Московский Государственный Университет им. М. В. Ломоносова
факультет Вычислительной математики и кибернетики

Отчет по "Доктору"

Киреев Даниил Сергеевич
420 группа ВМК МГУ

Москва, 2015

Содержание

1. Упражнения 1-6

Упражнение 1. Расширил репертуар системы (функции qualifier и hedge) двумя новыми фразами "why do you think that" и "say anything else" соответственно.

Упражнение 2. В шаблоне функция change-person брала пару слов, которые должны заменяться в тексте, проходила по фразе и применяла замену к каждому слову этой фразы (если она, конечно, была необходима), затем аналогично данная функция поступала с каждой последующей парой слов. В результате после отработки пар (i you) и (you i) слово i оставалось самим собой, что не верно. Функция была изменена таким образом, что она проходила по каждому слову и применяла к нему замены из списка пар (если замена применена, то переходим к следующему слову). В результате функция отрабатывает так, что каждое слово подвергается только одной замене.

Упражнение 3. В историю добавляются все предложения по отдельности, кроме вопросительных (потому что они имеют особую структуру). История - список из предложений. Если рандом выбрал стратегию earlier, то случайно выбирается одно предложение из истории.

Упражнение 4. Добавил функцию doctor, которая вызывает функцию visit-doctor для разных пользователей. Как только встречается пользователь с именем "suppertime", доктор прекращает прием пациентов.

Упражнение 5. Для хранения информации о ключевых словах и репликах, которые им соответствуют, я создал список:

```
(define keys
  (list
    (list '(depressed suicide)
          '(when you feel depressed |,| go out for ice cream) '(depression is a disease that can be treated))
    (list '(mother father parents)
          '(tell me more about your *) '(why do you feel that way about your * ?))))
```

В нем хранится информация о "блоках" ключевых слов. Каждый "блок" - список, в котором на первой позиции перечисляются ключевые слова, а далее идут фразы, которые ассоциируются с данными ключевыми словами. Список можно без проблем дополнить.

Упражнение 6. Аналогично предыдущему пункту я создал список, в котором хранится информация о стратегиях: лямбда-функция, которая определяет применима ли данная стратегия; функция, с которой ассоциирована данная стратегия и вес стратегии. Сначала проверяются все лямбда-функции и формируется список "хороших" стратегий, и позже из них случайно выбирается одна с учетом её веса.

2. Упражнение 7

Решение упражнения 7 отсутствует.

3. Весна. Обучение

Для хранения обученной модели я использую racket хеш-таблицу (воспользовался советом из текста задания). В качестве ключей я использую pair из symbol’ов - пара слов, которые связанны дугой. Значение для каждого ключа - число, которое отображает сколько раз в тексте была встреченная данная пара именно в таком порядке.

Первоначально из текста формируется список по тому же алгоритму, что и в стандартном "докторе", а именно:

1. Скобочки в высказываниях являются неким уточнением, дополнением. Например: Я иду домой (давно не был здесь). В ответном высказывании это уточнение не нужно: Ранее вы сказали, что вы идете домой. Уточнение здесь ни к чему. Соответственно я удаляю скобочки из высказываний. При этом я контролирую баланс скобочек, если есть лишние скобочки вне баланса, то я их просто удаляю.
2. Все вопросительные предложения удаляются. В историю они тоже не идут.
3. Если в запросе пользователя "-" идет слитно со словом, то он считается дефисом, если отдельно, то тире.
4. " ' " считается частью слова, i'm you're your's добавил в change-person.
5. Кавычки отделяются пробелами с обеих сторон (нельзя надежно определить является ли кавычка открывающейся или закрывающейся).
6. Итого знаки пунктуации, которые распознаются: " , . ! ? ‘ и скобочки (которые удаляются).

Далее по данному списку я пробегаю функцией, которая вносит пары слов в модель.

После создания и заполнения хеш-таблицы она сериализуется с помощью функций из racket и записывается в бинарный файл. Обучение данной модели на подготовленном тексте заняло примерно 8 секунд. Доступ к необходимой информации осуществляется достаточно быстро, чтобы не задумываться о его длительности.

Кроме того есть возможность дообучить модель, для этого необходимо запустить ту же самую функцию, но с другим флагом. В этом случае модель считывается из файла, текст обрабатывается, модель дополняется и будет вновь сохранена в тот же самый файл.

Формат запуска функции обучения:
(learning modelname textname flag)
modelname - имя файла с моделью
textname - имя файла с текстом
flag - либо "new", если нужно создать новую модель; либо "add", если нужно дообучиться

Тексты были взяты с сайта, который выполняет поиск по статьям: scholar.google.com. Использовался запрос "psychoanalyst", по данному запросу было просмотрено более 10 страниц из поисковой выдачи и отобраны 4 текста (к сожалению, информация о авторах и названиях была утеряна). В результате их обработки и объединения получился единый текст длинной более 10000 слов.

4. Весна. Генерация

Для генерации ответов я реализовал три дополнительные стратегии: прямой, обратный и смешанный подходы, которые были описаны в нашем задании. При прямом и обратном подходах генерация ответов начинается с точки, а при смешанном - со случайного слова из реплики пользователя (если случайное слово - конец предложения, то стратегия со смешанным подходом считается плохой, так как ответы получаются слишком длинными).

Смешанный подход является результатом слияния прямого и обратного:
результат_обратного_подхода случайное_слово результат_прямого_подхода

Функции, которые используются при прямом и обратном подходе аналогичны, так что я рассмотрю в данном отчете только алгоритмы, применяемые при прямом подходе.
Изначально мы преобразовали хеш-таблицу в список, чтобы было удобнее с ней работать с помощью функции hash->list. Теперь профильтруем список так, чтобы остались только те пары ключ-значение, в которых на первой позиции в ключе стоит необходимое слово. У нас получился новый список. Посчитаем сумму весов в этом списке: она нам понадобится, чтобы с помощью функции prob (описана в тексте оригинального задания "доктора") выбрать случайное слово для продолжения предложения. Когда новое слово получено, мы проверяем является ли оно концом предложения, если нет, то мы и дальше продолжаем дополнять наше предложение. В результате данных действий мы получили необходимый нам ответ.

5. Результаты

Формат вызова доктора:
(doctor modelname)
modelname - имя модели, которая используется для формирования ответов в "весне"

> (doctor "model")
(next!)
(Who are you?)
**user
Hello, user!
(what seems to be the trouble?)
**I love scheme!
this incomplete penetrance raises the data consist in plasma acth and for psychoanalysis, so novel investigative power and psychosomatic data consist in behavior so novel investigative power of this raises the failure, in limbic neocortical association and behavior

В данном примере сработала стратегия "весны", что не удивительно, ведь у неё стоит самый большой вес. Было сгенерировано предложение, которое выдалось пользователю в качестве ответа.

**I love scheme!
many people have the same sorts of feelings

Здесь сработала самая обычная функция hedge, которая возвращает ответ никак не смотря на запрос пользователя.

**I love scheme!
why do you believe that you love scheme

Здесь же сработала функция qualifier, которая заменила местоимения в реплике пользователя и выдала их в склейке с заготовленным шаблоном.

**I love my father and mother.
why do you feel that way about your father?
**I love my father and mother.
why do you feel that way about your mother?

В данном случае демонстрируется, как программа может среагировать на запрос, в котором есть несколько ключевых слов. В списке реакций на данные слова есть элемент "*", который может быть заменен на любое из ключевых слов из запроса пользователя (какое именно - выбирается случайным образом).

**I love scheme!
earlier you said that you love your father and mother

В данном примере показывается результат работы программы, в котором ответ на реплику пользователя был взят из истории, конечно же местоимения были в нем заменены.
При тестировании "весны" я подавал на обучение простенькие тексты. Качество итоговых ответов было значительно ниже по сравнению с компиляцией профессиональных статей: ответы повторялись и были короткими (вероятность выйти на конец предложения была выше).

Заключение

Изначально, когда я только приступил к освоению scheme, мне было тяжело привыкнуть к новому подходу, новой парадигме. Не уверен, что даже сейчас понял и проникся ею. На мой взгляд, даже на функциональном языке я пытался писать как на обычном императивном. В любом случае, я вынес для себя какие-то особенности работы с данным языком.

Если присмотреться к моему коду, который был написан для разных этапов, то можно проследить появление понимания многих аспектов языка. Я не стал переписывать функции, которые писал раньше (они не так понятны, как последние), прослеживать изменения действительно забавно.

В итоге, ближе к концу, когда я начал писать "весну", я начал получать удовольствие от написания программы. Этому поспособствовал как язык, так и само задание, которое было действительно интересным. С Яндекс.Весной я был знаком ещё до выдачи нам данного задания, как собственно и корчевателем, но я не понимал, какие методы они используют. Теперь же я реализовал более простой аналог сам, что не может не радовать.

Суммарно на выполнение всех этапов я потратил примерно 30 часов. Большая часть времени ушла, конечно, на то, чтобы разобраться с самыми основами, далее было всё проще и интереснее, основной упор был сделан именно на написание задания. Не могу не упомянуть превосходную документацию, пользоваться которой было невероятно удобно. Общий код занял примерно 350 строчек, обычно на такого рода задания их было побольше. Возможно этому поспособствовал язык, а возможно моя лень, объективно оценить я не могу.

Список литературы

[1] Racket Documentation https://docs.racket-lang.org
[2] Doctor https://ejudge.ru/study/5sem/doctor.pdf
[3] Задание «Весна». 2015-16 учебный год http://sp.cs.msu.ru/scheme/vesna.html

Приложение. Код программы

doctor.rkt

```
#lang scheme/base
(require racket/serialize)

(define (doctor modelname)
        (define in (open-input-file modelname))
        (define s-ht (read in))
        (close-input-port in)

        (define ht (hash->list (deserialize s-ht)))

        (define (visit-doctor name)
                (define (doctor-driver-loop name history)
                        (define (reply user-response history)
                                (define (change-person phrase)
                                        (many-replace '((i you) (me you) (you i) (am are) (are am) (my
your) (your my)
                                                                (i'm you're) (you're i'm) (your's
mine)) phrase))

                                (define user-list
                                        (give-user-list-true (give-user-list user-response '())))

                                (define random-word
                                        (symbol->string (pick-random user-response))
                                )

                                (define (forward-vesna history)
                                        (make-forward ht ".")
                                )

                                (define (back-vesna history)
                                        (append (cdr (reverse (make-back ht "."))) (list 'I.I))
                                )

                                (define (mix-vesna history)
                                        (make-mix ht random-word)
                                )

                                (define (qualifier history)
                                        (append
                                        (pick-random '((you seem to think)
                                                                (you feel that)
                                                                (why do you believe that)
                                                                (why do you say that)
                                                                (why do you think that)))
(change-person (pick-random user-list))))

                                (define (hedge history)
                                        (pick-random '((please go on)
                                                                (many people have the same
sorts of feelings)
```

```
                                                    (many of my patients have told
me the same thing)
                                                    (please continue)
                                                    (say anything else))))

                        (define (earlier history)
                                (append '(earlier you said that)
                                        (change-person (list-ref history (random (length
history))))))

                        (define keys
                                (list
                                        (list '(depressed suicide) '(when you feel depressed I,I
go out for ice cream) '(depression is a disease that can be treated))
                                        (list '(mother father parents) '(tell me more about your
*) '(why do you feel that way about your * ?))))

                        (define keylist
                                (give-keylist keys))

                        (define goodkeys
                                (give-goodkeys keys user-response))

                        (define perfectkeys ;without *
                                (give-perfectkeys goodkeys))

                        (define (key-words history)
                                (pick-random perfectkeys))

                        (define (good-strategy strategy user-response history keylist)
                                (cond ((null? strategy) '())
                                        (else (if ((car (car strategy)) user-response history
keylist random-word)
                                                (cons (car strategy) (good-strategy (cdr
strategy) user-response history keylist))
                                                (good-strategy (cdr strategy) user-response
history keylist)))))

                        (define strategy
                                (list
                                        (list (lambda (user-response history keylist rword) (and
(in-ht rword ht) (not (equal? rword ".")) (not (equal? rword "!")) (not (equal? rword "?")))) mix-vesna
10)
                                        (list (lambda (user-response history keylist rword) #t)
forward-vesna 10)
                                        (list (lambda (user-response history keylist rword) #t)
back-vesna 10)
                                        (list (lambda (user-response history keylist rword) (not
(null? (find-keyword user-response keylist)))) key-words 4)
                                        (list (lambda (user-response history keylist rword) (not
(null? (give-user-list-true (give-user-list user-response '())))))) qualifier 3)
                                        (list (lambda (user-response history keylist rword) #t)
hedge 2)
                                        (list (lambda (user-response history keylist rword) (< 2
(length history))) earlier 5)))
```

```
                                (define (sum_str gs)
                                        (if (null? gs) 0
                                        (+ (car (cdr (cdr (car gs)))) (sum_str (cdr gs)))))

                                (define (exec-strategy gs user-response history)
                                        (if (null? gs) (hedge history)
                                        (if (prob (car (cdr (cdr (car gs)))) (sum_str gs))
                                                ((car (cdr (car gs))) history)
                                                (exec-strategy (cdr gs) user-response history))))

                                (exec-strategy (good-strategy strategy user-response history keylist)
user-response history))

                        (newline)
                        (print '**)

                        (let ((user-line (string-downcase (read-line (current-input-port)))))
                                (cond ((equal? user-line "goodbye")
                                                (printf "Goodbye, ~a!\n" name)
                                                (printf "See you next week")
                                                (newline)
                                                (doctor modelname))
                                        (else (print-reply (reply (line-to-response user-line) history))
                                                (doctor-driver-loop name (append history (give-user-
list-true (give-user-list (line-to-response user-line) '()))))))
                        )

                (printf "Hello, ~a!\n" name)
                (print '(what seems to be the trouble?))
                (doctor-driver-loop name '()))

        (print '(next!))
        (newline)
        (print '(Who are you?))
        (newline)
        (print '**)
        (let ((name (string-downcase (read-line (current-input-port)))))
                (if (equal? name "suppertime")
                        (print '(Time to go home!))
                        (visit-doctor name))))

(define (fifty-fifty)
        (= (random 2) 0))

(define (pick-random lst)
        (list-ref lst (random (length lst))))

(define (required-word replacement-pairs word)
        (define (loop i)
                (cond ((= (length replacement-pairs) i) word)
                        ((equal? (car (list-ref replacement-pairs i)) word) (cadr (list-ref replacement-
pairs i)))
                        (else (loop (+ i 1)))))
        (loop 0))
```

```
(define (many-replace replacement-pairs lst)
	(cond ((null? lst) '())
		(else
			(cons (required-word replacement-pairs (car lst))
				(many-replace replacement-pairs (cdr lst))))))

(define (required-keyword word keylist)
	(define (loop i)
		(cond ((= (length keylist) i) '())
			((equal? (list-ref keylist i) word) word)
			(else (loop (+ i 1)))))
	(loop 0))

(define (find-keyword phrase keylist)
	(cond ((null? phrase) '())
		(else (if (null? (required-keyword (car phrase) keylist)) (find-keyword (cdr phrase)
keylist)
			(cons (required-keyword (car phrase) keylist) (find-keyword (cdr phrase)
keylist))))))

(define (prob n1 n2)
	(< (random n2) n1))

(define (give-keylist keys)
	(cond ((null? keys) '())
		(else (append (car (car keys)) (give-keylist (cdr keys))))))

(define (check-word word phrase)
	(cond ((null? phrase) '())
		(else (if (equal? word (car phrase))
			word (check-word word (cdr phrase))))))

(define (check-key key phrase)
	(cond ((null? key) '())
		(else (if (null? (check-word (car key) phrase)) (check-key (cdr key) phrase)
			(cons (check-word (car key) phrase) (check-key (cdr key) phrase))))))

(define (give-key key phrase)
	(if (null? (check-key (car key) phrase))
		'() (cons (check-key (car key) phrase) (cdr key))))

(define (give-goodkeys keys phrase)
	(cond ((null? keys) '())
		(else (if (null? (give-key (car keys) phrase))
			(give-goodkeys (cdr keys) phrase) (cons (give-key (car keys) phrase) (give-
goodkeys (cdr keys) phrase))))))

(define (word-replace words phrase)
	(cond ((null? words) '())
		(else (cons (many-replace (list (append '(*) (list (car words)))) phrase) (word-replace
(cdr words) phrase)))))

(define (key-replace words phrases)
	(cond ((null? phrases) '())
```

```
                (else (append (word-replace words (car phrases)) (key-replace words (cdr
phrases))))))

(define (give-perfectkeys keys)
        (cond ((null? keys) '())
                (else (append (key-replace (car (car keys)) (cdr (car keys))) (give-perfectkeys (cdr
keys))))))

; USER-LIST

(define (give-user-list-true lst)
        (cond ((null? lst) '())
                (else (cons (reverse (car lst)) (give-user-list-true (cdr lst))))))

(define (give-user-list lst tmp-lst)
        (cond ((null? lst) (if (not (null? tmp-lst)) (list tmp-lst) '()))
                ((or (equal? (car lst) 'l.l) (equal? (car lst) 'l!l)) (if (null? tmp-lst) (give-user-list (cdr lst)
'()) (cons tmp-lst (give-user-list (cdr lst) '()))))
                ((equal? (car lst) 'l?l) (give-user-list (cdr lst) '()))
                (else (give-user-list (cdr lst) (cons (car lst) tmp-lst)))))

; LINE-TO-RESPONSE

(define (my-char-punctuation? my-char)
        (or (char=? #\" my-char) (char=? #\, my-char) (char=? #\. my-char) (char=? #\! my-char)
(char=? #\? my-char) (char=? #\- my-char)))

(define (line-to-response line)
        (line-to-response-utility line (make-string 0) 0))

(define (line-to-response-utility line word bracket)
        (cond ((string=? line "") (if (not (string=? word "")) (list (string->symbol word)) '()))
                ((and (char=? #\( (string-ref line 0)) (string=? word "")) (line-to-response-utility
(substring line 1) (make-string 0) (+ bracket 1)))
                ((and (char=? #\( (string-ref line 0)) (not (string=? word ""))) (cons (string->symbol
word) (line-to-response-utility (substring line 1) (make-string 0) (+ bracket 1))))
                ((and (> bracket 0) (not (char=? #\) (string-ref line 0)))) (line-to-response-utility
(substring line 1) (make-string 0) bracket))
                ((and (> bracket 0) (char=? #\) (string-ref line 0))) (line-to-response-utility (substring
line 1) (make-string 0) (- bracket 1)))
                ((or (char-alphabetic? (string-ref line 0)) (char-numeric? (string-ref line 0)) (char=? #
\' (string-ref line 0))) (line-to-response-utility (substring line 1) (string-append word (string (string-ref
line 0))) bracket))
                ((and (char=? #\- (string-ref line 0)) (not (string=? word ""))) (line-to-response-utility
(substring line 1) (string-append word (string (string-ref line 0))) bracket))
                ((my-char-punctuation? (string-ref line 0)) (if (string=? word "") (cons (string->symbol
(string (string-ref line 0))) (line-to-response-utility (substring line 1) (make-string 0) bracket))

(cons (string->symbol word) (cons (string->symbol (string (string-ref line 0))) (line-to-response-
utility (substring line 1) (make-string 0) bracket))) ))
                (else (if (string=? word "") (line-to-response-utility (substring line 1) (make-string 0)
bracket) (cons (string->symbol word) (line-to-response-utility (substring line 1) (make-string 0)
bracket))))))

; PRINT-REPLY
```

```
(define (print-reply lst)
	(if (and (my-char-punctuation? (string-ref (symbol->string (car lst)) 0)) (not (char=? #
\" (string-ref (symbol->string (car lst)) 0))) (not (char=? #\- (string-ref (symbol->string (car lst)) 0))))
		(printf (symbol->string (car lst))) (printf (string-append " " (symbol->string (car lst)))))
	(if (not (null? (cdr lst))) (print-reply (cdr lst)) '()))

; MAKE-FORWARD

(define (firstword-filter ht word)
	(cond ((null? ht) '())
		(else (if (equal? (symbol->string (caaar ht)) word) (cons (car ht) (firstword-filter (cdr
ht) word)) (firstword-filter (cdr ht) word))))
)

(define (secondword-filter ht word)
	(cond ((null? ht) '())
		(else (if (equal? (symbol->string (car (cdr (car (car ht))))) word) (cons (car ht)
(secondword-filter (cdr ht) word)) (secondword-filter (cdr ht) word))))
)

(define (ht-sum lst)
	(cond ((null? lst) 0)
		(else (+ (cdr (car lst)) (ht-sum (cdr lst)))))
)

(define (fchoose-word lst sum last-word)
	(cond ((null? lst) last-word)
		(else (if (prob (cdr (car lst)) sum) (cdr (car (car lst))) (fchoose-word (cdr lst) sum last-
word))))
)

(define (schoose-word lst sum last-word)
	(cond ((null? lst) last-word)
		(else (if (prob (cdr (car lst)) sum) (car (car (car lst))) (schoose-word (cdr lst) sum
last-word))))
)

(define (make-forward ht word)
	(define lst (firstword-filter ht word))
	(define sum (ht-sum lst))
	;(define last-word (if (pair? lst) (cdr (car (car lst))) (cdr (car lst))))
	(define last-word (cdr (car (car lst))))

	(define new-word (car (fchoose-word lst sum last-word)))

	(if (or (equal? (symbol->string new-word) ".") (equal? (symbol->string new-word) "!")
(equal? (symbol->string new-word) "?")) (list new-word) (cons new-word (make-forward ht (symbol-
>string new-word))))
)

(define (make-back ht word)
	(define lst (secondword-filter ht word))
	(define sum (ht-sum lst))
	;(define last-word (if (pair? lst) (car (car (car lst))) (car (car lst))))
```

```
        (define last-word (car (car (car lst))))

        (define new-word (schoose-word lst sum last-word))

        (if (or (equal? (symbol->string new-word) ".") (equal? (symbol->string new-word) "!")
(equal? (symbol->string new-word) "?")) (list new-word) (cons new-word (make-back ht (symbol-
>string new-word))))
)

(define (make-mix ht word)
        (append (cdr (reverse (make-back ht word))) (list (string->symbol word)) (make-forward ht
word))
)

; IN-HT

(define (in-ht word ht)
        (cond ((null? ht) #f)
                (else (if (equal? word (symbol->string (caaar ht))) #t (in-ht word (cdr ht)) )))
)
```

learning.rkt

```
#lang scheme/base
(require racket/serialize)

(define (learning modelname textname flag)
  (define in (open-input-file textname))
  (define text (string-downcase (read-line in)))
  (close-input-port in)

  (define lst (text-to-lst text))

  (define ht (cond ((equal? flag "new") (begin
      (if (file-exists? modelname) (delete-file modelname) '())

      (make-hash)
    ))

    ((equal? flag "add") (begin
      (define in (open-input-file modelname))
      (define s-ht-in (read in))
      (close-input-port in)

      (deserialize s-ht-in)
    ))

    (else (exit))
  ))

  (parse-lst ht lst)

  (display "Learning complete\n")

  (define s-ht (serialize ht))

  (if (file-exists? modelname) (delete-file modelname) '())
  (define out (open-output-file modelname))
  (write s-ht out)
  (close-output-port out)
)

; PARSE-LST

(define (parse-lst ht lst)
  (cond ((null? (cdr lst)) ht)
    (else (begin (define pr (list (car lst) (car (cdr lst))))
      (if (hash-has-key? ht pr) (hash-set! ht pr (+ (hash-ref ht pr) 1)) (hash-set! ht pr 1))
      (parse-lst ht (cdr lst))))
  )
)

; TEXT-TO-LST

(define (my-char-punctuation? my-char)
```

```
  (or (char=? #\" my-char) (char=? #\, my-char) (char=? #\. my-char) (char=? #\! my-char) (char=? #
\? my-char) (char=? #\- my-char)))
(define (text-to-lst line)
  (text-to-lst-utility line (make-string 0) 0))
(define (text-to-lst-utility line word bracket)
  (cond ((string=? line "") (if (not (string=? word "")) (list (string->symbol word)) '()))
    ((and (char=? #\( (string-ref line 0)) (string=? word "")) (text-to-lst-utility (substring line 1) (make-
string 0) (+ bracket 1)))
    ((and (char=? #\( (string-ref line 0)) (not (string=? word ""))) (cons (string->symbol word) (text-to-
lst-utility (substring line 1) (make-string 0) (+ bracket 1))))
    ((and (> bracket 0) (not (char=? #\) (string-ref line 0)))) (text-to-lst-utility (substring line 1) (make-
string 0) bracket))
    ((and (> bracket 0) (char=? #\) (string-ref line 0))) (text-to-lst-utility (substring line 1) (make-string
0) (- bracket 1)))
    ((or (char-alphabetic? (string-ref line 0)) (char-numeric? (string-ref line 0)) (char=? #\' (string-ref
line 0))) (text-to-lst-utility (substring line 1) (string-append word (string (string-ref line 0))) bracket))
    ((and (char=? #\- (string-ref line 0)) (not (string=? word ""))) (text-to-lst-utility (substring line 1)
(string-append word (string (string-ref line 0))) bracket))
    ((my-char-punctuation? (string-ref line 0)) (if (string=? word "") (cons (string->symbol (string
(string-ref line 0))) (text-to-lst-utility (substring line 1) (make-string 0) bracket))
                    (cons (string->symbol word) (cons (string->symbol (string (string-ref line 0))) (text-
to-lst-utility (substring line 1) (make-string 0) bracket))) ))
    (else (if (string=? word "") (text-to-lst-utility (substring line 1) (make-string 0) bracket) (cons
(string->symbol word) (text-to-lst-utility (substring line 1) (make-string 0) bracket))))))
```