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

Отчёт по «Доктору»

Автор: Богатенкова Анастасия Олеговна
гр. 428

Москва, 2020

Содержание

1 Упражнения 1-7

Согласно описанию задания:

«Доктор» – это название программы, которая имитирует психоаналитика, ведущего диалог с пациентом. Программа принимает реплики пациента (в виде строк) и генерирует ответные реплики (также в виде строк).

Изначально в программе были реализованы две стратегии ответа доктора:

- замена лица во фразе пациента и добавление какого-то замечания в начало (*qualifier-answer*);
- выбор реплики из списка заранее определённых реплик (*hedge*).

Кроме того, изначально работа проводилась со списками символов, которые задавались с помощью "цитирования". Для более удобной работы с программой, все функции были переписаны так, чтобы работа проводилась в терминах строк.

1. *Задание упражнения №1* – добавить в каждую из перечисленных выше стратегий не менее трёх новых заготовленных фраз.

   Для этого в телах функций *qualifier-answer* и *hedge* при вызове функции *pick-random* в структуру-аргумент были добавлены новые строки (в случае *qualifier-answer*) или списки, состоящие из одной строки (в случае *hedge*).

2. *Задание упражнения №2* – написать новую версию функции *many-replace* с хвостовой рекурсией. Функция *(many-replace replacement-pairs lst)* осуществляет все замены в списке *lst* по ассоциативному списку *replacement-pairs*.

   Реализована функция *many-replace-1*: с помощью именованного let осуществляется проход по списку *lst* и накопление ответа в *result*. Голова списка ищется в ассоциативном списке *replacement-pairs*, если поиск был удачен, то в ответ добавляется замена, иначе в ответ помещается голова списка без изменений.

3. *Задание упражнения №3* – написать ещё одну версию функции *many-replace* так, чтобы тело новой версии состояло только из вызова map.

Реализована функция *many-replace-2*: тело функции, как требуется в задании, состоит только из вызова функции map. Первым аргументом map является анонимная функция, которая в случае успешного поиска в ассоциативном списке очередного элемента исходного списка возвращает замену, иначе возвращается сам элемент. В итоге возвращается список с произведенными заменами.

4. *Задание упражнения №4* – реализовать новую стратегию ответа (*history-answer*) на основе предыдущих реплик пациента.

   Для этого внутри цикла в функции *doctor-driver-loop* добавлен новый параметр *answers*, в котором накапливается история реплик пациента. Этот параметр также передается в функцию *reply*, которая может выбрать в качестве стратегии ответа функцию *history-answer*, если параметр *answers* не пуст.

   Функция *(history-answer answers)* возвращает список строк, состоящий из фразы *"earlier you said that"*, добавленной к произвольной фразе, выбранной из списка *answers* предыдущих реплик пациента (с заменой лица в реплике).

5. *Задание упражнения №5* – сделать программу многопользовательской, то есть добавить возможность принятия доктором нескольких пациентов. Кроме того, необходимо предусмотреть способ завершения работы программы либо после использования стоп-слова в качестве имени очередного пациента, либо при исчерпании количества принимаемых пациентов.

   Для этого в качестве аргументов основной функции *visit-doctor* были добавлены *stop-word* и *max-iter*. Стоп-слово *stop-word* – набор символов, переводящийся впоследствии в строку; число итераций *max-iter* – число итераций в цикле функции *visit-doctor*. На каждой итерации запрашивается имя пациента и сравнивается со стоп-словом, при совпадении имени пациента и стоп-слова или при *max-iter = 0* работа программы завершается, иначе вызывается функция *doctor-driver-loop*, после завершения которой работа программы продолжается (новая итерация цикла) со значением *max-iter* меньшим на единицу. Кроме того, добавлена функция *ask-patient-name*, осуществляющая ввод имени пациента.

6. *Задание упражнения №6* – реализовать ещё одну стратегию генерации ответных реплик, зависящую от ключевых слов в реплике пациента.

Для этого была добавлена специальная структура-список *templates*, в которой элементами являются списки следующего вида: список ключевых слов по определенной теме и список шаблонов для данных ключевых слов. Каждый шаблон – это список строк, среди которых может быть строка *"*"*, которая впоследствии заменяется на конкретное ключевое слово.

Кроме того, реализованы следующие функции:

- *(template-answer user-response keyword-list)* – функция, в которой *user-response* – ответ пациента в виде списка слов-строк, *keyword-list* – список ключевых слов, которые есть структуре *templates*. Функция находит ключевые слова, повстречавшиеся в реплике пациента, случайным образом выбирает из них одно слово. По ключевому слову строится список шаблонных ответных фраз, из которых случайным образом выбирается одна и все *"*"* в ней заменяются на выбранное ключевое слово. Полученная реплика в виде списка строк возвращается в качестве ответа.
- *(extract-keywords user-response keyword-list)* – функция, которая получает список всех ключевых слов, встретившихся в реплике пациента (с повторениями).
- *(make-templates-list keyword)* – функция, которая получает объединённый перечень всех шаблонов, относящихся к каждой группе, куда входит ключевое слово *keyword*.
- *(contains-keyword user-response keyword-list)* – функция-предикат, проверяющая, есть ли в реплике пользователя хотя бы одно ключевое слово из списка *keyword-list*.
- *get-keywords* – вычисляется один раз и возвращает список ключевых слов из структуры templates (без повторений). *get-keywords* используется в качестве *keyword-list* во всех предыдущих функциях.

7. *Задание упражнения №7* – создать обобщённую версию функции *reply*, которая будет работать с любым подаваемым ей на вход перечнем стратегий.

   Для этого была добавлена специальная структура-список *strategies*, элементами которой являются списки, содержащие следующие элементы:

1) анонимная функция-предикат, проверяющая, применима ли данная стратегия;
2) вес стратегии - натуральное число;
3) анонимная функция, вызывающая в своем теле функцию для определённой стратегии.

Для выбора стратегии с учетом веса была написана функция *(pick-random-with-weight lst-with-weights)*, где *lst-with-weights* – это список пар (вес, элемент), в данном случае элементом является выбираемая стратегия, а веса - это натуральные числа. В функции сначала вычисляется сумма всех весов и берется произвольное число от 0 до значения суммы. Затем осуществляется проход по списку пар и из полученного числа вычитается очередной вес до тех пор, пока текущий вес не станет больше текущего значения числа. После завершения прохода описанным образом возвращается текущий элемент, на котором был остановлен просмотр списка. Такой алгоритм позволяет учесть значение веса при случайном выборе элемента списка, так как чем больше вес, тем выше вероятность попасть в тот промежуток, который составляет часть суммы весов для конкретного элемента.

Наконец, была переписана функция *(reply user-response answers keywords strategies)*, где *user-response* – список строк, составляющих реплику пользователя; *answers* – история реплик пользователя; *keywords* – список ключевых слов из структуры *templates*; *strategies* – список стратегий, описанный выше. В теле функции сначала строится список стратегий, применимых на текущий момент (с помощью функций-предикатов из *strategies*), затем с помощью функции *pick-random-with-weight* с учетом веса выбирается стратегия и вызывается функция для конктретной стратегии, возвращающая ответную реплику доктора.

Помимо упражнений, описанных выше, необходимо было сделать ввод-вывод реплик в виде строк, а не списков символов, как это было сделано изначально. Для этого были использованы функции для ввода и вывода строк – *read-line* и *printf*. Текст реплики пациента может состоять из нескольких предложений, поэтому была реализована стратегия, при которой анализируется только первое предложение реплики. Чтобы не менять весь код целиком, сохранилась работа с репликой как со списком строк, при этом после прочтения реплики пациента строка превращалась в список строк, а перед

выводом ответа список строк переводился в строку. Для этого были написаны функции с использованием регулярных выражений:

- *(split-answer str)* – функция, с помощью которой строка разбивается на предложения по . ! ? и каждое предложение разбивается на слова по пробелам; знаки препинания выделяются в отдельные слова, все символы, кроме букв, цифр и знаков препинания убираются; возвращается список списков: слова для каждого предложения.

- *(join-answer lst)* – функция, которая соединяет список строк-слов в одну строку, при этом знаки препинания прибавляются без пробела слева.

2 Упражнение 8

Решение упражнения №8 отсутствует.

3 Весна. Обучение

Задание посвящено добавлению в «Доктор» новой стратегии построения ответов. Добавляемый способ состоит в генерации случайной последовательности слов. Для этого реализован генератор случайных фраз с помощью построения графов следования для текстов.

Далее под n-граммой будем понимать список, состоящий из n строк-слов.

В решении были использованы следующие структуры данных:

- *next-graph* – хеш-таблица с ключами в виде n-1-грамм – наборами слов, подряд встречающимися в тексте; значение – хеш-таблица, у которой ключ – слово, стоящее после данной n-1-граммы, значение – сколько раз встретилось это слово после данной n-1-граммы;

- *prev-graph* – хеш-таблица с ключами в виде n-1-грамм – наборами слов, подряд встречающимися в тексте; значение – хеш-таблица, у которой ключ – слово, стоящее перед данной n-1-граммой, значение – сколько раз встретилось это слово перед данной n-1-граммой;

- *begin-graph* – хеш-таблица с ключами в виде n-1-грамм – наборами слов, с которых начинаются предложения текста; значение – число, показывающее, сколько раз данная n-1-грамма встретилась в начале предложения;

- *end-graph* – хеш-таблица с ключами в виде n-1-грамм – наборами слов, которыми заканчиваются предложения текста; значение – число, показывающее, сколько раз данная n-1-грамма встретилась в конце предложения;

- *frequency-graph* – хеш-таблица с ключами в виде n-1-грамм – наборами слов, подряд встречающимися в тексте; значение – число, показывающее, сколько раз данная n-1-грамма встретилась в тексте.

Для заполнения перечисленных выше структур данных были реализованы следующие функции:

1. *(split-line line)* – разбивает строку на предложения, а предложения на слова аналогично функции разбиения на слова строки с репликой пациента;

2. *(read-loop input-file)* – построчно читает строки из файла с тренировочными текстами, разделяет каждую строку на предложения и добавляет в графы, описанные выше, всю необходимую информацию;

3. *(add-first-n-gram sentence)* – добавляет для данного предложения первую n-1-грамму в граф *begin-graph*, увеличивая счетчик n-1-грамм на единицу;

4. *(add-last-n-gram sentence)* – добавляет для данного предложения последнюю n-1-грамму в граф *end-graph*, увеличивая счетчик n-1-грамм на единицу;

5. *(add-sentence-to-next-graph sentence)* – добавляет для данного предложения все его n-1-граммы в граф *next-graph*, обновляет счетчик *frequency-graph*;

6. *(add-sentence-to-prev-graph sentence)* – добавляет для данного предложения все его n-1-граммы в граф *prev-graph*;

7. *(update-graph graph key value)* – вспомогательная функция для добавления n-1-граммы в графы следования;

8. *(update-frequency-graph graph key)* – вспомогательная функция для добавления n-1-граммы в графы-счётчики;

9. *(get-n-gram lst n)* – вспомогательная функция для выделения n-граммы (в обратном порядке) из списка: выделяется n первых элементов (в обратном порядке) из начала списка *lst*, если длина списка меньше, выделяется столько элементов, сколько есть в списке.

Для сохранения в файл и загрузки из файла полученных структур данных реализованы следующие функции:

1. *(save-structures output-file)* – сохраняет пять описанных выше графов в файл с именем *output-file*;

2. *(init-structures mode)* – инициализирует все необходимые структуры данных - либо формирует, вызывая функцию *read-loop*, либо читает графы из файла, вызывая функцию *load-structures*, в зависимости от значения *mode*;

3. *(load-structures input-file)* – читает пять описанных выше графов из файла с именем *input-file*.

В качестве тренировочных текстов были выбраны отрывки из книги психотерапевта В. Франкла «Человек в поисках смысла» [1] (для плохой базы) и тексты диалогов из EmpatheticDialogues [2] (для хорошей базы). Плохая база выбрана с целью показать, что есть возможность генерировать осмысленные фразы на тему психологии, однако реплики больше подходят для письменного, а не устного общения. Хорошая база подбиралась по теме «Любовь», фразы более похожи на ответ человека, участвующего в диалоге.

Тексты для обеих баз имеют достаточно большой размер – плохая база содержит 193136 слов, хорошая – 83410 слов. Тексты хорошей базы предварительно обрабатывались с помощью скрипта, написанного на языке python, были удалены лишние символы и предложения, не относящиеся к выбранной теме.

Построенные графы для плохой и хорошей баз сохранялись в текстовых файлах «bad_out.txt» и «good_out.txt» соответственно.

4 Весна. Генерация

По заданию необходимо было реализовать генератор, способный по результатам обучения строить фразы, состоящие из одного или более предложений, «прямым» и «смешанным» способами. Прямой способ подразумевает произвольный выбор какой-либо n-1-граммы из начала предложения и построение предложения на основе графа следования: очередное слово предложения выбирается произвольным образом из списка слов-значений в графе с учетом их веса, после этого первое слово из n-1-граммы удаляется, а в конец добавляется выбранное слово. Обратный способ построения предложения выбирает произвольную n-1-грамму из конца предложения и добавляет к ней слова, которые встречались перед ней. Смешанный способ подразумевает использование прямого и обратного методов – наборы слов из реплики пациента ищутся в списке всех n-1-грамм, и если подходящая n-1-грамма найдена, от нее в двух направлениях строится предложение-ответ.

Для реализации генератора были написаны следующие функции:

- *(direct-generator first-n-gramma)* – реализация прямого способа генерации реплик на основе первой n-1-граммы;
- *(reverse-generator first-n-gramma)* – реализация обратного способа генерации реплик;
- *(compound-generator sentence)* – реализация смешанного способа генерации реплик, *sentence* – предложение (список слов), на основе которого нужно построить ответную реплику. Функция работает следующим образом:
  - ищутся n-1-граммы, которые есть в графе всех n-1-грамм;
  - если таких n-1-грамм не найдено, ищутся n-1-граммы, в которых встречается максимальное число слов из предложения (порядок слов не учитывается);
  - если таких n-1-грамм не найдено, применяется прямой способ генерации;
  - если найдена n-1-грамма, наращивается ответ с двух сторон с помощью двух графов.

- *(pick-random-from-hash hash-with-weights)* – вспомогательная функция, аналогичная описанной выше функции *pick-random-with-weight*, в которой вместо ассоциативного списка в качестве аргумента передается хеш-таблица с весами в роли значений;

- *(find-best-reply sentence)* – вспомогательная функция для поиска n-1-грамм, в которых встречается максимальное число слов из предложения-реплики пациента. Из списка найденных n-1-грамм выбирается произвольная n-1-грамма, на основе которой генерируется предложение-ответ; если n-1-грамм не найдено, применяется прямой способ генерации на основе произвольно выбранной n-1-граммы.

Для предотвращения зацикливания при построении ответной реплики максимальная длина генерируемого ответа была ограничена 100 словами, однако такого значения длины предложения на практике получать не удавалось. Кроме того, перед тем, как применять описанный метод генерации ответа, к реплике пользователя применялась функция замены лица во фразе для того, чтобы сгенерированные реплики были более похожи на ответ.

Число n для размера n-грамм подбиралось эмпирически. Для плохой базы $n = 3$ (поэтому ответы могут быть слабо связаны с исходной репликой), для хорошей базы $n = 5$ (ответы могут хорошо соотвествовать теме разговора, однако вероятность совпадения части реплики пациента и какой-либо n-1-граммы становится ниже).

5 Результаты

В данной главе представлены фрагменты протоколов работы доктора. Доктору на вход подаются одинаковые фразы и результаты работы анализируются на соответствие теме разговора. Фразы пациента начинаются с «**».

5.1 Протокол работы доктора на «плохой базе»

> (visit-doctor 'bye 1)

next!

who are you?

**Nasty

Hello, Nasty!

what seems to be the trouble?

**i need to talk about my feelings

Сгенерированная фраза очень длинная и слабо соответствует теме

this is the appeal addressed to this: it seems to me, i involuntarily remembered the story of a divorced woman, he told me how prisoners, the other hand, a person is no work, it will be in doubt about your

**it is about love and relationship

Сгенерировалось что-то более осмысленное, но не слишком подходящее

it is sense of the case.

**i feel a lot of lonely last time

Стратегия №2

why do you say that you feel a lot of lonely last time

**sometimes i miss my friends

Генерация на основе двух слов, взятых в произвольном порядке, также наполовину осмысленный текст

the moment i shape my destiny, as we say that you can

**i miss my family, i haven't seen them for a long time

Стратегия №3

earlier you said that it is about love and relationship

**it is to hard to fall in love with somebody

Сгенерировалась очень длинная фраза

but if we talk about unconscious or subconscious spirituality such, it is not a single such person, we must emphasize over and over again, the concept of duty, managed to save her life meaningless is not devoid of meaning until the very possibility of course, one of the person may not be assumed that the irreplaceability and irreplaceability of a person is his psychological fate, and not to everyone's words: god is hard to spiritual suffering and mental traits of his books.

**but i have no regrets about the past

Стратегия №3

earlier you said that you feel a lot of lonely last time

**i were super brave in relationship but now i'm afraid

Стратегия №3

earlier you said that you feel a lot of lonely last time

**i have such good friends in my life

Сгенерировалось нечто глубокомысленное

psychologically, of course, you have you least according to their social destiny or not.

**but now nobody can help me with my trouble

Стратегия №2

why do you believe that but now nobody can help you with your trouble

**i hope you can understand the need

Стратегия №2

you feel that you hope i can understand the need

**recently i broke up with my boyfriend

Стратегия №1

it is interesting and i should think more about it

**i don't know if we can still be friends

Стратегия №3

earlier you said that but you have no regrets about the past

**goodbye

Goodbye, Nasty!

see you next week

time to go home

5.2 Протокол работы доктора на «хорошей базе»

> (visit-doctor 'bye 1)

next!

who are you?

**Nasty

Hello, Nasty!

what seems to be the trouble?

**i need to talk about my feelings

Сгенерировалось почти в тему, достаточно понятно и коротко

if you need anyone to talk to let me know.

**it is about love and relationship

Стратегия №2

the way i see it you mean to say that it is about love and relationship

**i feel a lot of lonely last time

Стратегия №3

earlier you said that it is about love and relationship

**sometimes i miss my friends

Не совсем уместная для доктора фраза, но очень подходит к теме разговора

you miss your buddy, huh.

**i miss my family, i haven't seen them for a long time

Короткая фраза, неудачная по соответствию теме

yall must have been friends for a long time.

**it is to hard to fall in love with somebody

Стратегия №2

you feel that it is to hard to fall in love with somebody

**but i have no regrets about the past

Стратегия №3

earlier you said that it is about love and relationship

**i were super brave in relationship but now i'm afraid

Удачный ответ, соответствующий теме

you were, i'm glad it sounds like you were super brave.

**i have such good friends in my life

Стратегия №2

why do you believe that you have such good friends in your life

**but now nobody can help me with my trouble

Стратегия №2

why do you believe that but now nobody can help you with your trouble

**i hope you can understand the need

Удачный ответ, соответствующий теме

i can understand the feeling.

**recently i broke up with my boyfriend

Удачный ответ, соответствующий теме

sometimes you just have to be reminded why you broke up.

**i don't know if we can still be friends

Неудачная по соответствию теме фраза

i have a lot of safety features you don't have a car to drive or the real need to.

**goodbye

Goodbye, Nasty!

see you next week

time to go home

Таким образом, реплики доктора, полученные на основе хорошей базы, получаются более короткими и соответствующими теме разговора. Это объясняется тем, что средняя длина предложений плохой базы существенно больше средней длины для хорошей базы, а размер n-1-грамм для графов хорошей базы больше размера n-1-грамм для графов плохой базы.

Заключение

Таким образом, работа программы «Доктор» была расширена добавлением трех новых способов построения реплик. Кроме того, был реализован ввод-вывод реплик доктора и пациента в виде строк.

Были добавлены следующие стратегии:

- сохранение истории реплик пациента и построение ответа на основе этой истории;
- построение ответа по ключевым словам, найденным в реплике пациента;
- генерация предложения-ответа на основе слов из реплики пациента, которые были найдены в заранее построенных графах предыдущих и следующих слов.

Список литературы

[1] *Frankl, Viktor E.* Mans Search for Meaning / Viktor E. Frankl. — 2nd Edition edition. — Beacon Press, 1947.

[2] EmpatheticDialogues. `https://github.com/facebookresearch/EmpatheticDialogues`.

Приложение. Код программы

Упражнения по "Доктору": файл doctor.rkt

```
#lang scheme/base
#lang scheme/base
(require racket/string)
(require racket/format)
(require racket/require)
(require "spring.rkt")
(init-structures 'test)

(define (visit-doctor stop-word max-iter)
  (let loop ((name (ask-patient-name))
             (iter max-iter))
    (cond ((or (= iter 0) (equal? name (~a stop-word)))
           (printf  "time to go home\n"))
          (else (printf "Hello, ~a!\n" name)
                (printf "what seems to be the trouble?")
                (doctor-driver-loop name)
                (loop (if (= iter 1) name (ask-patient-name))
                      (- iter 1))
                )
          )
    )
  )

(define (split-answer str)
  (foldl (lambda (x y) (let ((sentence
                              (filter
                               (lambda (z) (not (equal? z "")))
                               (regexp-split #px"\\s+" x))))
                         (if (null? sentence) y
```

```
                                (cons sentence y))))
          null
          (regexp-split #px"[\\.\\?!]"
                        (regexp-replace* #px"([\\)\\(;,:'])"
                        (regexp-replace* #px"[^\\w\\.\\)\\(
\\?!;,:'\\s\\-]+" str "") " \\1 ")))
  )

(define (join-answer lst)
  (regexp-replace* #px"([\\)']) "
                   (regexp-replace* #px" ([\\);,:'\\.])"
                                    (string-join lst) "\\1")
                   "\\1")
  )

(define (ask-patient-name)
  (begin
    (printf "next!\n")
    (printf "who are you?\n")
    (print '**)
    (read-line)
    )
  )

(define (doctor-driver-loop name)
  (let loop ((name name) (answers null)
                         (keywords get-keywords)
                         (strategies strategies))
    (newline)
    (print '**)
    (let ((user-response (read-line)))
      (cond
```

```
        ((equal? user-response "goodbye")
         (printf "Goodbye, ~a!\n" name)
         (printf "see you next week\n"))

        (else (let ((answer (split-answer user-response)))
                (printf (join-answer (reply (car answer)
                                            answers
                                            keywords
                                            strategies)))
                (loop name (foldl cons answers answer)
                      keywords
                      strategies))
              )
        )
      )
    )
  )

(define (reply user-response answers keywords strategies)
  (let ((possible-strategies (foldl (lambda (x result)
                                      (if ((car x)
                                           user-response
                                           answers
                                           keywords)
                                          (cons (cdr x) result)
                                          result))
                                    null
                                    strategies)))
    (if (and (not (null? possible-strategies))
             (not (null? (cdr possible-strategies))))
        ((pick-random-with-weight possible-strategies)
         user-response answers keywords)
```

```
        ((car possible-strategies)
         user-response answers keywords)
        )
    )
  )

(define (qualifier-answer user-response)
  (cons (pick-random '("you seem to think that"
                       "you feel that"
                       "why do you believe that"
                       "why do you say that"
                       "i would like to know more
about your story because"
                       "if i am not mistaken you said that"
                       "the way i see it you mean to say that")
                     )
        (change-person user-response)
        )
  )

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

(define (change-person phrase)
  (many-replace-2 '(("am" "are")
                    ("are" "am")
                    ("i" "you")
                    ("me" "you")
                    ("mine" "yours")
                    ("my" "your")
                    ("myself" "yourself")
```

```
                    ("you" "i")
                    ("your" "my")
                    ("yours" "mine")
                    ("yourself" "myself"))
                  phrase)
  )

(define (many-replace replacement-pairs lst)
  (cond ((null? lst) lst)
        (else (let ((pat-rep (assoc (car lst) replacement-pairs)))
                (cons (if pat-rep (cadr pat-rep)
                          (car lst)
                          )
                      (many-replace replacement-pairs (cdr lst))
                      )
                )
              )
        )
  )

(define (many-replace-1 replacement-pairs lst)
  (let loop ((lst lst) (result null))
    (cond ((null? lst) (reverse result))
          (else (let ((pat-rep (assoc (car lst)
                                      replacement-pairs)))
                  (loop (cdr lst)
                        (cons (if pat-rep (cadr pat-rep)
                                  (car lst))
                              result)
                        )
                  )
                )
```

```
          )
    )
  )

(define (many-replace-2 replacement-pairs lst)
  (map (lambda (x)(let ((pat-rep (assoc x replacement-pairs)))
                    (if pat-rep (cadr pat-rep)
                        x)))
       lst)
  )

(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")
                 ("i completely understand you")
                 ("would you like to tell me more about it?")
                 ("it is interesting and
i should think more about it")
                 )
               )
  )

(define (history-answer answers)
  (cons "earlier you said that"
        (change-person (pick-random answers))
        )
  )

(define (template-answer user-response keyword-list)
```

```
  (let ((keyword
         (pick-random
          (extract-keywords user-response keyword-list)
          )
         )
        )
    (many-replace-2
     (list (list "*" keyword))
     (pick-random
      (make-templates-list keyword)
      )
     )
    )
  )

(define (extract-keywords user-response keyword-list)
  (filter (lambda (x)(member x keyword-list)) user-response)
  )

(define (make-templates-list keyword)
  (foldl append
         null
         (map (lambda (x) (cadr x))
              (filter (lambda (y)(member keyword (car y)))
                      templates)))
  )

(define (contains-keyword user-response keyword-list)
  (ormap (lambda (y)
           (ormap (lambda (x) (equal? x y))
                  keyword-list))
         user-response)
```

```
  )

(define templates
  '(
    (
     ("depressed" "suicide" "exams" "university")
     (
      ("when you feel depressed, go out for ice cream")
      ("depression is a disease that can be treated")
      ("your life is more important than studying")
      ("do not think about" "*" "so much")
      )
     )
    (
     ("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" "*" "?")
      ("does your" "*" "make you unhappy?")
      ("have your any difficulties with your" "*" "?")
      )
     )
    (
     ("university" "scheme" "lections" "exams")
     (
      ("your education is important")
      ("how many time do you spend to learning?")
      ("you think so much about your" "*")
      ("do you think that" "*" "is really important for you?")
      )
```

```
    )
   (
    ("love" "passion" "tenderness" "affection" "fondness")
    (
     ("*" "is great feeling")
     ("were you happy when you felt" "*" "?")
     ("all people love somebody")
     ("many people said me that they felt" "*")
     )
    )
   (
    ("hatred" "disgust" "aversion" "contempt")
    (
     ("how often do you feel" "*" "?")
     ("try not to feel in that way")
     ("you should be more patient")
     ("all people make some mistakes, try not to judge them")
     )
    )
   )
  )

(define get-keywords
  (foldl (lambda (x y) (append (filter
                                (lambda (z) (not (member z y)))
                                x) y))
         null
         (map car templates))
  )

(define (pick-random-with-weight lst-with-weights)
  (let ((weight-sum (foldl
```

```
                    (lambda (x y) (+ (car x) y))
                    0 lst-with-weights)))
    (let loop ((cur-sum (random weight-sum))
               (lst lst-with-weights))
      (if (or (null? lst) (null? (car lst)))
          null
          (let ((cur-weight (caar lst)))
            (if (> cur-weight cur-sum)
                (cadar lst)
                (loop (- cur-sum cur-weight) (cdr lst))
                )
            )
          )
      )
    )
  )

(define strategies
  (list
   (list (lambda (x y z) #t) 2
         (lambda (repl history keywords)
           (qualifier-answer repl)))
   (list (lambda (x y z) #t) 1
         (lambda (repl history keywords)
           (hedge)))
   (list (lambda (repl answers keywords)
           (not (null? answers)))
         2
         (lambda (repl answers keywords)
           (history-answer answers)))
   (list (lambda (repl answers keywords)
           (contains-keyword repl keywords))
```

```
        3
        (lambda (repl answers keywords)
          (template-answer repl keywords)))
   (list (lambda (repl answers keywords)
          (not (null? repl)))
        5
        (lambda (repl answers keywords)
          (compound-generator (change-person repl))))
   )
  )
```

Задание "Весна": файл spring.rkt

```
#lang scheme/base

(require racket/string)
(provide (all-defined-out))

(define input-file "data/texts.txt")
(define output-file "good_out.txt")

(define next-graph (make-hash))

(define prev-graph (make-hash))

(define begin-graph (make-hash))

(define end-graph (make-hash))

(define frequency-graph (make-hash))

(define n 5)
```

```
(define (split-line line)
  (foldl (lambda (x y)
           (let ((sentence (filter
                            (lambda (z) (not (equal? z "")))
                            (regexp-split #px"\\s+" x))))
             (if (null? sentence) y
                 (cons sentence y))))
         null
         (regexp-split
          #px"[\\.\\?!]"
          (regexp-replace*
           #px"([;,:'])"
           (regexp-replace* #px"[^a-z\\.\\?!;,:'\\s\\-]+"
                            (string-downcase line)
                            "")
           " \\1 "))))
  )

(define (read-loop input-file)
  (define in (open-input-file input-file))
  (let loop ((line (read-line in)))
    (if (eof-object? line) (println "all structures created")
        (begin
          (let ((sentences (split-line line)))
            (if (null? sentences) null
                (begin
                  (map add-first-n-gram sentences)
                  (map add-sentence-to-next-graph
                       sentences)
                  (map add-last-n-gram sentences)
                  (map add-sentence-to-prev-graph
                       sentences))))
```

```
          (loop (read-line in))))
    )
  (close-input-port in)
  )

(define (add-first-n-gram sentence)
  (let ((n-gram (get-n-gram sentence n)))
    (if (< (length n-gram) n)
        (update-frequency-graph begin-graph
                                (reverse n-gram))
        (update-frequency-graph begin-graph
                                (reverse (cdr n-gram)))))
  )

(define (add-last-n-gram sentence)
  (let ((n-gram (get-n-gram (reverse sentence) n)))
    (if (< (length n-gram) n)
        (update-frequency-graph end-graph n-gram)
        (update-frequency-graph end-graph (cdr n-gram))))
  )

(define (add-sentence-to-next-graph sentence)
  (let ((n-gram (get-n-gram sentence n)))
    (if (< (length n-gram) n)
        (let ((n-1-gram (reverse n-gram)) (next-word "."))
          (update-graph next-graph n-1-gram next-word)
          (update-frequency-graph frequency-graph n-1-gram))
        (begin
          (let ((next-word (car n-gram))
                (n-1-gram (reverse (cdr n-gram))))
            (update-graph next-graph n-1-gram next-word)
            (update-frequency-graph frequency-graph n-1-gram))
```

```
          (add-sentence-to-next-graph (cdr sentence))
          )
        )
    )
  )

(define (add-sentence-to-prev-graph sentence)
  (define r-sentence (reverse sentence))
  (let ((n-gram (get-n-gram r-sentence n)))
    (if (< (length n-gram) n)
        (update-graph prev-graph n-gram ".")
        (begin
          (update-graph prev-graph
                        (cdr n-gram)
                        (car n-gram))
          (add-sentence-to-prev-graph
           (reverse (cdr r-sentence)))
          )
        )
    )
  )

(define (update-graph graph key value)
  (if (hash-has-key? graph key)
      (hash-set! graph key
                 (let ((word-hash (hash-ref graph key)))
                   (hash-set word-hash value
                             (if (hash-has-key?
                                  word-hash value)
                                 (add1 (hash-ref
                                        word-hash value))
                                 1))))
```

```
      (hash-set! graph key
                 (make-immutable-hash
                  (list (cons value 1)))))
  )

(define (update-frequency-graph graph key)
  (if (hash-has-key? graph key)
      (hash-set! graph key (add1 (hash-ref graph key)))
      (hash-set! graph key 1))
  )

(define (get-n-gram lst n)
  (let loop ((lst lst) (iter n) (res null))
    (if (or (= iter 0) (null? lst)) res
        (loop (cdr lst) (sub1 iter) (cons (car lst) res))))
  )

(define (pick-random-from-hash hash-with-weights)
  (let ((weight-sum (foldl + 0
                           (hash-values hash-with-weights))))
    (let loop ((cur-sum (random weight-sum))
               (lst (hash-keys hash-with-weights)))
      (if (null? lst)
          null
          (let ((cur-weight (hash-ref hash-with-weights
                                      (car lst))))
            (if (> cur-weight cur-sum)
                (car lst)
                (loop (- cur-sum cur-weight) (cdr lst))
                )
            )
          )
```

```
      )
    )
  )

(define (direct-generator first-n-gramma)
    (let loop ((cur-n-gram first-n-gramma)
               (result null) (iter 100))
      (if (or (< iter 0)
              (equal? (car (reverse cur-n-gram)) "."))
          (append (reverse result) cur-n-gram)
          (loop (append (cdr cur-n-gram)
                        (list (pick-random-from-hash
                               (hash-ref
                                next-graph
                                cur-n-gram))))
                (cons (car cur-n-gram) result) (sub1 iter)))
      )
  )

(define (reverse-generator first-n-gramma)
    (let loop ((cur-n-gram first-n-gramma)
               (result null) (iter 100))
      (if (or (< iter 0) (equal? (car cur-n-gram) "."))
          (append (cdr cur-n-gram) result)
          (loop (cons (pick-random-from-hash
                       (hash-ref prev-graph cur-n-gram))
                      (reverse (cdr (reverse cur-n-gram))))
                (cons (car (reverse cur-n-gram)) result)
                (sub1 iter)))
      )
  )
```

```
(define (compound-generator sentence)
  (define variants (make-hash))
  (let loop ((cur-sentence sentence))
    (let ((n-1-gram (reverse
                     (get-n-gram cur-sentence (sub1 n)))))
      (if (< (length n-1-gram) (sub1 n))
          null
          (begin
            (if (hash-has-key? frequency-graph n-1-gram)
                (hash-set! variants n-1-gram
                           (hash-ref frequency-graph n-1-gram))
                null)
            (loop (cdr cur-sentence))
            ))
      ))
  (if (hash-empty? variants)
      (find-best-reply sentence)
      (let ((base-n-gramma (pick-random-from-hash variants)))
        (append (reverse-generator base-n-gramma)
                (list-tail (direct-generator base-n-gramma)
                           (sub1 n)))
        ))
  )

(define (find-best-reply sentence)
  (define variants (make-hash))
  (define (update-variants graph-n-gram n-gram)
    (if (= (foldl (lambda (x y)
                    (if (member x graph-n-gram) (add1 y) y))
                  0 n-gram) (length n-gram))
        (hash-set! variants graph-n-gram
                   (hash-ref frequency-graph graph-n-gram))
```

```
        null)
    )
  (let loop-1 ((word-number (sub1 n)))
    (let loop-2 ((cur-sentence sentence))
      (let ((n-gram (get-n-gram cur-sentence word-number)))
        (if (or (< (length n-gram) word-number)
                (null? cur-sentence))
            null
            (begin
              (map (lambda (x) (update-variants x n-gram))
                   (hash-keys frequency-graph))
              (loop-2 (cdr cur-sentence))
              ))
        ))
    (if (hash-empty? variants)
        (if (> word-number 0)
            (loop-1 (sub1 word-number))
            (direct-generator
             (pick-random-from-hash begin-graph)))
        (let ((base-n-gramma
               (pick-random-from-hash variants)))
          (append (reverse-generator base-n-gramma)
                  (list-tail
                   (direct-generator base-n-gramma)
                   (sub1 n)))
          )
        )
    )
  )

(define (join-string lst)
  (regexp-replace* #px"(') "
```

```
                    (regexp-replace* #px" ([;,:'\\.])"
                                     (string-join lst) "\\1")
                    "\\1")
  )

(define (init-structures mode)
  (if (equal? mode 'train)
      (begin
        (read-loop input-file)
        (save-structures output-file))
      (load-structures output-file)
      )
  )

(define (save-structures output-file)
  (define out (open-output-file output-file #:exists 'replace))
  (write next-graph out)
  (write prev-graph out)
  (write begin-graph out)
  (write end-graph out)
  (write frequency-graph out)
  (close-output-port out)
  )

(define (load-structures input-file)
  (define in (open-input-file input-file))
  (set! next-graph (read in))
  (set! prev-graph (read in))
  (set! begin-graph (read in))
  (set! end-graph (read in))
  (set! frequency-graph (read in))
  (close-input-port in)
```

```
  )

(define (test-1)
  (println (join-string (direct-generator
                         (pick-random-from-hash begin-graph))))
  (newline)
  (println (join-string (reverse-generator
                         (pick-random-from-hash end-graph)))))

(define (test-2)
  (let ((sentence (read-line)))
    (println (join-string (compound-generator
                           (car (split-line sentence)))))
    )
  )

(init-structures 'train)
```