А.В.Матохина, А.А. Соколов, С.Е.Драгунов

Технологии проектирования систем искусственного интеллекта

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

А.В.Матохина, А.А. Соколов, С.Е.Драгунов

Технологии проектирования систем искусственного интеллекта

Учебное пособие

Волгоград
2021

Оглавление

Благодарность 4
Введение 4
Регулярные выражения 4
Материалы для самостоятельного изучения 7
Шпаргалка по регулярным выражениям (http://website-lab.ru/article/regexp/shpargalka_po_regulyarnyim_vyirajeniyam/) 8
Задание на лабораторную работу. Разработка регулярного выражения. 8
Варианты заданий 10
Разработка чат-бота. 15
Варианты заданий 16
Особенности чат-ботов в Telegram 22
Создание чат-бота в Telegram 24
Программирование логики бота 27
Чат-бот для ВКонтакте 31
Материалы для самостоятельного изучения 33
Мультиагентные системы. 33
Теория 33
Программирование агентов в игре Screeps. 33
Технические проблемы 34
Первая версия 37
isolated-vm **Ошибка! Закладка не определена.**

Благодарность

Разработчики пособия выражают благодарность студентам и магистрантам кафедры САПР и ПК Волгоградского государственного технического университета, а именно Кутыркину Кириллу, Володиной Дарье, Поповой Светлане, Леонтьевой Марие, Шашков Владислав…за помощь в оформлении и наполнении пособия.

Введение

Учебное пособие представляет собой практикум по технологиям проектирования систем искусственного интеллекта**.** В курсе предполагается изучение методов разработки чат-ботов, интеллектуальных агентов.

Пособие может быть использовано при изучении дисциплин «Системы искусственного интеллекта», а также может быть применено для курсового и дипломного проектирования. Рекомендовано для студентов обучающихся по специальностям: 09.04.01, 09.03.01.

Регулярные выражения

Компиляция – сборка программы, включающая трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера) или непосредственно на машинном языке или ином двоичнокодовом низкоуровневом командном языке и последующую сборку исполняемой машинной программы.

Трансляция программы – преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.

Трансляция осуществляется тремя последовательно соединенными блоками: лексическим, синтаксическим и генератором кода. На рисунке 1 ниже показаны все блоки традиционного транслятора.

Лексический анализ («токенизация», от англ. tokenizing) – процесс аналитического разбора входной последовательности символов на распознанные группы – лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах).

Лексический анализатор – это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии: сканирование и оценка.

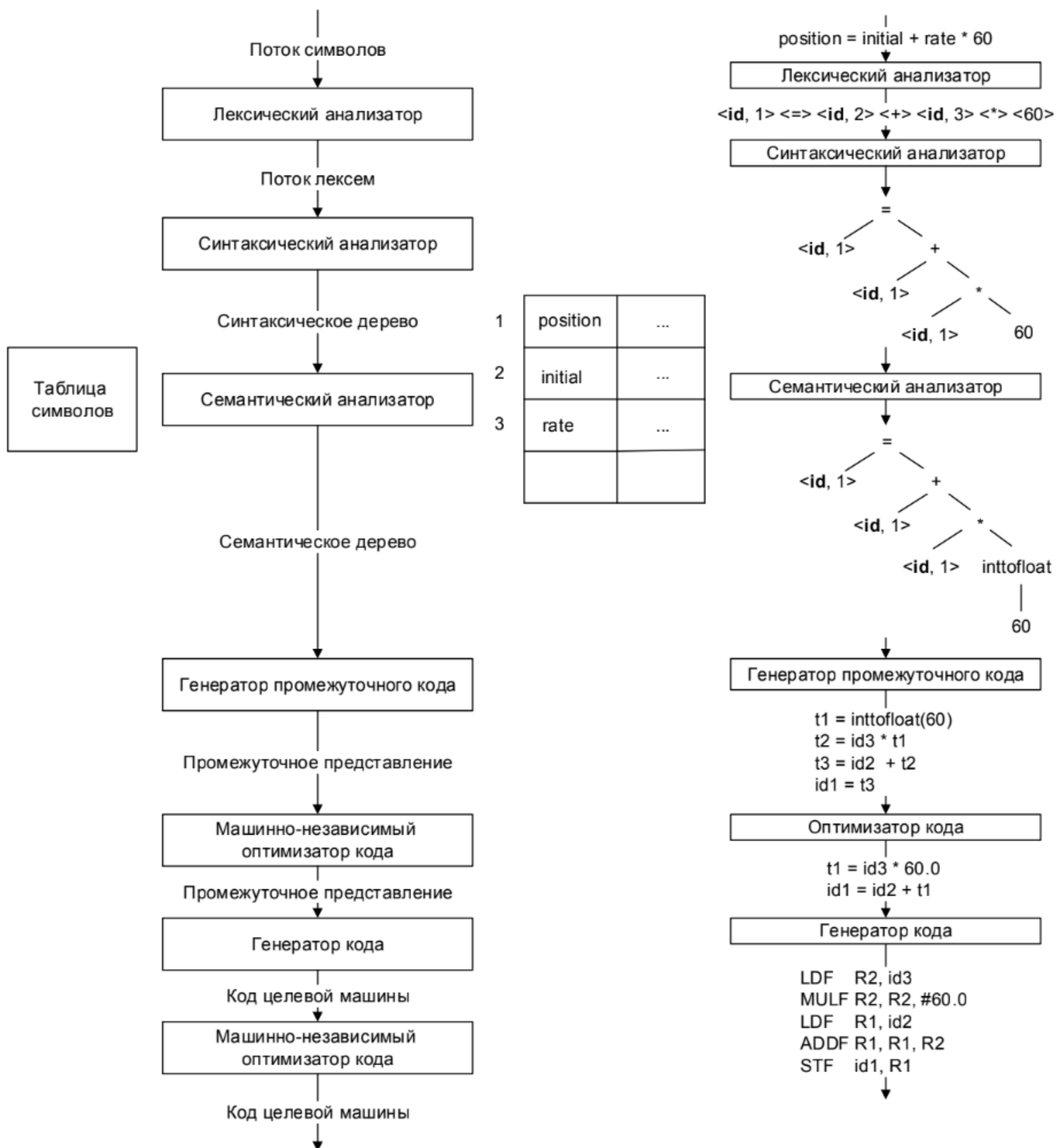


Рисунок 1 – Полный цикл трансляции

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого **регулярными выражениями.** В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах. Например, токен «целое число» может содержать любую последовательность десятичных цифр.

Регулярные выражения (англ. regular expressions) – формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов. Для поиска используется

строка-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска. Это довольно мощный инструмент, который может пригодиться во многих случая – поиск, проверка на корректность строки и т.д. Сейчас регулярные выражения используются многими текстовыми редакторами и утилитами для поиска и изменения текста на основе выбранных правил. Многие языки программирования уже поддерживают регулярные выражения для работы со строками. Например, Perl и Tcl имеют встроенный в их синтаксис механизм обработки регулярных выражений. Набор утилит (включая редактор sed и фильтр grep), поставляемых в дистрибутивах Unix, одним из первых способствовал популяризации понятия регулярных выражений.

Материалы для самостоятельного изучения

Хорошо о регулярных выражениях написано в https://ru.wikipedia.org/wiki/Регулярные_выражения.

Также важно понимание написанного регулярного выражения. Удобный инструмент для визуализации выражений - Regexper: https://regexper.com/

Литература (помимо огромного множества информации в Интернете):

- Регулярные выражения для новичков: https://tproger.ru/articles/regexp-for-beginners/
- Регулярные выражения: https://ru.wikibooks.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F
- Примеры регулярных выражений: как новичку разобраться в регулярках: https://netpeak.net/ru/blog/kak-novichku-razobratsya-v-regulyarnyh-vyrazheniyah/
- Упражнения для тренировки навыков регулярных выражений RegexOne - Learn Regular Expressions - Lesson 1: An Introduction, and the ABCs: https://regexone.com/

Шпаргалка по регулярным выражениям

(http://website-lab.ru/article/regexp/shpargalka_po_regulyarnyim_vyirajeniyam/)

Регулярные выражения

Якоря	
^	Начало строки +
\A	Начало текста +
$	Конец строки +
\Z	Конец текста +
\b	Граница слова +
\B	Не граница слова +
\<	Начало слова
\>	Конец слова

Символьные классы	
\c	Управляющий символ
\s	Пробел
\S	Не пробел
\d	Цифра
\D	Не цифра
\w	Слово
\W	Не слово
\xhh	Шестнадцатиричный символ hh
\Oxxx	Восьмиричный символ xxx

Символьные классы POSIX	
[:upper:]	Буквы в верхнем регистре
[:lower:]	Буквы в нижнем регистре
[:alpha:]	Все буквы
[:alnum:]	Буквы и цифры
[:digit:]	Цифры

<table>
<tr><th colspan="2">Образцы шаблонов</th></tr>
<tr><td>([A-Za-z0-9-]+)</td><td>Буквы, числа и знаки переноса</td></tr>
<tr><td>(\d{1,2}\/\d{1,2}\/\d{4})</td><td>Дата (напр., 21/3/2006)</td></tr>
<tr><td>([^\s]+(?=\.(jpg|gif|png))\.\2)</td><td>Имя файла jpg, gif или png</td></tr>
<tr><td>(^[1-9]{1}$|^[1-4]{1}[0-9]{1}$|^50$)</td><td>Любое число от 1 до 50 включительно</td></tr>
<tr><td>(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)</td><td>Шестнадцатиричный код цвета</td></tr>
<tr><td>((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})</td><td>От 8 до 15 символов с минимум одной цифрой, одной заглавной и одной строчной буквой (полезно для паролей).</td></tr>
<tr><td>(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})</td><td>Адрес email</td></tr>
<tr><td>(\<(/?[^\>]+)\>)</td><td>HTML теги</td></tr>
</table>

Примечание *Эти шаблоны предназначены для ознакомительных целей и основательно не проверялись. Используйте их с осторожностью и предварительно тестируйте.*

Кванторы	
*	0 или больше +
*?	0 или больше, нежадный +
+	1 или больше +
+?	1 или больше, нежадный +
?	0 или 1 +
??	0 или 1, нежадный +
{3}	Ровно 3 +
{3,}	3 или больше +
{3,5}	3, 4 или 5 +
{3,5}?	3, 4 или 5, нежадный +

<table>
<tr><th colspan="2">Диапазоны</th></tr>
<tr><td>.</td><td>Любой символ,
кроме переноса строки (\n) +</td></tr>
<tr><td>(a|b)</td><td>a или b +</td></tr>
<tr><td>(...)</td><td>Группа +</td></tr>
<tr><td>(?:...)</td><td>Пассивная группа +</td></tr>
<tr><td>[abc]</td><td>Диапазон (a или b или c) +</td></tr>
<tr><td>[^abc]</td><td>Не a, не b и не c +</td></tr>
<tr><td>[a-q]</td><td>Буква между a и q +</td></tr>
<tr><td>[A-Q]</td><td>Буква в верхнем регистре между A и Q +</td></tr>
<tr><td>[0-7]</td><td>Цифра между 0 и 7 +</td></tr>
</table>

Задание на лабораторную работу. Разработка регулярного выражения.

В лабораторной работе необходимо написать регулярное выражение для поиска подстроки по указанному в задании шаблону. Поиск подстроки производится исходной строке с любыми символами. Необходимо представить написанное регулярное выражение и продемонстрировать результаты его работы на нескольких тестовых примерах в одном из онлайн-редакторов.

На данной лабораторной работе программу писать не нужно. Составлять регулярные выражения и проверять их работоспособность можно в одном из специализированных онлайн-редакторов, например, Online regex tester and debugger: PHP, PCRE, Python, Golang and JavaScript, RegExr: Learn, Build, & Test RegEx, Regular Expression Tester.

В качестве ответа на данную лабораторную работу необходимо залить протокол в виде файла Word, в котором написать стандартную "шапку" протокола с вашим ФИО и номером лабораторной работы, задание по выбранному варианту, написанное регулярное выражение и скриншоты одного из перечисленных выше онлайн-редакторов, где отражено, что выражение находит искомую строку. Можно также приложить визуализацию из сервиса Regexper (см. ниже).

Страница задания на EOS для загрузки результатов работы:
http://eos.vstu.ru/mod/assign/view.php?id=95547

Пример протокола для этой и последующей лабораторных работ приложен:
https://drive.google.com/open?id=1qOSkxOOYxtTBiL7Lw7ZW6F6_2t1Cofi5

В дальнейшем содержимое протокола уже определяете сами, главное, чтобы оно раскрывало суть решения лабораторной и было опрятно оформлено: “шапка”, задание, предлагаемое решение, последовательность проделанных шагов, скриншоты, блок-схемы и другие диаграммы, написанный код, тестовые наборы и т.п.

Разобранный вариант задания

2. Правильный формат ввода домашнего адреса

Необходимо составить регулярное выражение для поиска строк с правильным

форматом домашнего адреса

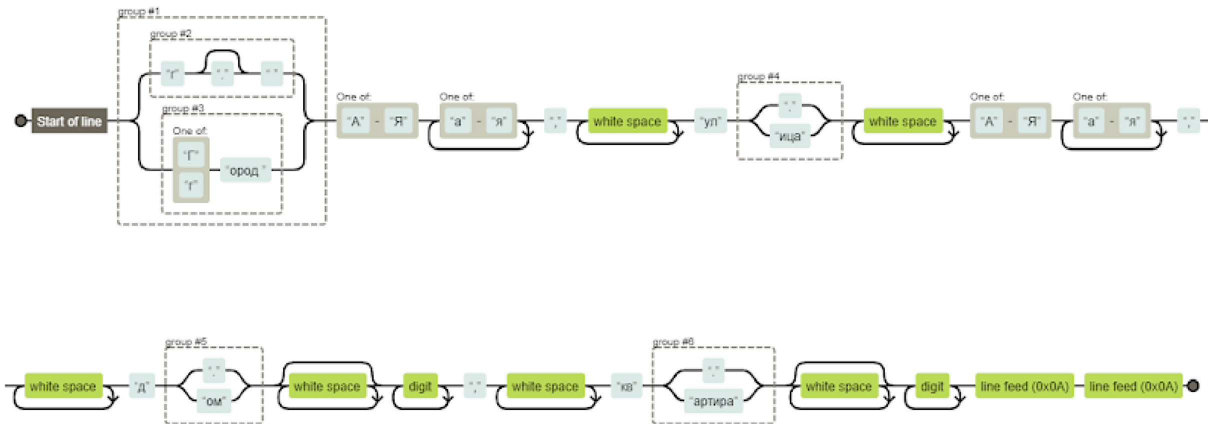
г. Волгоград, ул. Невская, д.18, кв.43

Город Москва, улица Молодежная, дом 20, квартира 50

Полученное выражение:

^((г\.?)|([Гг]ород))[А-Я]{1}[а-я]+\,\s+ул(\.|ица)\s+[А-Я]{1}[а-я]+\,\s+д(\.|ом)\s*\d+\,\s+кв(\.|артира)\s*\d+

Визуализация данного выражения в Regexper (https://regexper.com/):



Варианты заданий

Уникальный вариант самостоятельно выбирается каждым студентом.

1. Буквы в алфавитном порядке

Необходимо составить регулярное выражение для поиска строк с последовательностью букв в алфавитном порядке (для английского и русского алфавита)

abcdefgh

абгджик

2. Правильный формат ввода домашнего адреса

разобран выше

3. Является ли строка ФИО

Составляется регулярное выражения для определения, является ли строка ФИО

Якунин Влад Иванович

В.И. Якунин

Якунин В.И.

4. Доменные имена

Поиск доменных имен для протоколов http и https, с необязательным слешем в конце, после которого идет путь к файлу и параметры запроса

http://www.example.com/path/to/index.html?name=John&lastname=Doe

http://example.com/

http://example.com

5. Поиск дат

Осуществить поиск дат по следующему шаблону

DD.MM.YYYY HH:MI:SS

DD-MM-YYYY HH:MI:SS

DD/MM/YYYY HH.MI.SS

DD/MM/YYYY HH:MI:SS

6. Числа

Выбрать числа с запятой или пробелом, в качестве разделителя разрядов.

8,205,500.4672

8 205 500.4672

8,205,500

8 205 500

7. Поиск IPv6 адресов

Адреса IPv6 отображаются как восемь четырехзначных шестнадцатеричных чисел (то есть групп по четыре символа), разделенных двоеточием. Пример адреса:

2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d

Если одна или более групп подряд равны 0000, то они могут быть опущены и заменены на двойное двоеточие (::). Незначащие старшие нули в группах могут быть опущены. Например, 2001:0db8:0000:0000:0000:0000:ae21:ad12 может быть сокращен до 2001:db8::ae21:ad12, или 0000:0000:0000:0000:0000:0000:ae21:ad12 может быть сокращен до ::ae21:ad12.

8. Поиск химических элементов

Выбираем первые 70 элементов таблицы Менделеева. Решение должно быть максимально коротким

9. Поиск HTML-тэгов в строке

Причем тэги могут быть как состоящими из пар - открывающий и закрывающий (<p></p>) или пустыми (<img />, <br>). Они могут иметь атрибуты. (https://html5book.ru/html-tags/)

Пример тега p без атрибутов:

<p>This is a paragraph</p>

Пример тэга a с атрибутом href:

Example Link

10. Поиск номера мобильного телефона

Поиск производить по шаблонам:

+7 (999) 999-99-99

8 999 999-99-99

8 999 9999999

89999999999

11. Поиск номера домашнего телефона

Поиск производить по шаблонам:

+7 (8442) 99-99-99

+7 (999) 999-99-99

8 (8442) 99-99-99

8 (999) 999-99-99

999-999

99-99-99

12. Проверка на корректность адреса электронной почты

example123@example123.com

тест@тест.рф

13. Поиск однострочных и многострочных комментариев в коде на C++ (C#)

```
// Это однострочный комментарий, единственный на строке
x = 1; // Это однострочный комментарий после строки с кодом

/* Это однострочный комментарий, но написанный в одной строке */
/* Это многострочный
   комментарий. Он чаще всего используется, когда
   нужно написать много строк комментариев. */
```

14. Поиск однострочных и многострочных комментариев в коде на Python

Это однострочный комментарий, единственный на строке

def complicated_function(s): # Это однострочный комментарий после строки с кодом

"""

Это многострочный комментарий.

Он чаще всего используется, когда

нужно написать много строк комментариев.

"""

15. Поиск ссылок в HTML тегах

Пример:

<h1>Example</h1>

16. Поиск никнеймов

Никнейм начинается с одного или более символа: _, <, = или точки и заканчивается так же, только угловая скобка должна быть закрывающей (>). Первым и последним перед и после символов обязательно должно быть число. После числа могут быть символы русского и английского алфавита в верхнем или нижнем регистре.

17. Поиск пути к файлу в Windows

Примеры:

C:\полный\путь\к\файлу.txt

\\локальный_сервер\путь\к\файлу.pdf

18. Поиск представления цвета в шестнадцатеричном виде или в виде функции RGB

http://htmlbook.ru/css/value/color

Имя	Цвет	Код	RGB
white		#ffffff или #fff	rgb(255,255,255)
silver		#c0c0c0	rgb(192,192,192)
gray		#808080	rgb(128,128,128)
black		#000000 или #000	rgb(0,0,0)
maroon		#800000	rgb(128,0,0)
red		#ff0000 или #f00	rgb(255,0,0)
orange		#ffa500	rgb(255,165,0)
yellow		#ffff00 или #ff0	rgb(255,255,0)
olive		#808000	rgb(128,128,0)
lime		#00ff00 или #0f0	rgb(0,255,0)
green		#008000	rgb(0,128,0)
aqua		#00ffff или #0ff	rgb(0,255,255)
blue		#0000ff или #00f	rgb(0,0,255)
navy		#000080	rgb(0,0,128)
teal		#008080	rgb(0,128,128)
fuchsia		#ff00ff или #f0f	rgb(255,0,255)
purple		#800080	rgb(128,0,128)

19. Поиск информации в лог-файле

Лог-файл имеет следующий формат:

(<дд.мм.гггг> <чч.мм.сс>): Сообщение об ошибке (<имя файла>.<расширение>:<номер строки>). Пример:

(03.03.2019 22:33:44): Fatal Error at widget.List.makeView(ListView.java:1727)

20. Поиск IPv4 адресов

Они могут быть в десятичном или шестнадцатеричном представлениях. С точками. Подробнее про IP адреса можно узнать в википедии

Примеры:

99.198.122.146

0xFF.255.0377.0x12

Разработка чат-бота.

Цель: разработать чат-бот на языке Python, выполняющий заданные действия по варианту задания.

Задачи:

1. разработка регулярного выражения для поиска в произвольном тексте команд чат-боту и параметров выполнения команд;
2. реализация требуемых действий чат-бота в предварительной заготовке программы на языке Python;
3. разработка заготовки самого чат-бота, принимающего сообщения от пользователя и отправляющего сообщения пользователю;
4. компоновка всех перечисленных выше этапов в рамках чат-бота.

Протокол к лабораторной работе должен содержать описание всех задач, с приведением полученного регулярного выражения, алгоритма в виде блок-схемы реализации логики работы чат-бота, с расшифровкой поддерживаемых команд и с пояснениями к выполняемыми действиями, никнейма чат-бота в мессенджере для тестирования на отчёте и скриншотов диалога с чат-ботом с демонстрацией выполнения заданных действий чат-ботом. Также в архив необходимо приложить .py-файл исходного кода чат-бота.

ВСЕ ВЫПОЛНЕННЫЕ РАБОТЫ ДОЛЖНЫ БЫТЬ ЗАГРУЖЕНЫ НА EOS2.

Общие требования к чат-боту:

- чат-бот должен выполнять требуемые в задании действия по вводу команд, задаваемых пользователем, с опциональными параметрами для указания особенностей выполнения заданных действий;
- некоторые параметры команды чат-боту могут быть опциональными, т.е. если их не указать, подразумевается значение по умолчанию, заданное студентом;
- чат-бот может принимать одну или несколько различных команд для реализации выполнения задания курсовой работы;
- чат-бот не должен реагировать на сообщения-шум, которые не являются зарезервированными командами чат-бота;
- поиск в тексте сообщения команд и параметров необходимо реализовать с помощью регулярных выражений;

- если для команды не хватает каких-то параметров или они указаны неверно, чат-бот может сообщать об ошибке или не выполнять действия, но он должен говорить обо всех ошибках, возникающих в процессе выполнения заданных действий по команде, если они возникли по какой-то причине;
- чат-бот должен выдавать ответ на заданную команду об успешности или неуспешности выполнения команды, результаты выполнения команды или выводить требуемую пользователем информацию по команде (в зависимости от задания);
- внешний вид и содержание сообщений с информацией, выдаваемой чат-ботом задаётся студентом самостоятельно, однако они должны быть понятны, читаемы, опрятно оформлены;
- чат-бот должен проверять корректность формата вводимых данных (ФИО, почтовый адрес, адрес электронной почты, номер телефона и т.д.), логичности данных (например, не указывать дату позднее или ранее, чем текущий момент, по заданию, т.е. не вносить данные из будущего, не делать уведомления в прошлое и т.п.) и выводить сообщение об некорректности вводимых данных или игнорировать команду (упрощенный вариант);
- для хранения данных в файле можно использовать формат JSON.

Варианты заданий

1. Реализовать отправку письма на электронную почту с помощью чат-бота. В параметрах команды указать тему письма, адреса получателя (один и более), текст письма и время отправки в течение дня или «сейчас» (порядок на усмотрение студента). О работе с электронной почтой в Python можно посмотреть, например, здесь: https://python-scripts.com/send-email-smtp-python, https://realpython.com/python-send-email/ или в справке: https://docs.python.org/3/library/email.examples.html. Для реализации отправки писем необходимо создать свой аккаунт в почтовом сервисе. Пример возможного варианта команды: «/send Привет! Как дела? Как погода? -t Тестовое письмо -r ivanov@example.com, petrov@example.com -w сейчас»

2. Реализовать отправку чат-ботом напоминаний, создаваемых пользователем. В параметрах команды указать текст, о чём надо

напомнить, когда (разные варианты параметра: через заданный период времени или в указанное время, дату, или «сегодня, завтра»), делать ли напоминание ежедневно. Напоминание может быть не менее, чем через 10 секунд. Здесь понадобятся модули для работы со временем (time, datetime). Реализовать функционал, позволяющий пользователю запросить список всех созданных напоминаний и удалять те, которые не нужны (например, по номеру в списке). Пример возможного варианта команды добавления напоминания: «/add Купить хлеба -t завтра 10:00 no repeat»

3. Реализовать с помощью чат-бота возможность отслеживания своих доходов и расходов. Пользователь отправляет чат-боту сообщение с указанием, на что он потратил деньги, когда (дата или «сегодня» и время), но не позднее текущей даты, какую сумму (в рублях), а также откуда он получил деньги (также с отметкой, когда и какую сумму). Данные сохраняются в файл. Пользователь может запросить вывод списка всех расходов и доходов с указанием баланса на текущий момент по имеющемуся списку и очистить файл. Пример возможного варианта команды добавление расходов: «/spend Хлеб, молоко -c 85.50 -d 01.01.2020»

4. Реализовать генерацию результатов бросков игровых кубиков чат-ботом (в частности для игр в DnD, по аналогии с ресурсом https://rolz.org/). Пользователь вводит команду в формате: <кол-во_кубов1>d<кол-во_граней1> +/- … +/- <кол-во_кубовN>d<кол-во_гранейN> +/- <любое число> (в произвольном порядке), а чат-бот выводит полученное значение суммы/разности случайно сгенерированных чисел. Если количество кубов не указано перед «d», то считается, что куб один. Пример возможного варианта команды: «2d6 + d10 – 10 + 3d20»

5. Реализовать записную книжку в чат-боте. Пользователь может заносить в чат-бот информацию о людях: номер мобильного телефона, ФИО, перечисление учетных записей в мессенджерах/соц. сетях (ни одной или одна и более, через запятую). Данные сохраняются в файл. Пользователь может запросить вывод списка всех записей людей, отсортированных по ФИО, удалить выбранную запись, задавая номер в списке, и очистить файл. Пример возможного

варианта команды: «/add 89992233223 Иванов Иван, vk:ivanivan, t:ivan_v_telege»

6. Реализовать получение прогноза погоды от чат-бота. Прогноз может выдаваться на указанную дату (или «сегодня, завтра»), на период времени с одной даты по другую, пользователь может указать город для прогноза, прогноз может быть коротким или подробным (содержание каких данных в прогнозе – на усмотрение студента). Пример возможного варианта команды: «/weather –t завтра –l г. Волгоград -full». Для реализации получения прогноза погоды понадобится работа с API какого-либо сервиса, например: https://rapidapi.com/blog/weather-api-python/

7. Реализовать сохранение закладок по тематикам с помощью чат-бота. Пользователь может добавлять тематику. Потом для каждой из добавленных тематик он может сохранять адреса различных сайтов, статей, которые хочет прочесть позже. Данные сохраняются в файл. После он может запросить, что прочесть по заданной тематике и чат-бот выводит список ссылок. Пользователь может удалять ссылку из тематики, а также тематику целиком (со всеми ссылками в ней). Пример возможного варианта команды добавления закладки: «/add Программирование на Python -u https://example.com». Если такой тематики нет, она создаётся.

8. Реализация чат-бота собеседника с помощью библиотеки nltk, пакета chat. Необходимо заготовить словарь ответов по заданным регулярным выражениям и, используя пакет chat библиотекb nltk выдавать ответы пользователю. Подробнее можно прочесть здесь: https://medium.com/datadriveninvestor/build-your-own-chat-bot-using-python-95fdaaed620f, https://towardsdatascience.com/build-your-first-chatbot-using-python-nltk-5d07b027e727 пакет chat: https://www.nltk.org/api/nltk.chat.html. Необходимо использовать метод chat.response(<строка сообщения от пользователя>). Пример возможного варианта команды: «-Привет! Как дела? –Нормально! А у тебя? –Тоже хорошо! –Это здорово!»

9. Реализовать получение рекомендаций от чат-бота о том, что приготовить из заданного набора продуктов, имеющихся в

холодильнике у пользователя. Данные хранятся в файле в формате JSON, он заполняется студентом заранее. Формат представлен ниже:

```
{
        “dish”: “<наименование блюда>”,
        “type”: “<завтрак/обед/ужин/перекус>”,
        “ingredients”: [
                “<ингредиент 1>”,
                …
                “<ингредиент N>”
        ]
}
```

Пользователь вводит список продуктов, которые у него имеются в наличии, тип блюда и чат-бот возвращает список доступных блюд. Пример возможного варианта команды: «/dish -t перекус -i хлеб, масло, колбаса, сыр, молоко, яйца»

10. Реализовать напоминания о днях рождения с помощью чат-бота. Пользователь добавляет в список дней рождений записи, в которых содержатся ФИО, дата рождения и почтовый адрес места обычного места встречи/проживания именинника. Чат бот должен по утрам напоминать, что у кого-то из списка через день состоится день рождения, если есть такая запись. Пользователь может запросить список записей о днях рождения и удалить одну из записей по номеру в списке. Пример возможного варианта команды добавления записи о дне рождения: «/add –p Иванов Иван –w 01.01.1990 –l ул. Пушкина, 22, кв. 34»
11. Реализовать чат-бот для проведения голосования. Пользователь создаёт голосование, задавая вопрос для голосования, варианты выбора, дату и время окончания голосования. Идентификаторы пользователей, которые инициировали беседу с чат-ботом, должны храниться в файле, не повторяясь. Чат-бот рассылает всем пользователям из файла и текущему пользователю опрос: вопрос голосования и пронумерованные варианты выбора. Пользователи голосуют, отправляя номер варианта. По окончании голосования в заданные дату и время чат-бот выводит процентные соотношения количеств голосов за каждый из вариантов. Каждый пользователь, кто добавил себе чат-бота может создать только одно текущее

голосование. Он может его остановить по отдельной команде с выводом набранных результатов с пометкой, что оно остановлено. Другие пользователи могут проголосовать только один раз. Пример возможного варианта команды создания голосования: «/poll Где сегодня встречаемся нашей компанией друзей? -a В парке -a В кафе -a На набережной -a У Ивана дома -till 01.01.2020 15:00»

12. Реализовать чат-бот для записи на консультацию по предмету в университете. Студент запрашивает список доступных консультаций на неделю, чат-бот ему его выводит. Список выводится в формате: название предмета, дата, время, аудитория. Затем студент пишет чат-боту команду на запись на консультацию, в которой содержится: его ФИО, группа, предмет, дата и время. Чат-бот выдаёт ответ об успешной записи на консультацию, если в имеющемся списке консультаций на неделю есть выбранная консультация с таким предметом, датой и временем. Данные о консультациях хранятся в файле в формате JSON, он заполняется вами заранее. Формат представлен ниже:

```
[
    {
        “class”: “Наименование предмета консультации 1”,
        “date”: “Дата и время консультации 1”,
        “room”: “аудитория и корпус 1 (например, В-1001)”
    },
    …
    {
        “class”: “Наименование предмета консультации N”,
        “date”: “Дата и время консультации N”,
        “room”: “аудитория и корпус N (например, В-1001)”
    }
]
```

Пример возможного варианта команды записи на консультацию: «/addme Иванов Иван Иванович -r ВТВ-268 -c Мат. анализ -d 01.01.2020 15:00»

13. Реализовать чат-бот для записи своих результатов выполнения упражнений в тренажерном зале. Пользователь отправляет чат-боту сообщение с указанием даты (не ранее текущей даты), какое упражнение он делал, какой вес брал, сколько подходов по сколько повторений. Вес может быть опциональным, чтобы учитывать тренировки без веса. Данные сохраняются в файл. Пользователь может запросить вывод списка всех записей, сгруппированных по датам и очистку файла. Пример возможного варианта команды добавление расходов: «/add 01.01.2020, Жим лёжа, 4, 12, 60» (вес добавляется в конце)

Особенности чат-ботов в Telegram

Мессенджер Telegram позволяет его пользователям создавать своих чат-ботов. В нём, и в других мессенджерах Бот — это не просто «автоответчик», его правильнее считать автоматизированным помощником. Это многофункциональный инструмент, как для бизнеса, так и для развлечений. Боты могут использоваться для:

- индивидуальных уведомлений и контента
- интеграции с другими сервисами и получения от них оповещений (почта, переводчик, погода, новости, YouTube и др.)
- создания пользовательских инструментов, игр и др.

В данной работе рассматриваются чат-боты для Telegram, поскольку их создание является одним из самых простых и позволит сфокусироваться на реализации логики работы бота, а не на его создании и подготовке к работе.

Telegram Bots – это специальные учетные записи, для настройки которых не требуется дополнительный номер телефона. Пользователи могут взаимодействовать с ботами двумя способами:

- отправлять сообщения и команды ботам, открывая с ними чат или добавляя их в группы
- отправлять запросы из любого чата, набрав @username бота и запрос. Это позволяет отправлять контент из встроенных ботов прямо в любой чат, группу или канал

Сообщения, команды и запросы, отправленные пользователями, передаются программному обеспечению, написанному вами для бота, работающему на вашем компьютере (сервере). Сервер-посредник Telegram обрабатывает все задачи шифрования и связи с Telegram API. Вы общаетесь с этим сервером через простой HTTPS-интерфейс, который предлагает упрощенную версию Telegram API – Bot API. Подробное его описание здесь: https://core.telegram.org/bots/api

Особенности ботов в Telegram:

- у ботов нет онлайн-статуса и последних посещенных меток времени, вместо этого на интерфейсе отображается метка «бот»
- у ботов ограниченное облачное хранилище - старые сообщения могут быть удалены сервером вскоре после их обработки

- боты не могут инициировать разговоры с пользователями. Пользователь должен либо добавить их в группу, либо сначала отправить им сообщение. Люди могут использовать ссылки t.me/<bot_username> или поиск по имени пользователя, чтобы найти вашего бота
- имена пользователей ботов всегда должны заканчиваться на «бот» (например, @TriviaBot, @GitHub_bot)
- при добавлении в групповой чат по умолчанию боты не получают все сообщения от пользователей к пользователям (только в особенных случаях, см. в справке), а получают только:
    - сообщения, начинающиеся с косой черты '/' (см. ниже и в справке)
    - ответы на собственные сообщения бота
    - служебные сообщения (люди добавлены или удалены из группы и т. д.)
    - сообщения с каналов, на которых они зарегистрированы
- если в группе несколько ботов, можно добавить имена команд ботов в команды, чтобы избежать путаницы:
    - /start@TriviaBot
    - /start@ApocalypseBot

Чтобы облегчить пользователям использование ботов, разработчики Telegram просят всех разработчиков поддерживать у своих ботов несколько основных команд. Приложения Telegram будут иметь ярлыки интерфейса для этих команд:

- /start - начало взаимодействия с пользователем, например, отправка приветственного сообщения
- /help - возвращает справочное сообщение. Это может быть краткий текст о том, что может делать ваш бот, и список команд
- /settings - (если применимо) возвращает настройки бота для этого пользователя и предлагает команды для редактирования этих настроек

Создание чат-бота в Telegram

Сперва нужно зарегистрироваться в Telegram. Наиболее удобно использовать веб-клиент для знакомства с основными принципами работы ботов и API: https://web.telegram.org/

Для создания ботов есть специальный бот BotFather (https://t.me/botfather), откройте приложение, найдите @BotFather, начните с ним беседу и выполните несколько простых шагов, инструкция здесь: https://core.telegram.org/bots#6-botfather. После того, как вы создали бота и получили свой **токен авторизации (можно сказать, уникальный идентификатор бота)**, перейдите к руководству по Bot API, чтобы узнать, чему вы можете научить своего бота. **Токен нужно держать в секрете, чтобы никто не мог получить доступ к вашему боту и не начал его использовать в своих корыстных целях.** В случае чего, его можно пересоздать или отозвать (см. документацию у BotFather).

Примеры ботов (в том числе и на Python) приведены здесь: https://core.telegram.org/bots/samples

Вам нужно начать беседу с вашим ботом. Найдите своего бота в поиске пользователей. Введите в поисковой строке его имя и нажмите на кнопку /start. Пока что ваш бот ничего не умеет, но уже может принимать сообщения. Отправьте сообщение, например, «Привет». Это первое сообщение очень важно, поскольку оно станет первым обновлением, которое получит ваш бот.

Все запросы к Telegram Bot API должны обслуживаться по HTTPS и должны быть представлены в этой форме:

https://api.telegram.org/bot<token>/ИМЯ_МЕТОДА_ЗАПРОСА

Если вы в первый раз работаете с API, то разобраться вам поможет браузер. Откройте новую вкладку и воспользуйтесь Telegram API, отправив в строку браузера запрос:

https://api.telegram.org/bot<ваш_токен>/getUpdates

Таким образом, вы отправляете Get-запрос на сервер Telegram для вашего бота.

По сути, каждый раз, когда вам нужно получить, обновить или удалить информацию, хранящуюся на сервере, вы отправляете запрос и получаете ответ.

Подробнее здесь: https://ru.wikipedia.org/wiki/HTTP

Про запрос методом Get здесь: https://webkyrs.info/post/http-zapros-metodom-get

Отправлять данные боту обратно будем через Post-запрос: https://ru.wikipedia.org/wiki/POST_(HTTP)

Открыв этот адрес в браузере, вы отправите запрос на сервер Telegram, и он ответит вам в формате JSON, который разбирался на прошлой лабораторной работе.

Если вы своему боту ничего не писали, то там будет такая строка:


```
{"ok":true,"result":[]}
```


Т.е. бот работает, но он ещё ничего не получал, поэтому массив “result” пуст. Если «ok» равно true, запрос был успешным, и результат запроса можно найти в поле «result». В случае неудачного запроса «ok» равно false, а ошибка объясняется в «description».

В браузере JSON-строка от бота отображается сплошной строкой. Можете поместить этот текст в сервис JSON-форматтера, как говорилось на предыдущей лабораторной работе, чтобы посмотреть, как она выглядит с расстановкой переносов и подсветкой синтаксиса. Вы увидите что-то вроде такого:


```
{
 "ok":true,
 "result":[{
 "update_id":523349956,
 "message":{
 "message_id":51,
 "from":{
        "id":303262877,
        "first_name":"YourName"
 },
 "chat":{
        "id":303262877,
```

```
        "first_name":"YourName",
        "type":"private"
  },
  "date":1486829360,
  "text":"Hello"
  }
  }]
}
```


Назначение содержимого этого JSON-объекта разобрано в документации: https://core.telegram.org/bots/api#making-requests

Словарь обновлений, возвращаемый ботом, состоит из двух элементов: ok и results. Нас интересует вторая часть – список всех обновлений, полученных ботом за последние 24 часа.

Подробнее о методе getUpdates: https://core.telegram.org/bots/api#getupdates

Чтобы отправить сообщение от лица бота, нужно отправить запрос /sendMessage (см. в документации: https://core.telegram.org/bots/api#sendmessage). Вы увидите, что он принимает два параметра: chat_id и text. Т.е. мы указываем кому бот отправляет сообщение и с каким текстом. Вы можете создавать цепочки параметров в адресной строке браузера, используя ? для первого и & для всех последующих параметров. Команда для отправки сообщения будет выглядеть так:

https://api.telegram.org/bot<token>/sendMessage?chat_id=303262877&text=Hello

Попробуйте получить ответ от вашего бота, подставив в качестве chat_id значение вашего идентификатора чата, полученное после вызова /getUpdates (в нашем примере — 303262877). Текст сообщения может быть любым.

Программирование логики бота

Если у вас имеется возможность использовать мессенджер Telegram на компьютере, то вы можете разрабатывать его логику с помощью Python и среды разработки, установленных у вас на ПК. В противном случае вы будете получать ошибку о недоступности ресурса Telegram по понятным причинам (пример решения можно посмотреть в этой статье: https://habr.com/ru/post/448310/). В этом случае вам необходимо использовать онлайн-блокнот, о которых говорилось на второй лабораторной работе. И в этом случае лучше использовать блокнот, доступный только вам, например, в Google Colab (https://colab.research.google.com/), опять же, для сохранности вашего токена.

В случае работы на вашем компьютере убедитесь, что у вас установлен pip, обычно он устанавливается вместе с Python (https://pypi.org/project/pip/)

Необходимо установить пакет requests при помощи pip, вызываемого в консоли Windows (или см. документацию к онлайн-блокноту, а, например, в Google Colab он уже установлен, можно пропустить этот шаг):

$ pip install requests

Мы будем использовать данный модуль, больше о нём можно прочесть здесь: https://requests.readthedocs.io/en/master/

Напишем скрипт, который будет проверять обновления и отвечать на новые сообщения.

Сперва бот должен проверить обновления. Первое сообщение можно расценивать как самое свежее, но getUpdates возвращает все обновления за последние 24 часа. Напишем небольшой скрипт, чтобы получить самое последнее обновление, доставать chat_id из обновления и отправлять сообщение.

```
import requests


# Адрес нашего бота (без указания запроса)

url = 'https://api.telegram.org/bot<токен_вашего_бота>/'
```

```
# Получаем JSON-строку ответа на запрос getUpdates

def get_updates_json():

    response = requests.get(url + 'getUpdates')  # Выполняем Get-запрос, возвращается объект ответа

    return response.json()                          # Из объекта ответа берём JSON



# Получить последнее событие от бота

def last_update(response_json):

    results = response_json['result']         # Обращаемся к словарю по ключу result

    last_update_index = len(results) - 1       # Последнее событие - последнее по порядку минус 1, т.к. нумерация с нуля

    return results[last_update_index]      # Возвращаем последнее событие



# Получаем идентификатор чата, от которого пришло сообщение

def get_chat_id(result_json):

    return result_json['message']['chat']['id']       # Смотрим по JSON строке: message -> chat -> id



# Отправляем пришедшее событие назад

def send_message(chat_id, text):

    params = {'chat_id': chat_id, 'text': text}              # Формируем параметры запроса: куда и что отправяем

    return requests.post(url + 'sendMessage', data=params)  # Отправляем Post-запрос, возвращаем объект ответа



# Каскадно выполняем запрос к боту, получаем последнее событие и id чата отправителя
```

```
current_chat_id = get_chat_id(last_update(get_updates_json()))

send_message(current_chat_id, "Привет!")
```

Помните, как мы объединяли параметры при помощи «?» и «&»? Вы можете сделать то же самое, добавив словарь в качестве второго дополнительного параметра в функциях get/post из пакета requests.

Скрипт готов, но он не идеален. Главным минусом является необходимость запускать его каждый раз, когда мы хотим, чтобы бот отправил сообщение. Исправим это. Чтобы бот слушал сервер и получал обновления, нам нужно запустить основной цикл. На новой строке, после import requests, добавьте

from time import sleep

Из модуля работы со временем мы импортируем функцию sleep – пауза работы скрипта на указанное количество секунд.

После этого замените две последние строки на следующий код:

```
# Функция - точка входа в программу

def main():

  last_update_id = get_update_id(last_update(get_updates_json()))  # Получаем идентификатор самого последнего события



  # Бот работает в вечном цикле, нужно будет принудительно останавливать свою программу

  while True:

    updates_json = last_update(get_updates_json())  # Получаем последнее событие на текущий момент

    current_update_id = get_update_id(updates_json)  # Получаем его идентификатор



    # Если было новое событие, отправляем сообщение и запоминаем его идентификатор

    if current_update_id != last_update_id:
```

```
        send_message(get_chat_id(updates_json), "Привет!")
        last_update_id = current_update_id

    sleep(1)     # Ждём одну секунду, не забудьте про эту функцию, чтобы наш код не слал постоянно большое количество запросов на сервер Telegram


if __name__ == '__main__':
    main()
```

Хотя мы и добавили таймаут в 1 секунду, пример выше можно использовать только в обучающих целях, поскольку он использует частые опросы (short polling). Это плохо влияет на сервера Telegram, поэтому их нужно избегать. Если мы будем использовать способ получения обновлений через getUpdates без параметров, то запросы будут происходить слишком часто.

Есть ещё два способа получения обновлений через API — длинные опросы (long polling) и вебхуки (webhooks). Общая разница между polling и webhooks:

- **Polling** (через get_updates) периодически подключается к серверам Telegram для проверки новых обновлений. Мы отправляем запрос, получаем ответ.
- **Webhook** - это URL, который вы передаете Telegram один раз. Когда приходит новое обновление для вашего бота, Telegram отправляет это обновление по указанному URL.

Для реализации работы через webhooks есть свои определённые задачи, которые нужно выполнить, подробнее можно почитать здесь: https://github.com/python-telegram-bot/python-telegram-bot/wiki/Webhooks, но мы пока будем пользоваться polling.

Long polling отличается от short polling тем, что в параметре timeout метода getUpdate есть ненулевое значение (https://core.telegram.org/bots/api#getupdates). Т.е. когда мы делаем данный запрос с параметром timeout=60, результат будет возвращён сразу, если есть новые обновления, или он будет ожидать 60 секунд пока обновления не

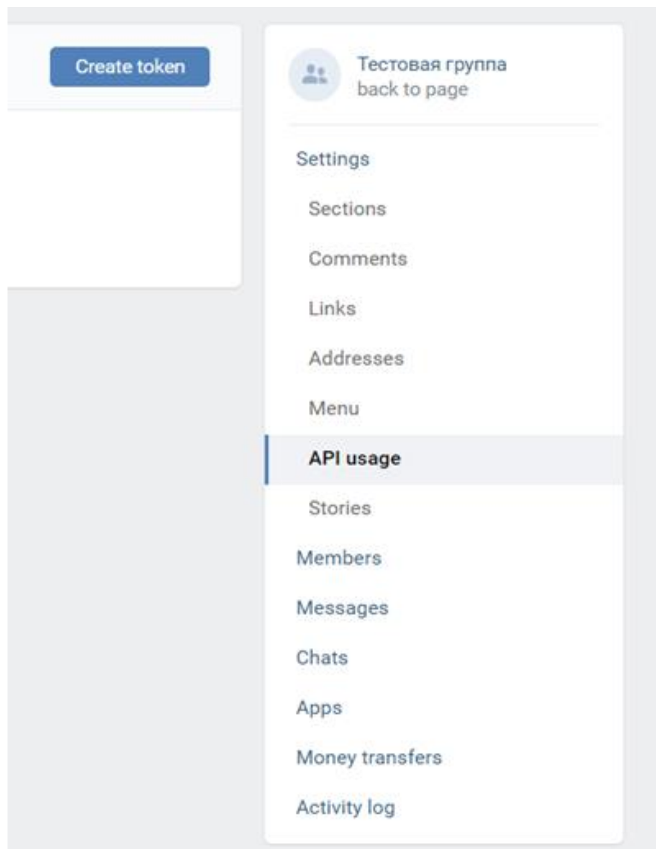
появятся и только после этого вернёт значение (либо обновление появились, либо так и не появились).

Поскольку мы начали использовать в скрипте основной цикл, мы должны переключиться на длинные опросы. Сперва изменим первую функцию, добавив в неё параметр timeout. Сам по себе он не уменьшит частоту проверки обновлений и будет работать только в том случае, когда обновлений нет. Чтобы помечать уже просмотренные обновления, нужно добавить параметр сдвига offset (тоже пока использовать не будем):

```
# Получаем JSON-строку ответа на запрос getUpdates

def get_updates_json():

    params = {'timeout': 100, 'offset': None}       # Ждём ответа на протяжении заданного в параметре timeout числа секунд

    response = requests.get(url + 'getUpdates', data=params)  # Выполняем Get-запрос, возвращается объект ответа

    return response.json()                            # Из объекта ответа берём JSON
```

Чат-бот для ВКонтакте

Если у кого-то возникли затруднения с созданием чат-бота для Telegram или работой с ним из-за блокировок, альтернативой является создание чат-бота для соц. сети ВКонтакте. Наиболее простым вариантом будет создать чат-бота сообщества (группы). Для этого, соответственно, необходимо создать для себя тестовое, можно закрытое, сообщество. В его настройках необходимо сделать доступными сообщения сообщества. Процесс создания данного чат-бота будет похожим на тот, что расписан для чат-бота в Telegram. Access token, аналогично, необходимо получить в настройках группы в разделе «Использование API» (описано в статьях по ссылкам ниже).



В остальном понадобится также только Python.
Работа с VK API достаточно подробно расписана в справке:

- https://vk.com/dev/first_guide
- https://vk.com/dev/bots_docs

Есть ряд статей, рассказывающих как создать своего чат-бота для ВКонтакте:

- https://habr.com/ru/post/427691/
- https://habr.com/ru/post/428507/
- https://habr.com/ru/post/335106/

Есть готовая библиотека для работы с VK API:

- https://github.com/python273/vk_api
- https://vk-api.readthedocs.io/en/latest/

Материалы для самостоятельного изучения

О дальнейшей работе с ботом можно посмотреть в статье, которая бралась за основу этого материала: https://tproger.ru/translations/telegram-bot-create-and-deploy/ Деплоить бот не нужно.

Также за основу взята вот эта подробная инструкция по ботам: https://core.telegram.org/bots

В этой лабораторной мы разобрали самостоятельную работу с ботом, однако также существует библиотека-обёртка для работы с Bot API, которая значительно упрощает работу с API, защищает нас от допущения множества ошибок при работе с ботом и привносит дополнительный полезный функционал: https://github.com/python-telegram-bot/python-telegram-bot

Ещё статьи на эту тему:

- https://habr.com/ru/post/448310/ (здесь описана другая библиотека, упрощающая работу с ботами в Telegram)
- https://www.freecodecamp.org/news/learn-to-build-your-first-bot-in-telegram-with-python-4c99526765e4/

Таким образом, по аналогии, вы можете писать чат-боты и для других мессенджеров или платформ: для ВК, для Viber, Facebook Messenger и других. Логика работы бота сохраняется, изменяется только способ создания чат-бота на платформе и способы (или даже только адреса) запросов обновлений и отправки сообщений.

Мультиагентные системы.

Теория

Разработка агентов для

Программирование агентов в игре Screeps.

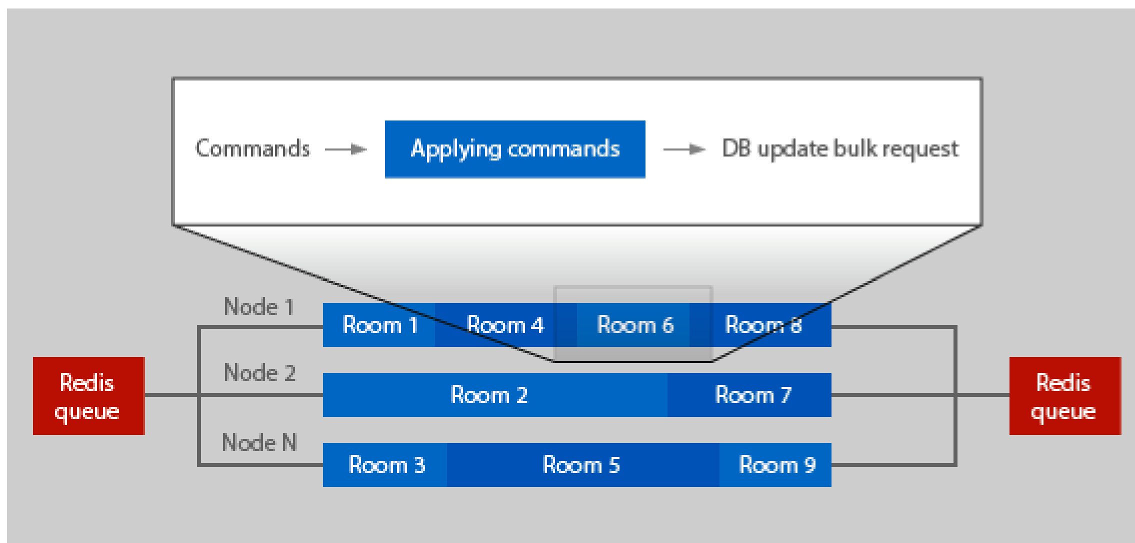
Screeps — игра для программистов: управляйте виртуальным миром с помощью JavaScript

стратегическая MMO-песочница реального времени с единым большим *persistent world*, в котором игроки не имеют никакого влияния на происходящее кроме как через написание скриптов AI для своих игровых юнитов. Все

механики обыкновенной стратегической игры — добыча ресурсов, создание юнитов, строительство базы, захват территорий, производство и торговля — требуется программировать самому игроку через JavaScript API, который предоставляется миром игры. Отличие от разных соревнований по написанию AI в том, что мир игры, как и полагается онлайновому игровому миру, постоянно работает и живет своей жизнью в реальном времени 24/7 на протяжении последних 4 лет, запуская AI каждого игрока каждый игровой такт.

Технические проблемы

Суть механики игрового мира заключается в следующем: весь мир разбит на комнаты, которые связаны между собой выходами по четырем сторонам света. Одна комната является атомарной единицей процесса обработки состояния игрового мира. В комнате могут быть некие объекты (например, юниты), у которых есть свое состояние, и на каждом игровом такте они получают команды от игроков. Серверный обработчик берет по одной комнате за раз, выполняет эти команды, меняя состояние объектов, и коммитит новое состояние комнаты в базу. Эта система хорошо горизонтально масштабируется: можно добавлять больше обработчиков в кластер, и так как комнаты архитектурно изолированы друг от друга, то параллельно может обрабатываться столько комнат, сколько запущено обработчиков.



На данный момент у нас в игре 42 060 комнат. Серверный кластер из 36 четырехъядерных физических машин содержит 144 обработчика. Для формирования очередей мы используем Redis, весь бекенд написан на Node.js.

Это был один этап работы игрового такта. Но откуда берутся команды игроков? Специфика игры в том, что нет никакого интерфейса, где можно было бы кликнуть на юнита и сказать ему отправиться в определенную точку или

построить определенное сооружение. Максимум, что можно сделать в интерфейсе — поставить нематериальный флаг в нужном месте комнаты. Чтобы юнит пришел в это место и сделал необходимое действие, нужно, чтобы ваш скрипт на протяжении нескольких игровых тактов выполнял примерно следующее:

```
module.exports.loop = function() {
  let creep = Game.creeps['Creep1'];
  let flag = Game.flags['Flag1'];
  if(!creep.pos.isEqualTo(flag.pos)) {
    creep.moveTo(flag.pos);
  }
}
```

Получается, на каждом игровом такте нужно взять функцию `loop` игрока, выполнить её в полноценном JavaScript-окружении этого конкретного игрока (в котором существует сформированный для него объект `Game`), получить набор приказов для юнитов, и отдать их на следующий этап процессинга. Кажется, все довольно просто.

Проблемы начинаются, когда дело доходит до нюансов реализации. На данный момент у нас 1600 активных игроков в мире. Скрипты отдельных игроков уже язык не поворачивается назвать "скриптами" — некоторые из них содержат до 25к строк кода, компилируются из TypeScript или даже из C/C++/Rust через WebAssembly (да, мы поддерживаем wasm!), и реализуют концепцию настоящих миниатюрных ОС, в которых игроки разработали собственный пул игровых задач-процессов и их менеджмент через ядро, которое берет столько задач, сколько получается выполнить на данном игровом такте, выполняет их, а невыполненные откладывает в очередь до следующего такта. Так как на каждом такте ресурсы CPU и памяти игрока ограничены, то такая модель

хорошо работает. Хотя и не является обязательной — для начала игры новичку достаточно взять скрипт из 15 строк, который к тому же уже написан в рамках туториала.

Но теперь давайте вспомним, что скрипт игрока должен работать в настоящей машине JavaScript. И что игра работает в реальном времени — то есть JavaScript-машина каждого игрока должна постоянно существовать, работая с неким заданным темпом, чтобы не замедлять игру в целом. Этап выполнения игровых скриптов и формирования приказов юнитам работает примерно по такому же принципу, как обработка комнат — каждый скрипт игрока является задачей, которую берет на себя один обработчик из пула, в кластере работает множество параллельных обработчиков. Но в отличие от этапа процессинга комнат, здесь уже кроется немало трудностей.

Во-первых, нельзя просто распределять задачи по обработчикам случайным образом на каждом такте так, как это возможно делать в случае комнат. JavaScript-машина игрока должна работать без останова, каждый следующий такт — это просто новый вызов функции `loop`, но глобальный контекст должен продолжать существовать тот же. Грубо говоря, в игре разрешается делать примерно так:

```
let counter = 0;
let song = ['EX-', 'TER-', 'MI-', 'NATE!'];

module.exports.loop = function () {
  Game.creeps['DalekSinger'].say(song[counter]);
  counter++;
  if(counter == song.length) {
    counter = 0;
  }
}
```

Такой крип будет петь по одной строчке песни каждый игровой такт. Номер строчки песни `counter` хранится в глобальном контексте, который сохраняется между тактами. Если каждый раз выполнять скрипт этого игрока в новом процессе обработчика, то контекст будет теряться. Значит, все игроки должны быть распределены по конкретным обработчикам, и менять их должны как можно реже. Но как тогда быть с балансировкой нагрузки? Один игрок может затратить 500мс выполнения на этой ноде, а другой игрок — 10мс, и очень трудно спрогнозировать это заранее. Если на одну ноду вдруг попадут 20 игроков по 500мс, то работа такой нода займет 10 секунд, в течение которых все остальные будут ждать её завершения и простаивать. А чтобы перебалансировать этих игроков и перекинуть на другие ноды, приходится терять их контекст.

Во-вторых, окружение игрока должно быть хорошо изолировано от других игроков и от серверного окружения. И это касается не только безопасности, но и комфорта для самих пользователей. Если соседний игрок, выполняющийся на той же ноде в кластере, что и я, творит что попало, генерирует много мусора, и вообще ведет себя неподобающе, то я не должен это чувствовать. Так как ресурсом CPU в игре является время выполнения скрипта (он подсчитывается с момента старта и до конца метода `loop`), то трата ресурсов на посторонние задачи во время выполнения моего скрипта могут быть очень чувствительными, ведь расходуются из моего бюджета ресурсов CPU.

В попытках справиться с этими проблемами мы пришли к нескольким решениям.

Первая версия

Первая версия движка игры была основана на двух базовых вещах:

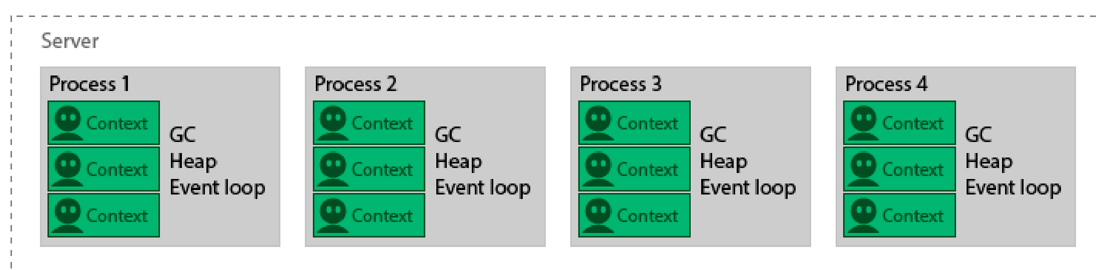
- штатный модуль `vm` в поставке Node.js,
- форки рантайм-процессов.

Выглядело это примерно следующим образом. На каждой машине в кластере существовало 4 (по числу ядер) процесса обработчиков игровых скриптов. При получении новой задачи из очереди игровых скриптов, обработчик запрашивал нужные данные из базы и передавал их в дочерний процесс, который поддерживался в постоянно запущенном состоянии, перезапускался в случае сбоя и переиспользовался разными игроками. Дочерний процесс, будучи изолированным от родительского (в котором содержалась бизнес-логика кластера), умел только одно: создать объект `Game` из полученных данных и

запустить виртуальную машину игрока. Для запуска использовался модуль `vm` в Node.js.

Почему это решение было неидеальным? Строго говоря, здесь не решались вышеописанные две проблемы.

`vm` работает в таком же однопоточном режиме, что и сам Node.js. Поэтому чтобы иметь на 4-ядерной машине четыре параллельных обработчика на каждом ядре, нужно иметь 4 процесса. Перемещение "живущего" в одном процессе игрока на другой процесс приводит к полному пересозданию глобального контекста, даже если это происходит в рамках одной и той же машины.



Кроме того, `vm` на самом деле не создает полностью изолированную виртуальную машину. Что оно делает, так это лишь создает изолированный *контекст*, или область видимости, но выполняет код в том же экземпляре виртуальной машине JavaScript, откуда происходит вызов `vm.runInContext`. А значит — в том же экземпляре, в каком запускаются и другие игроки. Хоть игроки и разделены по изолированным глобальным контекстам, но, будучи частью одной и той же виртуальной машины, имеют общую heap-память, общий garbage collector и генерируют мусор совместно. Если игрок "А" сгенерировал много мусора за время выполнения своего игрового скрипта, закончил работу, и управление перешло к игроку "Б", то в этот момент вполне может вызваться сбор *всего* мусора процесса, и игрок "Б" заплатит своим временем CPU за сбор чужого мусора. Не говоря уже о том, что все контексты работают в одном и том же event loop, и теоретически возможно выполнение чужого промиса в любой момент, хотя мы и пытались это предотвращать. Также `vm` не позволяет контролировать, сколько heap-памяти выделяется под выполнение скрипта, доступна вся память процесса.

isolated-vm

Живет на свете такой замечательный человек по имени Марсель Лаверде. Для одних он в свое время стал замечателен тем, что написал библиотеку node-fibers, для других — тем, что взломал Facebook и был нанят там работать. А

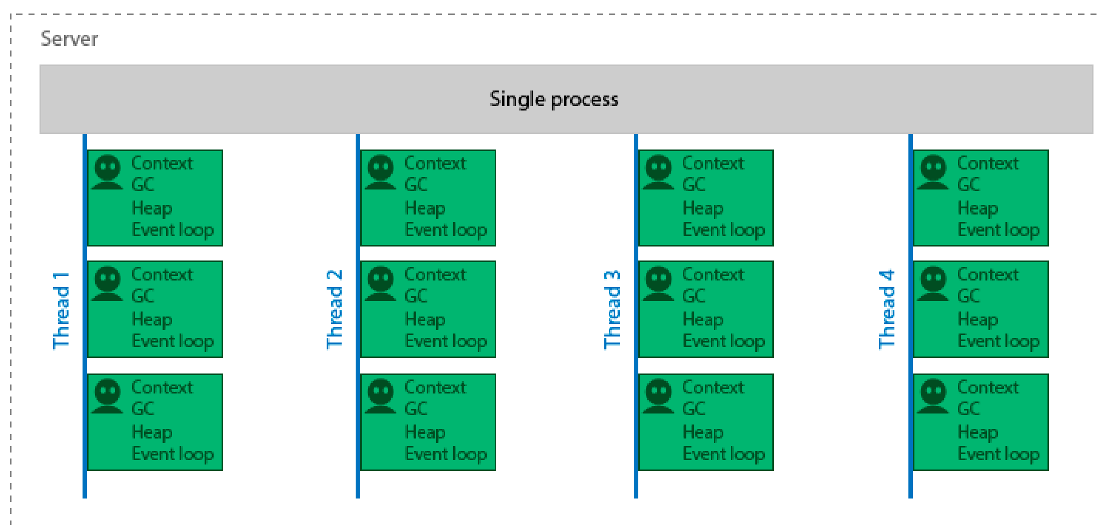
для нас он замечателен потому, что щедро участвовал в нашей самой первой краудфандинговой кампании и по сей день является большим фанатом Screeps.

Наш проект уже несколько лет как вышел в опенсорс — сервер игры опубликован на GitHub. Хотя официальный клиент и продается платно через Steam, но существуют альтернативные его версии, а сам сервер доступен для изучения и модификации в любых масштабах, что мы всячески поощряем.

И вот как-то раз Марсель пишет нам: «Ребята, у меня хороший опыт в нативной разработке C/C++ под Node.js, и мне нравится ваша игра, но не во всем нравится как она работает — давайте мы с вами напишем совершенно новую технологию запуска виртуальных машин под Node.js специально для Screeps?».

Так как денег Марсель не просил, мы не смогли отказаться. Через несколько месяцев нашего сотрудничества на свет появилась библиотека isolated-vm. И это поменяло абсолютно все.

`isolated-vm` отличается от `vm` тем, что изолирует не *контекст*, а *isolate* в терминах V8. Не вдаваясь в детали, это означает, что создается полноценный отдельный экземпляр JavaScript-машины, который обладает не только собственным глобальным контекстом, но и собственной heap-памятью, сборщиком мусора и работает в рамках отдельного event loop. Из минусов: на каждую запущенную машину требуется небольшой оверхед RAM (порядка 20 Мб), а также внутрь машины невозможно передавать объекты или вызывать функции напрямую, весь обмен надо сериализовать. На этом минусы заканчивается, в остальном — это просто панацея!



Теперь стало действительно возможным запускать скрипт каждого игрока в своем собственном полностью изолированном пространстве. У игрока есть свои 500 Мб хипа, если он закончился — то это значит, что закончился именно твой собственный хип, а не хип общего процесса. Если сгенерировал мусор — то это твой собственный мусор, тебе его и собирать. Повисшие промисы выполнятся только тогда, когда твоему изоляту перейдет управление в следующий раз, и не ранее. Ну и секьюрность — ни при каких обстоятельствах невозможно получить доступ куда-то вовне изолята, только если найти где-то уязвимость на уровне V8.

Но что насчет балансировки? Еще один плюс isolated-vm в том, что он запускает машины из этого же процесса, но в отдельных тредах (здесь пригодился опыт работы Марселя над node-fibers). Если у нас 4-ядерная машина, мы можем создать пул из 4 тредов, и запускать в один момент времени 4 параллельных машины. При этом находясь в рамках одного и того же процесса, а значит, имея общую память, мы можем перекидывать любого игрока из одного треда в другой внутри этого пула. Хоть каждый игрок и остается привязанным к одному конкретному процессу на одной конкретной машине (чтобы не терять глобальный контекст), но балансировки между 4 тредами оказывается достаточно, чтобы решить проблемы распределения "тяжелых" и "легких" игроков между нодами так, чтобы все обработчики заканчивали работу одновременно и вовремя.

После обкатки этой функции в экспериментальном режиме мы получили огромное количество положительных отзывов от игроков, скрипты которых стали работать гораздо лучше, стабильнее и предсказуемее. И теперь это наш движок по умолчанию, хотя игроки до сих пор могут по желанию выбрать legacy runtime чисто в целях обратной совместимости со старыми скриптами (некоторые игроки осознанно ориентировались на специфику shared-окружения в игре).