Embedded languages:
Common Lisp practice, 2

January 16, 2001

In what follows it is assumed that you have a tall Emacs window, a program `lisp` in your path, and a copy of the Common Lisp Hyperspec on hand. Start an inferior Lisp process under Emacs.

1- Simple Sentence Generation Create a Lisp file `simple-english.lisp` with the following contents:

```
;;;; -*- Lisp -*-
;;;; A simple grammar-based generator of English sentences.
;;;; Based on (Norvig, 1992) chapter 2.

;;; 1. Hard-coded grammar rules
(defun sentence ()    (append (noun-phrase) (verb-phrase)))
(defun noun-phrase () (append (Article) (Noun)))
(defun verb-phrase () (append (Verb) (noun-phrase)))
(defun Article ()     (one-of '(the a)))
(defun Noun ()        (one-of '(man ball woman table)))
(defun Verb ()        (one-of '(hit took saw liked)))

(defun one-of (set)
  "Pick one element of set, and make a list of it."
  (list (random-elt set)))

(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

Mark the whole buffer and evaluate the region to include these definitions in your inferior Lisp. C-x h C-c C-r

Test this code by running `(sentence)` several times. Also test `(noun-phrase)`, `(verb-phrase)`, etc. Trace all of these functions:

```
(trace sentence noun-phrase verb-phrase article
 noun verb)
```

and inspect the derivation produced by a new call to `(sentence)`. What does `lisp-show-variable-documentation` do? C-c C-f

2- Rule-based Sentence Generation A better solution uses an explicitly-declared set of grammar rules. Add the following code to your file:

```
(defparameter *simple-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defvar *grammar* *simple-grammar*
  "The current grammar, initially *simple-grammar*.")
```

Evaluate each function as you type it. The grammar is stored in an *association list*; test it with `(assoc 'noun *grammar*)`. Two helper functions extract parts of rules, and one more finds all possible rewritings of a given category:

C-c C-e

```
(defun rule-lhs (rule) "left-hand side" (car rule))
(defun rule-rhs (rule) "right-hand side" (cddr rule))

(defun rewrites (category)
  "Find all possible rewritings for this category"
  (rule-rhs (assoc category *grammar*)))
```

We use these rules to generate sentences, by randomly choosing between the possible rewritings of every phrase:

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (cond ((listp phrase)
         (mappend #'generate phrase))
        ((rewrites phrase)
         (generate (random-elt (rewrites phrase))))
        (t (list phrase))))
```

Test this new version using `(generate 'sentence)`.

1. Why does `rule-rhs` use `cddr`?
2. What is the difference between `append` and `mappend`?
3. Under what conditions is the default case of the `cond` in textttgenerate executed?

Walk through the execution using the debugger.

3- Same data, different program Copy the function `generate` as `generate-tree`, change the `mappend` to `mapcar`, and `cons` the category onto the front of each rewrite. What does the derivation tree look like?

M-{ C-SPC
M-} M-w C-y

4- Same program, different data Test your programs on the larger grammar from +PAIP/simple.lisp+.