Lisp zum Schreiben eines Computers lehren

Dieser Beitrag wurde ursprünglich auf Chinesisch verfasst und auf CSDN veröffentlicht, https://blog.csdn.net/lzw_java

---

Die meisten Codes und Ideen basieren auf “Ansi Common Lisp”P138~P141.

Problem: Gegeben sei ein englischer Text. Wie kann ein Computer basierend darauf einen zufälligen, aber lesbaren Text erzeugen? Zum Beispiel:

> 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 .

Dies ist ein zufälliger Text, der von einem Computer nach dem Lernen einiger Artikel von Paul Graham generiert wurde. Er erweitert sich zu einem Satz vom Wort “Venture”aus. Überraschenderweise ist der Text oft lesbar.

Algorithmus: Die Wörter, die nach jedem Wort erscheinen, und die Anzahl, wie oft sie erscheinen, werden aufgezeichnet. Zum Beispiel, wenn “I leave”im ursprünglichen Text 5 Mal und “I want”3 Mal erscheint und “I”vorher keinem anderen Wort vorausgeht, dann gibt es eine 5/8 Wahrscheinlichkeit, dass “leave” als nächstes Wort gewählt wird, wenn “I”getroffen wird. Wenn “leave”gewählt wird, dann wird überprüft, welche Wörter nach “leave”erschienen sind, und der Prozess wird wiederholt.

Nun lösen wir das Problem mit Lisp.

Der Symboltyp von Lisp kann verschiedene Zeichenketten und Interpunktionen gut aufzeichnen, daher werden wir ihn für die Aufzeichnung verwenden. Wir werden die eingebaute Hashtabelle verwenden, um eine Liste zu erstellen:

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

Wie erstellen wir die Liste?

```
(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)))
```

Das aktuelle Wort ist der Schlüssel, und die assoc-Liste ist der Wert unter diesem Schlüssel. Zum Beispiel gibt es unter “I”((|leave| . 5) (|want| . 3)). Wenn das Wort nicht existiert, dann wird (Wort . 1) gepusht.

Wie wählen wir ein zufälliges Wort aus?

```
(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))))))
```

Hier wird die Reduce-Funktion kluge verwendet.

Nun, überlegen wir, wie man ein gegebenes Wort in einen Satz auf beiden Seiten erweitert?

1) Den Text umkehren, um eine umgekehrte Liste zu erhalten, d.h. “I leave, I want”wird zu “leave I, want I”.
2) Die Hashtabelle umkehren, um eine andere Hashtabelle zu erhalten, wobei das spätere Wort der Schlüssel ist und die möglichen vorhergehenden Wörter und deren Anzahlen eine assoc-Liste bilden.
3) Versuche dein Glück, starte die Erweiterung des Satzes von einem Interpunktionszeichen, bis das gegebene Wort erscheint.

Ich habe die zweite Methode verwendet:

```
(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*))
```

Durchlaufe die ursprüngliche Hashtabelle und füge jedes Wortpaar in eine andere Hashtabelle mit umgekehrter Reihenfolge ein. Hier ist der Code zur automatischen Erzeugung von bidirektional erweiterten Sätzen:

```
(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)
```