Usando Lisp para Enseñar a una Computadora a Escribir

Este post fue originalmente escrito en chino y publicado en CSDN, https://blog.csdn.net/lzw_java/article/details/1159

La mayoría del código y las ideas se basan en “Ansi Common Lisp”P138~P141.

Problema: ¿Cómo puede una computadora generar texto aleatorio pero legible a partir de un texto en inglés? Por ejemplo:

> The National Venture Capital Association estimates that wealth associated with a deal a big spending by regulations that will spend one another’s main reason these projects.

Este es un texto aleatorio generado por una computadora después de aprender algunos artículos de Paul Graham. Se extiende en una oración desde la palabra “Venture”. Sorprendentemente, el texto es a menudo legible.

Algoritmo: Registra las palabras que aparecen después de cada palabra y el número de veces que aparecen. Por ejemplo, si “I leave”aparece 5 veces en el texto original y “I want”aparece 3 veces, y “I”no aparece antes de ninguna otra palabra, entonces al generar texto aleatorio, cuando se encuentre “I”, hay una probabilidad de 5/8 de elegir “leave”como la próxima palabra. Si se elige “leave”, luego verifique qué palabras han aparecido después de “leave”y repita el proceso.

Ahora, resolvamos el problema usando Lisp.

El tipo de símbolo de Lisp puede registrar varias cadenas y marcas de puntuación bien, por lo que lo usaremos para registrarlo. Usaremos la tabla hash incorporada para crear una lista:

```
(defparameter *words* (make-hash-table :size 10000))
```

¿Cómo creamos la lista?

```
(let ((prev '|.|))
  (defun see (sym)
    (let ((pair (assoc sym (gethash prev *words*))))
      (if pair
      (incf (cdr pair))
      (push (cons sym 1) (gethash prev *words*))))
    (setf prev sym)))
```

La palabra actual es la clave y la lista de asociaciones es el valor bajo esa clave. Por ejemplo, bajo “I” tenemos ((|leave| . 5) (|want| . 3)). Si la palabra no existe, luego empuje (word . 1).

¿Cómo seleccionamos aleatoriamente una palabra?

```
(defun random-word (word ht)
  (let* ((choices (gethash word ht))
     (x (random (reduce #'+ choices :key #'cdr))))
    (dolist (pair choices)
      (decf x (cdr pair))
      (if (minusp x)
      (return (car pair))))))
```

Aquí, la función reduce se utiliza ingeniosamente.

Ahora, piensa en cómo extender una palabra dada en una oración en ambos lados?

1) Invierta el texto para obtener una lista invertida, es decir, “I leave, I want”se convierte en “leave I, want I”.
2) Invierta la tabla hash para obtener otra tabla hash, donde la palabra posterior es la clave y las palabras anteriores posibles y sus conteos forman una lista de asociaciones.
3) Intenta tu suerte, comienza a extender la oración desde una marca de puntuación hasta que aparezca la palabra dada.

Usé el segundo método:

```
(defparameter *r-words* (make-hash-table :size 10000))

(defun push-words (w1 w2 n)
  (push (cons w2 n) (gethash w1 *r-words*)))

(defun get-reversed-words () ; a cat -> cat a
  (maphash #'(lambda (k lst)
           (dolist (pair lst)
         (push-words (car pair) k (cdr pair))))
       *words*))
```

Recorre la tabla hash original y luego inserta cada par de palabras en otra tabla hash con su orden invertido. Aquí está el código para generar automáticamente oraciones extendidas bidireccionales:

```
(defparameter *words* (make-hash-table :size 10000))
(defconstant maxword 100)
(defparameter nwords 0)
(defconstant debug nil)
(let ((prev '|.|))
  (defun see (sym)
```

```
    (incf nwords)
    (let ((pair (assoc sym (gethash prev *words*))))
      (if pair
      (incf (cdr pair))
      (push (cons sym 1) (gethash prev *words*))))
    (setf prev sym)))

(defun check-punc (c) ; char to symbol
  (case c
    (#\. '|.|) (#\, '|,|)
    (#\; '|;|) (#\? '|?|)
    (#\: '|:|) (#\! '|!|)))

(defun read-text (pathname)
  (with-open-file (str pathname :direction :input)
    (let ((buf (make-string maxword))
      (pos 0))
      (do ((c (read-char str nil 'eof)
          (read-char str nil 'eof)))
      ((eql c 'eof))
    (if (or (alpha-char-p c)
        (eql c #\'))
        (progn
          (setf (char buf pos) c)
          (incf pos))
        (progn
          (unless (zerop pos)
        (see (intern (subseq buf 0 pos)))
        (setf pos 0))
          (let ((punc (check-punc c)))
        (if punc
            (see punc)))))))))

(defun print-ht (ht)
  (maphash #'(lambda (k v)
        (format t "~A ~A~%" k v))
         ht))

(defparameter *r-words* (make-hash-table :size 10000))
```

```
(defun push-words (w1 w2 n)
  (push (cons w2 n) (gethash w1 *r-words*)))

(defun get-reversed-words () ; a cat -> cat a
  (maphash #'(lambda (k lst)
           (dolist (pair lst)
         (push-words (car pair) k (cdr pair))))
       *words*))

(defun print-a-word (word ht)
  (maphash #'(lambda (k lst)
           (if (eql k word)
           (format t "~A ~A~%" k lst)))
       ht))

(if debug
    (print-a-word '|leave| *r-words*))

(defun punc-p (sym) ; symbol to char, nil when fails.
  (check-punc (char (symbol-name sym) 0)))

(defun random-word (word ht)
  (let* ((choices (gethash word ht))
     (x (random (reduce #'+ choices :key #'cdr))))
    (dolist (pair choices)
      (decf x (cdr pair))
      (if (minusp x)
      (return (car pair))))))

(defun gen-former (word str)
  (let ((last (random-word word *r-words*)))
    (if (not (punc-p last))
    (progn
      (gen-former last str)
      (format str "~A " last)))))

(defun gen-latter (word str)
  (let ((next (random-word word *words*)))
    (format str "~A " next)
    (if (not (punc-p next))
```

```
        (gen-latter next str))))

;(gen-latter '|leave| t)

(defun get-a-word (ht) ; get a random word
  (let ((x (random nwords)))
    (maphash #'(lambda (k v)
         (dolist (pair v)
           (decf x (cdr pair))
           (if (minusp x)
               (return-from get-a-word (car pair)))))
         ht)))
;(get-a-word *words*)
(defun gen-sentence (word str)
  (gen-former word str)
  (format str "~A " word)
  (gen-latter word str))

(defun test ()
  (setf nwords 0)
  (read-text "essay.txt")
  (get-reversed-words)
  (let ((word (get-a-word *words*)))
    (print word)
    (gen-sentence word t)))
(test)
```