Томита-парсер

Быстрый старт

29.12.2016

Яндекс

Томита-парсер. Быстрый старт. Версия 1.0
Дата сборки документа: 29.12.2016.
Этот документ является составной частью технической документации Яндекса.
Сайт справки к сервисам Яндекса: http://help.yandex.ru

Предупреждение об исключительных правах

Контактная информация

ООО «ЯНДЕКС»
http://www.yandex.ru
Тел.: +7 495 739 7000
Email: pr@yandex-team.ru
Главный офис: 119021, Россия, г. Москва, ул. Льва Толстого, д. 16

Содержание

Об учебнике

Томита-парсер — это инструмент для извлечения структурированных данных (фактов) из текста на естественном языке. Извлечение фактов происходит при помощи контекстно-свободных грамматик и словарей ключевых слов.

Настоящая инструкция предназначена для начинающих пользователей парсера. Она даёт общее представление о том, как составлять грамматики и словари, как настраивать и запускать парсер. Описание синтаксиса грамматик, списки используемых помет, спецификации форматов генерируемых файлов приведены в Руководстве пользователя.

Исходные файлы простого проекта

Для запуска Томита-парсера необходимо создать файлы, перечисленные в следующей таблице.

Содержание	Формат	Примечания
config.proto — конфигурационный файл парсера. Сообщает парсеру, где искать все остальные файлы, как их интерпретировать и что делать.	Protobuf	Нужен всегда.
dic.gzt — корневой словарь. Содержит перечень всех используемых в проекте словарей и грамматик.	Protobuf / Gazetteer	Нужен всегда.
mygram.cxx — грамматика	Язык описания грамматик	Нужен, если в проекте используются грамматики. Таких файлов может быть несколько.
facttypes.proto — описание типов фактов	Protobuf	Нужен, если в проекте порождаются факты. Парсер запустится без него, но фактов не будет.
kwtypes.proto — описания типов ключевых слов	Protobuf	Нужен, если в проекте создаются новые типы ключевых слов.

Таким образом, минимальный набор файлов для запуска парсера включает конфигурационный файл и корневой словарь (см. пример **minimal**). При этом не будут использоваться грамматики и пользовательские типы ключевых слов, а также не будут порождаться факты. Типовой сценарий использования парсера подразумевает наличие всех пяти типов файлов.

Исходные файлы проектов-примеров в этом руководстве

Название	файл конфигурации	корневой словарь	грамматика	типы фактов	типы ключевых слов	тексты
minimal	config.proto	mydic.gzt				
tutorial1	config.proto	mydic.gzt	first.cxx			test.txt
tutorial2	config.proto	mydic.gzt	adjperson.cxx			test1.txt, test2.txt, test3.txt
tutorial3	config.proto	mydic.gzt	fioborn.cxx			test.txt
tutorial4	config.proto	mydic.gzt	date.cxx	facttypes.proto		test.txt
tutorial5	config.proto	mydic.gzt, animals_dict.gzt, mammals.txt	main.cxx, date.cxx	facttypes.proto	kwtypes.proto	test.txt

Простейшие правила

Томитные грамматики работают с цепочками. Одна цепочка соответствует одному предложению в тексте. Из цепочки выделяются подцепочки, которые, в свою очередь, интерпретируются в разбитые по полям факты.

Правила грамматики, которые выделяют подцепочки, выглядят так:

```
S -> Noun;
```

Это правило выделяет из цепочки все существительные. Например, из текста **(1)** оно выделит подцепочки *труд* и *человек*.

(1) *Труд облагораживает человека*

Обратим внимание, что при выделении цепочек по умолчанию происходит их нормализация, т.е. они приводятся к начальной форме.

В данном правиле `Noun` является терминалом. Терминал может стоять только в правой части правила. В Томите к терминалам относятся названия частей речи (`Noun`, `Verb`, `Adj`) , некоторые символы (`Comma`, `Punct`, `Ampersand`, `PlusSign`), а также леммы (`'лес'`, `'бежать'`) (см. список терминалов).

`S` — это нетерминал, который строится из терминалов и должен хотя бы один раз встретиться в левой части правила. Если нетерминал встречается только в левой части и никогда — в правой, значит, это вершина грамматики. Ее также можно указать явно:

```
#GRAMMAR_ROOT S    // При помощи директивы #GRAMMAR_ROOT указывается вершина грамматики – нетерминал S
S -> Noun;
```

Также в начале можно указать кодировку, в которой написана грамматика:

```
#encoding "utf-8"    // сообщаем парсеру о том, в какой кодировке написана грамматика
#GRAMMAR_ROOT S      // указываем корневой нетерминал грамматики
S -> Noun;             // правило, выделяющее цепочку, состоящую из одного существительного
```

Чтобы запустить грамматику, нам надо создать корневой словарь. Каждую новую грамматику, которую мы будем создавать, нужно будет добавлять туда отдельной статьей. Создадим файл first.cxx, где мы будем писать нашу первую грамматику. Пока там будет только 3 строчки, приведенные выше.

Далее создадим файл mydic.gzt — корневой словарь, и запишем в него следующее:

```
encoding "utf8";                // указываем кодировку, в которой написан этот файл
import  "base.proto";                   //  подключаем  описания  protobuf-типов (TAuxDicArticle и прочих)
import "articles_base.proto";  // Файлы base.proto и articles_base.proto встроены в компилятор.
                                  // Их необходимо включать в начало любого gzt-словаря.
// статья с нашей грамматикой:
TAuxDicArticle "наша_первая_грамматика"
{
    key = { "tomita:first.cxx" type=CUSTOM }
}
```

Подробнее о файлах с раширением `.gzt` будет сказано ниже.

Далее надо создать файл с параметрами, где мы укажем путь к корневому словарю, путь к входному и выходному файлу и другую информацию, необходимую для запуска грамматики. Создадим файл `config.proto` и запишем туда следующее:

```
encoding "utf8"; // указываем кодировку, в которой написан конфигурационный файл
TTextMinerConfig {
  Dictionary = "mydic.gzt"; // путь к корневому словарю
  PrettyOutput = "PrettyOutput.html"; // путь к файлу с отладочным выводом в уд-
обном для чтения виде
  Input = {
    File = "test.txt"; // путь к входному файлу
  }
  Articles = [
    { Name = "наша_первая_грамматика" } // название статьи в корневом словаре,
                                        // которая содержит запускаемую
грамматику
  ]
}
```

Нам осталось только создать файл `test` и записать туда предложения, которые мы хотим обработать нашей грамматикой:

Труд облагораживает человека.

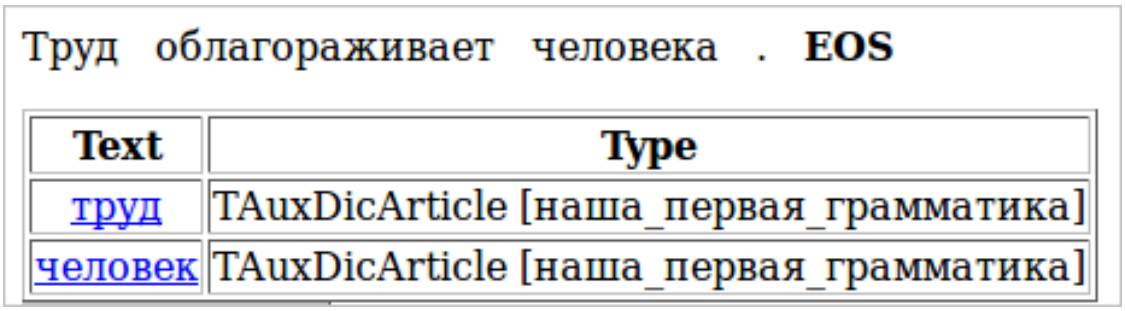
Для запуска грамматики нам понадобится программа tomitaparser, параметром которой будет файл `config.proto`. Итак, в командной строке нам надо написать такую команду:

```
В Linux, FreeBSD и прочих *nix системах:
./tomitaparser config.proto
В Windows:
tomitaparser.exe config.proto
```

На экране появятся сообщения о начале и завершении работы парсера с указанием даты и времени. После завершения работы печатается пустой XML файл, в котором могли бы быть факты, если бы грамматика их выделяла (подробно о фактах написано в последующих разделах учебника).

```
[15:10:12 18:20:01] - Start.  (Processing files.)
[15:10:12 18:20:01] - End.  (Processing files.)
<?xml version='1.0' encoding='utf-8'?><fdo_objects></fdo_objects>
```

После этого можно открыть файл `PrettyOutput.html` и посмотреть, какие цепочки выделила наша грамматика. В нашем примере `PrettyOutput.html` будет выглядеть так:



Труд облагораживает человека . **EOS**

Text	Type
труд	TAuxDicArticle [наша_первая_грамматика]
человек	TAuxDicArticle [наша_первая_грамматика]

Чтобы извлекать из текста не только отдельные существительные, но и, например, именные группы, применяется операция конкатенации. В качестве оператора конкатенации выступает обычный пробел:

```
S -> Adj Noun;
```

(2) Тяжелый труд облагораживает человека.

(3) *Тяжелый труд облагораживает хорошего человека.*

Теперь мы можем извлечь из текста **(2)** цепочку *тяжелый труд*. Заметим, что человека мы в данном случае проигнорировали: наличие прилагательного перед существительным является обязательным условием выделения цепочки. Исправим ситуацию, немного подкорректировав исходное предложение **(3)**. Теперь на выходе у нас две цепочки: *тяжелый труд* и *хороший человек*.

Тяжелый труд облагораживает хорошего человека . **EOS**

Text	Type
тяжелый труд	TAuxDicArticle [наша_первая_грамматика]
хороший человека	TAuxDicArticle [наша_первая_грамматика]

А если нас не интересует, какой бывает труд, а интересует только, какие бывают люди, можно написать такую грамматику:

```
S -> Adj 'человек';
```

Исходные файлы проекта tutorial1

tutorial1/config.proto	Конфигурационный файл парсера
tutorial1/mydic.gzt	Корневой словарь
tutorial1/first.cxx	Грамматика
tutorial1/test.txt	Текст

Ограничения-пометы

У всех людей есть имена. Поэтому, чтобы расширить свои знания о том, какие бывают люди, нам необходимо извлекать не только прилагательные, которые идут перед словом «человек», но и прилагательные, определяющие имена собственные.

Отличительной особенностью имен собственных является то, что все они пишутся с заглавной буквы. Чтобы сообщить эту информацию Томите, надо наложить соответствующее ограничение в правиле на терминал или нетерминал, обозначающий человека.

Для наложения ограничений на терминал или нетерминал в Томите используются специальные пометы (см. список помет). В нашем случае нам нужна помета `h-reg1`, указывающая на то, что первая буква слова должна стоять в верхнем регистре. Если первая буква будет в нижнем регистре, то правило с такой пометой не сработает. Тогда мы получим такое правило:

```
S -> Adj Word<h-reg1>;
```

Кроме пометы `h-reg1` есть еще помета `h-reg2`, обозначающая, что все буквы слова должны быть в верхнем регистре (как, например, в слове `МОСКВА`), и помета `h-reg3`, обозначающая так называемый верблюжий регистр (или CamelCase), в котором первая и, как минимум, еще одна буква в слове стоят в верхнем регистре (как, например, в слове `СитиИнвест`)

Обработав следующий текст при помощи правила `S -> Adj Word<h-reg1>` мы узнаем, с какими именно эпитетами упоминаются в нем литературные персонажи.

(4) В красном фраке и белоснежных панталонах ездит этот изнеженный Чичиков по российским просторам и встречает не людей… а кукол фарфоровых: то парочку Маниловых (щечки розовые, оборочки розовые), то лохматого Собакевича с супругой, то оглушительного Ноздрева в венгерке на голый торс. (Гоголь Н.В., «Мертвые души»)

Text	Type
изнеженный Чичиков	TAuxDicArticle [наша_первая_грамматика]
лохматый Собакевича	TAuxDicArticle [наша_первая_грамматика]
оглушительный Ноздрева	TAuxDicArticle [наша_первая_грамматика]

В этом примере правила сработали на нужных словах, но результат нормализации нас вряд ли устроит, т.к. к словарной форме приведены только прилагательные (Чичикову повезло только потому, что его фамилия была употреблена в нужной форме). Поскольку мы ничего не сказали о связи между `Adj`

и `Word<h-reg1>` кроме того, что они стоят рядом и именно в этом порядке, то парсер не знает о том, что прилагательное изменяется вместе с существительным (`лохматый Собакевич`, `лохматого Собакевича`, `лохматому Собакевичу`, ...). По умолчанию к словарной форме приводится только главное слово в цепочке. Если не указано иного, то главным словом цепочки является первое. О том, как сообщить парсеру о связях между словами, и как указать главное слово в цепочке, читайте в следующих разделах этого учебника.

Операторы

Возьмем другое предложение **(5)** и применим к нему правило `S -> Adj Word<h-reg1>`, обсуждавшееся в предыдущем разделе.

(5) Многоуважаемый Константин Иванович Деревяшкин сначала категорически воспретил ругаться в рупор и даже топнул ногой, но потом, после некоторого колебания, увлеченный этой идеей, велел позвать из соседнего дома бывшего черноморца — отчаянного ругателя и буяна.

Text	Type
многоуважаемый Константин	TAuxDicArticle [наша_первая_грамматика]

Наше правило не выделяет фамилию и отчество Константина. В качестве следующего шага в грамматике необходимо учесть, что после прилагательного может стоять одно, два или сразу три имени (т.е. слова с заглавной буквы). Это можно записать в виде трех отдельных правил.

```
S -> Adj Word<h-reg1>;                                  // "изнеженный Чичиков"
S -> Adj Word<h-reg1> Word<h-reg1>;                     // "новоявленный Аль Капоне"
S -> Adj Word<h-reg1> Word<h-reg1> Word<h-reg1>;   // "многоуважаемый Константин
Иванович Деревяшкин"
```

К счастью, эту грамматику можно сократить до одного правила при помощи оператора “+”, обозначающего, что терминал или нетерминал может встретиться в цепочке один или более раз. В результате грамматика будет выглядеть так:

```
S -> Adj Word<h-reg1>+;
```

Text	Type
многоуважаемый Константин Иванович Деревяшкин	TAuxDicArticle [наша_первая_грамматика]

Знак «+» и другие операторы описаны в Руководстве пользователя в разделе Операторы.

Далее стоит учесть, что прилагательных может быть больше одного, и они могут разделяться запятой или сочинительным союзом или не разделяться ничем:

(6) В центре большого персидского ковра поставили объемистую вазу с крюшоном, а вокруг нее радиусами разлеглась вся компания, не исключая и Яблоньки, которой пылкий влюбленный Новакович смастерил царственное ложе: шкура белого медведя, на шкуре плюшевый плед, а на пледе — Яблонька, положившая круглый алебастровый подбородок на огромную пушистую голову страшного зверя.

(7) Но прокурор, по-прежнему минуя оружейный магазин, катил каждое утро к зданию судебных установлений, с грустью поглядывая на фигуру Фемиды, державшей весы, в одной чашке которых он явственно видел себя санкт-петербургским прокурором, а в другой — розового и наглого Воробьянинова.

(8) В спальную вошла толстая, краснощекая Розалия Карловна и остановилась в ожидательной позе.

Для разбора примера под номером 6 нам необходимо добавить к терминалу Adj уже известный нам оператор “+”. Чтобы справиться с примерами 7 и 8 потребуется дописать несколько дополнительных правил:

```
AdjCoord -> Adj;                                          // вырожденный случай, когда
цепочка пригалательных - однородных членов
                                                                  // состоит из одного
прилагательного
AdjCoord -> AdjCoord<gnc-agr[1]> ',' Adj<gnc-agr[1]>; // пригалательные - одно-
родные члены могут быть записаны через запятую
AdjCoord -> AdjCoord<gnc-agr[1]> 'и' Adj<gnc-agr[1]>; // или через сочинительный
союз 'и'
```

Помета `gnc-agr` означает, что отмеченные ею слова должны быть согласованы по роду, числу и падежу. Подробнее об этом будет написано чуть позже в этом руководстве и в справочнике.

Теперь вспомним о том, что человек может обозначаться не только именем собственным, но и непосредственно словом «человек» и добавим его к терминалу, обозначающему имя собственное, с помощью оператора дизъюнкции: “|”.

Наконец, между именем собственным и определяющим его прилагательным нередко появляется форма обращения – товарищ, господин, мистер и т.д. Чтобы указать, что элемент цепочки является факультативным (может быть, а может и нет), его ставят в круглые скобки.

В итоге мы получаем такую грамматику:

```
#encoding "utf-8"
#GRAMMAR_ROOT S
ProperName ->  Word<h-reg1>+;                              // задание имени собствен-
ного
Person -> ProperName | 'человек';                     // человек может обозначаться
именем собственным
                                                             // или словом “человек”
FormOfAddress -> 'товарищ' | 'мистер' | 'господин';    // перечисление возможных
форм обращения
AdjCoord -> Adj;                                          // вырожденный случай, когда
цепочка пригалательных - однородных членов
                                                                  // состоит из одного
прилагательного
AdjCoord -> AdjCoord<gnc-agr[1]> ',' Adj<gnc-agr[1]>; // пригалательные - одно-
родные члены могут быть записаны через запятую
AdjCoord -> AdjCoord<gnc-agr[1]> 'и' Adj<gnc-agr[1]>; // или через сочинительный
союз 'и'
S -> Adj+ (FormOfAddress) Person;                          // для случаев, когда
прилагательные идут друг за другом
S -> AdjCoord (FormOfAddress) Person;                      // для случаев, когда
прилагательные разделены запятой
                                                           // или сочинительным союзом
```

Если применить эту грамматику к примерам 6-8, Томита-парсер выделит 3 цепочки:

Text	Type
которая пылкий влюбленный Новакович	TAuxDicArticle [наша_первая_грамматика]
розовый и наглого Воробьянинова	TAuxDicArticle [наша_первая_грамматика]
толстая , краснощекая Розалия Карловна	TAuxDicArticle [наша_первая_грамматика]

Мы видим, что в первую цепочку попало лишнее слово – которой. Это произошло потому, что оно является местоимением-прилагательным, а значит, удовлетворяет правилу `S -> Adj+ (FormOfAddress) Person;`

Вторая цепочка выделилась правильно, но неправильно нормализовалась. Это объясняется тем, что нормализации подвергается только главное слово в цепочке (по умолчанию – первое) и связанные с ним слова, а в наших правилах явным образом не указана связь слов друг с другом.

Обе эти проблемы можно решить, указав в правилах, что существительное и определяющее его прилагательное должны быть согласованы по роду, числу и падежу.

В синтаксисе Томита-парсера это записывается следующим образом:

```
S -> Adj<gnc-agr[1]>+ (FormOfAddress) Person<gnc-agr[1]>;
```

gnc-agr расшифровывается как GenderNumberCase-Agreement (т.е. согласование по роду, числу и падежу), а в квадратных скобках стоит идентификатор согласования, который указывает, какие элементы правой части правила согласуются между собой.

Обратите внимание, что в одном правиле можно использовать несколько разных согласований с разными идентификаторами. Например, в правиле `S -> Participle<gnc-agr[2]> Adj<gnc-agr[1]> Noun<gnc-agr[1], gram='тв'> Noun<gnc-agr[2], gram='им', rt>;` требуется согласование между первым и четвертым терминалами и между вторым и третьим терминалами. Такое правило будет корректно обрабатывать цепочку «обожаемый местным населением напиток». Используемые в этом примере пометы `gram='тв'` и `gram='им'` означают, что слова должны стоять в указанных падежах (творительном и именительном). Подробно об этой помете написано в следующих разделах и в справочнике.

Добавим согласование в нашу грамматику:

```
#encoding "utf-8"
ProperName ->  Word<h-reg1>+;
Person -> ProperName | 'человек';
FormOfAddress -> 'товарищ' | 'мистер' | 'господин';
AdjCoord -> Adj;
AdjCoord -> AdjCoord<gnc-agr[1]> ',' Adj<gnc-agr[1]>;
AdjCoord -> AdjCoord<gnc-agr[1]> 'и' Adj<gnc-agr[1]>;
S -> Adj<gnc-agr[1]>+ (FormOfAddress) Person<gnc-agr[1]>;
S -> AdjCoord<gnc-agr[1]> (FormOfAddress) Person<gnc-agr[1]>;
```

Text	Type
пылкий влюбленный Новакович	TAuxDicArticle [наша_первая_грамматика]
розовый и наглый Воробьянинов	TAuxDicArticle [наша_первая_грамматика]
толстая , краснощекая Розалия Карловна	TAuxDicArticle [наша_первая_грамматика]

В результате работы этой грамматики, в первом примере извлечется цепочка пылкий влюбленный Новакович без слова которая, т.к. которая не согласуется по роду с Новакович, а во втором нормализуется вся выделенная цепочка, а не только главное слово, т.к. слова явным образом связаны друг с другом через согласование.

Главное слово

В каждой цепочке, собираемой правилом, есть главное слово. Его грамматические признаки наследуются всей цепочкой как многословной единицей. По умолчанию главным словом назначается первое слово цепочки. В приведенной в предыдущем примере грамматике главными словами цепочек, собираемых правилами `AdjCoord` и `S` будут прилагательные, соответственно собранные цепочки тоже будут считаться прилагательными. В случае с `AdjCoord` это правильно, а в случае с `S` — нет, т.к. если мы, в дальнейшем, захотим включить эти цепочки в другие правила, то нужно будет, чтобы они функционировали как существительные и наследовали граммемы нетерминала `Person`. Для того, чтобы указать парсеру, какое слово нужно считать главным, используется помета `rt`.

Помета `rt` также оказывает влияние на нормализацию слов, включенных в согласование. При нормализации граммемы задаются от главного слова к зависимому, а не наоборот. Это становится очевидным в случае, когда в согласовании участвуют прилагательное и существительное среднего рода: «фразцузское посольство», «вкусное безе» . Нормализованной формой прилагательного является именительный падеж мужского рода единственного числа. Соответственно при нормализации этих цепочек, собранных правилом `S -> Adj<gnc-agr[1]> Noun<gnc-agr[1]>;` получим «французский посольство» и «вкусный безе.» Чтобы при нормализации прилагательные приняли форму среднего рода, необходимо указать, что главным словом является существительное, и именно оно указывает, в какую форму надо поставить согласованное с ним прилагательное: `S -> Adj<gnc-agr[1]> Noun<gnc-agr[1], rt>;`.

Возвращаясь к грамматике из предыдущего примера, окончательный ее вариант должен быть таким:

```
#encoding "utf-8"
ProperName ->  Word<h-reg1>+;
Person -> ProperName | 'человек';
FormOfAddress -> 'товарищ' | 'мистер' | 'господин';
AdjCoord -> Adj;
AdjCoord -> AdjCoord<gnc-agr[1]> ',' Adj<gnc-agr[1]>;
AdjCoord -> AdjCoord<gnc-agr[1]> 'и' Adj<gnc-agr[1]>;
S -> Adj<gnc-agr[1]>+ (FormOfAddress) Person<gnc-agr[1], rt>;
S -> AdjCoord<gnc-agr[1]> (FormOfAddress) Person<gnc-agr[1], rt>;
```

Text	Type
пылкий влюбленный Новакович	TAuxDicArticle [наша_первая_грамматика]
розовый и наглый Воробьянинов	TAuxDicArticle [наша_первая_грамматика]
толстая , краснощекая Розалия Карловна	TAuxDicArticle [наша_первая_грамматика]

Внимание!
В конфигурационном файле указано, что анализируемый текст читается не из файла, а из стандартного ввода. Поэтому при запуске нужно передать содержимое тестовых файлов в поток стандартного ввода.

Таким образом, не меняя файла `config.proto` можно запустить проект как на каком-то одном файле, так и на всех трех сразу:

```
# В Linux, FreeBSD и прочих *nix системах:
# на всех файлах
cat test?.txt | ./tomitaparser config.proto
# на одном файле
cat test1.txt | ./tomitaparser config.proto
rem В Windows:
rem на всех файлах
type test?.txt | tomitaparser.exe config.proto
rem на одном файле
tomitaparser.exe config.proto < test1.txt
```

Для того, чтобы парсер читал текст из файла, нужно добавить в конфигурационный файл следующий текст перед строкой «`Articles = [`» :

```
Input = {
    File = "test3.txt"; // путь к входному файлу
  }
```

Ограничения-поля

Какие бывают женщины? Чтобы ответить на этот непростой вопрос, нам необходимо добавить к нашим правилам немного морфологической информации, а именно, указать, что нас интересуют не все имена собственные, а только женские. В Томите для этого используются ограничения-поля. В отличие от ограничений-помет (`h-reg1`, `fw`, `rt` и т.д.), поля могут иметь разные значения, которые задаются как `имя_поля = 'значение'`. В нашем случае нам нужно поле `gram`, которому приписано значение «женский род».

Немного измененная грамматика для извлечения информации о женщинах будет выглядеть так:

```
#encoding "utf-8"
ProperName ->  Word<h-reg1, gram='жен'>+;
Person -> ProperName | 'женщина';
FormOfAddress -> 'товарищ' | 'мисс' | 'миссис' | 'госпожа';
AdjCoord -> Adj;
AdjCoord -> AdjCoord<gnc-agr[1]> ',' Adj<gnc-agr[1]>;
AdjCoord -> AdjCoord<gnc-agr[1]> 'и' Adj<gnc-agr[1]>;
S -> Adj<gnc-agr[1]>+ (FormOfAddress) Person<gnc-agr[1], rt>;
S -> AdjCoord<gnc-agr[1]> (FormOfAddress) Person<gnc-agr[1], rt>;
```

Такая грамматика сработает только на последнем из примеров **(6)-(8)**.

Text	Type
толстая , краснощекая Розалия Карловна	TAuxDicArticle [наша_первая_грамматика]

В качестве значения поля `gram` может выступать любая граммема, приписанная данному слову в морфологическом словаре. Например, чтобы увеличить точность выделения имен собственных, можно указать, что они могут состоять из фамилии, имени и отчества, а не просто из слов, написанных подряд с большой буквы:

```
ProperName -> (Word<h-reg1, gram="фам, жен">)
              Word<h-reg1, gram="имя, жен">
              (Word<h-reg1, gram="отч, жен">);
```

Регулярные выражения

Иногда для работы с текстом удобно использовать шаблоны, или регулярные выражения. С помощью шаблонов можно, например, научиться извлекать из текста год рождения персонажа. Для передачи этой информации в тексте обычно используется шаблон «X родился в YEAR», где X – имя персонажа биографии. Мы уже умеем извлекать имена собственные, слово «родился» можно задать леммой с соответствующей грамматической информацией, а для задания даты нам потребуется написать регулярное выражение:

```
#encoding "utf-8"
ProperName -> Word<h-reg1, gram='имя'>
              Word<h-reg1, gram='отч'>
              Word<h-reg1, gram='фам'>;
S -> ProperName 'родиться'<gram='прош'> 'в'
     AnyWord<wff=/[1-2]?[0-9]{1,3}г?\.?/> ('год' <gram='ед, дат'>);
```

Регулярное выражение, как правило, пишется в поле wff (подробно см. список помет) и его синтаксис почти полностью совпадает с синтаксисом регулярных выражений в Perl-е .

С помощью написанного правила из текста

(10) Иван Иванович Иванов родился в 1875 году в небольшом городке в поместье своего отца.

мы без труда узнаем подробности биографии Ивана Ивановича.

Исходные файлы проекта tutorial3

`tutorial3/config.proto`	Конфигурационный файл парсера
`tutorial3/mydic.gzt`	Корневой словарь
`tutorial3/fioborn.cxx`	Грамматика
`tutorial3/test.txt`	Текст

В конфигурационном файле указано, что анализируемый текст нужно читать из файла test.txt, поэтому проект запускается обычным способом.

Типы ключевых слов (kwtype)

Все мы знаем детский стишок Маршака про воробья:

(11) — Где обедал, воробей?

— В зоопарке у зверей.

Пообедал я сперва

За решеткою у льва.

Подкрепился у лисицы.

У моржа попил водицы.

Ел морковку у слона.

С журавлем поел пшена.

Погостил у носорога,

Отрубей поел немного.

Побывал я на пиру

У хвостатых кенгуру.

Был на праздничном обеде

У мохнатого медведя.

А зубастый крокодил

Чуть меня не проглотил.

Из первых двух строк следует, что воробей обедал у каких-то зверей, которые живут в зоопарке. Попробуем написать несложную систему правил, с помощью которых мы сможем узнать, у каких же именно зверей обедал воробей.

Для такой грамматики нам потребуется не только грамматическая, но и семантическая информация о словах из текста. Для этого нам надо построить мини-словарь.

Мы сталкивались со словарем в самом начале, когда создавали корневой файл `mydic.gzt`. Теперь рассмотрим этот формат подробнее.

Словарь в Томите называется газеттиром и пишется в отдельном файле с расширением `gzt`. Как и любой словарь, газеттир состоит из статей. Создадим файл `animals_dict.gzt` и напишем в нем простейшую статью, в которой перечислим некоторых животных:

```
encoding "utf8";
TAuxDicArticle "животные"
{
    key = "собака" | "кот" | "лошадь" | "корова" | "лев" | "слон" | "волк" |
          "кенгуру" | "крокодил"
}
```

В этом примере `TAuxDicArticle` – тип статьи, «животные» — ее название, а в поле `key` мы перечисляем все слова, входящие в эту статью.

Созданный словарь обязательно надо импортировать в корневой словарь. Для этого в файле `mydic.gzt` надо добавить следующую строчку:

```
import "animals_dict.gzt";
```

Теперь на созданную статью можно ссылаться из грамматики. Для этого используются пометы `kwtype` и `kwset`, в качестве значения которых выступает тип или имя статьи. Тип `TAuxDicArticle` является типом по умолчанию и используется во многих статьях, поэтому мы ссылаемся на название, которое уникально:

```
#encoding "utf-8"
S -> 'у' (Adj<gnc-agr[1]>) Noun<kwtype='животные', gram='род', rt, gnc-agr[1]>;
S -> 'с' (Adj<gnc-agr[1]>) Noun<kwtype='животные', gram='твор', rt, gnc-agr[1]>;
```

Если мы запустим эту грамматику на стишке про воробья, мы получим следующие подцепочки:

```
у лев
у слон
у хвостатый кенгуру
```

Естественно, полнота результата напрямую зависит от полноты словаря: в нашем примере у нас выделились только те животные, которых мы включили в словарь.

Названий животных очень много, и перечислять их всех внутри словаря не слишком удобно. Поэтому в газеттире есть возможность из статьи сослаться на текстовый файл, в котором перечислены все слова, входящие в данную статью. Перепишем наш словарь с учетом этой возможности:

```
TAuxDicArticle “животные”
{
    key = { "animals.txt" type=FILE }
}
```

В поле key указывается название текстового файла и тип ключа — FILE. Соответственно, в файле animals.txt перечислены названия животных:

```
собака
кот
лошадь
корова
лев
слон
волк
кенгуру
крокодил
```

С помощью газеттира можно решать более сложные задачи, например, мы можем узнать, с какими птицами обедают воробьи. Для этого нам нужно разделить животных по разным статьям в зависимости от их класса:

```
TAuxDicArticle "млекопитающие"
{
    key = "собака" | "кот" | "лошадь" | "корова" | "лев" | "слон" | "волк" |
          "кенгуру" | "морж" | "лисица" | "носорог" | "медведь"
}
TAuxDicArticle "птицы"
{
    key = "воробей" | "журавль" | "павлин"
}
TAuxDicArticle "рептилии"
{
    key = "черепаха" | "крокодил"
}
```

Чтобы сохранить информацию о том, что все млекопитающие, птицы и рептилии являются животными, целесообразно вместо стандартного типа статьи TAuxDicArticle использовать свой — например, animal. Типы статей описываются в специальном формате в отдельном файле. Создадим файл kwtypes.proto и запишем туда следующее:

```
import "base.proto";
import "articles_base.proto";
message animal : TAuxDicArticle {}
```

Первые две строчки импортируют встроенные в парсер файлы `base.proto` и `article_base.proto` аналогично тому, как это делается в корневом словаре, а в последней описывается тип статьи `animal`, производный от базового типа TAuxDicArticle. Все типы статей должны быть производными от TAuxDicArticle. Создание пользовательских типов статей позволяет объединить несколько статей в группу, чтобы в дальнейшем можно было бы использовать их все, указав только

их тип при помощи пометы kwtype (например, kwtype=animal), но не перечисляя каждую в отдельности.

Теперь мы можем переписать газеттир с использованием нового класса:

```
encoding "utf-8";
import "kwtypes.proto"; //импортируем файл, в котором описаны используемые в словаре типы статей
animal "млекопитающие"
{
    key = "собака" | "кот" | "лошадь" | "корова" | "лев" | "слон" | "волк" |
          "кенгуру" | "морж" | "лисица" | "носорог" | "медведь"
}
animal "птицы"
{
    key = "воробей" | "журавль" | "павлин"
}
animal "рептилии"
{
    key = "черепаха" | "крокодил"
}
```

Интерпретация

В начале мы сказали, что Томита выделяет подцепочки и интерпретирует их в разбитые по полям факты. До этого мы учились только выделять подцепочки, а теперь попробуем преобразовывать их факты.

Пусть у нас есть биография, и мы хотим извлечь из нее все даты. Для дальнейшей обработки даты удобнее извлекать не единой цепочкой, а сразу разбивать по полям: день недели, день, месяц, год. Для этого нам нужно создать факт «Дата» с перечисленными полями. Для описания фактов создаем отдельный файл — facttypes.proto. Его надо импортировать в корневой словарь — добавляем в файл mydic.gzt такую строчку:

```
import "facttypes.proto";
```

В сам файл facttypes.proto запишем следующее:

```
import "base.proto";           // описание protobuf-типов
import "facttypes_base.proto"; // описание protobuf-типа NFactType.TFact
message Date: NFactType.TFact
{
    optional string DayOfWeek = 1;
    optional string Day = 2;
    optional string Month = 3;
    optional string Year = 4;
}
```

В строке message Date: NFactType.TFact Date является названием типа факта, который мы описываем. Тип факта Date наследуется от базового типа NFactType.TFact, от которого нужно наследовать все типы фактов. Далее в фигурных скобках описываются поля факта Date. Вначале указывается, обязательно ли поле должно быть заполнено (optional — может быть пустым, required — обязательно должно содержать значение, иначе факт не будет сформирован). Далее указан тип поля string — это самый общий тип. В поля типа string можно положить любые строки текста.

DayOfWeek — название поля. Все поля нумеруются по порядку, начиная с единицы.

Также нам понадобится список дней недели и месяцев — для этого нужно создать соответствующие статьи в газеттире. Чтобы не создавать отдельный файл, их можно добавить прямо в корневой словарь:

```
TAuxDicArticle "месяц"
{
    key = "январь" | "февраль" | "март" | "апрель" | "май" | "июнь" |
          "июль" | "август" |   "сентябрь" | "октябрь" | "ноябрь" | "декабрь"
}
TAuxDicArticle "день_недели"
{
    key = "понедельник" | "вторник" | "среда" | "четверг" | "пятница" | "суббота"
| "воскресенье"
}
```

Теперь напишем саму грамматику, выделяющую в тексте даты и интерпретирующую их в факты. Назовем ее date.cxx.

```
#encoding "utf-8"
DayOfWeek -> Noun<kwtype="день_недели">;        // используем слова из статьи
"день_недели"
Day -> AnyWord<wff=/([1-2]?[0-9])|(3[0-1])/>; // число от 1 до 31
Month -> Noun<kwtype="месяц">;                    // используем слова из статьи
"месяц"
YearDescr -> "год" | "г. ";
Year -> AnyWord<wff=/[1-2]?[0-9]{1,3}г?\.?/>; // число от 0 до 2999 с возможным
"г" или "г." в конце
Year -> Year YearDescr;
        // день недели, запятая, число, месяц и год:
        // "понедельник, 3 сентября 2012г."
Date -> DayOfWeek interp (Date.DayOfWeek) (Comma)
        Day interp (Date.Day)
        Month interp (Date.Month)
        (Year interp (Date.Year));
        // число, месяц и год: "10 января 2011"
Date -> Day interp (Date.Day)
        Month interp (Date.Month)
        (Year interp (Date.Year));
        // месяц и год: "июнь 2009"
Date -> Month interp (Date.Month)
        Year interp (Date.Year);
```

Чтобы интерпретировать желаемую подцепочку в факт, надо написать слово interp и после него в скобках указать имя факта и имя поля внутри этого факта, в которое должна попасть подцепочка.

Чтобы запустить эту грамматику, нам надо во-первых, добавить в корневой словарь новую статью:

```
TAuxDicArticle "дата"
{
    key = { "tomita:date.cxx" type=CUSTOM }
}
```

Во-вторых, в файл config.proto, содержащий параметры запуска, надо добавить информацию о том, какие факты мы будем использовать в запускаемых грамматиках:

```
Facts = [
    { Name = "Date" }
]
```

Также можно добавить выходной файл, куда будут записываться факты (если этого не сделать, то они будут печататься прямо на экран):

```
Output = {
    File = "facts.txt";
    Format = text;        // можно использовать следующие форматы:
                          // proto (Google Protobuf), xml, text
}
```

Факты будут сохранены в файл facts.txt (для последующей обработки программами), а также появятся в PrettyOutput.html в удобном для просмотра виде. Например, если мы дадим на вход грамматике отрывок биографии Гагарина:

(12) Юрий Алексеевич Гагарин родился в пятницу, 9 марта 1934 года. Согласно документам, это произошло в деревне Клушино Гжатского района Западной области РСФСР (ныне Гагаринский район Смоленской области), то есть по месту жительства (прописки) родителей. По происхождению является выходцем из крестьян: его отец, Алексей Иванович Гагарин (1902—1973), — плотник, мать, Анна Тимофеевна Матвеева (1903—1984), — работала на молочнотоварной ферме. Его дедушка, рабочий Путиловского завода Тимофей Матвеевич Матвеев, жил в Санкт-Петербурге, в Автове, на Богомоловской (ныне Возрождения) улице в конце XIX века. 1 сентября 1941 года Юрий пошел в школу, но 12 октября деревню заняли немцы, и его учеба прервалась. Почти полтора года деревня Клушино была оккупирована немецкими войсками. 9 апреля 1943 года деревню освободила Красная армия, и учеба в школе возобновилась.

В PrettyOutput.html мы увидим следующее:

Юрий Алексеевич Гагарин родился в пятницу , 9 марта 1934 года . **EOS**

Согласно документам , это произошло в деревне Клушино Гжатского района Западной области РСФСР (ныне Гагаринский район Смоленской области) , то есть по месту жительства (прописки) родителей . **EOS**

По происхождению является выходцем из крестьян : его отец , Алексей Иванович Гагарин (1902 — 1973) , — плотник , мать , Анна Тимофеевна Матвеева (1903 — 1984) , — работала на молочнотоварной ферме . **EOS**

Его дедушка , рабочий Путиловского завода Тимофей Матвеевич Матвеев , жил в Санкт-Петербурге , в Автове , на Богомоловской (ныне Возрождения) улице в конце XIX века . **EOS**

1 сентября 1941 года Юрий пошел в школу , но 12 октября деревню заняли немцы , и его учеба прервалась . **EOS**

Почти полтора года деревня Клушино была оккупирована немецкими войсками . **EOS**

9 апреля 1943 года деревню освободила Красная армия , и учеба в школе возобновилась . **EOS**

Date			
DayOfWeek	**Day**	**Month**	**Year**
пятница	9	март	1934 года
	1	сентябрь	1941 года
	12	октябрь	
	9	апрель	1943 года

Text	Type
пятница , 9 марта 1934 года	TAuxDicArticle [дата]
1 сентября 1941 года	TAuxDicArticle [дата]
12 октября	TAuxDicArticle [дата]
9 апреля 1943 года	TAuxDicArticle [дата]

Исходные файлы проекта tutorial4

`tutorial4/config.proto`	Конфигурационный файл парсера
`tutorial4/facttypes.proto`	Описание типов фактов
`tutorial4/mydic.gzt`	Корневой словарь
`tutorial4/date.cxx`	Грамматика для дат
`tutorial4/test.txt`	Текст «Юрий Алексеевич Гагарин родился в пятницу ...»

Включение грамматик (kwtypes)

Задача выделения дат является очень распространенной при извлечении фактов из текста. Чтобы не писать одни и те же правила каждый раз, мы можем включать уже написанную грамматику в другие грамматики.

Допустим, в стишке про воробья у нас не только сказано, у кого он обедал, но и уточняется, когда:

(13) — В зоопарке у зверей.

— Где обедал, воробей?

Пообедал я сперва

За решеткою у льва.

Подкрепился у лисицы 14 августа.

1 сентября у моржа попил водицы.

Ел морковку у слона 29 декабря 2011 года.

С журавлем поел пшена.

и т.д.

Мы помним, что у нас уже есть грамматика, извлекающая точные даты, и грамматика, извлекающая информацию о том, у кого обедал воробей. Чтобы выяснить, у кого и когда обедал воробей, достаточно включить одну грамматику в другую. Это можно делать двумя способами:

1. С помощью директивы include.

```
#encoding "utf-8"
#include <date.cxx>
Animal -> Noun<kwtype=animal>;
WithWho -> 'y' (Adj<gnc-agr[1]>) Animal<gram='род', rt, gnc-agr[1]> interp
(Sparrow.Who);
WithWho -> 'c' (Adj<gnc-agr[1]>) Animal<gram='твор', rt, gnc-agr[1]> interp
(Sparrow.Who);
S -> Date interp (Sparrow.When) WithWho;
S -> WithWho Date interp (Sparrow.When);
```

   Директива include включает текст из указанного файла вместо самой себя. Т.е. парсер «увидит» вот такое текст:

```
#encoding "utf-8"
// грамматика извлечения дат из date.cxx
DayOfWeek -> Noun<kwtype="день_недели">;
Day -> AnyWord<wff=/([1-2]?[0-9])|(3[0-1])/>;
Month -> Noun<kwtype="месяц">;
YearDescr -> "год" | "г. ";
Year -> AnyWord<wff=/[1-2]?[0-9]{1,3}г?\.?/>;
Year -> Year YearDescr;
Date -> DayOfWeek interp (Date.DayOfWeek) (Comma) Day interp (Date.Day) Month
interp (Date.Month) (Year interp (Date.Year));
Date -> Day interp (Date.Day) Month interp (Date.Month) (Year interp
(Date.Year));
Date -> Month interp (Date.Month) Year interp (Date.Year);
//основная грамматика
Animal -> Noun<kwtype=animal>;
WithWho -> 'y' (Adj<gnc-agr[1]>) Animal<gram='род', rt, gnc-agr[1]> interp
(Sparrow.Who);
WithWho -> 'c' (Adj<gnc-agr[1]>) Animal<gram='твор', rt, gnc-agr[1]> interp
(Sparrow.Who);
S -> Date interp (Sparrow.When) WithWho;
S -> WithWho Date interp (Sparrow.When);
```

   Таким образом, директива include равносильна тому, что мы скопировали текст одной грамматики и вставили его в другую.

2. С помощью kwtype'ов газеттира.

   Мы помним, что каждой грамматике соответствует статья в коревом словаре. По сути, такая статья содержит в себе все цепочки, которые собирает данная грамматика, а значит, использовать одну грамматику в другой можно с помощью уже известной нам пометы kwtype. Выглядеть это будет так:

```
#encoding "utf-8"
Date -> AnyWord<kwtype='**дата**'>;  // используем статью "дата" из словаря
Animal -> Noun<kwtype=animal>;
WithWho -> 'y' (Adj<gnc-agr[1]>)
           Animal<gram='род', rt, gnc-agr[1]> interp (Sparrow.Who);
WithWho -> 'c' (Adj<gnc-agr[1]>)
           Animal<gram='твор', rt, gnc-agr[1]> interp (Sparrow.Who);
S -> Date interp (Sparrow.When) WithWho;
S -> WithWho Date interp (Sparrow.When);
```

В этом примере нетерминал Date соответствует любой цепочке, собираемой грамматикой date.cxx (которая в корневом словаре лежит в статье “дата”), например, 20 августа 2012 года.

Результат в первом и втором случае будет одинаковым.

Разница между первым и вторым способом включения грамматик заключается в том, что во втором случае сначала собираются цепочки, обозначенные kwtype’ами, а уже потом — цепочки, описываемые в правилах самой грамматики. Это означает, что во втором случае грамматика получает на вход следующий текст:

(14) — Где обедал, воробей?

— В зоопарке у зверей.

Пообедал я сперва

За решеткою у льва.

Подкрепился у лисицы 14_августа.

1_сентября у моржа попил водицы.

Ел морковку у слона 29_декабря_2011_года.

С журавлем поел пшена.

14_августа, 1_сентября и 29_декабря_2011_года – это теперь неделимые сущности, которые функционируют в тексте как одно слово. Грамматика уже не различает отдельные слова в их составе. Такие единицы наследует морфологические характеристики главного слова цепочки, из которой они были собраны.

Такое положение дел верно не только для kwtype’ов, за которыми стоят грамматики, но и для всех прочих. Например, если kwtype ссылается на статью с многословными ключами, то сначала эти многословные ключи объединятся в неделимое целое, а потом начнут работать правила грамматики. Так, если у нас есть такая статья в словаре:

```
animal "слон"
{
    key = "африканский слон"
}
```

А на вход дана следующая строчка:

(15) Ел морковку у африканского слона 29 декабря 2011 года.

То после применения kwtype’ов наша грамматика получит следующий текст:

(16) Ел морковку у африканского_слона 29_декабря_2011_года.

И именно его парсер будет обрабатывать правилами.

Исходные файлы проекта tutorial5

`tutorial5/config.proto`	Конфигурационный файл парсера
`tutorial5/kwtypes.proto`	Объявление kw-типа `animals`
`tutorial5/facttypes.proto`	Описание типов фактов
`tutorial5/mydic.gzt`	Корневой словарь
`tutorial5/animals_dict.gzt`	Словарь с названиями животных
`tutorial5/mammals.txt`	Список млекопитающих
`tutorial5/main.cxx`	Основная грамматика
`tutorial5/date.cxx`	Грамматика для дат
`tutorial5/test.txt`	Текст «Где обедал воробей ...» с датами

Яндекс

Томита-парсер

Быстрый старт

29.12.2016