Математические структуры и моделирование 2020. № 3(55). С. 103–116

УДК 004.42
DOI 10.24147/2222-8772.2020.3.103-116

АЛГОРИТМ РАСПОЗНАВАНИЯ ТЕКСТОВЫХ КОМАНД ПОСТРОЕНИЯ ПРОИЗВОЛЬНОЙ ФОРМЫ ВВОДА ДАННЫХ

Е.А. Тюменцев
преподаватель, e-mail: etyumentcev@gmail.com
Т.В. Мелешенко
студентка, e-mail: tanyamelesh@gmail.com

Омский государственный университет им. Ф.М. Достоевского, Омск, Россия

Аннотация. В данной статье предлагается решение задачи обработки запросов, записанных на русском языке для построения форм ввода данных. Описаны этапы лексического и синтаксического анализов запросов с примерами. В конце приведена ссылка на работающий проект и репозиторий с исходным кодом на языке Kotlin.

Ключевые слова: обработка, русский, естественный, язык, анализатор, лексема, правило, NLP, извлечение фактов из текста.

Введение

Приложения используют формы ввода данных.

В некоторых случаях форм становится настолько много, что разработка новых форм и доработка существующих составляют значительную часть всех затрат на проект.

Задачи по редактированию форм заключаются в повторяющихся однотипных операциях. Добиться снижения затрат на разработку форм предлагается за счёт голосового ввода команд и их автоматической обработки без участия программиста. Чтобы исключить программиста из данного процесса, система будет построена на следующих идеях:

1) разработать систему команд на естественном языке;

2) использовать WYSIWYG-редакор, чтобы максимально сократить обратную связь между командами пользователя и результатом, который он получает.

В настоящей статье будет описан алгоритм распознавания речевых команд пользователя по редактированию форм ввода данных.

Первичные результаты проекта были изложены на конференции «Молодёжь третьего тысячелетия XLII» 2019 [1].

1. Описание идеи

Задачу по определению команды из голосового запроса можно разделить на три подзадачи:

1) преобразование человеческой речи в текст;

2) распознавание текстовой команды и преобразование её во внутренний формат;

3) интерпретация команды из внутреннего формата.

Для решения первой подзадачи используется Google API, т. к. оно бесплатно и с приемлемым качеством преобразовывает голос в текст.

Распознавание текстовой команды представляет собой задачу извлечения смысла из текста. Наиболее популярными для решения подобных задач сейчас являются механизмы машинного обучения с использованием нейронных сетей. В нашем случае применение нейронных сетей затруднено отсутствием обучающего набора данных. Ожидается, что набор команд ограничен несколькими сотнями, поэтому было решено выбрать метод обучения с учителем без применения нейронных сетей. Безусловно, в этом подходе есть недостаток: качество результата сильно зависит от знаний учителя в предметной области. Однако в нашем случае он компенсируется небольшим размером предметной области.

Наш проект строится на основе гипотезы: каждую осмысленную последовательность слов, которую вводит человек, можно разбить на группы. Каждой группе можно по смыслу сопоставить команду.

Выделение смысловых групп будет осуществляться с помощью дерева принятия решений.

Для иллюстрации рассмотрим следующий пример.

1) «добавить фамилию»;

2) «добавить поле фамилия»;

3) «добавить выпадающий список»;

4) «добавить бежевый фон»;

5) «добавить ограничение в 10 символов»;

6) «добавить фамилия»;

7) «добавить фамилию в форму»;

8) «добавить фамилию в форму ввода паспорт».

Предположим, что мы хотим обработать вышеописанные команды.

Обратим внимание на примеры 1 и 6. Они практически идентичны за исключением лишь окончания слова «фамилия». Чтобы избавиться от необходимости

проводить морфологический разбор слов, а также сократить набор команд, решено приводить нормализацию слов и использовать алгоритм Стемминга[4] для отсечения конца слова. Особенностями реализованного стеммера является то, что отсекается только первое найденное совпадение с описанными внутри него правилами, а также и то, что он отсекает окончание только в том случае, если длина слова будет больше 4.

После нормализации команды будут выглядеть так:

1) «добав фамил»;

2) «добав поле фамил»;

3) «добав выпадающ список»;

4) «добав бежев фон»;

5) «добав ограничен в 10 символ»;

6) «добав фамилия»;

7) «добав фамилию в форм»;

8) «добав фамилию в форм ввод паспорт».

По алгоритму, представленному на рисунке 1, построим дерево принятия решений, которое может распознавать эти восемь команд. Рёбра будем помечать нормализованными словами из команды, а узлы, которые содержат правило, отмечать дополнительным кружком внутри. Таким образом, каждому конечному узлу будет соответствовать полностью распознанная фраза.

На рисунке 2 изображено дерево принятия решений, полученное с помощью алгоритма для вышеописанного примера.

Бывает так, что одна команда является частью другой, более длинной. Это можно наблюдать на примере запросов 1, 7 и 8. Чтобы избежать подобных ситуаций, было решено распознавать максимально длинную цепочку слов, т. е. выбирать правило, которое находится на максимально возможной глубине дерева.

Рассмотрим команды «создать поле фамилия» и «удалить поле фамилия». В вышеприведённом дереве им будут соответствовать два разных правила. Однако по смыслу обе команды представляют собой действия, совершаемые над полем «фамилия». Чтобы сократить количество правил в подобных случаях будем рассматривать не сами слова, а их смысловые значения, которое будем называть лексемами. А процесс получения лексемы из слова будем называть лексическим анализом.

Теперь так будут выглядеть приведённые ранее команды:

1) «действие» «частьИмени»;

2) «действие» «объект» «частьИмени»;

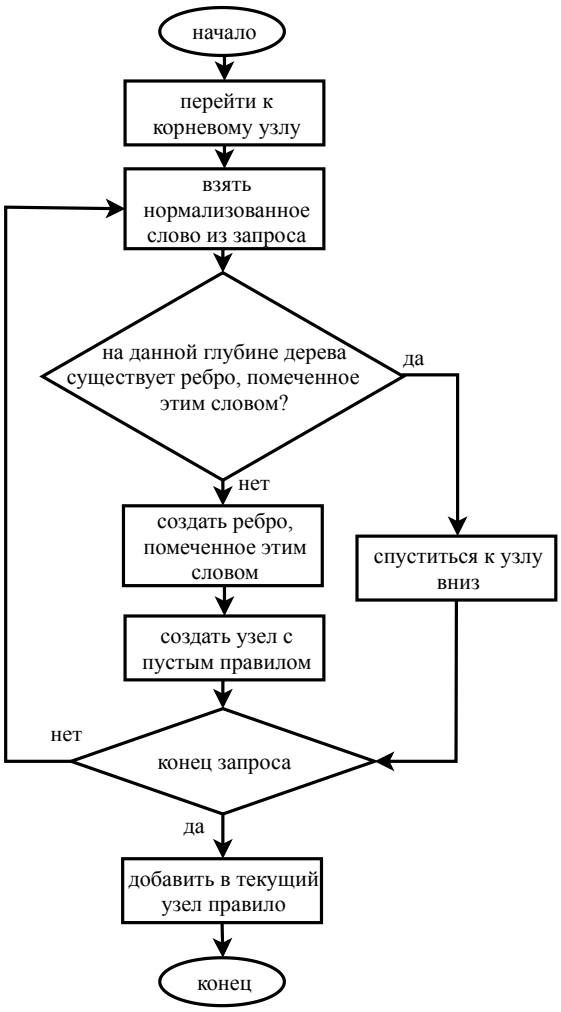

Рис. 1. Алгоритм построения дерева принятия решений

3) «действие» «видСписка» «видСписка»»;

4) «действие» «свойство» «имяСвойства»»;

5) «действие» «имяСвойства» «в» «свойство» «едСчисления»;

6) «действие» «частьИмени»»;

7) «действие» «частьИмени» «в» «объект»»;

8) «действие» «частьИмени» «в» «объект» «объект» «частьИмени»».

Из-за применения стеммера, а также и из-за неоднозначности естественного языка слову присваивается список лексем. В таком случае возникает вероятность выбрать неправильную лексему и не разобрать команду вовсе, но это было решено простым перебором списка, т. е. если первая лексема не подходит, мы спокойно заменим её на следующую. Например, два слова «другой» и «друг» после обработки стеммером будут сокращены до слова «друг», однако в

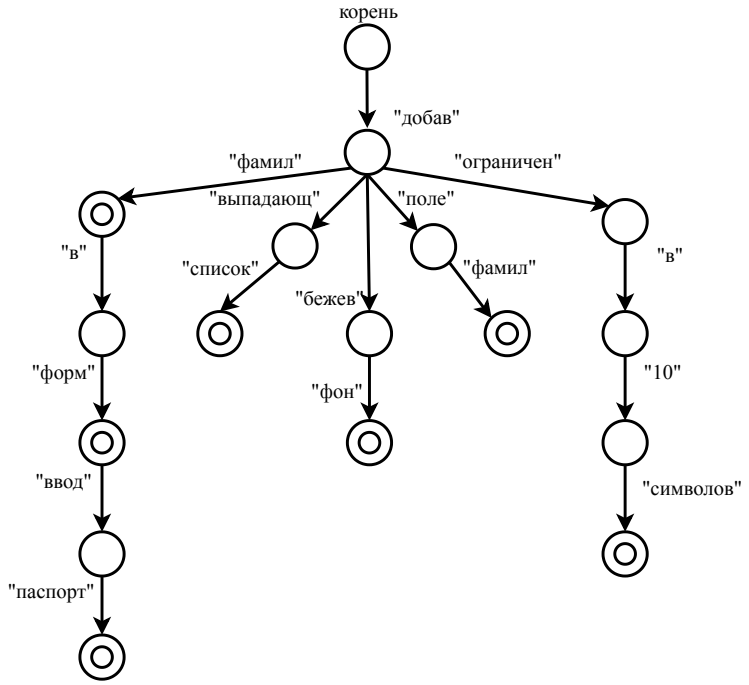

Рис. 2. Пример дерева принятия решений

первом случае скорее всего имеется ввиду ключевое слово, а во втором вероятнее — часть имени (о возможных значениях лексем будем говорить в главе 2). Сначала рассматривается вариант с ключевым словом, и если правила с таким набором лексем нет — она будет замена на «часть имени», поэтому на уровне лексического анализа эти неоднозначности не критичны.

Дерево принятия решений теперь будет строиться именно по лексемам. А когда дерево построено, по нему можно проверять все приходящие запросы. Перечислим все возможные шаги прохода по дереву в поисках правила:

1. Получаем из лексического анализатора объект, который хранит само слово из списка его лексем.

2. Берём его j-ю лексему и проверяем, содержится ли в дереве ребро, помеченное этой лексемой.

   (a) Если ребро, помеченное лексемой существует, то идём к шагу 3.

   (b) Если не содержит, в таком случае необходимо проверить существуют ли ещё лексемы у данного слова.

      i. Если есть, то просто увеличиваем j-ый счётчик и возвращаемся к шагу 1.

      ii. Если нет, нужно проверить, является ли текущее слово первым.

         A. Если да, тогда необходимо перейти к шагу 5.

         B. Если нет, тогда просто переходим к предыдущему слову, обнулив j-ый счётчик для i-го слова.

3. Посмотреть, существует ли правило на данном узле.

   (a) Если да, тогда нужно запомнить выбор с параметрами, при которых алгоритм достиг узла с правилом. И перейти к шагу 4.

4. Проверить, является ли текущее слово последним в запросе?

   (a) Если нет, то нужно увеличить i-ый счётчик на один, j-ый счётчик i-го слова задать равным нулю. И вернуться к шагу 1.

   (b) Если же слово последнее, то переходим к шагу 5.

5. Взять последний выбор и вернуть правило во внутреннем формате. На этом работа алгоритма завершена.

Для наглядности на рисунке 3 приведена блок-смеха описанного алгоритма.

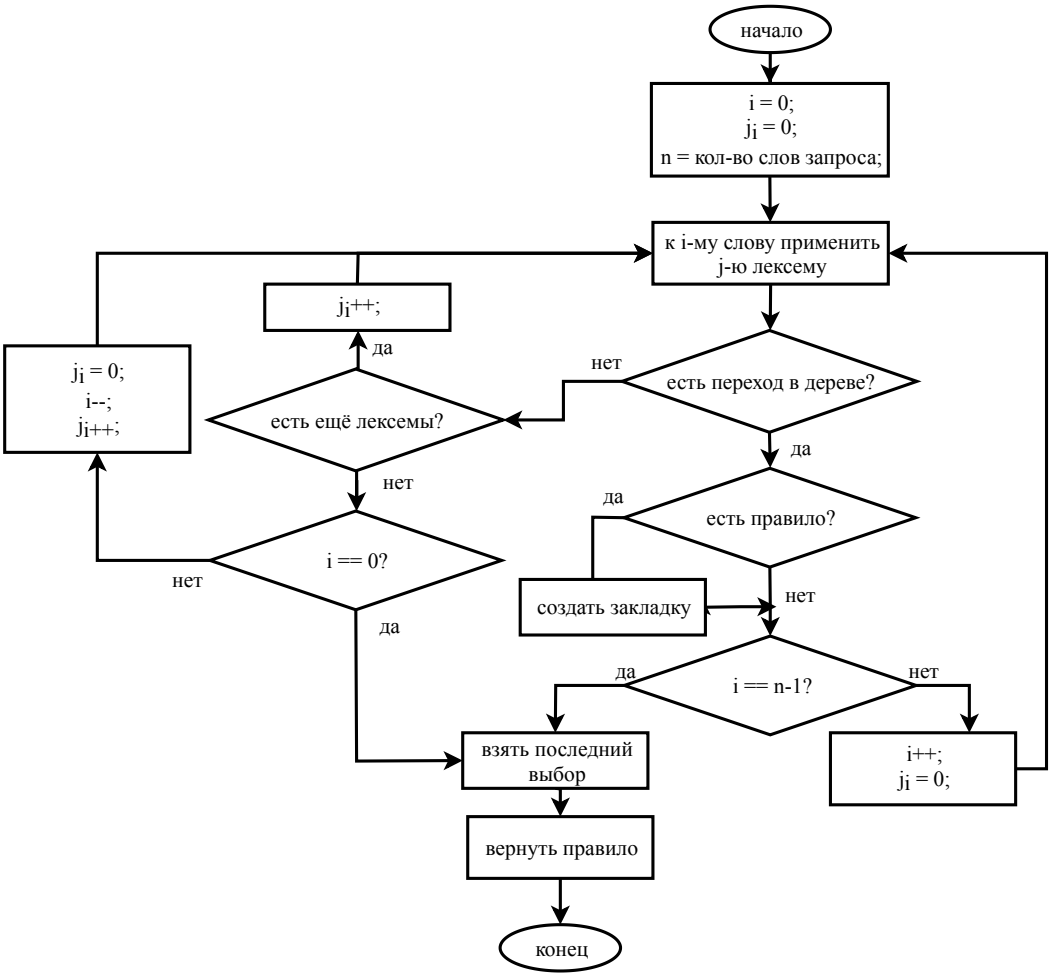

Рис. 3. Алгоритм разбора дерева принятия решений

2. Реализация

2.1. Лексический анализ

Описанный алгоритм реализован на языке Kotlin.

Перед тем, как непосредственно провести лексический анализ, необходимо полученную строку нормализовать. В рамках данного проекта «нормальной» является строка, содержащая только прописные буквы. Тогда приложение получится нечувствительным к регистру, однако этим недостатком было решено пренебречь. Теперь можно запустить анализатор, выделяющий условные «слова», которые представляют из себя последовательность букв русского и/или английского алфавитов и арабские цифры, в том числе не разделённые пробелом (например, «форма12»). Запуск анализатора производится с помощью вызова метода parse(), содержащегося в классе LexemeAnalyzer, диаграмма класса которого приведена на рисунке 4.

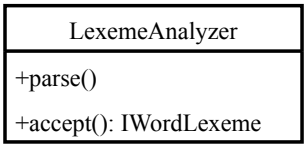

Рис. 4. Диаграмма класса LexemeAnalyzer

Анализатор построен на основе конечного автомата, представленного на рисунке 5.

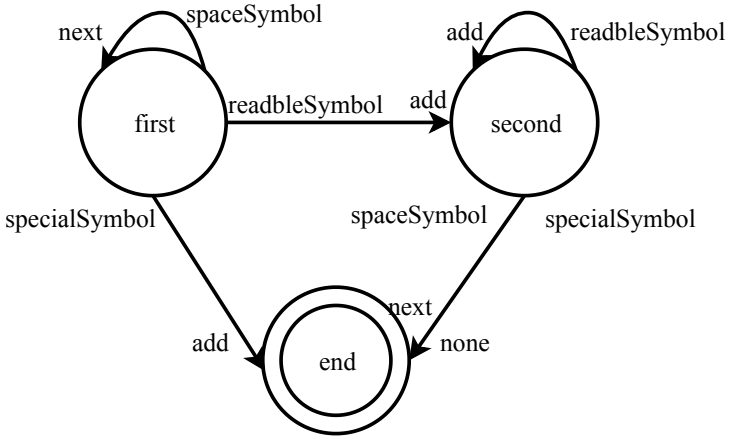

Рис. 5. Конечный автомат лексического анализатора

В таблице 1 опишем значения входящих сигналов автомата.

Для распределения всех входящих символов на описанные выше группы в программе имеется словарь.

В данном проекте выходные сигналы «next», «add», «none» представляют собой методы автомата лексического анализатора, которые будут выполнены при получении соответствующих входящих сигналов.

Таблица 1. Значения входящих сигналов

Входящий сигнал	Значение
«readableSymbol»	все буквы русского и латинского алфавитов; все римские цифры.
«specialSymbol»	символы-разделители
«spaceSymbol»	пробельные символы

Опишем более подробно значения выходных сигналов:

- «next» — перейти к следующему символу;
- «add» — прочли символ, который составляет очередное «слово» ;
- «none» — группа символов, которые составляют «слово», составлена.

После завершения работы автомата в его памяти будет лежать строка с последовательностью прочитанных символов. Её тоже нужно привести к «нормальному» формату. В этом случае «нормальным» считается обработанное стеммером слово без пробела на конце и/или в начале.

Результат в виде обработанного стеммером ключа — совсем не то, что мы ожидаем от лексического анализатора. Нам необходимо получить набор лексем. Для этого у нас есть специальный словарь этих лексем с таким форматом:

```
{
    "word":<слово>,
    "valuesList":<список лексем>
}
```

Теперь можно отметить, что все цифровые значения в словаре для простоты заменяются специальным обозначением «NUM», что происходит на этапе нормализации «слова». Например, по запросу «форма12» получим такой объект:

```
{
    "word":"формаNUM",
    "valuesList":["частьИмени","свойство"]
}
```

Обычное же числовое значение в запросе «скругление в 12» будет представлено как

```
{
    "word":"NUM",
    "valuesList":["частьИмени","свойство"]
}
```

Поговорим о возможных значениях "valuesList". Их можно условно поделить на три вида: символы, собственно-лексемы и неизменяемые слова. К первому относятся пробельные символы, знаки препинания и прочие символы. Ко второму относятся обобщающие слова, которые были выделены в процессе разработки анализатора. Вероятно, ещё будут вноситься изменения, однако на данный момент список включает в себя следующие обобщения: видЛиста, видПоля, видСписка, видТекстовогоПоля, едСчисления, действие, имяСвойства, объект, свойство, указатель, частьИмени. Названия обобщающих слов комментариев не требуют, а потому переходим к следующему виду.

Он представляет собой набор тех слов, которые можно выделить, как ключевые. В работе ключевыми считаются все простые предлоги, а также слова «режим», «новый» и некоторые другие. Список последних также может пополняться по мере дальнейшей разработки других анализаторов и рассмотрения новых команд. Конечно, все слова в словаре записаны в том представлении, в котором они получаются на выходе стеммера.

Возможно, данная группа при дальнейшей работе над проектом будет модифицирована или удалена вовсе. Сейчас же она необходима для упрощения разработки правил, по которым описываются действия.

Для наглядности проведём анализ команды «создать форму Опрос».

Посимвольно считываем строку до тех пор, пока не дойдём до конечного состояния.

Теперь память автомата содержит строку «создать», т. к. по переходам автомата (рисунок 5) пробелы не записываются в память. На выходе из стеммера получаем «созд».

Ищем набор лексем по ключу «созд», и итогом работы анализатора будет объект, созданный из этой строки формата JSON:

```
{
    "word":"созд",
    "valuesList":["действие","частьИмени"]
}
```

Лексема «действие» означает, что данное слово будет означать собственно действие, которое необходимо выполнить. Лексема «частьИмени» означает, что данное слово может быть частью имени объекта, например, в запросе «открыть форму какого цвета создать форму».

Запускаем автомат дальше и на выходе стеммера получим «форм», а объект:

```
{
    "word":"форм",
    "valuesList":["объект","имяСвойства"]
}
```

Здесь словоформа «форм» может принимать три значения в зависимости от контекста:

- «объект» — в том случае, если говорим об объекте, который имеет тип «форма»;

- «имяСвойства» — является названием свойства действия, например, в запросе «установить отступ формы», оба слова «отступ», «формы» являются «имяСвойства».

С помощью автомата получим «опрос» и подадим на вход стеммера, на выходе ничего не поменялось, поэтому остаётся «опрос». Получим такой объект:

```
{
    "word":"опрос",
    "valuesList":["частьИмени"]
}
```

На этом разбор примера закончим. Далее перейдём к рассмотрению работы синтаксического анализа.

2.2. Синтаксический анализ

На уровне синтаксического анализа необходимо выделить наборы синтаксических групп, которые могут представлять собой команду следующего формата:

```
{
    "lexemesList":<список лексем>
}
```

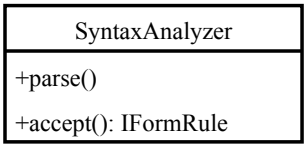

Рис. 6. Диаграмма класса SyntaxAnalyzer

В нашем проекте синтаксический анализ проводится с помощью класса SyntaxAnalyzer, диаграмму которого видно на рисунке 6. Класс содержит публичный метод accept(), который возвратит полученное правило в формате:

```
{
    "lexemesList": [
                        {
                            "first":<лексема>,
                            "second":<словоформа>
                        },
                    ]
}
```

В методе parse() для разбора входящего запроса используется дерево принятия решений. Его структуру можно описать следующим образом: пустой корень, от которого по рёбрам, описывающимися лексемами, можно спуститься на

уровень ниже — к узлу дерева, где может лежать правило, если оно существует в словаре правил, а может и не лежать.

Структуру дерева принятия решений изобразим графически. Для этого определим несколько правил, по которым построим дерево:

- {"lexemesList":["действие"]}
- {"lexemesList":["действие "объект"]}
- {"lexemesList":["действие "объект "частьИмени"]}
- {"lexemesList":["действие "объект "частьИмени "свойство "имяСвойства"]}
- {"lexemesList":["действие "свойство "в "объект "частьИмени"]}

Аналогично тому, как это было описано в главе 1, получаем дерево, которое представлено на рисунке 7.

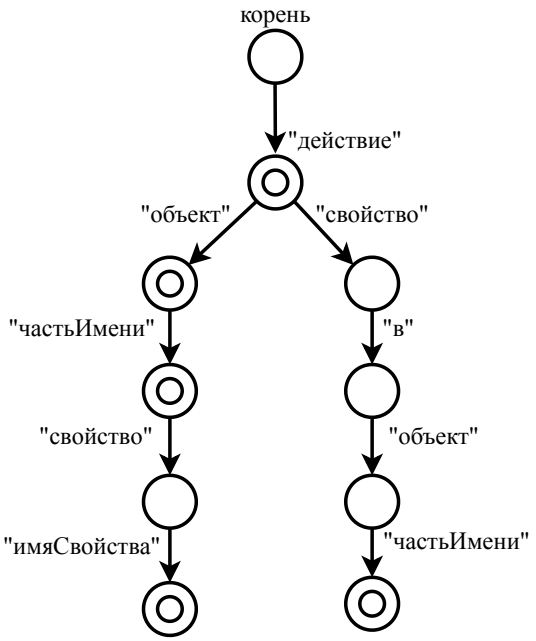

Рис. 7. Дерево принятия решений

Приведём простой пример разбора полученного выше дерева.

Пользователь дал запрос «создать форму Опрос».

Сначала мы обратимся к лексическому анализатору для получения первого «слова» и его лексем. Сразу возьмём результат этого запроса из прошлой главы: {"word":"созд "valuesList":["действие "свойство "частьИмени"]}.

Берём первую возможную лексему («действие») и обращаемся к дереву принятия решений. Исходя из его структуры (рисунок 7), видим, что ребро, помеченное как «действие», у нас действительно есть. Значит, проверяем на наличие правила. Узел содержит правило, поэтому надо запомнить выбор, т. е.

сохранить параметры, при которых достигли этого результата. Данное «слово» последним не является, поэтому переходим к следующему.

Теперь смотрим на объект {"word":"форм "valuesList":["объект "имяСвойства "свойство"]}, берём его первую лексему («объект»), ребро с такой пометкой имеется, переходим к узлу ниже, также существует правило, однако «слово» не последнее. И идём дальше.

Следующий объект {"word":"опрос "valuesList":["частьИмени"]}. Здесь всего одна возможная лексема, которая есть у текущего узла при переходе на нижний уровень. Сменили узел и замечаем, что он содержит правило. Это последнее «слово». Берём последний сохранённый выбор, возвращаем правило {"lexemesList":[{"first":"действие "second":"созд"}, {"first":"объект "second":"форм"}, {"first":"частьИмени "second":"опрос"}]}

Возможны ситуации, когда по одному и тому же набору «слов» можно распознать два правила. Такие ситуации в будущем планируется обрабатывать с помощью исключений.

На этом закончим разбор алгоритма работы синтаксического анализатора.

2.3. Неоднозначности

Предположительно, по набору слов не всегда можно получить одну определённую команду. Тогда результатом разбора будет являться не команда, а список команд. И в таком случае, вероятно, будет использоваться система диалога с пользователем для уточнения его команды.

Также на примере запроса «цвет фона красный» можно заметить, что не всегда в запросе содержится полная информация о том, где должно быть выполнено действие пользователя, это связано с тем, что человеку свойственна определённая компактность речи. Эту неоднозначность можно решить на уровне интерпретатора команды: запоминать последний текущий элемент и применять изменения к нему. Однако если в интерпретаторе не будет нужной информации, можно также использовать систему диалога.

Возможны и такие случаи, когда вообще не удастся распознать команду. Например, команда «Открыть» вызывает вопрос и у человека: «Что открыть?», поэтому и в таком случае можно использовать диалоговую систему.

Вероятнее всего, найдётся некоторое множество примеров команд, которые будет невозможно распознать данным алгоритмом. Однако мы ожидаем, что оно будет значительно меньше подлежащего обработке множества.

Далее подробно обсудим реализацию идеи, написанную на языке Kotlin.

Заключение

Получены алгоритмы и их реализации лексического и синтаксического анализов голосовых команд редактирования форм данных. На данный момент всего распознаётся 216 голосовых команд на уровне синтаксического анализа.

Исходный код можно посмотреть на GitLab[2]. Также дорабатывается семантический анализатор, после чего понадобится создать больший набор за-

просов для дальнейшей работы над проектом и его совершенствованием.

Литература

1. Бутерус Д.В., Мелешенко Т.В, Молодцов И.Н., Тюменцев Е.А. К вопросу построения речевого интерфейса генератора форм ввода данных на основе искусственного интеллекта // Молодёжь третьего тысячелетия [Электронный ресурс] : сборник научных статей. Электрон. текст. дан. Омск : Изд-во Ом. гос. ун-та, 2019. С. 256–259.
2. Репозиторий проекта с анализаторами. URL: https://gitlab.com/tanyamelesh/forms (дата обращения 14.01.2020).
3. Репозиторий проекта для тестирования. URL: https://gitlab.com/tanyamelesh/formsweb (дата обращения 15.01.2019).
4. Lovins J.B. Development of a Stemming Algorithm // Mechanical Translation and Computational Linguistics, 1968. URL: http://mt-archive.info/MT-1968-Lovins.pdf (дата обращения 28.01.2020).

ALGORITHM FOR RECOGNIZING TEXT COMMANDS FOR CONSTRUCTING AN ARBITRARY FORM OF DATA INPUT

E.A. Tumentcev
Instructor, e-mail: etyumentcev@gmail.com
T.V. Meleshenko
Student, e-mail: tanyamelesh@gmail.com

Dostoevsky Omsk State University, Omsk, Russia

Abstract. The article proposes a solution to the problem of processing requests recorded in Russian to building data entry forms. Lexeme and syntactic analyzes stages request with examples are described. In the end of article, there is link to working project and repository with source code in Kotlin language.

Keywords: processing, russian, natural, language, analyzer, lexem, rule, NLP, extracting facts from text.

References

1. Buterus D.V., Meleshenko T.V, Molodtsov I.N., and Tyumentsev E.A. K voprosu postroeniya rechevogo interfeisa generatora form vvoda dannykh na osnove iskusstvennogo intellekta. Molodezh' tret'ego tysyacheletiya [Elektronnyi resurs] : sbornik nauchnykh statei, Elektron. tekst. dan., Omsk, Izd-vo Om. gos. un-ta, 2019, pp. 256–259. (in Russian)
2. Repozitorii proekta s analizatorami. URL: `https://gitlab.com/tanyamelesh/forms`. (in Russian)

3. Repozitorii proekta dlya testirovaniya. URL: `https://gitlab.com/tanyamelesh/formsweb`. (in Russian)
4. Lovins J.B. Development of a Stemming Algorithm, Mechanical Translation and Computational Linguistics, 1968. URL:`http://mt-archive.info/MT-1968-Lovins.pdf`.

Дата поступления в редакцию: 13.07.2020