Utiliser Lisp pour enseigner à un ordinateur à écrire

Ce message a été écrit en chinois et publié sur CSDN, https://blog.csdn.net/lzw_java/article/details/11599993

La plupart du code et des idées sont basés sur “Ansi Common Lisp”P138~P141.

Problème : Comment un ordinateur peut-il générer un texte aléatoire mais lisible à partir d’un texte en anglais ? Par exemple :

> 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 .

C’est du texte aléatoire généré par un ordinateur après avoir appris certains articles de Paul Graham. Il s’étend dans une phrase à partir du mot “Venture”. De manière surprenante, le texte est souvent lisible.

Algorithme : Enregistrez les mots qui apparaissent après chaque mot et le nombre de fois qu’ils apparaissent. Par exemple, si “I leave”apparaît 5 fois dans le texte original et “I want”apparaît 3 fois, et “I”ne précède aucun autre mot, alors lors de la génération de texte aléatoire, lorsqu’on rencontre “I”, il y a une probabilité de 5/8 de choisir “leave”comme mot suivant. Si “leave”est choisi, vérifiez ensuite quels mots ont suivi “leave”et répétez le processus.

Maintenant, résolvons le problème en utilisant Lisp.

Le type de symbole de Lisp peut enregistrer divers chaînes de caractères et signes de ponctuation, nous l’utiliserons donc pour l’enregistrement. Nous utiliserons la table de hachage intégrée pour créer une liste :

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

Comment créer la 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)))
```

Le mot actuel est la clé, et la liste assoc est la valeur sous cette clé. Par exemple, sous “I”nous avons `(|leave| . 5) (|want| . 3)`. Si le mot n’existe pas, alors push `(word . 1)`.

Comment choisir un mot de manière aléatoire ?

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

Ici, la fonction reduce est utilisée de manière astucieuse.

Maintenant, pensons à la manière de prolonger un mot donné en une phrase des deux côtés ?

1) Inversez le texte pour obtenir une liste inversée, c'est-à-dire "I leave, I want"devient "leave I, want I".
2) Inversez la table de hachage pour obtenir une autre table de hachage, où le mot suivant est la clé, et les mots précédents possibles et leurs comptes forment une liste assoc.
3) Tentez votre chance, commencez à prolonger la phrase à partir d'un signe de ponctuation jusqu'à ce que le mot donné apparaisse.

J'ai utilisé la deuxième méthode :

```
(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*))
```

Parcourez la table de hachage d'origine, puis insérez chaque paire de mots dans une autre table de hachage avec leur ordre inversé. Voici le code pour générer automatiquement des phrases prolongées des deux côtés :

```
(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)
```