

Томи́та-парсер

Томи́та-парсер

28.11.2024

Томита-парсер. Версия 1

Дата подготовки документа: 28.11.2024

Этот документ является составной частью технической документации Яндекса.

© 2008—2024 ООО «ЯНДЕКС». Все права защищены.

Предупреждение об исключительных правах и конфиденциальной информации

Исключительные права на все результаты интеллектуальной деятельности и приравненные к ним средства индивидуализации юридических лиц, товаров, работ, услуг и предприятий, которым предоставляется правовая охрана (интеллектуальную собственность), используемые при разработке, поддержке и эксплуатации службы, включая, но не ограничиваясь, программы для ЭВМ, базы данных, изображения, тексты, другие произведения, а также изобретения, полезные модели, товарные знаки, знаки обслуживания, коммерческие обозначения и фирменные наименования, принадлежат ООО «ЯНДЕКС» либо его лицензиарам.

Использование результатов интеллектуальной деятельности и приравненных к ним средств индивидуализации в целях, не связанных с разработкой, поддержкой и эксплуатацией службы, не допускается без получения предварительного согласия правообладателя. Настоящий документ содержит конфиденциальную информацию ООО «ЯНДЕКС». Использование конфиденциальной информации в целях, не связанных с разработкой, поддержкой и эксплуатацией службы, а равно как и разглашение таковой, не допускается. При этом под разглашением понимается любое действие или бездействие, в результате которых конфиденциальная информация в любой возможной форме (устной, письменной, иной форме, в том числе с использованием технических средств) становится известной третьим лицам без согласия обладателя такой информации либо вопреки трудовому или гражданско-правовому договору.

Отношения ООО «ЯНДЕКС» с лицами, привлекаемыми для разработки, поддержки и эксплуатации службы, регулируются законодательством Российской Федерации и заключаемыми в соответствии с ним трудовыми и/или гражданско-правовыми договорами (соглашениями). Нарушение требований об охране результатов интеллектуальной деятельности и приравненных к ним средств индивидуализации, а равно как и конфиденциальной информации, влечет за собой дисциплинарную, гражданско-правовую, административную или уголовную ответственность в соответствии с законодательством Российской Федерации.

Контактная информация

ООО «ЯНДЕКС»

<https://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

Быстрый старт.....	5
Исходные файлы простого проекта.....	5
Простейшие правила.....	6
Ограничения-пометы.....	8
Операторы.....	8
Главное слово.....	11
Ограничения-поля.....	12
Регулярные выражения.....	12
Исходные файлы проекта tutorial3.....	13
Типы ключевых слов (kwtype).....	13
Интерпретация.....	15
Исходные файлы проекта tutorial4.....	19
Включение грамматик (kwtypes).....	19
Исходные файлы проекта tutorial5.....	21
Руководство разработчика.....	21
Принцип работы.....	21
Алгоритм работы парсера на одном предложении и одной грамматике.....	21
Исходные файлы простого проекта.....	22
Синтаксис грамматик.....	23
Директивы.....	23
Правила.....	24
Операторы.....	25
Синтаксические пометы.....	27
Помета <code>rt</code>	27
Значение подстановки <code>#{ }</code>	27
Комментарии.....	27
Пометы-ограничения.....	27
Тип ключевого слова (помета <code>kwtype</code>).....	27
Множество типов ключевых слов (помета <code>kwset</code>).....	28
Грамматические характеристики (помета <code>gram</code>).....	28
Объединение грамматических характеристик (помета <code>GU</code>).....	29
Согласование в грамматиках.....	30
Регулярные выражения (пометы <code>wfm</code> , <code>wff</code> , <code>wfl</code>).....	31
Помета <code>no_hom</code>	31
Список всех помет.....	31
Подробно про кавычки.....	36
Список терминалов.....	36
Значения грамем.....	37
Интерпретация и нормализация.....	40
Более формально.....	40
Описание типов фактов.....	40
Поля фактов.....	40
Значения полей по умолчанию.....	41
Капитализация полей фактов.....	41
Сохранение лемм.....	41
Интерпретация фактов в грамматиках.....	42
Получение фактов из внешних грамматик (оператор <code>from</code>).....	42
Интерпретация с явным значением (оператор <code>=</code>).....	42
Конкатенация (оператор <code>+</code>).....	42
Одновременная интерпретация нескольких фактов.....	42

Нормализация.....	43
Пометы интерпретации.....	44
Синтаксис газеттира.....	44
Статья словаря в общем виде.....	44
Тип статьи.....	44
Предопределенные типы статей.....	45
Название статьи.....	45
Комментарии.....	45
Ключ.....	45
Грамматические пометы в описании ключей.....	46
Специальные типы ключей.....	47
Связь между газеттиром и грамматикой.....	47
Ссылка из грамматики в газеттир.....	47
Ссылка из газеттира в грамматику.....	47
Использование газеттира при нормализации.....	47
Грамматические признаки мультиворда.....	48
Неочевидные решения часто возникающих задач.....	48
Исключить неправильные частотные срабатывания.....	48
Описать только изменяемые существительные.....	49
Описать классы слов с регулярным словообразованием.....	49
Отладка грамматики.....	49
Пример работы парсера.....	49
Запуск парсера и файл конфигурации.....	53
Описание формата конфигурационного файла.....	53
Отладочный вывод.....	54
Примеры.....	55

Об учебнике

Томи́та-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 Томи́та-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.
factypes.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	factypes.proto		test.txt
tutorial5	config.proto	mydic.gzt, animals_dict.gzt, mammals.txt	main.cxx, date.cxx	factypes.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 расширением `.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 удобном 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 `tomitparser`, параметром которой будет файл `config.proto`. Итак, в командной строке нам надо написать такую команду:

```
В Linux, FreeBSD и прочих *nix системах:
./tomitparser config.proto
В Windows:
tomitparser.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 . EO	
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.gz	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 работает. 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_грамм
лохматый Собакевича	TAuxDicArticle [наша_первая_грамм
оглушительный Ноздрева	TAuxDicArticle [наша_первая_грамм

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 связи 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
<u>многоуважаемый Константин</u>	TAuxDicArticle [наша_первая_гр

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
<u>многоуважаемый Константин Иванович Деревяшкин</u>	TAuxDicA

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 кривоносом, 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 прилагательных
- однородных членов

AdjCoord -> AdjCoord<gnc-agr[1]> ',' Adj<gnc-agr[1]>; // состоит из одного прилагательного
- однородные 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 собственного
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 прилагательных
- однородных членов
AdjCoord -> AdjCoord<gnc-agr[1]> ' ' Adj<gnc-agr[1]>; // состоит из одного прилагательного
- однородные члены могут быть записаны через запятую // прилагательные
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	T
<u>которая пылкий влюбленный Новакович</u>	TAuxDicArticle [наша]
<u>розовый и наглого Воробьянинова</u>	TAuxDicArticle [наша]
<u>толстая , краснощекая Розалия Карловна</u>	TAuxDicArticle [наша]

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 напитков ». 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	T
<u>пылкий влюбленный Новакович</u>	TAuxDicArticle [наша]
<u>розовый и наглый Воробьянинов</u>	TAuxDicArticle [наша]
<u>толстая , краснощекая Розалия Карловна</u>	TAuxDicArticle [наша]

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: «французское 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	T
<u>пылкий влюбленный Новакович</u>	TAuxDicArticle [наша]
<u>розовый и наглый Воробьянинов</u>	TAuxDicArticle [наша]
<u>толстая , краснощекая Розалия Карловна</u>	TAuxDicArticle [наша]



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	T
толстая, краснощекая Розалия Карловна	TAuxDicArticle [наш

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 морковь 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 -> 'y' (Adj<gnc-agr[1]>) Noun<kwtype='животные', gram='под', rt, gnc-agr[1]>;
S -> 'c' (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 `kwtupe` (например, `kwtupe=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

Согласно документам , это произошло в деревне Клушино
(ныне Гагаринский район Смоленской области) , то
родителей . **EOS**

По происхождению является выходцем из крестьян : е
1973) , — плотник , мать , Анна Тимофеевна Матве
молочнотоварной ферме . **EOS**

Его дедушка , рабочий Путиловского завода Тимофей
в Автове , на Богомоловской (ныне Возрождения) у

1 сентября 1941 года Юрий пошел в школу , но 12
прервалась . **EOS**

Почти полтора года деревня Клушино была оккупирова

9 апреля 1943 года деревню освободила Красная арми

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 стихе 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 корневом 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='под', 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 наследуют 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 строка:

(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

ОСНОВНЫЕ ПОНЯТИЯ

Томи-парсер позволяет по написанным пользователем шаблонам (**КС-грамматикам**) выделять из текста разбитые на поля цепочки слов или факты. Например, можно написать шаблоны для выделения адресов. Здесь фактом является адрес, а его полями — «название города», «название улицы», «номер дома» и т.д.

Парсер включает в себя три стандартных лингвистических процессора: токенизатор (разбиение на слова), сегментатор (разбиение на предложения) и морфологический анализатор (**mystem**).

Основные компоненты парсера: **газеттир**, **набор КС-грамматик** и множество **описаний типов фактов**, которые порождаются этими грамматиками в результате процедуры **интерпретации**.

Газеттир — словарь ключевых слов, которые используются в процессе анализа КС-грамматиками. Каждая статья этого словаря задает множество слов и словосочетаний, объединенных общим свойством. Например, «все города России». Затем в грамматике можно использовать свойство «является городом России». Слова или словосочетания можно задавать явно списком, а можно «функционально», указав грамматику, которая описывает нужные цепочки. Например, цепочка ключевых слов «адрес» описывается соответствующей грамматикой и может быть использована в грамматике для выделения городских происшествий. Подробнее об этом будет в описании механизма каскадов.

Грамматика — множество правил на языке КС-грамматик, описывающих синтаксическую структуру выделяемых цепочек. Грамматический парсер запускается всегда на одном предложении. Перед запуском терминалы грамматики отображаются на слова (или словосочетания, об этом будет сказано ниже) предложения. Одному слову может соответствовать много терминальных символов. Таким образом, парсер получает на вход последовательность множеств терминальных символов. Например, в нашей грамматике есть всего два термина Verb и Noun, а входное предложение «Мама мыла стекло.» . Тогда парсер получит на вход такую последовательность: {Noun}, {Verb, Noun}, {Verb, Noun}. На выходе получаются цепочки слов, распознанные этой грамматикой.

Факты — таблицы с колонками, которые называются полями фактов. Факты заполняются во время анализа парсером предложения. Как и чем заполнять поля фактов указывается в каждой конкретной грамматике. Это называется **интерпретацией**. Типы факты описываются на специальном языке в отдельном файле.

Принцип работы

Алгоритм работы парсера на одном предложении и одной грамматике

1. Ищутся вхождения всех ключей из газеттира. Если ключ состоит из нескольких слов (например, «Нижний Новгород»), то создается новое искусственное слово, которое мы называем «мультиворд».
2. Из всех найденных ключей газеттира отбираются те, которые упоминаются в грамматике (см. **Помета kwtype**).

3. Среди отобранных ключей могут встречаться и мультиторды, пересекающиеся друг с другом или включающие в себя одиночные ключевые слова. Парсер пытается покрыть предложение непересекающимися ключевыми словами так, чтобы как можно большие куски предложения были охвачены ими.
4. Линейная цепочка слов и мультитордов подается на вход GLR-парсеру. Терминалы грамматики отображаются на входные слова и мультиторды.
5. На последовательности множеств терминалов GLR-парсер строит все возможные варианты. Из всех построенных вариантов также отбираются те, которые как можно шире покрывают предложение.
6. Затем парсер запускает процедуру [интерпретации](#) на построенном синтаксическом дереве. Он отбирает специально помеченные подузлы, а слова, которые им соответствуют, записываются в порождаемые грамматикой поля фактов.

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. 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.

Каскады

В качестве ключевых слов могут быть использованы результаты работы других грамматик. Тогда на шаге 2 предыдущего алгоритма парсер смотрит, какие ключевые слова задаются не списком слов, а конкретными грамматиками (см. [специальные типы ключей](#)). Если такие ключи есть, то он рекурсивно запускает тот же алгоритм на всех грамматиках, упомянутых в статьях для этих ключевых слов. Выделенные этими грамматиками цепочки становятся мультитордами и смешиваются с остальными ключевыми словами. Таким образом, можно строить целые деревья парсеров, подавая результаты работы одного в качестве входных символов другого.

Адресация

Главным элементом для парсера является [статья из газеттира](#). Статью можно представлять в виде функции на предложении, результатом работы которой всегда является выделенная подцепочка. [Имя статьи](#) тогда является именем этой функции, а [тип статьи](#) — множеством функций, применяемых в случайном порядке. Внутри этой статьи-функции явно указано, каким способом выделить нужную цепочку: грамматикой или поиском указанных слов и словосочетаний. Вся связь между грамматиками происходит через статьи газеттира, в которых в ключе [вместо слов указана грамматика](#). В [параметрах к парсеру](#) указываются также имена газеттирных статей, т.е. вызываются функции, выделяющие подцепочки.

Вес

Парсер может порождать много вариантов разбора одной и той же цепочки, которые отличаются только деревом разбора. Различные деревья могут порождать разные варианты заполнения полей фактов. Для этой цели правилу можно приписывать вес (см. описание пометы `weight` в разделе [операции над правилами](#)). По умолчанию вес у всех правил равен 1. Но его можно искусственно уменьшить, чтобы парсер выбрал вариант с большим весом, если такой построился.

Синтаксис грамматик

Файл грамматики состоит из двух частей: блока с директивами, влияющими на работу грамматики в целом, и множества правил.

Директивы

Директивы начинаются со знака `#` и заканчиваются переносом строки. Некоторые директивы совпадают по смыслу с аналогичными директивами препроцессора языка C++.

#encoding

Указывает кодировку данного файла с грамматикой. Кодировка по умолчанию: `utf-8`. Кодировка указывается в кавычках.

```
#encoding "windows-1251"
```

#include

Включает в данную грамматику текст другой грамматики. При этом фильтры грамматик суммируются, а корень включаемой грамматики (см. `#grammar_root`) игнорируется. Название включаемой грамматики указывается в кавычках.

```
#include "small_grammar.cxx"
```

#GRAMMAR_ROOT

Директива `#GRAMMAR_ROOT` указывает нетерминал, который является корнем данной грамматики. Корень грамматики можно не задавать явно, если в грамматике есть только один нетерминал, который ни разу не встречается в правой части правил грамматики.

```
#GRAMMAR_ROOT MainRule
```

#GRAMMAR_KWSET

Директива `#GRAMMAR_KWSET` позволяет явно указать имена или типы статей из газеттира, чьи найденные в предложении ключи должны передаваться парсеру в качестве терминалов. С точки зрения текущей грамматики, эти цепочки становятся `multiword`-ами (см. п. 2 [алгоритма](#) работы парсера). В частности директива `#GRAMMAR_KWSET` может использоваться для уточнения значения пометы `kwtype=none`. Пример использования см. [тут](#).

Синтаксис описания статей аналогичен синтаксису пометы `kwset`: перечисления в квадратных скобках `[]` через запятую.

```
#GRAMMAR_KWSET ["котики", cat_types];
```

#filter

Фильтры позволяют ускорить работу грамматик. Если предложение на входе не соответствует ни одному из объявленных фильтров, то грамматика на них не запускается.

Фильтр записывается как последовательность терминалов. Предложение проходит фильтр если в нем в указанном порядке встретились слова описываемые этими терминалами. Расстояние между словами может быть любым.

Перед названиями терминалов в фильтрах ставится амперсанд &. У терминалов могут быть их обычные пометы: `kwtype`, `wff` и другие. Операторы (+, * и другие) в фильтрах использовать нельзя. Между терминалами может быть указано максимально допустимое расстояние в квадратных скобках [].

Если в грамматику включается другая грамматика директивой `#include`, то фильтры включаемой грамматики тоже учитываются.

```
#filter &Word<kwtype=fio> &AnyWord<wff=".*\d.*",h-reg1> [10] &Hyphen;
```

Объявление подстановки: `#define`, `#undef`

Директива `#define` также используется для реализации подстановок. В этом случае у директивы два аргумента: имя подстановки и ее значение. Именем подстановки может быть любая последовательность из букв, цифр и символа подчеркивания, первый символ имени подстановки не может быть цифрой. Значением подстановки может быть любое завершённое выражение, в синтаксисе, разрешенном в правой части правил грамматики: список граммем, цепочка нетерминалов с ограничениями и интерпретациями и т.п. Знак переноса строки означает конец значения переменной.

После объявления имя подстановки можно использовать с помощью синтаксиса в фигурных скобках со знаком доллара перед ним: `#{ ... }`. Между знаком \$, фигурными скобками и именем подстановки пробелы не допускаются.

Следует понимать, что подстановка «подставляется» не буквально (как, например, в языке C++), а сама по себе является сложным токеном в тексте грамматики. Поэтому подстановки нельзя использовать внутри строкового литерала (там она просто не будет распознана), а также внутри идентификаторов и других длинных токенов.

```
#define ALL_CASES [nom,acc,gen,dat,ins,loc]
BetterStatus -> PostStatusCoord<gnc-agr[1]>
                FIO<rt,gnc-agr[1],GU=${ALL_CASES}>;
BetterStatus -> PostStatusCoord<gram='им'>
                FIO<rt,GU=${ALL_CASES}>;
#undef ALL_CASES
```

```
#define HERO_WITH_INTERP Hero<rt,gram='ед,им'> interp (HeroMaybe.Fio from Fio)
Maybe -> ${HERO_WITH_INTERP}
        TellVerb<gram='3л,ед'> ToSomeone Word;
Maybe -> ${HERO_WITH_INTERP}
        ToSomeone TellVerb<gram='3л,ед'> Word;
#undef HERO_WITH_INTERP
```

Все определенные с помощью `#define` имена действуют дальше по тексту грамматики, в том числе во всех включенных позже файлах. Подстановку можно отменить, указав ее имя после директивы `#undef`. Рекомендуется всегда отменять подстановки явно, чтобы избежать неожиданных эффектов от включения одной грамматики в другую.

`#NO_INTERPRETATION`

Вводит запрет на интерпретацию в рамках текущей грамматики. Все операторы `interp` перестают срабатывать.

Правила

Грамматики для Томита-парсера состоят из правил. У каждого правила есть левая и правая части, разделенных символом `->`. В левой части стоит один нетерминал (S в примере, приведенном ниже). В правой части стоит список терминалов или нетерминалов (S1 ... Sn), после которого указываются условия (Q), применяемые ко всему правилу в целом. Условия могут отсутствовать. Правило заканчивается точкой с запятой (;).

Общий вид правила в грамматике

```
S -> S1 ... Sn { Q };
```


Для любого нетерминала, упомянутого в правой части правила, должно существовать правило, в котором этот нетерминал находится слева от \rightarrow . Порядок перечисления правил в грамматике не имеет значения, т.е. не влияет на процесс применения этой грамматики к тексту.

Если во входном тексте находится цепочка, которая соответствует правой части, то правило «срабатывает» и грамматика использует эту цепочку как значение символа `LeftPart`.

```
MoscowWord -> "москва"<h-рег1>;
MoscowGroup -> Adj<gnc-agr[1]>* MoscowWord<rt,gnc-agr[1]>;
```

Возьмется текст «Красная Москва»;

Символы S_i в правой части правила разделяются пробелами или оператором «или» `|`, а также могут сопровождаться унарными операторами `*`, `+` и `()`.

Операторы

Оператор |

Оператор `|` используется для сокращенной записи правил с одинаковой левой частью. У пробела приоритет выше, чем у `|`. Следующий пример иллюстрирует применение оператора `|`.

Пример без оператора |

```
NP -> Noun; // правило 1: именная группа может состоять
// из одного существительного
NP -> Adj<gnc-agr[1]> Noun<gnc-agr[1]>; // правило 2: именная группа может включать
// согласованные прилагательное и существительное
NP -> Noun Noun<gram="рд">; // правило 3: именная группа может состоять из двух
// существительных, при этом второе стоит в родительном падеже
```

Пример с оператором |

```
NP -> Noun | Adj<gnc-agr[1]> Noun<gnc-agr[1]> | Noun Noun<gram="рд">;
```

Оператор *

Оператор `*` (звезда Клини) после (не)терминала означает что символ повторяется ноль или больше раз. Соответственно, в правой части правила всегда должен быть (не)терминал без оператора `*`, т.к. правила с пустой правой частью в Томите запрещены.

```
NounGroup -> Noun Noun<gram="gen">*;
//Препроцессор парсера преобразует это правило так:
ARTIFICIAL_Noun -> Noun<gram="gen">;
ARTIFICIAL_Noun -> ARTIFICIAL_Noun Noun<gram="gen">
NounGroup -> Noun ARTIFICIAL_Noun<gram="gen">;
NounGroup -> Noun;
```

Оператор `*` позволяет определить согласование между копиями (не)терминала. Согласование записывается после оператора `*` в квадратных скобках `[]`.

```
NounGroup -> Noun Noun<gram="gen">*[gn-agr];
```

Оператор +

Оператор `+` после (не)терминала означает что символ повторяется один или больше раз. Оператор `+` также позволяет определить согласование между копиями (не)терминала.

Пример

```
Adjectives -> Adj+[gnc-agr];
```

Это правило аналогично следующему:

```
Adjectives -> Adj<gnc-agr[1]> Adj<gnc-agr[1]>+;
```

Символы в правой части правила

Символы S_i в правой части состоит из трех частей: $N \langle P_1, \dots, P_n \rangle \text{ interp } (I_1; \dots; I_n)$, где N — имя терминала или нетерминала, P_i — [пометы-ограничения](#) на свойства терминала/нетерминала N , I_i — имя поля и факта, куда при [интерпретации](#) записывается цепочка слов, подходящая под нетерминал N . Обязательным является только имя терминала/нетерминала N . Названия нетерминалов могут состоять из латинских букв, цифр и знака подчеркивания $_$, должны начинаться с буквы и не могут совпадать с зарезервированными именами [терминалов](#).

Каждый символ N соответствует одному слову (если это терминал) или группе слов (если нетерминал), которые обладают большим количеством разных свойств: грамматические характеристики, регистр, принадлежность к заранее определенному множеству слов и т.д. Ограничения P_i позволяют управлять этими свойствами, сужая множество слов, которым может соответствовать символ N .

См. [Пометы—ограничения при терминалах и нетерминалах](#)

Операции над правилами

Помета	Семантика	Пример использования
outgram	В поле пометы outgram находится вектор грамем, который присваивается символу в левой части правила.	<pre>ForeignWord -> Word<lat> {outgram = 'nom, acc, gen, loc, dat, ins'};</pre>
count	В поле пометы count записывается максимальное число терминалов, которое может включать в себя символ в левой части правила. Количество терминалов должно быть строго меньше указанного значения.	<pre>S -> NP {count = 10};</pre>
weight	В поле пометы weight записывается вес (число, со значением от 0 до 1), который присваивается символу в левой части правила. Подробнее использование весов правил описано в разделе Веса .	<pre>NP -> Adj Noun {weight = 0.7};</pre>
trim	Помета trim означает, что из выделенной цепочки выбрасываются все терминалы, которые не входят в факты, построенные в ходе интерпретации.	<pre>S -> NP {trim};</pre>
not_hreg_fact	Если у группы, построенной по правилу с ограничением not_hreg_fact, все слова, входящие в значения заполненных полей факта, находятся в верхнем регистре, то анализ такого дерева объявляется неуспешным.	<pre>S -> NP {trim, not_hreg_fact};</pre>

Кавычки в правилах

Символы двойных и одинарных кавычек " " и ' ' заключают строки в правой части правил. Слово в кавычках должно стоять в словарной форме. Если требуется ограничить слово косвенной формой, то можно использовать помету gram.

Кроме того статьи с кириллическими названиями обязательно должны оформляться двойными кавычками " " .

Пример

```
CatSlangWord -> "котэ";
CatSlangWord -> Word<kwtype="котэ">;
```

Синтаксические пометы

Помета `rt`

Терминал или нетерминал с пометой `rt` (`root`, корень) становится синтаксической вершиной поддерева, которое соответствует данному правилу. Это означает, что нетерминал в левой части правила получает грамматические и другие характеристики (не)терминала с пометой `rt`. В правой части правила не может находиться более одного (не)терминала с пометой `rt`. По умолчанию помета `rt` приписывается самому левому (не)терминалу в правой части правила.

Значение подстановки `#{ }`

Сочетание `#{ }` с именем подстановки фигурных скобках сообщает препроцессору, что требуется заменить имя подстановки на его значение. Подробнее о подстановках см. [Объявление подстановки: `#define`, `#undef`](#).

Комментарии

Можно использовать однострочные и многострочные комментарии как в языке C++.

Однострочные комментарии предваряются двумя косыми чертами `//` и заканчиваются концом строки. Они могут начинаться в конце любого текста грамматики или в начале строки. Комментарии нельзя поставить внутри значащего текста грамматики.

Пример

```
// Это однострочный комментарий
```

Начало многострочного комментария предваряется знаком `/*` заканчивается знаком `*/`.

Пример

```
/* Это многострочный
   комментарий
   */
```

Пометы-ограничения

На множество цепочек, которое описывает терминал или нетерминал можно наложить ряд ограничений. Для этого после (не)терминала в угловых скобках `< >` через запятую записываются пометы-ограничения, которые уточняют свойства (не)терминала. Список таких помет-ограничений приведен в таблице в конце этой главы.

Ограничения можно применять только к (не)терминалам стоящим в правой части правила и к (не)терминалам в фильтрах. Ограничение при нетерминале применяется к синтаксически главному слову группы, которую описывает данный нетерминал. К большей части помет можно применить оператор отрицания `~`, который означает «допустимы любые значения ограничения, кроме перечисленных». Оператор отрицания ставится слева от пометы.

Ограничения могут быть разнообразными по своей структуре. Некоторые представляют собой унарный оператор, некоторые имеют поле, которое может быть заполнено разными значениями. Ограничение «согласование» определяется на паре (не)терминалов.

У некоторых ограничений существуют свои особенности: некоторые из них нельзя применять к нетерминалам, некоторые не употребляются с отрицанием. Все эти свойства также описаны в сводной таблице. Ниже перечислены пометы, требующие подробного описания.

Тип ключевого слова (помета `kwtype`)

Помета `kwtype` (тип ключевого слова, `keyword type`) указывает, что слово (или главное слово многословной сущности), которому соответствует данных (не)терминал, должно быть объектом заданного

типа. Значение поля `kwtype` заполняется названием статьи словаря или ее названием типа статей словаря. Информация о статьях и их типах находится в одном из `gzt`-словарей.

Каждая такая статья словаря является инструкцией для построения объекта. В свою очередь, объект может быть как ключевым словом из определенного словаря, так и синтаксической группой, выделенной другой грамматикой. Поле `kwtype` играет ключевую роль в построении правил грамматик, поскольку позволяет правилам использовать результаты работы других грамматик в качестве атомарных объектов.

```
Animal -> Noun<kwtype="животные_центральной_африки"> ;
```

У пометы `kwtype` также существует специальное значение `none`. Оно означает, что данный символ не может быть объектом соответствующим какой-либо статье. При проверке омонимов слова на соответствие помете `kwtype=none`, рассматриваются только те объекты, которые определены в правилах данной грамматики. Например, если некоторое слово — это название компании, но описывающая его статья «имя_компании» не упоминается в правилах данной грамматики, то это слово будет считаться простой единицей и удовлетворит ограничению `kwtype=none`.



Внимание:

Это ограничение работает по-разному для терминала `Word` и для всех остальных символов. Символ `Word<kwtype=none>` срабатывает только на словах, у которых для всех омонимов выполняется вышеуказанное условие. Для остальных символов эта помета срабатывает у слов, у которых есть хотя бы один омоним для которого выполняется это условие.

После срабатывания `kwtype` у символа остается только один из омонимов, для которого есть эта помета. Следовательно, выше в синтаксическом дереве применение к тому же символу еще одного ограничения `kwtype` с другим значением бессмысленно: ведь к данному символу приписан всего один омоним и правило, которое требует наличия другого омонима, просто не сработает. Этим помета `kwtype` отличается от пометы `kwset`.

Примечание:

Использовать помету `kwtype` у нетерминалов расположенных на высоких уровнях синтаксического дерева не рекомендуется, т.к. это может замедлить работу парсера.

Множество типов ключевых слов (помета `kwset`)

Ограничение `kwset` (множество типов ключевых слов, `keyword set`) выполняет ту же функцию, что и `kwtype`. Разница между этими ограничениями состоит в том, что `kwset` применяется ко всем омонимам слова. Во-первых, это означает, что полученный после применения `kwset` символ может содержать другие омонимы, если есть другие статьи словаря которые также описывают этот символ. Во-вторых в помете `kwset` можно перечислить сразу несколько названий статей или их типов.

Запись `kwset=["статья_1", "статья_2", ..., "статья_n"]` означает, что у слова должен существовать хотя бы один омоним, который описывается одной из перечисленных статей или типов. Запись с отрицанием `kwset=~["статья_1", "статья_2", ..., "статья_n"]` означает, что если хотя бы один омоним слова описывается одной из перечисленных статей или типов, то правило не сработает.

Использование символов косая черта / звездочка * в поле пометы `kwset` аналогично их использованию в поле пометы `kwtype`.

Грамматические характеристики (помета `gram`)

Помета `gram` ограничивает символ допустимыми значениями грамматических характеристик. Например, запись `gram="nom, pl"` означает, что у слова или у главного слова многословной сущности должны быть граммы `nom` (`nominative`, именительный падеж) и `pl` (`plural`, множественное число). В списке граммем можно указывать любые граммы, в том числе и часть речи, т.е. терминал `NOUN` можно записать как `Word<gram="S">`. Список используемых граммем и их значения перечислены в таблице [Значения граммем](#).

Если у слова есть два омонима, то помета `gram` последовательно проверяет значения грамматических характеристик у каждого из них. Например, у слова «леса» есть омоним с граммами `nom` и `pl` (именительный падеж, множественное число) и омоним с граммами `gen` и `sg` (родительный падеж,

единственное число). Это слово будет соответствовать правилу с пометой `gram="gen, sg"` или с пометой `gram="pl"`, но не будет соответствовать правилу с пометой `gram="gen, pl"`.

Отрицание граммы в поле пометы `gram` применяется сразу ко всем омонимам. Например, слово «леса» не удовлетворяет помете `gram="~sg"` несмотря на то что у него есть омоним множественного числа. В одной помете `gram` можно указывать одновременно граммы с отрицанием и без.

Объединение грамматических характеристик (помета GU)

Помета `GU` (`gramtag union`) предоставляет более широкие возможности использования грамм в грамматиках. В своей самой простой форме `GU=["nom, pl"]` эта помета аналогична помете `gram`: она проверяет грамматические характеристики (в примере выше — «именительный падеж множественного числа») у каждого омонима и если находится омоним, удовлетворяющий этому условию, то правило срабатывает.

Грамммы записываются через запятую в квадратных скобках `[]`. Отрицание отдельных грамм в этой записи запрещено. Отрицание `~` перед квадратными скобками означает, что пересечение перечисленных грамм с граммами каждого омонима пусто, т.е. если интерпретация хотя бы одного омонима подойдет по множеству грамм в квадратных скобках, то правило с таким ограничением не работает.

Пример без отрицания

```
S -> Noun<GU=[sg, acc], rt>;
```

Сработает следующим образом:

```
- табуретка // именительный падеж
+ табуретку // винительный падеж
+ стол      // именительный или винительный падеж
- стола    // родительный падеж
```

Пример с отрицанием

```
S -> Noun<GU=~[sg, acc], rt>;
```

Сработает следующим образом:

```
+ табуретка // именительный падеж
- табуретку // винительный падеж
- стол      // именительный или винительный падеж
+ стола    // родительный падеж
```

Амперсанд `&` перед квадратными скобками означает, что парсер будет рассматривать граммы не у каждого омонима по отдельности, а одновременно у объединения грамматических признаков всех омонимов.

Пример с амперсандом

```
S -> Noun<GU=&[sg, acc, nom], rt>;
```

Сработает следующим образом:

```
- табуретка // именительный падеж
- табуретку // винительный падеж
+ стол      // именительный или винительный падеж
- стола    // родительный падеж
```

Кроме того помета `GU` позволяет записывать дизъюнкцию нескольких таких условий. Через вертикальную черту `|` («или») можно записать несколько разных списков грамм и правило сработает в том случае, если слово удовлетворяет хотя бы одному из перечисленных условий.

```
GU=[sg, ins] | &[nom, acc, gen, dat, ins]
```

В этой записи требуется выполнение одного из следующих условий

- у нетерминала есть омоним в творительном падеже единственного числа
- объединение всех грамм нетерминала включает все падежные граммы (слово не изменяется по падежам)

```
- стол
```

+ столом
+ пальто

Согласование в грамматиках

Для проверки совпадения значений одного или нескольких признаков одновременно у двух символов в Томите реализован механизм согласования. Например, согласование по падежу (помета *c-agr*) в следующем примере сработает на словосочетании «человек и кошка» и не сработает на словосочетании «человеку и кошка»:

A -> Noun<c-agr[1]> 'и' Noun<c-agr[1]>

Согласование — это бинарное отношение, которое записывается при каждом из двух символов в правой части правила. Чтобы различить согласования одного типа внутри одного правила, паре символов приписывается идентификатор (целое число). Если проверка согласования прошла успешно и если один из участников согласования — вершина синтаксической группы, то ему приписывается вектор граммом, полученный пересечением граммом участников согласования.

Отрицание согласования означает, что у двух символов не может быть тождества значений заданных согласованием признаков. При этом часть значений признаков у этих символов может и совпадать.

Один символ может одновременно участвовать в нескольких отношениях согласования с разными участниками. Например, существительное может одновременно быть согласовано с определением и со сказуемым или с двумя прилагательными. Ниже приведены некоторые примеры согласований и примеры разбираемых (плюс в начале строки) и не разбираемых (минус в начале строки) ими цепочек:

S -> Adj<gnc-agr[1]> Noun<gnc-agr[1], gram='им', sp-agr[2]> Adv Verb<rt, sp-agr[2]>;

+ наша Маша громко плачет
- нашей Маша громко плачет
- наша Маша громко плачут

S -> Adj<gnc-agr[1]> Adj<gnc-agr[2]> Noun<rt, gnc-agr[1], gnc-agr[2] >;

+ новый эстонский премьер-министр
- новые эстонский премьер-министр
- новый эстонская премьер-министр

S -> Participle<gnc-agr[2]> Adj<gnc-agr[1]> Noun<gnc-agr[1], gram='тв'> Noun<gnc-agr[2], gram='им', rt>;

+ обожаемый местным населением напиток
- обожаемая местным населением напиток
- обожаемый местными населением напиток

S -> Noun<~gnc-agr[1], rt> Adj<~gnc-agr[1]>;

+ платформа Северный
- платформа Северная

Далее описаны специальные виды согласования, удобные при написании грамматик для русских текстов.

Согласование fem-c-agr

Согласование *fem-c-agr* — это расширенное согласование *gnc-agr*, которое допускает рассогласование по роду, если у одного из членов согласования есть граммы *fem* и *snrn*. Это согласование введено для женских имен и фамилий, так как они могут употребляться вместе с существительными мужского рода, например поэт Ахматова. Но цепочка поэтесса Маяковский не возьмется.

Согласование after-num-agr

Согласование пары прилагательное + существительное после числительного в русском.

S -> Adj<after-num-agr[1]> Noun<after-num-agr[1], rt>;

пять американских президентов

два американских президента

Согласование fio-agr

Согласование fio-agr устанавливается между двумя объектами типа fio. Это согласование создано чтобы уточнить что в одном предложении участвуют ФИО записанные в одном формате. Согласование fio-agr осуществляется по следующим правилам:

1. Если одно из ФИО состоит только из фамилии, а другое содержит имя или инициал, то проверка неуспешна.
2. Если оба ФИО содержат инициалы или полные имена, но они по-разному расположены относительно фамилии (слева или справа), то проверка неуспешна.
3. Во всех остальных случаях согласование успешно.

Регулярные выражения (пометы wfm, wff, wfl)

Символ проверяется на соответствие регулярному выражению, указанному в поле помет wfm, wff и wfl. Помета wfm применяет регулярное выражение к вершине синтаксической группы, соответствующей данному нетерминалу. Пометы wff и wfl, соответственно к первому и последнему слову цепочки. Если символу соответствует одно слово в тексте, то результаты применения к нему этих трех помет эквивалентны.

Разбор регулярного выражения происходит с помощью свободно распространяемой библиотеки [Perl Compatible Regular Expressions](#). Синтаксис регулярных выражений языка Perl описан в документации: [perlre](#). Небольшое, но важное отличие от описанного синтаксиса регулярных выражений состоит в том, препроцессор дополнительно приписывает ко всем выражениям из поля помет wfm, wff и wfl символы начала и конца цепочки ^ и \$. На практике это чаще удобно, чем не удобно. В последнем случае, чтобы скорректировать эту особенность достаточно прибавить к регулярному выражению с обеих сторон последовательность «точка-звездочка» .*.

Запись регулярного выражения в кавычках требует дополнительных escape символов по сравнению со стандартным синтаксисом регулярных выражений Perl. Так как обратная косая черта \ уже является escape символом для строк, он должен быть повторен два раза для того чтобы в регулярном выражении возникла одна косая черта. Таким образом, чтобы распознать регулярным выражением сам знак \, в поле wfm необходимо использовать запись «\\» (сравните с /\/ в Perl).

Также можно использовать традиционный формат описания регулярных выражений с двумя косыми чертами / / вместо кавычек " ". Эта форма записи интерпретирует escape символы обычным для Perl образом.

```
S -> Word<wfm=/[А-Я-]{3,10}>; // только заглавные буквы и дефис длиной от 3 до 10 символов
```

Помета no_hom

Помета no_hom требует чтобы символ состоял из омонимов с одной частью речи. Например, правило S->Word<no_hom>; сработает на слове стол, но не сработает на слове столовая, т.к. последнее в словаре описано как прилагательное и как существительное.

В случае если символу присвоен «географический» kwtupe (т.е. любой kwtupe, тип которого начинается с префикса geo_ то помета no_hom проверяет, что все омонимы принадлежат географическим словарям.

Список всех помет

Помета	Семантика	Терминал	Нетерминал	Фильтр	Отрицание	Синтаксис
Статьи с обращением к словарям						

Помета	Семантика	Терминал	Нетерминал	Фильтр	Отрицание	Синтаксис
kwtype	Символ ограничен статей или типом статей указанным в поле kwtype.	+	+	+	+	kwtype="статья1" kwtype="статья1" kwtype= type1
kwset	Символ ограничен одной из статей или типов статей, указанных в поле kwset.	+	+	-	+	kwset=[type1,"статья1"] kwset=[type1,"статья1"]
kwsetf	Ограничение аналогичное kwset, которое применяется к первому (а не к главному) слову группы.	+	+	-	+	kwsetf=[type1,"статья1"]
label	Символ ограничен списком из статьи указанной в поле label.	+	+	-	+	label="статья1"
gztweight	Добавляет вес к общему весу нетерминала в левой части правила. Прибавляемый вес находится в словарной статье в поле, название которого указывается в поле пометы gztweight. Помету gztweight можно использовать только вместе с пометой kwtype.	+	+	-	-	kwtype="type1", gztweight="type1weight"
Грамматические ограничения						
gram	Проверяет значения грамматических характеристик отдельно для каждого омонима.	+	+	-	+	gram="sg,pl"

Помета	Семантика	Терминал	Нетерминал	Фильтр	Отрицание	Синтаксис
GU	Проверяет значения грамматических характеристик отдельно для каждого омонима или для всех омонимов одновременно.	-	+	-	+	GU=[nom,sg] GU=[sg] GU=&[nom,acc]
Согласования						
gnc-agr	Согласование по роду, числу и падежу: gender number case.	+	+	-	+	Adj<gnc-agr[1]> Noun<gnc-agr[1]>
nc-agr	Согласование по числу и падежу: number case.	+	+	-	+	N1<nc-agr[2]> N2<nc-agr[2]>
c-agr	Согласование по падежу: case.	+	+	-	+	Noun<c-agr[3]> 'и' Noun<c-agr[3]>
gn-agr	Согласование по роду и числу: gender number.	+	+	-	+	
gc-agr	Согласование по роду и падежу: gender case.	+	+	-	+	
fem-c-agr	Расширенное согласование gnc-agr, которое допускает рассогласование по роду, если у одного из членов согласования есть граммемы "fem,famn".	+	+	-	+	Noun<fem-c-agr[1]> Noun<fem-c-agr[1]> Сработает так: + врач Анна - врач Михаил
after-num-agr	Согласование пары прилагательное+существительное после числительного в русском, типа «5 американских президентов», но «2 американских президента»	+	+	-		

Помета	Семантика	Терминал	Нетерминал	Фильтр	Отрицание	Синтаксис
sp-agr	Согласование между субъектом и предикатом: subject predicate.	+	+	-	+	Noun<sp-agr[4]> Verb<sp-agr[4]>
fio-agr	Согласование двух объектов типа fio по формату записи.	+	+	-	+	
geo-agr	Согласование двух объектов географического тезауруса по принадлежности одной ветви геотезауруса.	+	+	-		
Регулярные выражения						
wfm	Регулярное выражение применяется к словоформе, которая является вершиной синтаксической группы.	+	+	+	+	Word<wfm=".*банк(а е о м)/">
wff	Регулярное выражение применяется к первой словоформе синтаксической группы.	+	+	+	+	Word<wff="им\."
wfl	Регулярное выражение применяется к последней словоформе синтаксической группы.	+	+	+	+	AnyWord<wfl="[0-9]{3}-[0-9]{2}-[0-9]{2}">; // номер телефона
Регистр букв в слове						
h-reg1	Первая буква слова стоит в верхнем регистре. Помета применяется к началу фразы, а не к вершине.	+	+	+		Noun<h-reg1>

Помета	Семантика	Терминал	Нетерминал	Фильтр	Отрицание	Синтаксис
h-reg2	В верхнем регистре стоит первая буква слова и как минимум еще одна буква слова, как например, в слове «МосСтрой».	+	+	+		
h-reg	Синоним h-reg2					
l-reg	Все буквы слова в нижнем регистре.	+	+	+		
Кавычки						
quoted	Слово или группа слов в кавычках.	+	+	-	+	SomeQuote<quoted> SomeName<quoted>
l-quoted	Слово или группа слов с открывающей кавычкой перед первым символом и без закрывающей кавычки после последнего символа.	+	+	-	+	
r-quoted	Слово или группа слов с закрывающей кавычкой после последнего символа и без закрывающей кавычки перед первым символом.	+	+	-	+	
Специальное						
fw	Самое первое слово символа должно быть первым словом предложения.	+	-	-	+	Lead_in<fw> ProperName<fw>
mw	Многословная сущность (multiword).	+	-	-	+	Noun<mw> SimpleWord<mw>
lat	Слово состоит из букв латинского алфавита.	+	+	+	-	Word<lat>

Помета	Семантика	Терминал	Нетерминал	Фильтр	Отрицание	Синтаксис
no_hom	Символ должно состоять из омонимов с одной частью речи.	+	+	-	-	Word<no_hom>
cut	Слово или синтаксическая группа исключается из интерпретации.	+	+	-	-	MainWords Context<cut>
rt	Обозначает вершину получившейся синтаксической группы. Строго говоря, rt не является ограничением, а относится к синтаксическим операторам, описанным ранее.	+	+	-	-	NP -> Adj Noun<rt>;
Словарность						
dict	Слово должно быть в морфологическом словаре.	+	-	+	+	

Подробно про кавычки

	Внуково	"Внуково"	"Внуково	Внуково"	."
quoted	no	yes	no	no	no
~quoted	yes	no	yes	yes	yes
l-quoted	no	no	yes	no	no
~l-quoted	yes	yes	no	yes	yes
r-quoted	no	no	no	yes	yes
~r-quoted	yes	yes	yes	no	no

Список терминалов

Терминал может быть леммой слова заключенной в одинарные кавычки или одним из зарезервированных имен терминалов из приведенной ниже таблицы. Обратите внимание, что терминалы описывают не только отдельные слова, но и многословные сущности (уже собранные грамматикой или иным способом), вершины которых удовлетворяют определению терминала. Большинство терминалов совпадает с известными частями речи, однако часто это совпадение неполное. Например, терминал Adj не распознает краткие прилагательные.

Некоторые терминалы распознают пунктуацию, например Comma, LBracket. Знаки пунктуации для которых не существует соответствующего терминала могут быть описаны сочетанием терминала AnyWord с пометой wff (см. [СПИСОК ПОМЕТ](#)), например: Anyword<wff="!">;

Терминал	Значение
'...'	Морфологическая лемма. Записывается в одинарных кавычках
AnyWord	Любая последовательность символов без пробелов. Нужно быть осторожнее с конструкцией AnyWord*, так как парсер в этом случае будет строить очень много вариантов и его работа сильно замедлится.
Word	Любое слово, состоящее из букв русского или латинского алфавита. Также разрешаются слова записанные через дефис. Под это определение не попадают цепочки, которые содержат знаки пунктуации (кроме дефиса), специальные ASCII символы и цепочки цифр.
Noun	Существительное (слово с граммемой «S»). Сюда не входят имена, фамилии и отчества.
Adj	Прилагательное, причастие, порядковое числительное или местоименное прилагательное: слово с граммемой «A», «partcp», «ANUM» или «APRO». Краткие прилагательные сюда не входят.
OrdinalNumeral	Порядковое числительное: слово с граммемой «ANUM».
Adv	Наречие: слово с граммемой «ADV».
Participle	Причастие: слово с граммемой «partcp».
Verb	Глагол: слово с граммемой «V».
Prep	Предлог: слово с граммемой «PR».
UnknownPOS	Нераспознанное морфологией слово. К этому нетерминалу не относятся несловарные слова, которые описаны в газетире. Морфологический компонент строит парадигму для таких слов, и они становятся словарными, если соответствующие статьи упоминаются в данной грамматике.
SimConjAnd	Слово «и».
QuoteDbI	Двойные кавычки.
QuoteSng	Одинарные кавычки.
LBracket	Открывающая скобка.
RBracket	Закрывающая скобка.
Hyphen	Тире.
Punct	Точка.
Comma	Запятая.
Colon	Двоеточие.
Percent	Цепочка символов, включающая символ %.
Dollar	Цепочка символов, включающая символ \$.
PlusSign	Знак плюс +.
EOSent	Символ конца предложения.

Значения граммем

При написании грамматик можно использовать как кириллические, так и латинские названия граммем. Кириллические названия граммем обязательно должны оформляться двойными кавычками " ". Некоторые граммемы, например граммемы частей речи, не имеют кириллических аналогов. В устаревшей

документации также могут встретиться кириллические названия граммем, которые использовались до перевода парсера на Яндекс-морфологию. Эти названия не поддерживаются в текущей версии парсера.

Латинская запись	Значение	Кириллическая запись
Части речи		
A	прилагательное	–
ADV	наречие	–
ADVPRO	местоименное наречие	–
ANUM	порядковое числительное	–
APRO	местоименное прилагательное	–
COM	часть композита (первая часть сложных слов)	–
CONJ	союз	–
INTJ	междометие	–
NUM	числительное	–
PART	частица	–
PR	предлог	–
S	существительное	–
SPRO	местоимение	–
V	глагол	–
Одушевленность		
anim	одушевленность	од
inan	неодушевленность	неод
Число		
sg	единственное число	ед
pl	множественное число	мн
dual	дуалис, двойственное число	дв
Род		
m	мужской род	муж
f	женский род	жен
n	средний род	сред
mf	общий род	мж
Падеж		
nom	номинатив, именительный падеж	им
gen	генитив, родительный падеж	род
dat	датив, дательный падеж	дат
acc	аккузатив, винительный падеж	вин
ins	творительный падеж	твор
loc	локатив (в лесу)	местн
abl	аблатив, исходный падеж	пр
voc	звательный падеж	зват
Форма прилагательных		
brev	краткая форма	кр

Латинская запись	Значение	Кириллическая запись
plen	полная форма	полн
poss	притяжательные прилагательные	притяж
Степень сравнения		
supr	превосходная степень	прев
comp	сравнительная степень	срав
Лицо		
1p	первое лицо	1-л
2p	второе лицо	2-л
3p	третье лицо	3-л
Время		
praes	настоящее время	наст
inpraes	непрошедшее	непрош
praet	прошедшее время	прош
aor	аорист, прошедшее время	прош2
fut	будущее время	буд
Вид		
ipf	несовершенный вид	несов
pf	совершенный вид	сов
Залог		
act	действительный залог	действ
pass	страдательный залог	страд
Репрезентация и наклонение глагола		
partcp	причастие	прич
ger	деепричастие	деепр
inf	инфинитив	инф
indic	изъявительное наклонение	изъяв
imper	императив, повелительное наклонение	пов
subj	сослагательное наклонение	сослаг
Переходность		
tran	переходный глагол	пе
intr	непереходный глагол	нп
Прочие обозначения		
reflex	возвратность	возвр
praed	предикатив	прдк
geo	географическое название	гео
persn	имя	имя
famn	фамилия	фам
patrn	отчество	отч
awkw	образование формы затруднено	затр
obsol	устаревшая форма	устар

Латинская запись	Значение	Кириллическая запись
inform	разговорная форма	разг
rare	редко встречающееся слово	редк
obsc	обсценная лексика	обсц
abbr	аббревиатура или сокращение	сокр
dist	искаженная форма	искаж

Интерпретация и нормализация

Интерпретация — это способ сохранить результат анализа текста в табличной форме, удобной для последующей обработки, например, для сохранения в базу данных. Интерпретация является последним этапом обработки текста в Томита-парсере. Процесс интерпретации состоит в распределении подцепочек из выделенной грамматикой цепочки по полям факта.

Факт — это множество значений полей-атрибутов, связанных смысловыми отношениями. Например, факт «Meeting» может иметь поля «Place», «Time» и «Theme».

Более формально

Интерпретация — это процедура, которая позволяет отобразить древесную синтаксическую структуру во множество линейно организованных фактов. Каждый факт можно представить как таблицу из одной строки и одной или нескольких колонок: в каждой ячейке этой таблицы содержится объект. Цель интерпретации состоит в сопоставлении множества строящихся нетерминалов грамматики множеству выделяемых для определенного типа факта объектов, т.е. в распределении подцепочек слов по заранее заданным полям в таблице.

Описание типов фактов

Любой факт, порожаемый грамматикой, должен быть описан. Факты описываются в словаре типа `gzt`. Название факта задается служебным словом `message`, после имени факта записывается его тип. Базовый тип всех фактов `NFactType.TFact`. Поля фактов перечисляются в фигурных скобках `{ }`.

```
message SomeFactName : NFactType.TFact
{
  ...
}
```

Типы фактов могут наследоваться. При этом все поля, входящие в базовый тип факта, войдут и в производный тип. Для того, чтобы от базового типа факта можно было наследовать производные, нужно указывать диапазон номеров полей, зарезервированных для использования в производных типах: `extensions 2 to 10`.

```
message BasicFact : NFactType.TFact
{
  required string field1 = 1;
  extensions 2 to 10;
}
message DerivedFact : BasicFact
{
  required string field2 = 2;
  required string field3 = 3;
}
```

Язык описания типов Protobuf описан в документации [Protocol Buffers/Language Guide](#).

Поля фактов

Все поля фактов имеют следующие характеристики: имя поля, тип поля, обязательность заполнения поля. Явное указание всех трех характеристик обязательно при описании типа факта.

Можно использовать Поля фактов относятся к одному из четырех типов, перечисленных в таблице.

Тип поля	Описание	Примеры значений
string	Строка. Это универсальное поле, может содержать любые строковые значения	расписная черепаха 11.5 news.yandex.ru
bool	Булевский тип данных, допускает одно из двух значений: true и false	true false
NFactType.TFio	Тип для хранения ФИО: полных фамилии, имени и отчества или инициалов.	Мюррей Билл Калугина Л.П.
NFactType.TDate	Тип для хранения даты и времени или временного интервала, выделенных статьей одного из двух специальных типов: date и date_chain.	1997-11-23 00:00:00 2011-06-17 19:30:00 2012-02-20 00:00:00 - 2012-02-22 00:00:00

Обязательность заполнения поля факта задается служебным словом `required` или `optional`, затем следует название типа поля (см. таблицу выше), а затем его название и после знака равенства = номер поля факта и точка с запятой. Слова `required` или `optional` означают, что поле факта соответственно обязательное или факультативное. Если в процессе построения дерева у факта остается незаполненным хотя бы одно обязательное поле, то факт не извлекается.

```
optional string CountryName = 1;
```

Номера поля должны быть уникальными для каждого факта. Поэтому важно иметь в виду, что если факт унаследовал свой тип от другого факта, то он также унаследовал и его поля с их номерами.

Значения полей по умолчанию

При описании поля факта ему можно приписать значение по умолчанию. В этом случае если будет заполнено хотя бы одно другое поле этого факта, то в этом поле уже будет заданное по умолчанию значение. Значение по умолчанию изменится, если при интерпретации это поле будет заполнено другой цепочкой.

Значение по умолчанию задается в квадратных скобках [] после объявления поля, в параметре `default`, как это сделано в следующем примере:

```
optional string Animal = 1 [ default = "неведома зверушка" ];
```

Капитализация полей фактов

При описании фактов можно указать желаемую капитализацию их значений при выводе. Пометы, влияющие на капитализацию, пишутся в квадратных скобках [] после описания полей. Разделителем между пометами служит запятая. Такие пометы и их значения перечислены ниже:

Тип поля	Описание
[(NFactType.normcase) = TITLE_CASE]	Капитализировать первую букву первого слова
[(NFactType.normcase) = CAMEL_CASE]	Капитализировать первую букву у всех слов

Сохранение лемм

Помета [(NFactType.info) = true] указывает парсеру, что необходимо сохранять в выходной файл леммы слов и помечать главные слова.

Интерпретация фактов в грамматиках

Для записи цепочки в поле факта интерпретации в Томите используется оператор `interp`. Цепочка, распознанная символом за которым следует `interp`, записывается в поле факта, которое обозначается в скобках после `interp`.

```
Animal -> Cat interp (Animal.Small);
```

В этом примере цепочка собранная нетерминалом `Cat` записывается в поле `Small` факта `Animal`. Один символ можно интерпретировать в несколько полей, в том числе и в несколько полей разных фактов. Для этого поля перечисляются через точку с запятой ;.

```
Animal -> Cat interp (Animal.Small; Object.Animate);
```

Записываемое в поле факта значение должно соответствовать типу данных этого поля. Если в конце построения дерева у факта остались незаполненные поля типа `required`, то факт не будет сохранен в выходной файл.

Получение фактов из внешних грамматик (оператор `from`)

В тех случаях, когда интерпретируемый символ представляет собой отдельный объект, построенный другой грамматикой, в которой были извлечены некоторые факты, эти факты можно получить в текущей грамматике. Оператор `from` позволяет явно указать при интерпретации из какого поля другого факта нужно извлечь данные для текущей интерпретации. Поля обоих фактов должны быть одинакового типа.

Оператор `from` в следующем примере `Animal<kwtype="cats"> interp (Circus.Staff from Animal.Small)` означает, что в поле `Staff` факта `Circus` надо положить данные из поля `Small` факта `Animal`, которое было получено интерпретацией в грамматике заданной статьей `"cats"`.

Название факта после оператора `from` можно опустить: `Animal<kwtype="cats"> interp (Circus.Staff from Small)`. В этом случае интерпретация будет искать любой факт с полем `Small`.

Если имена полей обоих фактов совпадают, то оператор `from` можно не использовать. Интерпретация автоматически извлечет значение поля с таким же именем. В следующем примере в поле факта попадет значение поля `Staff` из любого факта собранного грамматикой `"cats"`, если такой факт существует: `Animal<kwtype="cats"> interp (Circus.Staff)`. Если такого факта нет, то вся цепочка, представленная нетерминалом `Animal` попадет в `Circus.Staff`.

Интерпретация с явным значением (оператор `=`)

Процедура интерпретации позволяет также указать значение факта явно. Для этого значение факта записывается в кавычках после названия поля факта и знака равенства.

```
SportVerb interp (News.Type="спорт")
```

Конкатенация (оператор `+`)

Оператор конкатенации `+` используется в тех случаях, когда необходимо собрать в поле факта объекты из цепочек, между которыми стоят другие слова. В этом случае у цепочки расположенной правее и выше в синтаксическом дереве ставится знак плюс `+` перед именем поля факта в интерпретации.

Например, для того чтобы из цепочки «эксперт Института экономики по нефтедобыче» получить факт «эксперт по нефтедобыче» можно использовать следующие правила:

```
CompanyAndPosition-> Position<kwtype=post> interp(Data.Position) CompanyName;
S -> CompanyAndPosition PrepositionalPhrase interp(+Data.Position);
```

Одновременная интерпретация нескольких фактов

Что произойдет, если в грамматика построит дерево, в котором поле одного факта будет интерпретировано несколько раз? В этом случае существенно, что интерпретация осуществляется по символам правой части правила пошагово: слева направо. Можно представить себе, что при анализе текста парсер всегда движется по дереву слева направо и снизу вверх. Таким образом, узлы, которые находятся в дереве правее и выше обрабатываются позже. Итак, ситуация, при которой в итоговом

синтаксическом дереве поле одного факта будет интерпретировано несколько раз, порождает три сценария, которые могут комбинироваться между собой.

1. Если в одно и то же поле факта попадают две цепочки, которые пересекаются, например:

```
Animal -> Adjective Elephant interp (Animal.Big);
Creature -> Animal interp (Animal.Big);
```

то новое значение интерпретации запишется поверх старого.

2. Если цепочки не пересекаются и у этого же факта есть другое поле, которые интерпретируется в дереве один раз, то это второе поле дублируется, и на выходе порождаются два факта с разными цепочками в первом поле и одинаковыми во втором. Например, для типа факта Person с полями Name, Position и Company для цепочки

<i>Вагит , Алекперов</i>	<i>Михаил и Ходорковский</i>	<i>Евгений , Швидлер</i>	<i>президент НК «Лукойл»</i>	<i>НК ОАО и «Юкос»</i>	<i>НК «Сибнефть»</i>
Person.Name	Person.Name	Person.Name	Person.Position	Person.Company	Person.Company

можно получить три факта с повторяющимся полем Person.Position.

Person		
Name	Position	Company
Вагит Алекперов	президент	Лукойл
Михаил Ходорковский	президент	Юкос
Евгений Швидлер	президент	Сибнефть

3. Если цепочки не пересекаются и у этого же факта нет поля, которое интерпретируется в дереве один раз, то факты интерпретируются попарно в порядке парсинга: слева направо. Например, цепочка

<i>Евгений и Швидлер</i>	<i>Герман , Хан</i>	<i>президент НК «Сибнефть»</i>	<i>директор Тюменской нефтяной компании</i>
Person.Name	Person.Name	Person.Position	Person.Company

получит интерпретацию

Person		
Name	Position	Company
Евгений Швидлер	президент	Сибнефть
Герман Хан	директор	Тюменская нефтяная компания

Данные три сценария могут комбинироваться.

Нормализация

При интерпретации осуществляется нормализация значений полей факта. Нормализация выделенных объектов реализует два принципа:

1. **Морфологическая нормализация.** Выбирается синтаксическая вершина цепочки и ставится в словарную форму. Если в цепочке есть слова связанные с вершиной согласованием, то соответствующие грамемы также принимают новые значения, так чтобы согласование было соблюдено. Например, если нормализации подвергается именная группа «новым министром», с вершиной «министром», то результатом нормализации будет цепочка «новый министр».
2. **Нормализация через gzt-словарь.** Если при создании цепочки, которая попала в поле факта, участвовала статья словаря с заполненным полем Lemma, то при нормализации часть цепочки описанной этой статьей заменяется значением поля Lemma этой статьи. При этом к новой цепочке,

взятой из поля `lemma`, применяются грамлеммы, которые применялись к исходной цепочке в процессе построения дерева. Например, если в словаре существует статья

```
TAuxDicArticle "россия_полное_название"
{
  key = "российская федерация"
  lemma = "россия"
}
```

а в поле факта попадает «цепочка *министра культуры Российской Федерации*», собранная при участии статьи "россия_полное_название", то результатом нормализации будет цепочка «*министр культуры России*».

Пометы интерпретации

Дополнительные пометы позволяют влиять ход интерпретации и нормализации. Пометы стоят на последнем месте в выражении оператора `interp` и записываются после двух двоеточий `::`. Список помет перечислен в конце этого раздела в таблице.

Несколько примеров:

```
StreetName interp (Address.Street::not_norm);
Addressee interp (Address.Street::norm="dat,sg");
```

Помета	Семантика
<code>norm</code>	При нормализации вершине именной группы приписываются грамлеммы в поле пометы <code>norm</code> .
<code>not_norm</code>	Интерпретация проходит без нормализации.
<code>only_from_fact</code>	Нормализация происходит только в том случае, если значение поля факта извлекается из другого поля факта, полученного в результате работы другой грамматики. См. Получение фактов из внешних грамматик (оператор <code>from</code>) .

Синтаксис газеттира

Статья словаря в общем виде

```
ТипСтатьи "НазваниеСтатьи"
{
  Ключ = "... "
  Поле1 = "... "
  ...
  ПолеN = "... "
}
```

Описание статьи словаря состоит из типа статьи, названия статьи и содержания статьи. Содержание статьи ограничивается фигурными скобками `{ }`. Содержание статьи – это названия полей и их значения после знака равенства `=`. Каждое следующее поле отделяется переносом строки или фигурными скобками.

Фигурные скобки также отделяют внутренние блоки статьи.

```
object "Гора"
{
  key = "Говерла"
}
```

Тип статьи

Тип статьи записывается перед ее названием. Название типа статьи может состоять из букв латинского алфавита с учетом регистра, цифр и символа подчеркивания (`_`). Цифры в начале названия типа статьи

запрещены. Существует базовый тип статьи в газеттире — `TAuxDicArticle`, остальные типы являются производными от него и должны быть заранее объявлены. Тип статьи определяет состав ее полей.

Примеры правильных названий типов статей: `TAuxDicArticle`, `funny_animal`, `object`.

Типы статей часто используются для того, чтобы объединить статьи в определенные группы, например, все названия городов в тип статьи `city`. Для этого определяют тип статьи, производный от `TAuxDicArticle` и не содержащий полей.

```
message city : TAuxDicArticle {}
```

Типы статей можно использовать в ограничениях `kwtype` и `kwset`.

Предопределенные типы статей

Несколько типов статей определены во встроенном в парсер файле `kwtypes_base.proto`. Эти названия нельзя переопределять, но ими можно пользоваться в словарях, создавая статьи этого типа, и в грамматиках, ссылаясь на эти типы.

Название типа	Семантика
<code>fio</code>	встроенный в парсер C++-алгоритм выделения ФИО
<code>fio_without_surname</code>	Выделенные тем же алгоритмом ФИО, но без фамилии
<code>date</code>	встроенный в парсер алгоритм выделения дат (например, «01.08.1012», «1 января 2011 года»)
<code>number</code>	встроенный в парсер алгоритм выделения чисел (например, «1342 тыс.», «1,342 миллиона»)

Название статьи

Название статьи записывается в кавычках. Оно состоит из букв любого алфавита с учетом регистра, цифр и символов подчеркивание (`_`) и косая черта (`/`). Цифры в начале названия статьи запрещены. Названия статей должны быть уникальными. Перед названием статьи обязательно указываться ее тип.

Примеры правильных названий статей: `Гора, mountain/83, _dağ1`.

Комментарии

Комментарии вводятся двумя косыми чертами (как в C++).

```
// комментарий
```

Ключ

Ключ — это основное поле статьи. В ключе указывается, как именно ищется цепочка. В одной статье может быть несколько ключей. Текст поля `key` записывается в кавычках. Слова разделяются пробелами. Регистр значения не имеет.

```
key = "дед мороз"
```

Варианты ключа (!)

Знак `|` («или») разделяет несколько вариантов ключа.

```
key = "кто-то" | "что-то"
```

Поиск по точной форме (!)

Знак `!` («искать по точной форме») сообщает газеттиру, что искомое слово должно совпадать с точной формой слова, указанного в ключе. Знак `!` распространяется только на слово, перед которым он находится.

```
key = { "не !дай !бог" }
```

Помета `morph = EXACT_FORM` аналогична знаку «!», поставленному перед каждым словом ключа. Например:

```
key = { "не дай бог" morph = EXACT_FORM }
```

Ссылка на другую статью (\$)

Статья, на которую ссылается этот ключ, должна быть описана выше. Название статьи должно быть указано полностью: не разрешается ссылаться на группу статей, объединенных одинаковым префиксом перед знаком косая черта /. Важно понимать, что когда статья передается в ключ таким образом, теряется информация о главных словах.

```
key = "дикий $название_животного"
```

Регистр (Case = UPPER)

Обозначает, что значение ключа статьи должно быть в верхнем регистре. Например:

```
key = { "только в верхнем регистре" Case = UPPER }
```

Главное слово статьи (mainword)

Помета указывает, какое из слов ключа является главным словом (вершиной) статьи. Грамматическая информация от этого слова присваивается цепочке, которую описывает статья. Номера слов считаются, начиная с 1. Например:

```
key = { "дикая собака" mainword = 2 }
```

Если `mainword` записывается снаружи ключа, то он относится ко всем ключам данной статьи. Например:

```
key = "дикая собака"
key = "дикий кот"
mainword = 2
```

Замена цепочки (lemma)

Поле `lemma` указывает значение, на которое заменяется цепочка, определенная статьей. Эта замена происходит в полученной цепочке, а также в фактах, в которых эта цепочка используется. Например:

В поле `lemma` можно указать параметры `always` и `indeclinable`:

* если значение параметра `always = 1`, то определенная статьей цепочка заменяется на значение поля `lemma`, даже если статья не участвовала в формировании цепочки, в которую она включается. По умолчанию `always = 0`.

* если значение `indeclinable = 1`, то при лемматизации значение поле `lemma` не меняется. По умолчанию `indeclinable = 1`

Грамматические пометы в описании ключей

Помета gram

Значения этого поля — граммы (см. список всех используемых грамм в разделе [Значения грамм](#)), которые применяются к ключу. Например, `gram=sg` означает, что статье соответствуют только формы ключа в единственном числе. Например:

```
key = { "стол" gram=sg }
```

Помета word

Помета `word` указывает, к какому из слов многословного ключа применяется помета `gram`. Если помета `word` отсутствует, то `gram` применяется ко всему ключу (надо проверить). Например:

```
key = { "право потребитель" gram = {"mn", word=1} gram = {"род", word=2} }
```

Согласование (помета arg)

Помета описывает согласование между двумя словами в ключе. Возможны два типа согласования:

* по роду, числу и падежу: `agr=gnc_agr` или `agr=GENDER+NUMBER+CASE`

* по падежу: `agr=CASE`

```
key = { "автономный область" | "автономный округ" agr=gnc_agr }
key = { "главный редактор" agr=GENDER+NUMBER+CASE }
key = { "вооруженный сила" gram="мн" agr=CASE }
```

Специальные типы ключей

Есть два специальных типа ключей:

* `type = CUSTOM` для подключения к словарю грамматик и встроенных в парсер алгоритмов

* `type = FILE` для файлов со списками слов и словосочетаний

* `key = { "alg:fio" type=CUSTOM }` — ссылается на C++ алгоритм выделения ФИО

* `key = { "tomita:geo/city.cxx" type=CUSTOM }` — ссылается на грамматику. Можно указывать относительный или полный путь.

* `key = { "animals.txt" type=FILE }` — ссылается на файл со списком слов.

Связь между газеттиром и грамматикой

Ссылка из грамматики в газеттир

Если мы хотим ограничить терминал или главное слово нетерминала каким-то множеством слов или словосочетаний, то делаем это с помощью `kwtype` ([подробнее о kwtype](#)) и `kwset` ([подробнее о kwset](#)), например:

`Noun<kwtype=имя_типа_или__статьи>`

или

`NP<kwset=[имя_типа_или_статьи1 , . . . , имя_типа_или_статьи2]`

При нетерминале лучше использовать `kwset`, так будет быстрее работать.

Если указывается тип статьи, то это равносильно явному перечислению всех имен статей с этим типом. Иногда нужно, чтобы на вход парсеру уже подавались заранее собранные цепочки — мультиторды. Например, мы пишем грамматику выделяющую имена, и мы не хотим выделять имя в цепочке «Г. Владимир». Тогда можно написать статью в газеттире, где явно задать список таких городов со словами «г» или «город» («Г. Киров», «город Владимир»). Затем с помощью директивы `#GRAMMAR_KWSET` дать ссылку на эту статью в грамматике про имена. Тогда парсер перед запуском грамматики на предложении «В Г. Кирове» в качестве входного символа построит мультиторд «Г_Владимир» и парсер просто не увидит отдельно слова «Владимир». Подробнее про этот прием см. [тут](#).

Ссылка из газеттира в грамматику

Если мы хотим написать газеттирную статью, **ключом** которой являются все цепочки, выделяемые грамматикой из файла `grammar.cxx`, то мы пишем так:

```
key = { "tomita:grammar.cxx" type=CUSTOM }
```

К тому же нет другого способа сделать так, чтобы парсер увидел, скомпилировал и запустил грамматику, кроме как включить в газеттир статью со ссылкой на нее.

Использование газеттира при нормализации

При заполнении полей факта парсер пытается нормализовать выделенную цепочку. Но он не только ставит главное и согласованные с ним слова в начальную форму. Если среди нормализуемых слов есть слово, которому была приписана какая-то газеттирная статья, в которой было поле `lemma`, то вместо оригинального слова будет записано то, что стоит в этом поле.

Например, есть грамматика:

```
Country -> Word<kwtype= geo_country >;
S-> Country interp(Geo.Country);
```

И статья, на которую она ссылается:

```
geo_country "гео_россия"
{
  key = "россия" | "рф"
  lemma = "россия"
}
```

Тогда на фразе «президент РФ» грамматика отработает и напишет в поле Country факта Geo слово «россия».

Грамматические признаки мультиворда

Если ключ газетирной статьи состоит больше чем из одного слова и на эту статью есть ссылка из грамматики, то грамматика видит этот ключ как одно слово. При обработке входного предложения из этих слов строится мультиворд. При этом грамматические характеристики, которые проверяются пометами gram или GU<ссылка>, наследуются от главного слова. Оно указывается в поле mainword (см. [Главное слово статьи \(mainword\)](#)).

Например, грамматика, ссылающаяся на статью heads:

```
N -> Word<kwtype=heads>;
Post -> N<gram='anim'>;
```

Газетир, в котором есть эта статья:

```
heads "ген_дир"
{
  key = "генеральный директор"
}
```

На фразе «Генеральный директор Петров» такая грамматика не сработает. У статьи «ген_дир» не указано главное слово, которое по умолчанию первое. Тогда мультиворд «генеральный_директор» берет свои грамматические характеристики от слова «генеральный», у которого нет категории одушевленности. Правильно написать:

```
heads "ген_дир"
{
  key = "генеральный директор"
  mainword = 2
}
```

Неочевидные решения часто возникающих задач

Исключить неправильные частотные срабатывания

Часто возникает такая задача: есть грамматика, выделяющая, например, даты. Нужно убрать очевидные частые исключения, например «улица 8 марта».

Можно перечислить такие названия в газетире, включив их в качестве ключей к статье с название «плохие_даты».

```
Some_type "плохие_даты"
{
  key = "улица 8 !марта"
}
```

А в грамматике дат в директиве #GRAMMAR_KWSET

упомянуть эту статью.

```
#GRAMMAR_KWSET [ "плохие_даты" ]
```


В этом случае парсер перед запуском грамматики на тексте «дом на улице 8 марта» превратит подцепочку «улице 8 марта» в один мультиворд и отдельно слова «8» и «марта» он уже не увидит.

Ту же задачу можно решить общее, если не перечислять жестким списком все плохие упоминания дат в именах собственных, а описать их в грамматике. Тогда в ключе статьи «плохие_даты» будет уже ссылка на грамматику.

```
Some_type "плохие_даты"
{
  key = { "tomita:bad_dates.cxx" type=CUSTOM }
}
```

Описать только изменяемые существительные

Для этого нужно использовать помету GU, в которой запретить существительные, у которых есть сразу все падежи.

```
Noun<GU=~[nom,acc,dat,instr,loc,gen]>
```

Описать классы слов с регулярным словообразованием

Иногда не хочется в словаре перечислять слова, которые образуются стандартным способом. Например «по-испански», «по-французски». Такие случаи легко описать с помощью регулярок.

```
S -> Word<wfm=/по-.*[цс]ки>;
```

Отладка грамматики

Пошагово отследить срабатывание грамматики нельзя, но можно воспользоваться параметрами `PrintTree` (см. [Дерево синтаксического разбора \(параметр PrintTree\)](#)) и `PrintRules` (см. [Срабатывание правил \(параметр PrintRules\)](#)) в файле конфигурации. С помощью этих параметров посмотреть построенное синтаксическое дерево (`PrintTree`) и узнать, какие правила сработали или не сработали (`PrintRules`).

Также можно создать факт с именем `Test` и несколькими необязательными полями.

```
message Test: NFactType.TFact
{
  optional string A = 1;
  optional string B = 2;
  optional string C = 3;
  optional string D = 4;
  optional string E = 5;
}
```

А затем вставлять интерпретацию этого факта во все интересующие места грамматики (только нужно не забыть при этом указать этот факт в [файле конфигурации](#), чтобы он печатался). Это аналог отладки с помощью `print` в Perl или `printf` в C++.

Пример работы парсера

Рассмотрим это на примере, в котором будем собирать информацию о том, кто в каком городе родился. При этом посмотрим последовательно на все этапы анализа текста. Для начала опишем поля факта. Факт можно представить себе как запись в таблице, а поля - как ее колонки. В данном случае нужны две колонки: одна для имени человека (назовем `Person`), другая — для места рождения (назовем `Place`). Сам факт назовем `BornFact`.

Файл `facttypes.proto`

```
import "base.proto";
import "facttypes_base.proto";
message BornFact: NFactType.TFact
{
  required string Person = 1;
  required string Place = 2;
}
```

Возьмем два простых предложения, в которых есть интересующая нас информация: Иван родился в Нижнем Новгороде и Михаил родился в Петербурге и напишем грамматику, которая может описать предложения такого типа:

Файл `bornin.cxx`

```
#encoding "utf-8"
Born -> Verb<kwtype=born>;
City -> Noun<kwtype=city>;
Person -> AnyWord<gram="имя">;
S -> Person interp(BornFact.Person) Born "в" City interp(BornFact.Place);
```

Правило `Born -> Verb<kwtype=born>` описывает возможные предикаты, которые могут указывать на рождение, и ссылается на kw-типе газеттира `born`. Аналогичным образом, правило `City -> Noun<kwtype=city>` описывает возможные названия городов и ссылается на kw-типе `city`. Правило `Person -> AnyWord<gram="имя">` сработает на словах, у которых есть граммема имя. Последнее правило собирает полную цепочку с предлогом `в`. Корневой словарь газеттира при этом будет выглядеть следующим образом:

Файл `dic.gzt`

```
encoding "utf8";
import "base.proto";
import "articles_base.proto";
import "kwtypes_my.proto";
import "facttypes.proto";
TAuxDicArticle "РодилсяВ"
{
  key = { "tomita:bornin.cxx" type=CUSTOM }
}
city "Нижний_Новгород"
{
  key = "Нижний Новгород";
  mainword = 2;
}
city "Санкт_Петербург"
{
  key = "Санкт-Петербург" | "Питер" | "Петербург";
  lemma = "Санкт-Петербург";
}
born "родиться"
{
  key = "родиться"
}
born "появиться_на_свет"
{
  key = "появиться на свет"
}
```

Статья `РодилсяВ` ссылается на грамматику, находящуюся в файле `bornin.cxx`. Статьи типа `city` и `born` перечисляют слова и словосочетания соответствующих типов, чтобы на них можно было ссылаться в грамматике при помощи ограничения `kwtype=...`. Свойство `mainword = 2` указывает, что в словосочетании `Нижний Новгород` главным словом является `Новгород`, а не первое слово, как было бы по умолчанию. Это важно, т.к. грамматические признаки главного слова приписываются всему словосочетанию. `Санкт-Петербург` может быть назван несколькими способами. Чтобы учесть и это, три варианта перечислены в поле `key`. Для того, чтобы в поле `BornFact.Place` все записи про `Санкт-Петербург` выглядели одинаково, добавлено поле `lemma=`, указывающее на тот вариант названия, который должен быть использован при нормализации.

kw-типы `city` и `born` объявлены в файле `kwtypes_my.proto`. Все используемые kw-типы должны быть описаны таким образом. Некоторые из них (ссылка?) уже описаны во встроенном файле `kwtypes_base.proto`.

Файл `kwtypes_my.proto`

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

Конфигурационный файл для запуска парсера выглядит следующим образом:

Файл `config.proto`

```
encoding "utf8";
```

```
TTextMinerConfig {
  Dictionary = "dic.gzt";           // корневой словарь газеттира
  PrettyOutput = "debug.html";     // файл с отладочным выводом
  Input = {
    File = "test.txt";             // файл с анализируемым текстом
    Type = dpl;                    // режим чтения "document per line" (каждая строка - отдельный документ)
  }
  Articles = [
    { Name = "РодилсяВ" }          // Запустить статью корневого словаря "РодилсяВ"
  ]
  Facts = [
    { Name = "BornFact" }          // Сохранить факт "BornFact"
  ]
  Output = {
    File = "facts.txt";            // Записать факты в файл "facts.txt"
    Format = text;                 // используя при этом простой текстовый формат
  }
}
```

Файл test.txt

```
Иван родился в Нижнем Новгороде.
Михаил появился на свет в Петербурге.
```

Для запуска парсера достаточно указать в качестве аргумента программы `tomitaparser.exe` имя конфигурационного файла `config.proto`:

```
tomitaparser.exe config.proto
```

В результате своей работы парсер напечатает сообщения об успешной компиляции корневого словаря и грамматики, а также о времени начала и завершения обработки текста:

```
Compiling "dic.gzt" ... OK
  Compiling bornin.cxx ... (10 unique symbols, 5 rules) OK
[25:10:12 13:35:05] - Start. (Processing files.)
[25:10:12 13:35:05] - End. (Processing files.)
```

В результате обработки будут сгенерированы два файла: `facts.txt` и `debug.html`. Первый файл (`facts.txt`) содержит обработанный текст и выделенные из него факты. Простой текстовый формат удобен для проверки выделения фактов во время разработки и не предназначен для последующей автоматической обработки. Для автоматической обработки используйте формат `Format=xml`. Второй файл (`debug.html`) содержит более подробную отладочную информацию, включая морфологический разбор (отображается при наведении указателя мышки на слово) и цепочки слов, соответствующие статьям корневого словаря.

Файл facts.txt

```
Иван родился в Нижнем Новгороде .
BornFact
{
  Person = иван
  Place = Нижний Новгород
}
Михаил появился на свет в Петербурге .
BornFact
{
  Person = михаил
  Place = Санкт-петербург
}
```

Файл debug.html

Иван родился в Нижнем Новгороде . EOS

Михаил появился на свет в Петербурге . EOS

BornFact	
Person	Place
иван	Нижний Новгород
михаил	Санкт-петербург

Text	Type
иван родился в Нижнем Новгороде	TAuxDicArticle [РодилсяВ]

Text	Type
михаил появился на свет в Санкт-петербург	TAuxDicArticle [РодилсяВ]

Далее рассмотрим подробно работу парсера на этих двух предложениях. В приведенных ниже таблицах последовательно показаны этапы анализа текста, начиная с морфологического разбора, результатом которого являются леммы и грамматические признаки. Далее ищутся ключи из упомянутых в грамматике статей газеттира. В первом предложении это ключ «родиться» (kwtype="born") и ключ «Нижний Новгород» (kwtype="city"). Поскольку kwtype="city" используется в грамматике, то цепочка «Нижний Новгород» попадет к ней на вход в виде мультиторда, на котором сработает терминал Noun<kwtype="city">. Аналогичным образом сработает и терминал Verb<kwtype="born"> на цепочке «появился на свет». Из терминалов, перечисленных в строке «терминалы», GLR-парсер соберет нетерминалы Person, Born и City по правилам, описанным в файле bornin.sxx. В правиле S -> Person interp(BornFact.Person) Born "в" City interp(BornFact.Place) выполняется интерпретация, в ходе которой цепочки, сопоставленные нетерминалам Person и City, нормализуются и интерпретируются (записываются) в поля BornFact.Person и BornFact.Place факта BornFact.

текст	<i>Иван</i>	<i>родился</i>	<i>в</i>	<i>Нижнем</i>	<i>Новгороде</i>
леммы	иван	родиться	в	нижний	новгород
грамматические признаки	S, persn, nom, sg, m, anim	V, praet, sg, indic, m, ipf, intr	PR	A, abl, sg, plen, m, n	S, geo, abl, sg, m, inan
kw-type		born		city	
терминалы	AnyWord<gram="person">kwtype="born">			Noun<kwtype="city">	
нетерминалы	Person	Born	"в"	City	
интерпретация (поля факта)	BornFact.Person			BornFact.Place	

текст	<i>Михаил</i>	<i>появился</i>	<i>на</i>	<i>свет</i>	<i>в</i>	<i>Петербурге</i>
леммы	Михаил	появиться	на	свет	в	Санкт-Петербург
грамматические признаки	S, persn, nom, sg, m, anim	V, praet, sg, indic, m, pf, intr	PR	S, nom, acc, sg, m, inan	PR	S, geo, abl, sg, m, inan
kw-type		born				city
терминалы	AnyWord<gram="verb">kwtype="born">				"в"	Noun<kwtype="city">

нетерминалы	Person	Born	"в"	City
интерпретация (поля факта)	BornFact.Person			BornFact.Place

Запуск парсера и файл конфигурации

Имя конфигурационного файла — единственный аргумент программы `tomitaparser`. Т.е. запускать надо вот так:

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

В конфигурационном файле указывается:

- где находятся тексты, которые нужно обработать;
- путь к корневому словарю;
- какие статьи корневого словаря запускать;
- какие факты нужно сохранить и в каком формате;
- куда сохранять факты;
- несколько вспомогательных параметров.

Описание формата конфигурационного файла

```
message TTextMinerConfig {
  message TInputParams {
    enum SourceFormat {
      plain = 0;
      html = 1;
    }
    enum SourceType {
      no = 0;
      dpl = 1;
      mapreduce = 2;
      arcview = 3;
      tar = 4;
      som = 5;
      yarchive = 6;
    }
    optional string File = 1 [default = ""]; // Если File пропущено - то STDIN
    optional string Dir = 2 [default = ""]; // Если читаем файлы из папки
    optional SourceFormat Format = 3 [default = plain]; // Если Format пропущено - то plain

    optional SourceType Type = 4; // Если Type пропущено - то читаем обычный файл (то же, что
    "no")
    optional string Encoding = 5 [default = "utf8"]; // Кодировка входных данных (если не задано - utf8)
    optional string SubKey = 6; // Для mapreduce
    optional int32 FirstDocNum = 7 [default = -1]; // Для arcview: первый документ, который надо анализировать
    // (если не задано - с начала архива)
    optional int32 LastDocDum = 8 [default = -1]; // Для
    arcview: последний документ, который надо анализировать
    // (если не задано - до конца архива)
    optional string Url2Fio = 9; // Таблица url2fio для interview
    optional string Date = 10; // Дата для форматов кроме arcview
  }
  message TOutputParams {
    enum OutputMode {
      append = 0;
      overwrite = 1;
    }
    enum OutputFormat {
      xml = 0;
      text = 1;
      proto = 2;
      mapreduce = 3;
    }
    optional string File = 1 [default = ""]; // Если не задано - то STDOUT
    optional OutputMode Mode = 2 [default = append]; // Если не задано явно, но файл существует - ошибка
    optional OutputFormat Format = 3 [default = xml]; // Если не задано - то xml
    optional string Encoding = 4 [default = "utf8"]; // Кодировка выходных данных, по умолчанию - utf8
    optional bool SaveLeads = 5 [default = false]; // Сохранять ли лиды
    optional bool SaveEquals = 6 [default = false]; // Сохранять ли equals facts
  }
  message TArticleRef {
    required string Name = 1;
  }
  message TFactTypeRef {
    required string Name = 1;
  }
}
```

```

    optional bool NonEquality = 2 [default = false];
  }
  required string Dictionary = 1; // Может быть папкой, тогда парсер будет работать
                                  // в режиме совместимости со старыми путями
  optional string PrettyOutput = 2; // Имя файла, куда будет писаться отладочный HTML файл
  optional uint32 NumThreads = 3 [default = 1]; // кол-во потоков (по умолчанию – один)
  optional string Language = 4 [default = "ru"]; // Язык (по умолчанию – русский)
  optional string PrintTree = 5; // см. parsedoc
  optional string PrintRules = 6; // см. parsedoc
  optional TInputParams Input = 7;
  optional TOutputParams Output = 8;
  repeated TArticleRef Articles = 9;
  repeated TFactTypeRef Facts = 10;
  optional bool SavePartialFacts = 11 [default = true];
  repeated TArticleRef ArticlesBeforeFragmentation = 12;
}

```

Отладочный вывод

Парсер может вывести на экран или сохранить в файл подробную информацию о ходе анализа текста. Формат вывода этой отладочной информации предназначен для удобства просмотра и не предназначен для автоматической обработки. Параметр `PrintTree` указывает файл, в который будет записаны деревья синтаксического разбора. Параметр `PrintRules` — применяемые правила. В качестве имени файла можно указать `STDOUT` или `STDERR`, чтобы вывести отладочную информацию в соответствующий поток.

Параметры `PrintTree` и `PrintRules` можно использовать одновременно.

Деревья синтаксического разбора (параметр `PrintTree`)

Параметр `PrintTree` позволяет указать файл, в который будут выведены синтаксические деревья, которые парсер строит в процессе разбора.

Пример

```

// грамматика
S -> Adj Noun;
// текст
Тяжелый труд облагораживает хорошего человека.
// фрагмент конфигурационного файла
PrintTree="tree.txt";

```

В результате работы парсера будет сформирован файл `tree.txt`, в который будут помещены все построенные парсером деревья.

`tree.txt`

```

=====first.cxx=====
coverage: 2, weight: 0.36666666
S -> Adj[*Тяжелый] Noun[труд] :: 0.43333333
|
|-- Adj -> Тяжелый :: 1
|
|-- Noun -> труд :: 1
coverage: 2, weight: 0.34
S -> Adj[*хорошего] Noun[человека] :: 0.43333333
|
|-- Adj -> хорошего :: 1
|
|-- Noun -> человека :: 1
=====multiwords=====
Тяжелый_труд: TAuxDicArticle,
             наша_первая_грамматика,
хорошего_человека: TAuxDicArticle,
                   наша_первая_грамматика,

```

Срабатывание правил (параметр `PrintRules`)

Параметр `PrintRules` позволяет указать файл, в который будет выведен список сработавших правил.

Пример

```

// грамматика
S -> Adj Noun;
// текст
Тяжелый труд облагораживает хорошего человека.
// фрагмент конфигурационного файла
PrintRules="rules.txt";

```

rules.txt

```

=====processing first.cxx=====

S -> Adj[*Тяжелый] Noun[труд] ::

S -> Adj[*хорошего] Noun[человека] ::

*****finished first.cxx*****

```

Примеры

Важно: не забывать указывать кодировку самого конфига

```

encoding "utf8";
TTextMinerConfig {
  Dictionary = "../tomita-work/simple.gzt";
  Articles = [
    { Name = "_статья" }
  ]
  Facts = [
    { Name = "TFdo" }
  ]
}

```

Будет читать текст с консоли как один документ в кодировке utf8. Запустит статью «_статья» из корневого gzt-файла «../tomita-work/simple.gzt». Напечатает XML с фактами типа TFdo в консоль.

```

encoding "utf8";
TTextMinerConfig {
  Dictionary = "../tomita-work/simple.gzt";
  Input = {
    File = "fdo.txt";
    Encoding = "windows-1251";
  }
  Articles = [
    { Name = "_статья" }
  ]
  Facts = [
    { Name = "TFdo" }
  ]
}

```

Прочитает файл fdo.txt как один документ в кодировке Windows-1251 (надо явно указывать, т.к. по умолчанию — utf8). Запустит статью «_статья» из корневого gzt-файла «../tomita-work/simple.gzt». Напечатает XML с фактами типа TFdo в консоль.

```

encoding "utf8";
TTextMinerConfig {
  Dictionary = "../tomita-work/simple.gzt";
  Input = {
    File = "fdo.txt";
    Encoding = "windows-1251";
  }
  Output = {
    File = "facts.proto";
    Format = proto;
  }
  Articles = [
    { Name = "_статья" }
  ]
  Facts = [
    { Name = "TFdo" }
  ]
}

```

Сохранит файл с фактами типа TFdo в facts.proto.

Обратная связь

Если вы не нашли ответ на свой вопрос в [документации](#), вы можете обратиться в службу поддержки. Просим вас максимально подробно описать возникшую проблему.