Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчет по «Доктору»

Чупахин Андрей Андреевич
421 группа

Москва, 2015

Содержание

Упражнения 1-6

Упражнение 1

Краткое описание задания:

Изменить процедуры qualifier и hedge, расширив репертуар системы.

Решение:

Увеличил количество элементов-фраз в списке, из которого при вызове qualifier случайно выбирается одна фраза. Для hedge решение аналогичное.

Код:

```
(define (qualifier)
    (pick-random '((you seem to think)
                   (you feel that)
                   (why do you believe)
                   (why do you say)
                   (what do you mean when say)
                   (You said)
                   (It |'| s very interesting What can you say except)
                   (We are clearly in the right direction |,| if you said)))
  )
(define (hedge)
    (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)
                   (can you say in more detail about your problem)
                   (Do you believe in what you said)
                   (We are on the right track)
                   (If you need anything |,| just ask)
                   (Make yourself at home)
                   (How long have you had these thoughts)
                (If you could wave a magic what positive changes would you make happen in your life))))
```

Упражнение 2

Краткое описание задания:

Улучшенная замена личных местоимений. Вторая стратегия ответа.

Решение:

Раньше функция many-replace просматривала все пары из списка replacement-pairs, для каждой пары вызывалась функция replace, которая заменяла в исходном списке слов все слова, совпадающие с первым словом из пары на второе слово из пары. Такой подход приводил к ошибочной работе(слова не менялись, оставались такими же), если в списке пар были противоположные пары, например, (I am) и (am I).

Чтобы сформировать правильный результат необходимо пробегаться по словам исходного списка и менять каждое слово в соответствии со списком replacement-pair.

В итоге изменена функция many-replace и replace. Many-replace пробегается по всем словам, вызывает для каждого replace с параметрами replacement-pair и word (текущее просматриваемое слово). А replace в свою очередь пробегается по парам replacement-pair(ищем совпадение по первому слову из пары); если первое слово из пары совпадает с word, то в качестве результата возвращаем второе слово из пары, иначе продолжаем поиск совпадения; если совпадений не было, то возвращаем word.

Код:

```
(define (replace replacement-pairs word)
  (cond ((null? replacement-pairs) word)
        ((equal? (caar replacement-pairs) word) (cadar replacement-pairs))
        (else (replace (cdr replacement-pairs) word ))))


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

Упражнение 3

Краткое описание задания:

Запоминание истории сообщений. Третья стратегия выбора ответа: берем произвольную реплику из истории, изменяем в ней лицо с первого на второе и добавляем вводную фразу “early you said that”.

Решение:

Чтобы запоминалась история, я ввел еще один входной параметр(список) в функцию doctor-driver-loop: old-phrases. Эта функция работает рекурсивно до тех пор, пока клиент не скажет goodbye. Поэтому для запоминания я при рекурсивном вызове передавал в качестве истории уже новый список – (update-history user-response old-phrases). Update-history – функция, которая предложения клиента добавляет в историю. Так как был модифицирован ввод, то теперь user-response – это список предложений. Чтобы не изменять доктора, я сделал каждое предложение списком из элементов типа symbol. После встраивания нового ввода сам доктор практически не поменялся.

Третья стратегия реализуется функцией answer-old-phrase. Она использует уже написанную функцию change-person(которая в свою очередь использует many-replace).

Код:

Добавлен параметр old-phrase.

```
(define (doctor-driver-loop name old-phrases)
...
)
```

```
(define (update-history lst history)
    (if (null? lst) history
        (update-history (cdr lst) (unique-push (car lst) history))))
```

Третья стратегия генерации ответа. Эта функция объявлена внутри doctor-driver-loop

```
(define (answer-old-phrase client-response)
    (list
     (cond ((null? old-phrases) '(You haven't said anything before that))
           (else (append '(early you said that) (change-person '((i you) (I you) (me you) (am are) (my your) (you i) (You I) (are am) (your my)) (pick-random old-phrases)))))))
```

Рекурсивный вызов

(doctor-driver-loop name (update-history user-response old-phrases)

Упражнение 4

Краткое описание задания:

Последовательная работа с несколькими клиентами.

Решение:

Добавил в тело функции visit-doctor вызов функции ask-patient-name. В ask-patient-name используется модифицированный ввод – doctor-read. Добавил также в visit-doctor cond, в котором есть проверка на ввод специального имени клиента suppertime, при вводе которого доктор завершает работу. В doctor-driver-loop считываются все предложения, введенные клиентом, с помощью функции doctor-read. Если пользователь ввел goodbye, то доктор завершает работу с текущим клиентом и считывает имя нового клиента.

Код:

```
(define (ask-patient-name)
   (begin
   (print '(NEXT!))
   (newline)
   (print '(Who are you?))
   (doctor-read)))


(define (visit-doctor)
...
 (cond ((equal? (doctor-print name) "suppertime") (print '(Time to sleep)))
       (else (begin
                (printf "Hello, ~a!\n" (if (null? name) "My friend"  (doctor-print name)))
                (print '(what seems to be the trouble?))
                (doctor-driver-loop (if (null? name) '((My friend)) name) '())
                (visit-doctor)))))
```

Упражнение 5

Краткое описание задания:

Реализация пятой стратегии генерации ответа. Ответ генерируется в зависимости от ключевых слов в реплике клиента.

Решение:

Генерация ответа – keyword-strategy. В предложении клиента может быть несколько ключевых слов, поэтому функция keyword-strategy возвращает список предложений, каждое соответствует отдельному обработанному ключевому слову. Keyword-strategy – это обертка для функции check-key-words, которая пробегается по всем словам из ответа клиента и вызывает для каждого из них функцию check-all-keys. Check-all-keys для каждого keys(см.код KEYWORDS-LIST) из keylst(см.код KEYWORDS-LIST) вызываем функцию check-keys. Check-keys делает следующее: проверяет наличие word в первом подсписке keys, если есть, то возвращаем рандомный подсписок, отличный от первого, т.е. один из sentence1 …. sentenceM; если во втором подсписке списка keys есть символ '*, то выбираем произвольную фразу из (cdr keys) и вызываем функцию уже известную функцию many-replace.

Код:

```
(define KEYWORDS-LIST ;такой список назовем keylst
  '( ((depressed suicide) ; каждый внутренний подсписок keylst вида
                          ;( (keyword1...keywordN) (sentence1)... (sentenceM)) назовем keys
     (when you feel depressed |,| go out for ice cream)
     (depression is a disease that can be threated)
   )
   ((mother father family parents)
    (tell me more about your *)
    (why do you feel that way about your * ?)
    (Are your * good?)
   )
   ((rain snow)
    (Do you like *)
    (What thoughts arise for you when the * ?)
   )))
```

```
;Задание №5. 5. Проверяет наличие word в списке lst
    (define (exist? word lst)
      (and (not (null? lst))
        ( or (equal? word (car lst))
          (exist? word (cdr lst)))))
;Задание №5. 4. Проверяем наличие word в первом подсписке keys, если есть,
;то возвращаем рандомный подсписок, отличный от первого(в котором хранятся ключевые
      слова)
;Если во втором подсписке списка keys есть символ '*, то выбираем произвольную фразу из
      (cdr keys)
;и вызываем функцию replace-all-words-in-lst
;Используем уже написанную функцию many-replace для замены * на ключевое слово
    (define (check-keys word keys)
     (cond ((exist? word (car keys))
          (cond ((exist? '* (cadr keys)) (many-replace (list(list '* word)) (pick-random (cdr keys))))
            ( else (pick-random (cdr keys)))))
        (else '())))
;Задание №5. 3. Для каждого keys из keyslst вызываем функцию check-all-keys
    (define (check-all-keys keyslst word)
     (cond ((null? keyslst) '())
        (else (let ((result (check-keys word (car keyslst))))
            ( if (null? result) (check-all-keys (cdr keyslst) word)
              result)))))
;Задание №5. 2.Пробегаемся по всем словам из ответа клиента вызываем функцию check-
key-words
    (define (check-key-words lst keyslst)
     (cond ((null? lst) '())
        (else (let ((result (check-all-keys keyslst (car lst))))
            (if (null? result) (check-key-words (cdr lst) keyslst)
              (cons result (check-key-words (cdr lst) keyslst)))))))
;Задание №5. 1. Обертка для check-key-words
```

(define (keywords-strategy client-response)

(check-key-words client-response KEYWORDS-LIST))

Упражнение 6

Краткое описание задания:

Обобщить структуру программы, используя набор предикатов.

Решение:

Я создал список pred-F(см. Код). Pred-F -это список списков, каждый список состоит из предиката и ассоциированного с ним набора функций. Основная функция – execute-strategy. Она в качестве результата выдает список списков (список из ответов). После мы произвольным образом выбираем один из ответов(см. Код::Вызов execute-strategy). Execute-strategy, получив на вход фразу клиента, пробегается по списку pred-F, выполняет предикат, если он выдал true – значит выбираем произвольную ассоциированную с этим предикатом функцию и выполняем ее, полученный результат кладем в результирующий список, так проделываем для всех предикатов.

Код:

;Задание №6 1. Предикаты и соответствующий им список функций

(define pred-F (list (list (lambda (x) (<= (length x) 5)) trite-expression answer-old-phrase)

(list (lambda (x) (intersect? '(i you I me am are my your You) x)) answer-change-pronoun trite-expression)

(list (lambda (x) (intersect? (keywords KEYWORDS-LIST) x)) keywords-strategy)))

;Задание №6 2. Пробегаемся по списку, выполняем предикат (первый элемент каждого подсписка),

;если #t, следовательно, выполняем рандомную функцию(вместо pick-random можно сделать выбор с некоторой вероятностью/весами),

;соответствующую данному предикату, иначе ничего не делаем

;После всегда переходим к другому подсписку

(define (execute-strategy pred-func-lst client-response)

(cond ((null? pred-func-lst) '())

(else (let ((pred-func (car pred-func-lst)))

(if ((car pred-func) client-response)

(let ((result ((pick-random (cdr pred-func)) client-response)))

(cond ((null? result) (execute-strategy (cdr pred-func-lst) client-response))

(else (append result (execute-strategy (cdr pred-func-lst) client-response)))

))

(execute-strategy (cdr pred-func-lst) client-response)

)))))

Вызов execute-strategy

(pick-random (execute-strategy pred-F response))

Комментарии к модифицированному вводу

Решение:

Основные функции doctor-read и doctor-print. При считываании действуем по следующей схеме: считываем всю строку с помощью read-line. Из строки делаем список элементов типа char; затем к этому списку применяем функцию convert-letters-lst-to-internal-presentation(См. Приложение), эта функция возвращает список списков, каждый внутренний подсписок - это отдельное предложение клиента, состоящее из элементов типа symbol, так было сделано, чтобы в доктор вносить небольшие изменения. Она последовательно просматривает символы: если встречается буква или цифра, то сохраняем в специальном списке word; если пробельный word сохраняем в sentence , word обнуляем; если встретился терминатор(| . |, | , |, !, ?), то заканчиваем предложение и добавляем его res(результат работы функции) и т.д.

Код:

Большая часть в Приложении

(define (doctor-read)

(begin

(let ((input (read-line)))

(reverse2 (convert-letters-lst-to-internal-presentation (string-to-letters-lst input))))))

(define (doctor-print lst)

(define (loop lst res)

(cond ((null? lst) res)

((null? (cdr lst)) (loop (cdr lst) (string-append res (doctor-string (car lst)))))

(else (loop (cdr lst) (string-append res (doctor-string (car lst)) " "))))

(loop lst ""))

Упражнение 7

Решение упражнения 7 отсутствует

Весна. Обучение

Основные функции(более подробное описание функций можно посмотреть в Приложении):

(train inputfile outputfile) – читаем файл inputfile с обучающим текстом, делаем граф и сохраняем результат в outputfile.

(extend-knowledges file1 file2 file3) - ситываем текст из file1, создаем graph1. Считываем текст из file2, создаем graph2. Сливаем эти графы и результат записываем в file3.

(merge-graphs graph1 graph2) – функция слияния двух графов.

(create-graph text) – создает граф из списка, элементы которого имеют тип symbol.

Структура данных:

Так как генерировать ответную фразу нужно уметь и прямым способом, и обратным, была выбрана следующая сруктура:

{ “word1” : [{“predword1”: frequency1, …, “predwordP1”: frequencyP1}

{“nextword1”: frequency1, …, “nextwordN1”: frequencyN1}]

…

“wordK” : [{“predword1”: frequency1, …, “predwordPK”: frequencyPK}

{“nextword1”: frequency1, …, “nextwordNK”: frequencyNK}]

}

{} - обозначение racket hash-table

[] - scheme-список

word1 …. wordK – это слова(каждое слово имеет тип symbol) из считанного текста – ключи хэш-таблицы.

Значение каждого ключа wordi – это список из двух хэш-таблиц. Первая хранит информацию о словах, предшествующих wordi, а вторая о словах, которые идут после wordi.

Frequency – количество появлений слова в тексте.

Таким образом, при генерации фразы(и прямым, и обратным способом, и смешанным) за константное время можно получить доступ к frequency-значению, а это значение влияет на выбранное слово.

Алгоритм обучения:

Считывание из файла производится с помощью doctor-read-from-string(которая практически совпадает с doctor-read, её мы рассмотрели первой главе). После считывания

работает функция create-graph, которая строит структуру, описанную в пункте *Структура данных.* Код можно посмотреть в приложении.

Работа с файлами:

Хранится результирующий граф в текстовом файле. Считывание тоже из текстового файл происходит.

(read-file filename) – считывание файла с обучающим тестом. Считываетсяя построчно, пока не будет достигнут EOF.

(save-graph graph filename) – запись графа в файл. Запись производится функцией print

(read-graph filename) – считывание графа из файла. Считывание производится функцией read

Обучающие тексты:

1. http://www.tandfonline.com/doi/full/10.1080/10481885.2015.1013826
2. http://www.tandfonline.com/doi/full/10.1080/00107530.2014.905903
3. http://www.tandfonline.com/doi/full/10.1080/1551806X.2015.979467
4. http://www.tandfonline.com/doi/full/10.1080/15228878.2013.856329

Выбранные тексты достаточно большие, больше 2500 слов. Каждый текст – это статья, специально не брались выдержки из книг по психоанализу, так как там слишком специфический язык.. Некоторые тексты собержат диалоги, для доктора это то, что нужно. Тексты программную обработку не проходили. Были убраны вручную(из-за малого количества) сокращения.

Весна. Генерация

Основные функции(более подробное описание функций можно посмотреть в Приложении):

(generator-hybrid-method word graph) – построение фразы смешанным способом. От текущего слова строится фраза в двух направлениях: к концу предложения(прямой способ) с помощью (generator-direct-method init-word graph), к началу(обратный способ) - с помощью (generator-reverse-method init-word graph).

(generator-direct-method init-word graph) – строит предложение от слова init-word до тех пор, пока не достигнет терминального символа, или длина не станет равной константе PHRASE_LENGTH_LIMIT. Внутри используется функция (take-next-word graph prev-word), которая, используя граф, выдает следующее за prev-word слово.

(take-next-word graph prev-word) – в 0.25 случаев выдает слово, которое чаще всего встречается после prev-word, в остальных произвольное слово, которое может следовать за prev-word. Произвольный выбор был сделан, чтобы всегда не выдавалось слово максимальной длины.

(generator-reverse-method init-word graph) – строит предложение от слова init-word к до тех пор, пока не достигнет терминального символа, или длина не станет равной константе PHRASE_LENGTH_LIMIT. Но теперь в отличие от прямого способа используется функция

(take-prev-word graph next-word), которая, используя граф, выдает предыдущее за next-word слово.

(take-prev-word graph next-word) – работает так же , как и (take-next-word graph prev-word), только смотрим на предыдущие слова.

Результаты

Доктор имеет 5 стратегий выбора ответа:

I) Ответ произвольной фразой из заранее заданного списка – функция trite-expression

Удачные реплики:

***I have recently thought about suicide*

How long have you had these thoughts

Неудачные реплики:

***How are you?*

many of my patients have told me the same thing

II) Замена первого лица на второе во фразе клиента - функция answer-change-pronoun

Удачные реплики:

***You are being very helpful to me*

We are clearly in the right direction, if you said I am being very helpful to you

Неудачные *реплики*:

***I like rain*

what do you mean when say you like rain

III) Генерация ответа, используя историю сообщений - функция answer-old-phrase

Удачные реплики:

***Can you help me*

early you said that you have recently thought about suicide

Неудачные реплики:

***I like rain*

early you said that you have recently thought about suicide

IV) Генерация ответа в зависимости от ключевого слова во фразе клиента функция keyword-strategy

Удачные реплики:

***Let's talk about my mother*

tell me more about your mother

***When i was caught in the snow, i thought about suicide.*

What thoughts arise for you when the snow?

Неудачные реплики:

***When i was caught in the snow, i thought about suicide.*

Do you believe in what you said

V) Формирование ответа при помощи генератора из задания “Весна” - функция generate-phrase

Удачные реплики:

***Let' talk about my problem*

The problem is how our own separate and the vision of the world within the traumatic bifurcation with the beginning of the present day the of so - called mental illness, it up.

Неудачные реплики:

***Can you say me some interesting story?*

Sharon's idea, as to generate the next time I was not an exposure to see, Gergely, but also be more interesting, jurist, and the job of myself?

Комментарии по качеству работы “Доктора”

Иногда “Доктор” может давать ответ, связанный с вопросом клиента, но это скорее всего исключение, чем правило. Доктор имеет достаточно простые стратегии I-IV – они очень слабо зависят от текущей фразы клиента: зависимость максимум в одном слове – одного слова мало для формирования хорошего ответа. Пятая стратегия тоже зависит от одного слова. Пятая стратегия может работать в двух режимах:

1. Хороший режим - генерировать связанное по смыслу хорошее предложение(в этом случаем скорее всего это предложение просто вырвано из обучающего текста либо состоит из больших вырванных из исходного текста кусков).

2. Плохой режим – сгенерированная фраза не связана по смыслу.

Анализ влияния обучения на качество генерируемых реплик

Так как сам подход в генерации результирующей фразы зависит от одного слова из предложения клиента, то качество ответа не зависит от количества текстов, на которых обучился доктор(См. пункт Комментарии по качеству работы “Доктора”). Чем больше обучающих текстов, тем более разнообразные будут ответы доктора. Они будут более осмысленными, но связь с предложениями клиента не появится.

Заключение

Доктор имеет разные стратегии генерации ответа, но эти стратегии очень редко позволяют получить осмысленный ответ, так как осмысленный ответ может быть получен только тогда, когда генерация ответной фразы зависит от нескольких клиентских слов, в идеале от контекста, но это либо невозможно сделать, либо очень сложно.

Код состоит из следующих основных частей:

I) Input – модифицированный ввод/вывод

II) Trainer – функции обучения

III) Generator - функции генерации ответа

IV) Doctor – основные функции “Доктора”

Код получился достаточно большой, так как были написаны много вспомогательных функций, например, проверки символов на принадлежность к определенным множестам(ко множеству цифр, латинских букв, терминальных символов).

Дополнительная литература и статьи не использовались, кроме статей с обучающей выборкой.

Приложение

```
#lang scheme/base
==================================INPUT==============================
(define (string-head string1)
  (cond ((> (string-length string1) 0) (substring string1 0 1))
        (else "")))

(define (string-tail string1)
  (substring string1 1 (string-length string1)))

;create list of letters from str
;there is built-in function: (string->list str) -> (listof char?)
;(string-ref str k) -> char? возвращает символ строки в позиции k
(define (string-to-letters-lst str)
  (let ((letter (string-head str)))
    (if (equal? letter "") '()
      (cons (string-ref letter 0) (string-to-letters-lst (string-tail str))))))

;convert list of letters to string
;there is built-in function: (list->string lst) -> string?
(define (convert-letters-lst-to-string lst)
  (if (null? lst) ""
     (string-append (string (car lst)) (convert-letters-lst-to-string (cdr lst)))))

(define (exist? word lst)
  (and (not (null? lst))
       ( or (equal? word (car lst))
            (exist? word (cdr lst))
       )))

(define (reverse2 lst)
```

```
  (define (loop lst res)
    (if (null? lst) res
        (loop (cdr lst) (cons (car lst) res)
              )))
  (loop lst '()))


;'.' ',' ';' in quota presentation -> |,| |.| |;|
(define Separators '(#\, #\- #\: #\; #\'))
(define Terminators  '(#\. #\! #\?))
(define EnglishLetter '(#\a #\b #\c #\d #\e #\f #\g #\h #\i #\j #\k #\l #\m #\n #\o #\p #\q #\r #\s #\t #\u #\v #\w #\x #\y #\z
                        #\A #\B #\C #\D #\E #\F #\G #\H #\I #\J #\K #\L #\M #\N #\O #\P #\Q #\R #\S #\T #\U #\V #\W #\X #\Y #\Z))
(define Digits '(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9))


(define (isalpha? letter)
  (exist? letter EnglishLetter))


(define (isdigit? letter)
  (char-numeric? letter))


(define (isseparator? letter)
  (exist? letter Separators))


(define (isterminators? letter)
  (exist? letter Terminators))


(define (isspace? letter)
  (char-whitespace? letter))


(define (incorrectletter? letter)
```

```
  (not (or (isspace? letter) (isalpha? letter) (isdigit? letter) (isseparator? letter) (isspace? letter))))

(define (char->sym char)
  (string->symbol (string char)))

(define (last-elem lst)
  (if (null? (cdr lst)) (car lst) (last-elem (cdr lst))))

;lst - list of symbols
(define (last-is-term? lst)
  (if (null? lst) #f
      (let ((back (symbol->string (last-elem lst))))
        (if ( > (string-length back) 1) #f (exist? (string-ref back 0) Terminators)))))

;lst - list of chars
(define (first-is-term? lst)
  (if (null? lst) #t
      (or (isterminators? (car lst)) (incorrectletter? (car lst)))))

;(char-alphabetic? char) <-- return true if letter (Unicode)
;(char-numeric? char) <-- retunr true if digit (Unicode)
;!!!there is built-in function: (string->symbol str) -> symbol?   <----conversion string to quota

(define (create-word word)
  (if (null? word) '()
      (list (string->symbol (list->string (reverse word))))))

(define (convert-letters-lst-to-internal-presentation lst)
  (define (loop lst word seq res)
    (cond ((null? lst) (let ((last-seq (append seq (create-word word))))
                         (if (null? last-seq) res (cons last-seq res))))
```

```
        ((isalpha? (car lst)) (loop (cdr lst) (cons (car lst) word) seq res))
        ((isdigit? (car lst)) (loop (cdr lst) (cons (car lst) word) seq res))
        ((isseparator? (car lst)) (loop (cdr lst) '() (append seq (create-word word) (list (char->sym (car lst)))) res))

        ((isterminators? (car lst))
          (if (first-is-term? (cdr lst)) (loop (cdr lst) '() (append seq (create-word word) (list (char->sym (car lst)))) res )
              (loop (cdr lst) '() '() (cons (append seq (create-word word) (list (char->sym (car lst)))) res))
          )
        )
        ;((isterminators? (car lst)) (loop (cdr lst) '() '() (cons (append seq (create-word word) (list (char->sym (car lst)))) res)))
        ((isspace? (car lst)) (loop (cdr lst) '() (append seq (create-word word)) res))
        (else  ;(if (last-is-term? seq) (loop (cdr lst) '() '() (cons (append seq (create-word word) (list (char->sym (car lst)))) res))
               ;    (loop (cdr lst) word seq res)
               ;)
          (cond ((and (last-is-term? seq) (first-is-term? (cdr lst))) (loop (cdr lst) word seq res))
                ((and (last-is-term? seq) (not (first-is-term? (cdr lst)))) (loop (cdr lst) '() '() (cons (append seq (create-word word)) res)))
                ((and (not (last-is-term? seq)) (first-is-term? (cdr lst))) (loop (cdr lst) word seq res))
                (else (loop (cdr lst) word seq res)))))
  (loop lst '() '() '()))

(define (doctor-read)
  (begin
    (let ((input (read-line)))
      (reverse2 (convert-letters-lst-to-internal-presentation (string-to-letters-lst input)))
      )))

(define (doctor-read-from-string str)
```

```
  (reverse2 (convert-letters-lst-to-internal-presentation (string-to-letters-lst str))))

(define Punc (list "." "," ":" ";" "." "!" "?"))
(define WithoutSpace (list """))

(define (punctuation? str)
  (if (> (string-length str) 1) #f
      (exist? str Punc)))

(define (withoutspace? str)
    (if (> (string-length str) 1) #f
      (exist? str WithoutSpace)))

(define (doctor-string lst)
  (define (loop lst res)
    (cond ((null? lst) res)
          ;((null? (cdr lst)) (loop (cdr lst) (string-append res " " (symbol->string (car lst)))))
          ;((punctuation? (symbol->string (cadr lst))) (loop (cdr lst) (string-append res (symbol->string (car lst)))))
          ((or (punctuation? (symbol->string (car lst))) (withoutspace? (symbol->string (car lst)))) (loop (cdr lst) (string-append res (symbol->string (car lst)))))
          (else
           (if (withoutspace? (string (string-ref res (- (string-length res) 1))))   (loop (cdr lst) (string-append res (symbol->string (car lst))))
               (loop (cdr lst) (string-append res  " " (symbol->string (car lst))))))))
  (if (null? lst) ""
      (loop (cdr lst) (symbol->string (car lst)))))

(define (doctor-print lst)
  (define (loop lst res)
    (cond ((null? lst) res)
          ((null? (cdr lst)) (loop (cdr lst) (string-append res (doctor-string (car lst)))))
```

```
        (else (loop (cdr lst) (string-append res (doctor-string (car lst)) " "))))
   (loop lst ""))

(define (input-output)
  (doctor-print (doctor-read)))

================================TRAINER================================
============================get-text-from-lst-of-lst============================
(define (concate lst1 lst2)
  (define (loop lst1 lst2 res)
    (cond ((not (null? lst1)) (loop (cdr lst1) lst2 (cons (car lst1) res)))
          ((null? lst2) res)
          (else (loop '() (cdr lst2) (cons (car lst2) res)))
          ;alternative variant
          ;((not (null? lst2)) (loop '() (cdr lst2) (cons (car lst2) res)))
          ;(else res)))
  (reverse2 (loop lst1 lst2 '())))

(define (convert-to-lst lst-of-lst)
    (define (loop lst-of-lst res)
      (if (null? lst-of-lst) res
          (loop (cdr lst-of-lst) (concate res (car lst-of-lst)))))
  (loop lst-of-lst '()))

================================create-graph================================
(define (add-elem-to-hash hash elem)
  (if (hash-has-key? hash elem)
      (begin (hash-set! hash elem (+ (hash-ref hash elem) 1)) hash)
      (begin (hash-set! hash elem 1) hash)))

(define (add-new-word hash-table prev curr next)
```

```
  (if (hash-has-key? hash-table curr)
      (let ((lst (hash-ref hash-table curr)))
        (begin
          (add-elem-to-hash (car lst) prev)
          (add-elem-to-hash (cadr lst) next)
          hash-table))
      (begin
        (hash-set! hash-table curr (list (make-hash) (make-hash)))
        (hash-set! (car (hash-ref hash-table curr)) prev 1)
        (hash-set! (cadr (hash-ref hash-table curr)) next 1)
        hash-table)))

(define (create-graph text)
  (define (loop text hash-table prev)
    (cond ((= (length text) 1) (add-new-word hash-table prev (car text) '|.|))
          ;((= (length text) 2)   (add-new-word hash-table prev (car text) (cadr text)))
          (else (loop (cdr text) (add-new-word hash-table prev (car text) (cadr text)) (car text)))))
  ;Предполагаем, что в начале каждого текста стоит точка. Нужно добавить ее в словарь. Для этой начальной точки prev будет тоже точкой
  (add-new-word (loop text (make-hash) '|.|) '|.| '|.| (car text)))

==============================train-from-all==============================
(define (train-from-stdin)
  (create-graph (convert-to-lst (doctor-read))))

;text <-> result from doctor-read function, i.e. list of sequences(list of list)
(define (train-from-text text)
  (create-graph (convert-to-lst text))
)

;Read file line by line
```

```
(define (read-file filename)
  (define (loop file res)
    (let ((str (read-line file)))
       (if (eof-object? str) res
          (loop file (string-append res str "\n")))))
    (loop (open-input-file filename) ""))

;Read file and create graph
(define (train-from-file inputfile)
  (train-from-text (doctor-read-from-string (read-file inputfile))))

;if file - output is already exist then -> raise exception
(define (save-graph graph filename)
  (begin
    (define out (open-output-file filename #:exists 'truncate))
    (print graph out)
    (close-output-port out)))

(define (read-graph filename)
  (begin
    (define in (open-input-file filename))
    (define graph (read in))
    (close-input-port in)
    graph
  ))

;APPEND VALUE TO SIMPLE HASH
(define (add-value-to-simple-hash key value hash)
  (if (not (hash-has-key? hash key)) (begin (hash-set! hash key value) hash)
      (let ((old-value (hash-ref hash key)))
        (begin (hash-set! hash key (+ value old-value)) hash))))
```

```
;Merge two hash-> {key1:value1, ... keyN:valueN} keys - symbols values - interger number
(define (merge-hash src dst)
  (define (loop src src-keys res)
    (if (null? src-keys) res
        (let ((key (car src-keys)) (value (hash-ref src (car src-keys))) )
           (loop (begin (hash-remove! src key) src) (cdr src-keys) (add-value-to-simple-hash key value dst)))))
  (loop src (hash-keys src) dst))

;ADD ELEMENT TO HASH
;[{} {}] -> value
(define (add-value-to-hash key value hash)
  (if (not (hash-has-key? hash key)) (begin (hash-set! hash key value) hash)
      (let ((prev (car (hash-ref hash key))) (next (cadr (hash-ref hash key))))
        (begin (hash-set! hash key (list (merge-hash (car value) prev) (merge-hash (cadr value) next))) hash))))

;MAIN
;Сливаем graph1 в graph2
(define (merge-graphs graph1 graph2)
  (define (loop graph graph-key res)
    (if (null? graph-key) res
     (let ((key (car graph-key)) (value (hash-ref graph (car graph-key))))
      (loop (begin (hash-remove! graph key) graph) (cdr graph-key) (add-value-to-hash key value res)))))
  (loop graph1 (hash-keys graph1) graph2))

;MAIN
;Read file - inputfile, create graph and save it into outputfile
(define (train inputfile outputfile)
  (save-graph (train-from-file inputfile) outputfile))
```

```
;Считываем текст из file1, создаем graph1. Считываем текст из file2, создаем graph2. Сливаем эти графы и результат записываем в file3.
(define (extend-knowledges file1 file2 file3)
  (save-graph (merge-graphs (train-from-file file1) (train-from-file file2)) file3))
================================================================================
(define (pick-random lst)
  ( cond ((null? lst) '())
        ( else (list-ref lst (random (length lst))))))
==================================GENERATOR============================
;Генератор должен быть встроен в "Доктор"
;Считывание из файла см. (read-graph filename)
(define PHRASE_LENGTH_LIMIT 50)

(define (find-word-with-weight hash weight)
  (define (loop keys)
   (cond ((null? keys) '|.|)
        ((=(hash-ref hash (car keys)) weight) (car keys))
        (else (loop (cdr keys)))))
  (loop (hash-keys hash)))

(define (low-register word)
  (let ((list-of-char (string->list (symbol->string word))))
    (string->symbol (list->string (cons (char-downcase (car list-of-char)) (cdr list-of-char))))))

(define (up-register word)
  (let ((list-of-char (string->list (symbol->string word))))
    (string->symbol (list->string (cons (char-upcase (car list-of-char)) (cdr list-of-char))))))

;direct generation
(define (check-register word next-word)
```

```
        (if (end? word)
            (up-register next-word)
           (begin
            (low-register word)
            (low-register next-word))))

;Так как мы начинаем с точки и сами генерируем фразу, следовательно, слово word в графе
;всегда существует, и проверять его наличие не нужно
(define (take-next-word graph word)
  (begin (define next-words (cadr (hash-ref graph word)))
         (define max-weight (apply max (hash-values next-words)))
         (if (<= (random 20) 5)
           (find-word-with-weight next-words max-weight)
           (pick-random (hash-keys next-words)))))

;CHECK_WORD is terminator or not
(define (end? word)
  (if (=(string-length (symbol->string word)) 1) (isterminators? (string-ref (symbol->string word) 0))
      #f))

(define (phrase-complete? next-word phrase)
  (or (end? next-word) (>= (length phrase) PHRASE_LENGTH_LIMIT)))

;GENERATOR_DIRECT_METHOD
(define (generator-direct-method init-word graph)
  (define (result-phrase ph last-word)
    (if (end? last-word) (reverse2 (cons last-word ph))
        (reverse2 (cons '|.| (cons last-word ph)) )))
  (begin
    (define (loop prev-word res)
      (let ((next-word (take-next-word graph prev-word)))
```

```
      (if (phrase-complete? next-word res) (result-phrase res next-word)
          (loop next-word (cons (check-register prev-word next-word) res)))))
   ;(loop '|.| '())
   (loop init-word '())))
```

==========

```
(define (get-terminators-from-graph graph)
  (define (loop terminators res)
    (if (null? terminators) res
        (loop (cdr terminators)
              (if (hash-has-key? graph (char->sym (car terminators)))
                  (cons (char->sym (car terminators)) res)
                  res))))
  (loop Terminators '()))

(define (take-prev-word graph word)
  (begin (define prev-words (car (hash-ref graph word)))
         (define max-weight (apply max (hash-values prev-words)))
         (if (<= (random 20) 5)
             (find-word-with-weight prev-words max-weight)
             (pick-random (hash-keys prev-words)))))

;GENERATOR_REVERSE_METHOD
(define (generator-reverse-method init-word graph)
  (define (result-ph ph last-word)
    (cond ((null? ph) (cons last-word ph))
          ((not (list? ph)) (cons last-word ph))
          (else   (if (end? last-word) (cons (up-register (car ph)) (cdr ph))
                      (cons (up-register last-word) ph)
                  )
          )
    )
```

```
  )
  (begin
    (define (loop next-word res)
      (let ((prev-word (take-prev-word graph next-word)))
        (if (phrase-complete? prev-word res) (result-ph res prev-word)
            (loop prev-word (cons prev-word res))
        )
      )
    )
    ;(define last-word (pick-random (get-terminators-from-graph graph)))
    ;(loop  last-word (list last-word))
    (loop  init-word '())
  )
)

;GENERATOR_HYBRID_METHOD
(define (generator-hybrid-method word graph)
  (append (generator-reverse-method word graph) (cons word (generator-direct-method word graph)))
)
;================================DOCTOR================================
(define (visit-doctor)
  (define (doctor-driver-loop name old-phrases)
    (define (reply user-response)

      (define (change-person pairs phrase)
        (many-replace pairs phrase))

      (define (qualifier)
        (pick-random '((you seem to think)
                       (you feel that)
```

```
            (why do you believe)
            (why do you say)
            (what do you mean when say)
            (You said)
            (It |'| s very interesting What can you say except)
            (We are clearly in the right direction |,| if you said)))
    )

    (define (hedge)
      (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)
            (can you say in more detail about your problem)
            (Do you believe in what you said)
            (We are on the right track)
            (If you need anything |,| just ask)
            (Make yourself at home)
            (How long have you had these thoughts)
            (If you could wave a magic what positive changes would you make happen in your
life)))
    )

    (define (fifty-fifty)
      ( cond ((null? old-phrases) (random 3) )
            (else (random 4)))
    )

    ;Задание №5. Список списков вида ( (key1 ... keyN) (seq1) ... (seqM) )
    (define KEYWORDS-LIST
      '( ((depressed suicide)
```

```
      (when you feel depressed |,| go out for ice cream)
      (depression is a disease that can be threated)
    )
    ((mother father family parents)
     (tell me more about your *)
     (why do you feel that way about your * ?)
     (Are your * good?)
    )
    ((rain snow)
     (Do you like *)
     (What thoughts arise for you when the * ?)
    )
   )
)

(define (keywords keywords-list)
  (cond ((null? keywords-list) '())
        (else (append (caar keywords-list) (keywords (cdr keywords-list )) ))
  )
)

(define (intersect? lst1 lst2)
  (and  (not(null? lst1))
        (not(null? lst2))
        (or (exist? (car lst1) lst2)
        (intersect? (cdr lst1) lst2))
  )
)

(define (intersectOLD? lst1 lst2)
  (cond ((null? lst1) #f)
```

```
        ((null? lst2) #f)
        ((exist? (car lst1) lst2) #t)
        (else (intersectOLD? (cdr lst1) lst2))
    )
  )

  ;Задание №5. 6. Заменяем в списке все '* на word и возвращаем полученный список в качестве результата
  (define (replace-all-words-in-lst word lst)
    (cond ((null? lst) lst)
        ((equal? '* (car lst)) ( cons word (replace-all-words-in-lst word (cdr lst))))
        (else (cons (car lst) (replace-all-words-in-lst word (cdr lst))))
    )
  )

;Задание №5. 5. Проверяет наличие word в списке lst
  (define (exist? word lst)
     (and (not (null? lst))
        ( or (equal? word (car lst))
            (exist? word (cdr lst))
        )
     )
  )

  (define (existOLD? word lst)
    (cond ((null? lst) #f)
        ((equal? word (car lst)) #t)
        (else (existOLD? word (cdr lst)))
     )
  )
```

```
;Задание №5. 4. Проверяем наличие word в первом подсписке keys, если есть,
;то возвращаем рандомный подсписок, отличный от первого(в котором хранятся ключевые слова)
;Если во втором подсписке списка keys есть символ '*, то выбираем произвольную фразу из (cdr keys)
;и вызываем функцию replace-all-words-in-lst
;!!!!NEW      ;Используем уже написанную функцию many-replace для замены * на ключевое слово
(define (check-keys word keys)
  (cond ((exist? word (car keys))
         (cond ((exist? '* (cadr keys)) (many-replace (list(list '* word)) (pick-random (cdr keys))))
               ( else (pick-random (cdr keys)))
         )
        )
        (else '())
  )
)

;Задание №5. 3. Для каждого keys из keyslst вызываем функцию check-all-keys
(define (check-all-keys keyslst word)
  (cond ((null? keyslst) '())
        (else (let ((result (check-keys word (car keyslst))))
                ( if (null? result) (check-all-keys (cdr keyslst) word)
                     result
                )
              )
        )
  )
)


;Задание №5. 2.Пробегаемся по всем словам из ответа клиента вызываем функцию check-
```

key-words

```
(define (check-key-words lst keyslst)
  (cond ((null? lst) '())
        (else (let ((result (check-all-keys keyslst (car lst))))
                (if (null? result) (check-key-words (cdr lst) keyslst)
                    (cons result (check-key-words (cdr lst) keyslst))
                )
              )
        )
  )
)
```

;Все функции, реализующие различные стратегии ответа должны иметь функции обертки, которые имеют один входной параметр user-response

;Задание №5. 1. Обертка для check-key-words

```
(define INPUT_FOR_GENERATOR "/home/andrew/Dropbox/7SEMESTR/FUNC_PROG/Task_2_Vesna/part2/2.txt")

(define (keywords-strategy client-response)
  (check-key-words client-response KEYWORDS-LIST)
)

(define (trite-expression client-response)
  (list
   (hedge)
   )
)

(define (answer-old-phrase client-response)
  (list
   (cond ((null? old-phrases) '(You haven't said anything before that))
```

```
        (else (append '(early you said that) (change-person '((i you) (I you) (me you) (am are) (my your) (you i) (You I) (are am) (your my)) (pick-random old-phrases)))))
    )
  )

  (define (answer-change-pronoun client-response)
    (list
     (append (qualifier) (change-person '((i you) (I you) (me you) (am are) (my your) (you i) (You I) (are am) (your my)) client-response))
    )
  )

  ;Если все слова длины 1, следовательно, вернется точка
  (define (max-word lst)
    (foldr (lambda (a b) (if (> (string-length (symbol->string b)) (string-length(symbol->string a)) ) b a )) '|.| lst )
  )
  (define (generate-phrase client-response)
    (begin
      (define word (max-word client-response))
      (define graph (read-graph INPUT_FOR_GENERATOR))
      (if (hash-has-key? graph word)
          (generator-hybrid-method word graph)
          (generator-hybrid-method (pick-random (hash-keys graph)) graph))))

  ;Задание №6 1. Предикаты и соответствующий им список функций
  (define pred-F (  list ( list (lambda (x) (<= (length x) 5)) trite-expression answer-old-phrase)
                         ( list (lambda (x) (intersect? '(i you I me am are my your You) x) ) answer-change-pronoun trite-expression)
                         ( list (lambda (x) (intersect? (keywords KEYWORDS-LIST) x )) keywords-strategy)
                 )
```

```
)

;Задание №6 2. Пробегаемся по списку, выполняем предикат (первый элемент каждого подсписка),
;если #t, следовательно, выполняем рандомную функцию(вместо pick-random можно сделать выбор с некоторой вероятностью/весами),
;соответствующую данному предикату, иначе ничего не делаем
;После всегда переходим к другому подсписку
(define (execute-strategy pred-func-lst client-response)
  (cond ((null? pred-func-lst) '())
        (else (let ((pred-func (car pred-func-lst)))
                ( if ((car pred-func) client-response)
                     (let ((result ((pick-random (cdr pred-func)) client-response)))
                       (cond ((null? result) (execute-strategy (cdr pred-func-lst) client-response))
                             (else (append result (execute-strategy (cdr pred-func-lst) client-response))))
                     (execute-strategy (cdr pred-func-lst) client-response)))))

;New version - with predicates
(begin
  (let ((response (pick-random user-response)))
    (if (< (random 100) 20)
        (list (pick-random (execute-strategy pred-F response)))
        (list (generate-phrase response))))))

(define (unique-push element vector)
  (cond ((equal? element '() ) vector)
        ((null? vector) ( cons element vector))
        (else (let ((curr-element (car vector)))
                (cond ((equal? curr-element element) vector)
                      (else (cons curr-element (unique-push element (cdr vector)))))))))
```

```
  (define (update-history lst history)
    (if (null? lst) history
       (update-history (cdr lst) (unique-push (car lst) history))))

  (newline)
  (print '**)
  (let ((user-response (doctor-read)))
   (cond ((null? user-response) (begin (printf "Please continue\n") (doctor-driver-loop name old-phrases)))
         ((equal? (doctor-print user-response) "goodbye" )
          (printf "Goodbye, ~a!\n" (doctor-print name))
          (print '(see you next week))
          ;(print old-phrases) Можно распечатать список всех ответов клиента
          (newline))
         (else (begin (printf "HISTORY -> ") (print old-phrases) (newline))
              (printf (doctor-print (reply user-response)))
              ;unique push| old-phrases is a set
              (doctor-driver-loop name (update-history user-response old-phrases) ))))

 )

 (define (ask-patient-name)
  (begin
  (print '(NEXT!))
  (newline)
  (print '(Who are you?))
  (doctor-read)))

 (define name (ask-patient-name))

 (cond ((equal? (doctor-print name) "suppertime") (print '(Time to sleep)))
```

```
        (else (begin
                (printf "Hello, ~a!\n" (if (null? name) "My friend"  (doctor-print name)))
                (print '(what seems to be the trouble?))
                (doctor-driver-loop (if (null? name) '((My friend)) name) '())
                (visit-doctor)))))

(define (replace replacement-pairs word)
  (cond ((null? replacement-pairs) word)
        ((equal? (caar replacement-pairs) word) (cadar replacement-pairs))
        (else (replace (cdr replacement-pairs) word ) )
  )
)

;New many-replace function
(define (many-replace replacement-pairs lst)
        (cond ((null? lst) '())
              (else (let ((current-word (car lst)))
                      ( cons (replace replacement-pairs current-word)  (many-replace replacement-pairs (cdr
lst)))))))
```