different verb numbers based on whether our subject phrase is singular or plural, for example.

Exercise one for the reader: Use the `Lingua::EN::Stem` or `Text::Stem` modules from `http://www.cpan.org/` to fix the verb number for singular or plural subjects automatically. Exercise two for the reader: Replace our routines for choosing random words with the CPAN module `Data::Random::WordList`. We haven't used this approach because it doesn't save much code.

On the other hand, this gives us a reasonable first cut, and reasonable test input for many purposes, even if it does read a little oddly.

Next time, we'll see if we can take this one step further and build haiku from random elements. Until then, happy trails. ➸

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`*.*