Московский государственный университет имени М. В. Ломоносова
факультет Вычислительной математики и кибернетики
кафедра системного программирования.

Отчет по "Доктору"

Лебедев Никита Алексеевич
группа 428
г. Москва, 2020

Содержание

Предисловие

Многие функции, полученные в результате выполнения заданий, были изменены при выполнении последующих заданий и не вошли в финальную версию доктора. Тем не менее, код всех описанных функций приведен в этом отчёте.
Для удобства работы в текстовом редакторе названия некоторых функций были переведены с kebab-case на camelCase.

1. Упражнения 1-7

Упражнение 1

Измените функции qualifier-answer и hedge, добавив в каждую не менее трёх новых заготовленных фраз-реплик и/или фраз, с которых начинается ответ с заменой лица.

Описание функций:

Функция qualifierAnswer получает реплику пользователя в качестве значения параметра. В ней случайно выбирается одно из заготовленных начал ответа. Окончанием ответа является реплика пользователя, в которой у местоимений изменено лицо.

Функция hedge случайным образом выбирает ответ из заранее заготовленного списка ответов.

Структуры данных: фраза пользователя и ответ на нее представляют собой список символов.

Решение для qualifierAnswer:
Добавил 3 новых начала ответа:

```
(do you mean that)
(so you are saying that)
(why do you feel that)
```

Решение для hedge: Добавил 3 новых варианта ответа:

```
(interesting)
(I understand you)
(please tell more about it)
```

Упражнение 2

Напишите новую версию функции many-replace с хвостовой рекурсией и вызовите её в теле change-person. Составьте код нового many-replace без определения локальной вспомогательной функции, а с использованием "именованного" let.

Структуры данных:
replacement-pairs - список списков-пар слов, где на первом месте стоит слово, которое нужно заменить, а на втором - на которое нужно заменить.
lst - список слов, в котором нужно заменить слова.

Идея реализации:
Заводим именованный let с названием rec-replace. При помощи assoc он для каждого слова из списка lst пытается найти пару с соответствующим ключом и заменить его на второе слово из пары. Возвращается развернутый список, в котором произведены все необходимые замены. Осталось еще раз развернуть его при помощи reverse.

```
(define (many-replace replacement-pairs lst)
 (let
   ((reversed-replaced
     (let rec-replace
       (
         (left '())
         (right lst)
       )
       (if (null? right)
         left
         (let ((pat-rep (assoc (car right) replacement-pairs)))
           (let ((new-val (if pat-rep (cadr pat-rep) (car right))))
             (rec-replace (cons new-val left) (cdr right))
           )
         )
       )
     )
   ))
   (reverse reversed-replaced)
 )
)
```

Упражнение 3

Напишите ещё одну версию функции many-replace и замените вызов в теле change-person. Сделайте так, чтобы тело новой версии состояло только из вызова map. Дополнительную функцию не определяйте отдельно, а заведите как анонимную -- результат вычисления спецформы lambda.

При помощи функции map, примененной к списку слов lst, создадим новый список слов, в котором заменены все слова, совпадающие хотя бы с одним из ключей в replacement-pairs.

```
(define (many-replace replacement-pairs lst)
 (map
   (lambda (word)
     (let ((pat-rep (assoc word replacement-pairs)))
       (if pat-rep (cadr pat-rep) word)
     )
   )
   lst
 )
)
```

Упражнение 4

Измените программу таким образом, чтобы doctor-driver-loop сохранял список всех реплик пользователя. Новая версия функции reply будет выбирать одну стратегию из трёх. Новую стратегию реализуйте как отдельную функцию history-answer.

Добавим к doctor-driver-loop еще один парамер old-phrases, представляющий из себя список списков-реплик пользователя. При каждом новом вызове doctor-driver-loop список дополняется новым ответом пользователя.

```
(define (doctor-driver-loop name old-phrases)
  (newline)
  (print '**)
  (let ((user-response (read)))
    (cond
      ((equal? user-response '(goodbye))
        (printf "Goodbye, ~a!\n" name)
        (print '(see you next week))
      )
      (else
        (print (reply user-response old-phrases))
        (doctor-driver-loop name (cons user-response old-phrases))
      )
    )
  )
)
```

Функция reply выбирает один из 2 вариантов при первом ответе и 3 вариантов при последующих.

```
(define (reply user-response old-phrases)
 (let ((num-variants (if (null? old-phrases) 2 3)))
   (case (random num-variants)
       ((0) (qualifier-answer user-response))
       ((1) (hedge))
       ((2) (history-answer old-phrases))
   )
 )
)
```

Функция history-answer выбирает случайную фразу из ранее сказанных пользователем, заменяет лицо в местоимениях и добавляет в начало ‘earlier you said that’

```
(define (history-answer old-phrases)
 (append '(earlier you said that) (change-person (pick-random old-phrases)))
)
```

Упражнение 5

Измените программу таким образом, чтобы «Доктор» автоматически переходил к приему следующего пациента после прощания с предыдущим. Предусмотрите способ завершения работы многопользовательского «Доктора» или после использования стоп-слова в качестве имени очередного пациента, или при исчерпании количества принимаемых пациентов.

При каждом следующем рекурсивном вызове visit-doctor уменьшает счетчик оставшихся пациентов на 1. Когда он становится равен 0, доктор завершает рабочий день. Стоп-слово передается в рекурсивный вызов неизменным.

```
(define (visit-doctor stopWord num)
 (define (askPatientName)
   (begin
     (println '(next!))
     (println '(who are you?))
     (print '**)
     (car (read))
   )
 )
 (if (>= 0 num)
   '(time to go home)
   (let ((name (askPatientName)))
     (if (equal? name stopWord)
       '(working day finished)
       (begin
         (printf "Hello, ~a!\n" name)
         (print '(what seems to be the trouble?))
         (doctorDriverLoop name '())
         (visit-doctor stopWord (- num 1))
       )
     )
   )
 )
)
```

Упражнение 6

Реализуйте в программе стратегию построения ответов по ключевым словам.

Структуры данных:
word-groups - список структур данных, выглядящих так:

```
( ; ключевые слова
 (mother father parents brother sister uncle aunt grandma grandpa)
 ( ; шаблоны ответов
   (tell me more about your * , i want to know all about your *)
   (why do you feel that way about your * ?)
   (i dont have *)
   (at least your * is alive!)
 )
)
```

Функция keyword-answer:

1. выбирает одно ключевое слово из пользовательской реплики
2. выбирает один шаблон ответа, соответствующий выбранному слову
3. подставляет в него ключевое слово и возвращает результат

```
(define (keyword-answer user-response)
 (let*
   (
     (chosen-word (pick-random (find-keywords user-response)))
     (chosen-template (pick-random (find-templates chosen-word)) )
     (response (many-replace (list (list '* chosen-word)) chosen-template))
   )
   response
 )
)

(define (find-word-groups word)
 (filter (lambda (wg) (includes (car wg) word) ) word-groups )
)

(define (find-templates word)
 (foldl
   (lambda (wg init)
     (if (includes (car wg) word)
       (append (cadr wg) init)
       init))
```

```
  '()
  word-groups))
```

Упражнение 7

Переписать reply, создав обобщенную версию, которая будет работать с любым подаваемым ему на вход перечнем стратегий.

Структуры данных:
Перечень стратегий представляет из себя список из структур данных вида (предикат вес функция), где assocParamList - ассоциативный список ключами именами параметров.

```
; пример одной стратегии
(list
 (lambda (assocParamList) #t)
 1
 (lambda (assocParamList)
   (let ((userResponse (cadr (assoc 'userResponse assocParamList))))
     (qualifierAnswer userResponse)
   )
 )
)
```

```
(define (reply strategies assocParamsLst)
```

Функция reply, используя предикаты, выбирает доступные стратегии ответов. Из них, с учетом весов, выбирается функция. Эта функция вызывается с параметром assocParamsList, ее значение возвращается в качестве результата работы reply.

2. Упражнение 8

Решение упражнения №8 отсутствует.

3. Весна. Обучение

Структура данных:
(starts . order) - оба элемента в паре - хэш-таблицы.
starts = #(n-1-gram, count): в качестве ключей содержит n-1-граммы, с которых начинаются предложения, а в качестве значений - сколько предложений с них начинается.
order = #(n-1-gram, (nexts.prevs)): в качестве ключей содержит n-1-граммы, а в качестве значений - пару из хэш-таблиц.
nexts = #(word, count): в качестве ключей содержат слова, а в качестве значений - сколько раз они встречаются в тексте за указанной в order n-1-граммой.
prevs - аналогично nexts, только слова идут не после n-1-граммы, а до нее.

Алгоритм обучения:

1. В результате предварительной обработки текста мы получаем представление текста в виде списка слов-символов. При помощи функции slidingWindow с размером окна n+1 проходимся по всем n-1-граммам текста и заполняем обучаемую структуру данных:
   Добавляем n-1-грамму в хэш-таблицу order, если ее еще не было.
2. Если слова, идущего перед этой n-1-граммой нет в таблице nexts для этой n-1-граммы, то добавляем ее в эту таблицу в качестве ключа и со счетчиком 1. Иначе просто увеличиваем ее счетчик в этой таблице на 1.
3. Аналогичным образом добавляем слово, идущее перед n-1-граммой в таблицу prevs.
4. Если перед n-1-граммой идет одно из слов (!?.), то добавляем эту n-1-грамму в таблицу starts в качестве ключа со счетчиком 1, если ее там еще не было, либо просто увеличиваем ее счетчик на 1, если она там уже была.

Формат файла для хранения результатов обучения:
Выводим описанную выше структуру (starts.order) в файл при помощи функции print (см. функцию writeFile в коде). Чтобы восстановить данные читаем из файла функцией read (см. функцию readFileAsObject в коде).

Набор текстов для обучения:
Программа должна генерировать ответы на тему депрессии. Данные для обучения я искал в гугле по запросу “Что сказать человеку в депрессии”. Собрал около 60 коротких предложений на английском, на которых потом и обучал доктора.
В качестве неудачного текста для обучения была выбрана книга Джозефины Джексон по психоанализу “Outwitting Our Nerves A Primer of Psychotherapy”

Характеристики обучающих текстов и полученных в результате баз:
Хорошие данные: 57 предложений до обучения, 20КБ после обучения.
Плохие данные: 19300 строк до обучения, 1МБ после обучения.

4. Весна. Генерация

Генерация прямым способом:
Выбираем с учетом веса n-1-грамму из множества starts. Далее, при помощи таблицы nexts с учетом веса выбираем слово, идущее за выбранной n-1-граммой. Выбираем последние n-1 слов из полученной n-граммы и повторяем те же действия, пока не встретим слово из (!?.). Это означает, что предложение закончено.
Генерация смешанным способом:
Выбираем произвольную n-1-грамму. По ней генерируем предложения прямым и обратным способом, а затем склеиваем их (не забыв убрать повтор n-1-граммы). Обратный способ генерирует предложения аналогично прямому, но с использованием таблицы prevs.
Борьба с зацикливанием и слишком длинными предложениями:
При каждой генерации ответа выбирается случайное число в некотором диапазоне. Как только длина ответа начинает превышать это число, генератор стремится закончить предложение как можно скорее: если следующим словом может быть слово из (?!.), то, вне зависимости от веса, будет выбрано именно оно.
Праметры для генератора:
N было выбрано равным 3. Это обеспечило достаточное разнообразие генерируемых предложений при обучении на относительно небольшой выборке, при этом сохранив их осмысленность.

Основная функция генератора:

```
(define (generateAnswer starts.order userResponse)
 ; Функция wantEnd старается выбрать слово, являющееся концом предложения.
 ; Если это не возможно, то выбирает любое слово с учетом веса
 (define (wantEnd hashTable)
   (let
     ((a (or
       (and (hash-ref hashTable '|.| #f) '|.|)
       (and (hash-ref hashTable '|!| #f) '|!|)
       (and (hash-ref hashTable '|?| #f) '|?|)
     )))
     (or a (pickRandomKeyFromHT hashTable))
   )
 )
 ; Функция forward генерирует ответ прямым способом
 (define (forward order n-1gram lenLeft)
   (let*
     (
       (nexts (cdr (hash-ref order n-1gram "nothing")))
       (next (if (<= lenLeft 0)
         (wantEnd nexts)
         (pickRandomKeyFromHT nexts)
       ))
     )
```

```
    (if (isEnd? next)
      (append n-1gram (list next))
      (cons (car n-1gram) (forward order (append (cdr n-1gram) (list next)) (-
lenLeft 1)))
    )
  )
)
; Функция backward генерирует ответ обратным способом
(define (backward order st lenLeft)
  (let*
    (
      (prevs (car (hash-ref order st "nothing")))
      (prev (if (<= lenLeft 0)
        (wantEnd prevs)
        (pickRandomKeyFromHT prevs)
      ))
    )
    (if (isEnd? prev)
      st
      (append
        (backward order (cons prev (drop-right st 1)) (- lenLeft 1))
        (last-pair st)
      )
    )
  )
)
; Функция mixed генерирует ответ смешанным способом с использованием forward и
backward
(define (mixed order st lenLeft)
  (let*
    (
      (len (length st))
      (pre (drop-right (backward order st lenLeft) len))
      (post (list-tail (forward order st lenLeft) len))
    )
    (append pre st post)
  )
)
```

```
; (define (slidingWindow lst windowLength params fun)
 (let*
   (
     (starts (car starts.order))
     (order (cdr starts.order))
     (n-1 (length (car (hash-keys starts))))
     (stMatch (slidingWindow userResponse n-1 order (lambda (n-1gram order)
       (if (hash-ref order n-1gram #f)
         n-1gram
         #f
       )
     )))
     ; lenLeft - длина, после которой генератор будет стремиться завершить
предложение
     (lenLeft (random 15))
   )
   ; Если в пользовательской фразе есть n-1-грамма, которая встречалась в
обучающей выборке
   ; то генерируем ответ смешанным способом, отталкиваясь от этой n-1-граммы
   ; иначе генерируем ответ прямым способом
   (if stMatch
     (mixed order stMatch lenLeft)
     (forward order (pickRandomKeyFromHT starts) (* 2 lenLeft))
   )
 )
)
```

5. Результаты

Пример работы доктора, обученного на хорошем датасете “sentenses_for_learning.txt” с n-граммой длины 3.

who are you?
** Nikita
hello, nikita!
what seems to be the trouble?
** i feel depressed
you’re important to me.
** thank you doctor
is there anything you want to talk about it?
** it is hard for me to wake up in mornings
you’re a good person.
** thank you doctor
i’m here for you.
** i don't have energy for my work
this isn’t an ending.
** i don't know what to do with it
when did you begin feeling like this?
** month or two ago...
i’m sure we can figure it out together.
** i feel depressed
that sounds like it’s really hard.
** yes it is
this isn’t an ending.
** thank you doctor
depression can be defeated.
** i don't think so
your feelings are valid.
** it is hard to wake up in mornings
depression is a serious condition.
***** can you say that i am not alone?***
you’re not alone.
** goodbye
Goodbye, nikita!
See you next week!
next!
who are you?
** stop
working day finished

В выделенном жирным курсивом вопросе-ответе срабатывает генерация ответа по n-1-грамме (где n=3). Встретив “not alone”, генератор начал построение предложения в левую и правую стороны от нее. Слева он построил “you’re” и остановился, т.к. в качестве предыдущего слова ему выпала точка. Справа он добавил точку, как слово, следующие за “alone”.

Пример работы доктора, обученного на плохом датасете - книге психоаналитика Джексона “jackson.txt” с n-граммой длины 3.

next!
who are you?
** Nikita
hello, nikita!
what seems to be the trouble?
** i feel depressed
he has fed his flock.
** thank you doctor
no, thank you, and in many cases been dropped out of gear.
** i dont have energy for my work
again and is a costly proceeding, an inner and spiritual process very different from all foods finally reaches the rectum, it is far too compelling to be.
** i dont have energy for my work
all the petting he craves, is capable also of displacement or the energy for creative work.
** i dont know what to do with it
how many times it pays to know what made her play tennis every day.
** month or two ago...
suggestibility is one or two before the period of parental care.
** i feel depressed
i made her a stranger in the past.
** can you say i am not alone?
neither can you wash it out of the physician may be.
** goodbye
Goodbye, nikita!
See you next week!
next!
who are you?
** stop
working day finished

Мы видим, что обучающая выборка сильно влияет на качество генерируемых ответов. Хотелось бы иметь бОльшую обучающую выборку для хорошего варианта. Сначала я хотел написать собственный генератор обучающих предложений, но потом решил, что для нашего сравнения качество предложений в обучающей выборке важнее их количества, поэтому выбрал предложения написанные живыми людьми.

Заключение

В ходе проделанной работы был разработан чат-бот, изображающий психотерапевта и обладающий встроенным генератором предложений на тему “депрессия”. Как видно из приведенных в главе “Результаты” примеров диалогов, качество обучающей выборке крайне важно для такого генератора. Была разработана структура данных для хранения результатов обучения, а также написаны функции, позволяющие создавать и читать бэкап этой структуры, а также объединять структуры данных с результатами обучения на двух различных (или одинаковых) выборках.

Список литературы

Josephine Jackson. Outwitting Our Nerves A Primer of Psychotherapy - BiblioBazaar 2007, - 256 с.

Приложение. Код программы

```
#lang scheme/base

(require racket/string)
(require racket/port)
(require racket/hash)
(require racket/list)

; параметр num - сколько еще пациентов доктор может принять
(define (visit-doctor stopWord num starts.order)
 (define (askPatientName)
   (begin
     (printf "next!\n")
     (printf "who are you?\n")
     (printf "** ")
     (car (parseString (read-line)))
   )
 )
 (if (>= 0 num)
   '(time to go home)
   (let ((name (askPatientName)))
     (if (equal? name stopWord)
       (printf "working day finished")
       (begin
         (printf "hello, ~a!\n" name)
         (printf "what seems to be the trouble?")
         (doctorDriverLoop name '() starts.order)
         (visit-doctor stopWord (- num 1) starts.order)
```

```
       )
     )
   )
 )
)

(define (doctorDriverLoop name oldPhrases starts.order)
 (newline)
 (printf "** ") ; доктор ждёт ввода реплики пациента, приглашением к которому
является **
 (let ((userResponse (getFirstSentence (parseString (read-line)))))
   (cond
     ((equal? userResponse '(goodbye)) ; реплика '(goodbye) служит для выхода
из цикла
           (printf "Goodbye, ~a!\n" name)
           (print '(see you next week))
           (newline)
     )
     (else
       (printf
         (prepareForPrint
           (reply strategies (list
             (list 'userResponse userResponse)
             (list 'oldPhrases oldPhrases)
             (list '|starts.order| starts.order)
           ))
         )
       ) ; иначе Доктор генерирует ответ, печатает его и продолжает цикл
       (doctorDriverLoop name (cons userResponse oldPhrases) starts.order)
     )
   )
 )
)

(define strategies (list
 ; (predicate weight function)
 (list
   (lambda (assocParamList) #t)
   1
```

```
   (lambda (assocParamList)
     (let ((userResponse (cadr (assoc 'userResponse assocParamList))))
       (qualifierAnswer userResponse)
     )
   )
 )
 (list
   (lambda (assocParamList) #t)
   2
   (lambda (assocParamList) (hedge))
 )
 (list
   (lambda (assocParamList) (not (null? (cadr (assoc 'oldPhrases
assocParamList)))))
   3
   (lambda (assocParamList)
     (let ((oldPhrases (cadr (assoc 'oldPhrases assocParamList))))
       (historyAnswer oldPhrases)
     )
   )
 )
 (list
   (lambda (assocParamList)
     (let ((userResponse (cadr (assoc 'userResponse assocParamList))))
       (hasKeywords userResponse)
     )
   )
   4
   (lambda (assocParamList)
     (let ((userResponse (cadr (assoc 'userResponse assocParamList))))
       (keywordAnswer userResponse)
     )
   )
 )
 (list
   (lambda (assocParamList) #t)
   1000
   (lambda (assocParamList)
     (let*
```

```
      (
        (starts.order (cadr (assoc '|starts.order| assocParamList)))
        (userResponse (cadr (assoc 'userResponse assocParamList)))
      )
      (generateAnswer starts.order userResponse)
    )
  )
 )
))

; генерация ответной реплики по userResponse - реплике от пользователя
(define (reply strategies assocParamsLst)
 (define (filterByPredicate strategies assocParamList)
   (filter
     (lambda (strtg)
       (let ((predicate (car strtg)))
         (predicate assocParamList)
       )
     )
     strategies
   )
 )
 ; lst: ((weight function) (weight function) ...)
 (define (findByWeightAndRand lst rand)
   (let*
     (
       (this (car lst))
       (thisWeight (car this))
       (thisFunction (cadr this))
     )
     (if (< rand thisWeight)
       thisFunction
       (findByWeightAndRand (cdr lst) (- rand thisWeight))
     )
   )
 )
 ; lst: ((weight function) (weight function) ...)
 (define (pickRandomWithWeight lst)
   (let*
```

```
    (
      (maxRandom
        (foldl
          (lambda (x init) (+ init (car x)))
          0
          lst
        )
      )
      (rand (random maxRandom))
    )
    (findByWeightAndRand lst rand)
  )
 )

 (let*
   (
    (filteredStrategies (filterByPredicate strategies assocParamsLst))
    (weightedList (map cdr filteredStrategies))
    (function (pickRandomWithWeight weightedList))
   )
   (function assocParamsLst)
 )
)

(define (keywordAnswer userResponse)
 (let*
   (
    (chosenWord (pickRandom (findKeywords userResponse)))
    (chosenTemplate (pickRandom (findTemplates chosenWord)) )
    (response (manyReplace (list (list '* chosenWord)) chosenTemplate))
   )
   response
 )
)

; 1й способ генерации ответной реплики - замена лица в реплике пользователя и
приписывание к результату нового начала
(define (qualifierAnswer userResponse)
 (append
```

```
   (pickRandom
     '(
       (you seem to think that)
       (you feel that)
       (why do you believe that)
       (why do you say that)
       ;
       (do you mean that)
       (so you are saying that)
       (why do you feel that)
     )
   )
   (changePerson userResponse)
 )
)

(define (historyAnswer oldPhrases)
 (append '(earlier you said that) (changePerson (pickRandom oldPhrases)))
)

; случайный выбор одного из элементов списка lst
(define (pickRandom lst)
 (list-ref lst (random (length lst)))
)

; замена лица во фразе
(define (changePerson phrase)
       (manyReplace '((am are)
                      (are am)
                      (i you)
                      (me you)
                      (mine yours)
                      (my your)
                      (myself yourself)
                      (you i)
                      (your my)
                      (yours mine)
                      (yourself myself))
                    phrase)
```

```
)
 (define wordGroups '(
 (
   (depressed suicide exams university)
   (
     (when you feel depressed, go out for ice cream)
     (depression is a disease that can be treated)
     (depression can be defeated)
     (depression is for loosers!)
   )
 )
 (
   (mother father parents brother sister uncle ant grandma grandpa)
   (
     (tell me more about your * , i want to know all about your *)
     (why do you feel that way about your * ?)
     (i dont have *)
     (at least your * is alive!)
   )
 )
 (
   (university scheme lections)
   (
     (your education is important)
     (how many time do you spend to learning ?)
     (i guess, we are all talented in some way...)
     (nobody watches the diploma anyway)
   )
 )
 (
   (friend girlfriend wife husband)
   (
     (do you really need such * ?)
     (be careful, your * might be plotting against you...)
   )
 )
 (
   (money work debt loan)
   (
```

```
     (have you ever thought of becoming homeless? they are so carefree)
     (they killed my uncle because of his *)
   )
 )
))

(define keywords
 (foldl
   (lambda (wg init)
     (append (car wg) init)
   )
   '()
   wordGroups
 )
)

(define (findTemplates word)
 (foldl
   (lambda (wg init)
     (if (member word (car wg))
       (append (cadr wg) init)
       init
     )
   )
   '()
   wordGroups
 )
)

(define (findKeywords lst)
 (filter
   (lambda (word)
     (member word keywords)
   )
   lst
 )
)

(define (hasKeywords lst)
```

```
 (ormap
   (lambda (word) (member word keywords))
   lst
 )
)

(define (manyReplace replacementPairs lst)
 (map
   (lambda (word)
     (let ((patRep (assoc word replacementPairs))) ; пара (ключ значение) или ()
; Доктор ищет первый элемент списка в ассоциативном списке замен
       (if patRep (cadr patRep) word)
     )
   )
   lst
 )
)

; 2й способ генерации ответной реплики - случайный выбор одной из заготовленных
фраз, не связанных с репликой пользователя
(define (hedge)
 (pickRandom
   '(
     (please go on)
     (many people have the same sorts of feelings)
     (many of my patients have told me the same thing)
     (please continue)
     (intresting)
     (I understand you)
     (please tell more about it)
   )
 )
)

;================ input/output ================;



; (read-line)
;[.,:;()!?-] символы пунктуации
```

```
(define (parseString str)
 (define (removeBadChars str)
   (regexp-replace*
#rx"([&\\^\\*\\+\\=\\_\\|\\/\\%\\$\\#\\№\\@\\>\\<\\`\\~\\{\\}\\(\\)\"]|[[]|[]]|\
n)" str " " )
 )
 (define (addSpaces str)
   (regexp-replace* #rx"([.,:;«»()!?-])" str " \\1 ")
 )
 (define (splitBySpaces str)
   (regexp-split #rx" +" str)
 )
 (define (removeEmptyStr strList)
   (filter
     non-empty-string?
     strList
   )
 )
 (define (mapStringToSymbol strList) (map string->symbol strList))

 (mapStringToSymbol (removeEmptyStr (splitBySpaces (addSpaces (removeBadChars
(string-downcase str))))))
)

(define (prepareForPrint symbolList)
 (define (mapSymbolToString symbolList) (map symbol->string symbolList))
 (define (removeOddSpaces str)
   (regexp-replace* #rx" *- *"
     (regexp-replace* #rx" +([.,:;»)!?-])" str "\\1") ; убрал открывающую скобку
   "-")
 )
  (removeOddSpaces(string-join (mapSymbolToString symbolList)))
)

(define (readFileAsString path)
 (let*
   (
     (inputPort (open-input-file path))
```

```
      (data (port->string inputPort))
    )
    (close-input-port inputPort)
    data
  )
)

(define (readFileAsObject path)
  (let*
    (
      (inputPort (open-input-file path))
      (data (read inputPort))
    )
    (close-input-port inputPort)
    (println "readFileAsObject finished")
    data
  )
)

(define (writeFile data path)
  (let*
    (
      (outputPort (open-output-file path #:exists 'replace))
    )
    (print data outputPort)
    (close-output-port outputPort)
    "writeFile finished"
  )
)


; (hash-ref <hash> <key> <failval>)
; (hash-set! <hash> <key> <val>)
; val: (prevs . nexts)

(define (isEnd? symbol)
  (or
    (equal? symbol '|.|)
    (equal? symbol '|?|)
```

```
   (equal? symbol '|!|)
 )
)

;================ learning ================;


(define (slidingWindow lst windowLength params fun)
 (if (< (length lst) windowLength)
   #f
   (let
     (
       (thisVal (fun (take lst windowLength) params))
     )
     (if thisVal
       thisVal
       (slidingWindow (cdr lst) windowLength params fun)
     )
   )
 )
)

(define (incCount hashTable key increment)
 (let*
   (
     (count (hash-ref hashTable key 0))
     (newCount (+ increment count))
   )
   (hash-set! hashTable key newCount)
   newCount
 )
)

(define (addKeyToHashTable hashTable symbol valIfNot)
 (let
   (
     (value (hash-ref hashTable symbol #f))
   )
   (if value
```

```
     value
     (begin
       (hash-set! hashTable symbol valIfNot)
       valIfNot
     )
   )
 )
)

(define (learn starts.order N symbolList)
 (slidingWindow (cons '|.| symbolList) (+ N 1) starts.order
   (lambda (n+1gram starts.order)
     (let*
       (
         (starts (car starts.order))
         (order (cdr starts.order))

         (prev (car n+1gram))
         (next (last n+1gram))
         (n-1gram (drop-right (cdr n+1gram) 1))

         (prevs.nexts (addKeyToHashTable order n-1gram (cons (make-hash)
(make-hash))))
       )
       (if (isEnd? prev)
         (incCount starts n-1gram 1)
         "nothing"
       )
       (incCount (car prevs.nexts) prev 1)
       (incCount (cdr prevs.nexts) next 1)
       #f
     )
   )
 )
)

;=============== generating answer ===============;

(define (pickRandomKeyFromHT hashTable)
```

```
 (define sum (foldl
   (lambda (cur prev)
     (+ prev (hash-ref hashTable cur "noghing"))
   )
   0
   (hash-keys hashTable)
 ))
 (define rand (random sum))
 (define result "empty")
 (hash-for-each hashTable
   (lambda (key count)
     (if (not (equal? result "empty"))
       "nothing"
       (begin
         (set! rand (- rand count))
         (if (<= rand 0)
           (set! result key)
           "nothing"
         )
       )
     )
   )
 )
 result
)


(define (generateAnswer starts.order userResponse)
 ; Функция wantEnd старается выбрать слово, являющееся концом предложения.
 ; Если это не возможно, то выбирает любое слово с учетом веса
 (define (wantEnd hashTable)
   (let
     ((a (or
       (and (hash-ref hashTable '|.| #f) '|.|)
       (and (hash-ref hashTable '|!| #f) '|!|)
       (and (hash-ref hashTable '|?| #f) '|?|)
     )))
     (or a (pickRandomKeyFromHT hashTable))
   )
```

```
 )
 ; Функция forward генерирует ответ прямым способом
 (define (forward order n-1gram lenLeft)
   (let*
     (
       (nexts (cdr (hash-ref order n-1gram "nothing")))
       (next (if (<= lenLeft 0)
         (wantEnd nexts)
         (pickRandomKeyFromHT nexts)
       ))
     )
     (if (isEnd? next)
       (append n-1gram (list next))
       (cons (car n-1gram) (forward order (append (cdr n-1gram) (list next)) (-
lenLeft 1)))
     )
   )
 )
 ; Функция backward генерирует ответ обратным способом
 (define (backward order st lenLeft)
   (let*
     (
       (prevs (car (hash-ref order st "nothing")))
       (prev (if (<= lenLeft 0)
         (wantEnd prevs)
         (pickRandomKeyFromHT prevs)
       ))
     )
     (if (isEnd? prev)
       st
       (append
         (backward order (cons prev (drop-right st 1)) (- lenLeft 1))
         (last-pair st)
       )
     )
   )
 )
 ; Функция mixed генерирует ответ смешанным способом с использованием forward и
backward
```

```
 (define (mixed order st lenLeft)
   (let*
     (
       (len (length st))
       (pre (drop-right (backward order st lenLeft) len))
       (post (list-tail (forward order st lenLeft) len))
     )
     (append pre st post)
   )
 )

 ; (define (slidingWindow lst windowLength params fun)
 (let*
   (
     (starts (car starts.order))
     (order (cdr starts.order))
     (n-1 (length (car (hash-keys starts))))
     (stMatch (slidingWindow userResponse n-1 order (lambda (n-1gram order)
       (if (hash-ref order n-1gram #f)
         n-1gram
         #f
       )
     )))
     ; lenLeft - длина, после которой генератор будет стремиться завершить
предложение
     (lenLeft (random 15))
   )
   ; Если в пользовательской фразе есть n-1-грамма, которая встречалась в
обучающей выборке
   ; то генерируем ответ смешанным способом, отталкиваясь от этой n-1-граммы
   ; иначе генерируем ответ прямым способом
   (if stMatch
     (mixed order stMatch lenLeft)
     (forward order (pickRandomKeyFromHT starts) (* 2 lenLeft))
   )
 )
)

(define (getFirstSentence lst)
```

```
 (if (null? lst)
   lst
   (if (isEnd? (car lst))
     (list (car lst))
     (cons (car lst) (getFirstSentence (cdr lst)))
   )
 )
)

(define (merge pair1 pair2)
 (define (mergeHashTables ht1 ht2)
   (hash-for-each ht2
     (lambda (key count)
       (incCount ht1 key count)
     )
   )
 )
 (mergeHashTables (car pair1) (car pair2))
 (hash-for-each (cdr pair2)
   (lambda (symbol prevs2.nexts2)
     (let*
       (
         (prevs1.nexts1 (addKeyToHashTable (cdr pair1) symbol (cons (make-hash)
(make-hash))))
       )
       (mergeHashTables (car prevs1.nexts1) (car prevs2.nexts2))
       (mergeHashTables (cdr prevs1.nexts1) (cdr prevs2.nexts2))
     )
   )
 )
)


; ОБУЧАЕМСЯ
; (define bad (cons (make-hash) (make-hash)))
; (learn bad 2 (parseString (readFileAsString "_bad_tridon.txt")))
; (learn bad 2 (parseString (readFileAsString "_bad_jackson.txt")))
; (writeFile bad "backup_bad")
```

```
; (define good (cons (make-hash) (make-hash)))
; (learn good 4 (parseString (readFileAsString "_good_short_hilton.txt")))
; (learn good 4 (parseString (readFileAsString "_normal_sadger.txt")))
; (writeFile good "backup_good")

; ВСПОМИНАЕМ
(define good (readFileAsObject "backup_good"))
(define bad (readFileAsObject "backup_bad"))

; (define merged (cons (make-hash) (make-hash)))
; (merge merged good)
; (merge merged bad)

(visit-doctor 'stop 3 bad)
```