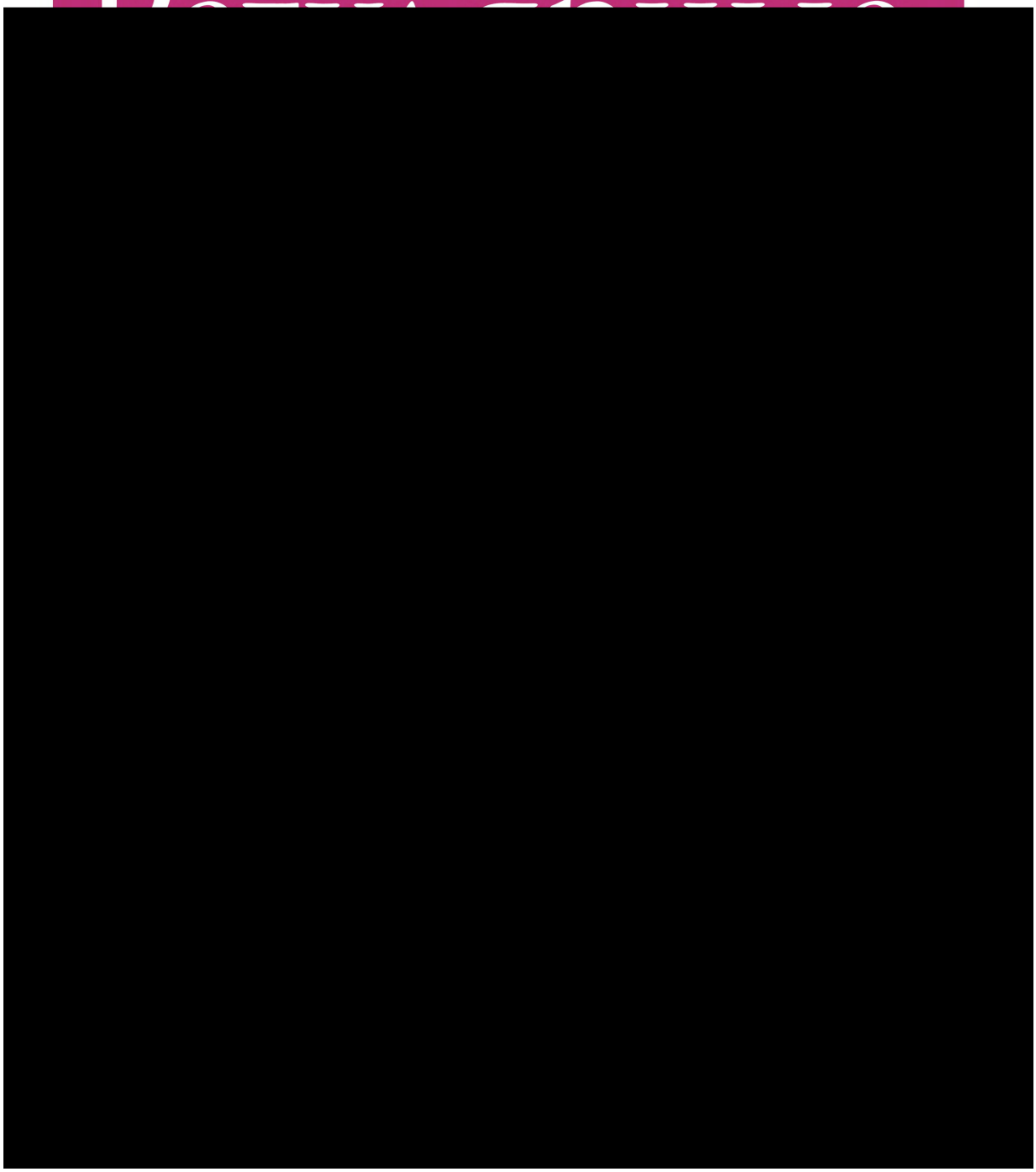
Лаконично, доходчиво, пошагово

Основы



Регулярные выражения

Основы

Introducing Regular Expressions

Michael Fitzgerald

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Регулярные выражения

Основы

Майкл Фицджеральд

Москва • Санкт-Петербург • Киев
2015

ББК 32.973.26-018.2.75
Ф66
УДК 681.3.07

Издательский дом "Вильямс"
Главный редактор *С.Н. Тригуб*
Зав. редакцией *В.Р. Гинзбург*
Перевод с английского и редакция канд. хим. наук *А.Г. Гузикевича*
По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Фицджеральд, Майкл.

Ф66 Регулярные выражения: основы. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 144 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1953-3 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Майкл Фицджеральд

Регулярные выражения: основы

Литературный редактор *И.А Попова*
Верстка *О.В Мишутина*
Художественный редактор *Е.П Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 06.04.2015. Формат 70x100/16
Гарнитура Times. Усл. печ. л. 11,61. Уч.-изд. л. 7,7
Тираж 400 экз. Заказ № 2112

Отпечатано способом ролевой струйной печати
в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1953-3 (рус.)
ISBN 978-1-449-39268-0 (англ.)

Оглавление

Содержание

Об авторе

Майкл Фицджеральд — программист и консультант, автор ряда книг и статей по программированию, опубликованных издательствами O'Reilly и John Wiley & Sons. Был членом комитета по разработке RELAX NG — языка описания структуры XML-документов.

Об изображении на обложке

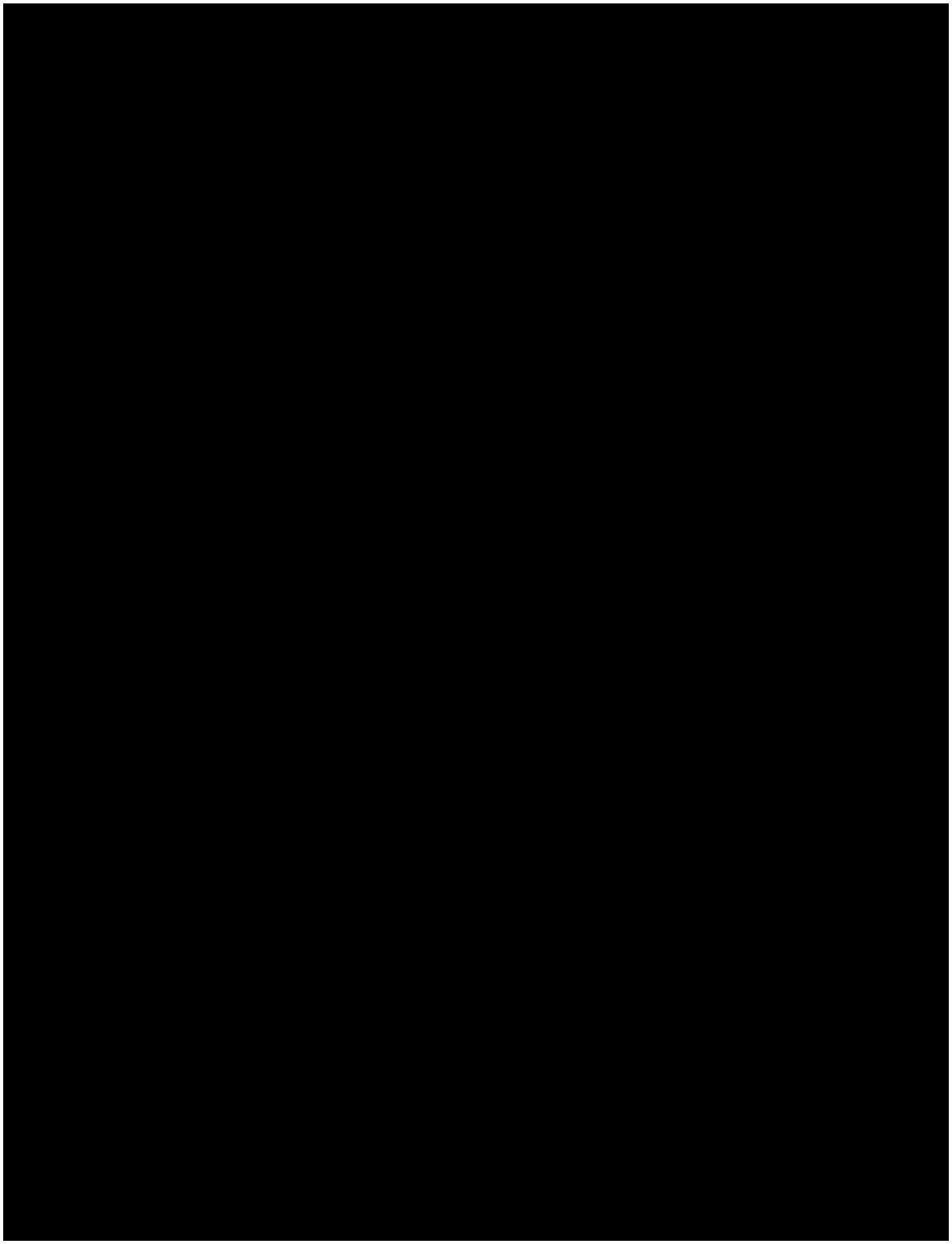
Существо на обложке книги — это *крылан*, рукокрылое млекопитающее, родственное летучей мыши, представитель семейства *Pteropodidae* подотряда *Megachiroptera*.

Латинское название крыланов "мегахироптера" (гигантские рукокрылые) не совсем точное. Крупные виды крыланов действительно отличаются большими размерами (размах крыльев до 1,7 м, длина тела до 40 см) и массой (до 1 кг), однако масса некоторых меньших их сородичей составляет всего 15 г, а длина туловища — 5 см. Другие названия крыланов — "летучие собаки", "летучие лисицы", "летучие мыши Старого Света", "фруктовые летучие мыши". При этом важно подчеркнуть, что между крыланами и летучими мышами (подотряд рукокрылых *Microchiroptera*) имеются существенные различия.

В полном соответствии с одним из своих названий, крыланы — большие любители фруктовых плодов и цветочного нектара. Одни из них прокусывают зубами кожуру плода и поедают мякоть, другие же просто высасывают сок с небольшим количеством мякоти, а остальное выплевывают. Поскольку многие крыланы питаются цветочным нектаром, они хорошие опылители и разносчики семян. По данным организации World Bat Sanctuary, занимающейся охраной этого биологического вида, около 95% всей новой растительности влажных тропиков вырастает из семян, разносимых крыланами. Подобные отношения между крыланами и растениями являются одной из разновидностей *мутуализма*, а явление перекрестного опыления у растений за счет летучих мышей (и крыланов) получило название *хироптерофилия*.

Крыланов можно обнаружить во многих уголках планеты, но преимущественные места их обитания — влажные тропические зоны, изобилующие фруктовыми деревьями и цветочной растительностью. Крыланы — отличные летуны, но их приземление выглядит довольно неуклюжим. Часто они буквально сваливаются на ветви дерева и зависают на нем, ухватившись лапами за ветви. Поэтому многие ошибочно полагают, будто крыланы слепы. В действительности, как показали результаты научных наблюдений, крыланы обладают самым острым зрением среди всех рукокрылых, которые в большинстве своем ориентируются в пространстве с помощью эхолокации. Крыланы же в поисках пищи и при передвижении наряду с развитым обонянием используют также зрение.

Изображение на обложке взято из книги Кассела *Natural History*.



Введение

Эта книга научит вас работать с регулярными выражениями на конкретных примерах. Цель книги — максимально упростить освоение регулярных выражений. Фактически каждое понятие обсуждается с приведением соответствующих примеров, которые читателю будет легко повторить и проверить.

Регулярные выражения упрощают поиск определенных образцов текста. Точнее говоря, они представляют собой текстовые строки, описывающие на специальном языке искомые шаблонные комбинации символов в наборах текстовых строк, в большинстве случаев — строк, хранящихся в документах или файлах.

Формальная теория регулярных выражений была впервые изложена математиком Стивеном Клином в его книге *Introduction to Metamathematics* (New York, Van Nostrand), опубликованной в 1952 году, однако ее основные концепции были разработаны еще в начале 1940-х годов. Широкую популярность среди компьютерных специалистов регулярные выражения приобрели в начале 1970-х годов после выхода операционной системы Unix (детища Брайана Кернигана, Денниса Ритчи, Кена Томпсона и других сотрудников корпорации AT&T Bell Laboratories), включающей такие утилиты, как *sed* и *grep*.

Насколько мне известно, одним из первых компьютерных приложений, в которых начали использоваться регулярные выражения, был текстовый редактор QED (сокр. от "Quick Editor" — быстрый редактор). Код этого редактора был написан для системы распределения времени Berkley Time-Sharing System, выполнявшейся на компьютере SDS 940 компании Scientific Data Systems. Версия QED, задокументированная в 1970 году, — это переписанный Кеном Томпсоном вариант существовавшего в то время редактора для системы Compatible Time-Sharing System, который был разработан сотрудниками вычислительного центра MIT и включал одну из ранних, если не самую первую, практических реализаций регулярных выражений для вычислительных целей. (Возможности регулярных выражений, предлагаемые в редакторе QED, описаны в табл. А.1 приложения.)

Для демонстрации примеров в книге применяются различные средства, и я надеюсь, что большинство из них окажутся для вас полезными и их использование не вызовет трудностей, однако некоторые средства могут быть недоступны для пользователей Windows. Конечно, вы сможете пропускать описания примеров, которые вам не удается воспроизвести в силу отсутствия того или иного средства. Однако я считаю, что каждый, кто всерьез задумывается о карьере компьютерного специалиста, должен обязательно ознакомиться с методами обработки регулярных выражений в Unix-подобных средах. Я работаю с подобными системами вот уже 25 лет и все равно каждый день нахожу для себя что-то новое.

> "Бедняги, не знакомые с UNIX, обречены заново изобретать велосипед", — Генри Спенсер

К некоторым из представленных в книге инструментов возможен доступ в Интернете с помощью браузера, что будет наиболее удобно для большинства читателей. Часть инструментов требует использования командной строки, тогда как другие инструменты доступны в виде настольных приложений. В случае отсутствия у вас нужного инструментария его можно легко загрузить из Интернета. Большинство инструментальных средств, о которых пойдет речь, бесплатны или стоят совсем недорого.

В этой книге я стараюсь описывать регулярные выражения простым и понятным языком. Специальная терминология употребляется весьма экономно и только в самой необходимой степени. Я придерживаюсь именно такого подхода, поскольку многолетний опыт научил меня, что чрезмерно насыщенный терминами текст нередко препятствует пониманию сути. Это соответствует принципу, положенному в основу книги: много полезного можно делать даже тогда, когда еще не до конца вник в суть проблемы.

Существует множество различных реализаций регулярных выражений. В частности, вы увидите, что они используются в таких инструментах командной строки Unix, как *vi* (*vim*), *grep* или *sed*. Регулярные выражения встроены в такие языки программирования, как Perl (а разве могло быть иначе?), Java, JavaScript, C#, Ruby и многие другие, а также в декларативные языки наподобие XSLT 2.0. Список реализаций может быть продолжен такими настольными приложениями, как Notepad++, Oxygen или TextMate.

Большинство из указанных реализаций регулярных выражений в чем-то сходны, а в чем-то различаются. Я не могу подробно обсудить все отличия в столь маленькой книге, но о многих расскажу. Любые попытки задокументировать *все* различия между *всеми* реализациями наверняка привели бы меня в больницу. Поэтому углубляться во все подробности такого рода я не буду. Данная книга предназначена лишь для ознакомления читателя с регулярными выражениями, и эта цель будет достигнута.

Для кого предназначена эта книга

Предполагаемые читатели книги — люди, которые за всю свою жизнь не написали еще ни одного регулярного выражения. Если вы новичок в этой области или в программировании вообще, то книга будет для вас хорошим вводным курсом. Другими словами, она предназначена для тех читателей, которые слышали кое-что о регулярных выражениях и заинтересовались ими, но пока еще не до конца понимают, что они собой представляют. Если вы относитесь к данной категории, то эта книга как раз для вас.

В целом, рассматривая свойства регулярных выражений, я буду придерживаться принципа "от простого к сложному". Иными словами, новые сведения будут преподноситься постепенно, небольшими порциями.

Если вы достаточно хорошо знакомы с регулярными выражениями и уверенно ими пользуетесь, вам лучше обратиться к другим книгам. Эта книга предназначена для новичков, которые нуждаются в том, чтобы их буквально вели за руку. Если вам уже приходилось сталкиваться с регулярными выражениями, но ваша практика работы с ними довольно ограниченна, то эта книга будет для вас полезной. Однако темпы рассмотрения материала могут показаться вам не столь быстрыми, как хотелось бы.

Могу порекомендовать несколько учебных пособий, которые имеет смысл изучить после прочтения данной книги. В книге Джеффри Фридла *Регулярные выражения, 3-е издание* (Символ-Плюс, 2008 г.) регулярные выражения рассмотрены гораздо более подробно. Кроме того, можете прочитать книгу Яна Гойвертса и Стивена Левитана *Регулярные*

выражения. Сборник рецептов, 2-е издание (Символ-Плюс, 2015 г.). Ян Гойвертс — создатель RegexBuddy, мощного настольного приложения для работы с регулярными выражениями (`http://www.regexbuddy.com/`), тогда как Стивен Левитан создал RegexPal, онлайновый процессор регулярных выражений (`http://www.regexpal.com`), который будет использован в первой главе.

Что необходимо для работы с книгой

Чтобы чтение книги принесло вам максимальную пользу, в вашем распоряжении должны быть некоторые средства Unix (Linux), доступ к которым на компьютерах Mac обеспечивается установкой операционной системы Darwin (разновидность BSD), а на компьютерах Windows — установкой приложения Cygwin, в дистрибутив которого включены многие инструменты GNU (см. `http://www.cygwin.com` и `http://www.gnu.org`).

Книга изобилует многочисленными примерами. Можете просто просматривать их, но, для того чтобы действительно чему-то научиться, лучше самостоятельно выполнить как можно больше примеров, поскольку, по моему глубокому убеждению, важнейший фактор обучения — закрепление получаемых знаний на практике. Я представлю вам веб-приложения, упрощающие освоение регулярных выражений за счет цветового выделения найденных совпадений, а также познакомлю с инструментальными "рабочими лошадками" из мира Unix и настольными приложениями для анализа регулярных выражений и их использования в контекстном поиске.

Примеры из книги можно найти на сайте Github по следующему адресу:

```
https://github.com/michaeljamesfitzgerald/↪
        Introducing-Regular-Expressions
```

Кроме того, полный архив всех примеров и тестовых файлов, используемых в книге, доступен для загрузки по следующим адресам:

```
http://examples.oreilly.com/0636920012337/examples.zip
http://www.williamspublishing.com/Books/978-5-8459-1953-3.html
```

Будет лучше, если вы сначала создадите рабочий каталог или папку на своем компьютере и загрузите туда эти файлы, а затем уже приступите к чтению книги.

Соглашения, принятые в книге

В книге использованы следующие типографские соглашения.

- *Курсивом* выделяются новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов и т.п.
- `Моноширинный шрифт` используется в листингах программ, а также в основном тексте для представления регулярных выражений, содержимого командной строки и других подобных элементов.

- Этой пиктограммой обозначены разделы текста, содержащие советы, рекомендации, а также замечания общего характера.

Использование кода примеров

Эта книга была написана для того, чтобы облегчить вам работу. Вообще говоря, вы можете свободно использовать приведенный в книге код в своих программах и документации. Получения какого-либо специального разрешения от нас, если только речь не идет о значительных объемах кода, не требуется. Например, использование в вашей программе нескольких фрагментов кода, взятых из книги, не требует разрешения. Однако продажа или распространение компакт-диска, содержащего примеры из книг, выпущенных издательством O'Reilly, без предварительного получения разрешения запрещена. Цитирование данной книги и использование кода примеров в ответах на вопросы не требует разрешения. Вместе с тем, если вы включаете в документацию своего продукта значительные объемы кода из приведенных в книге примеров, то получение соответствующего разрешения является обязательным условием.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152

ГЛАВА 1

Что такое регулярные выражения

Регулярные выражения — это специальные текстовые строки, которые используются в качестве шаблонов (образцов) для сопоставления с наборами других строк. Они были предложены в качестве способа описания регулярных языков еще в 1940-х годах, однако начало их широкого применения в программировании относится к 1970-м годам. Я впервые узнал о них из руководства к текстовому редактору QED, написанному Кеном Томпсоном:

> "Регулярное выражение — это шаблон, определяющий набор символьных строк; о таком шаблоне говорят, что он соответствует определенным строкам".

Вскоре регулярные выражения были встроены в целый ряд инструментальных средств, которые первоначально входили в состав операционной системы Unix, но впоследствии обрели самостоятельное существование. В частности, к их числу относятся текстовые редакторы *ed*, *sed* и *vi* (*vim*), а также утилита *grep* и интерпретируемый C-подобный язык *AWK*. В то же время способы реализации регулярных выражений в каждом конеретном случае имели свои особенности.

В этой книге я придерживаюсь индуктивного подхода, т.е. мы будем переходить от частного к общему. Поэтому чаще всего будут приводиться сначала примеры и лишь затем — соответствующие теоретические выкладки.

Регулярные выражения имеют репутацию трудной темы, однако все зависит от того, как приступить к их изучению. Существуют, например, совсем простые регулярные выражения:

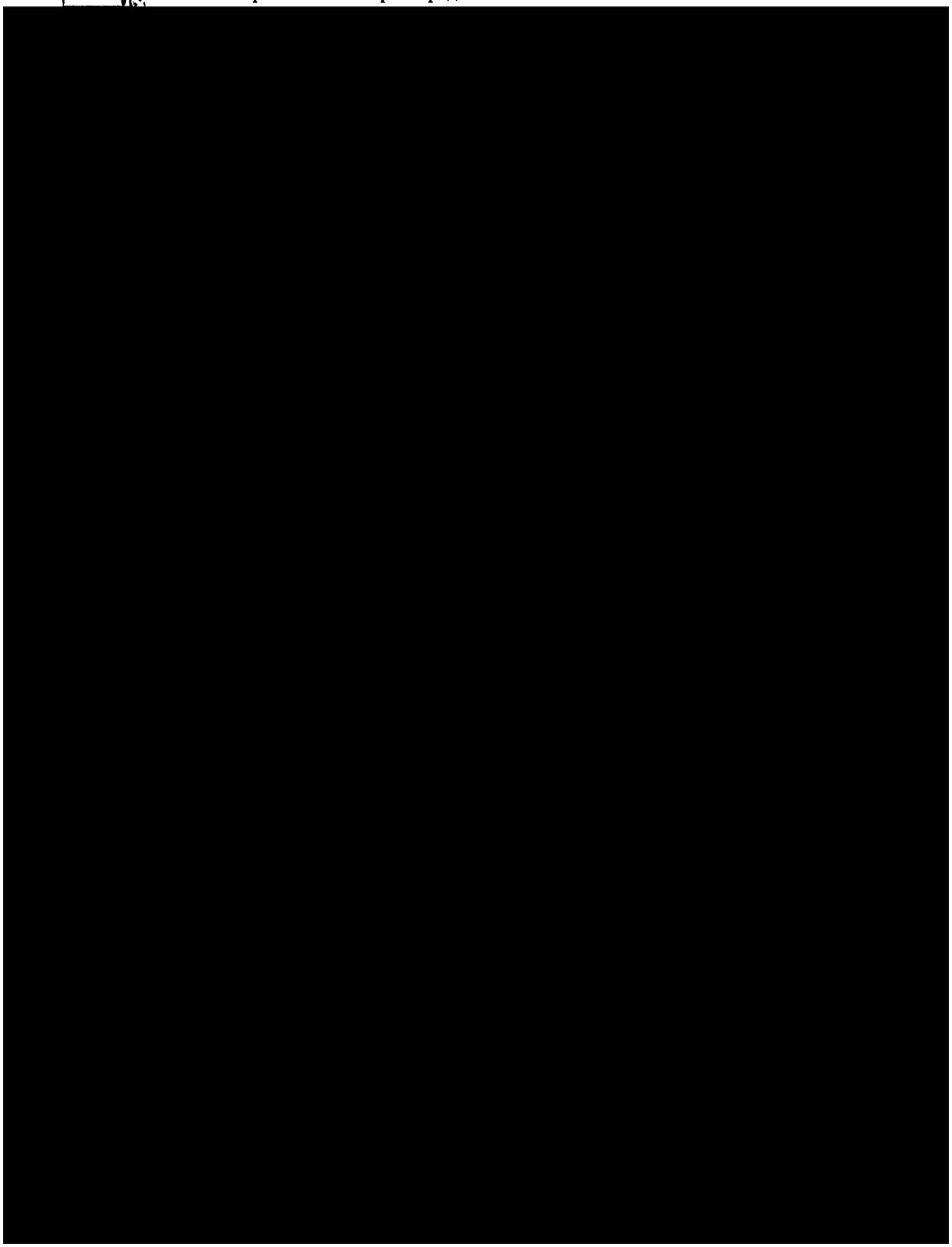
```
\d
```

где *метасимволу* `\d` соответствует любая одиночная цифра от 0 до 9. Постепенно усложняя это регулярное выражение, мы придем в конце главы к выражению следующего вида, которое можно использовать в качестве надежного шаблона для поиска телефонных номеров в формате записи, принятом в США и Канаде:

```
^(\(\d{3}\)|^\d{3}[.-]?)?\d{3}[.-]?\d{4}$
```

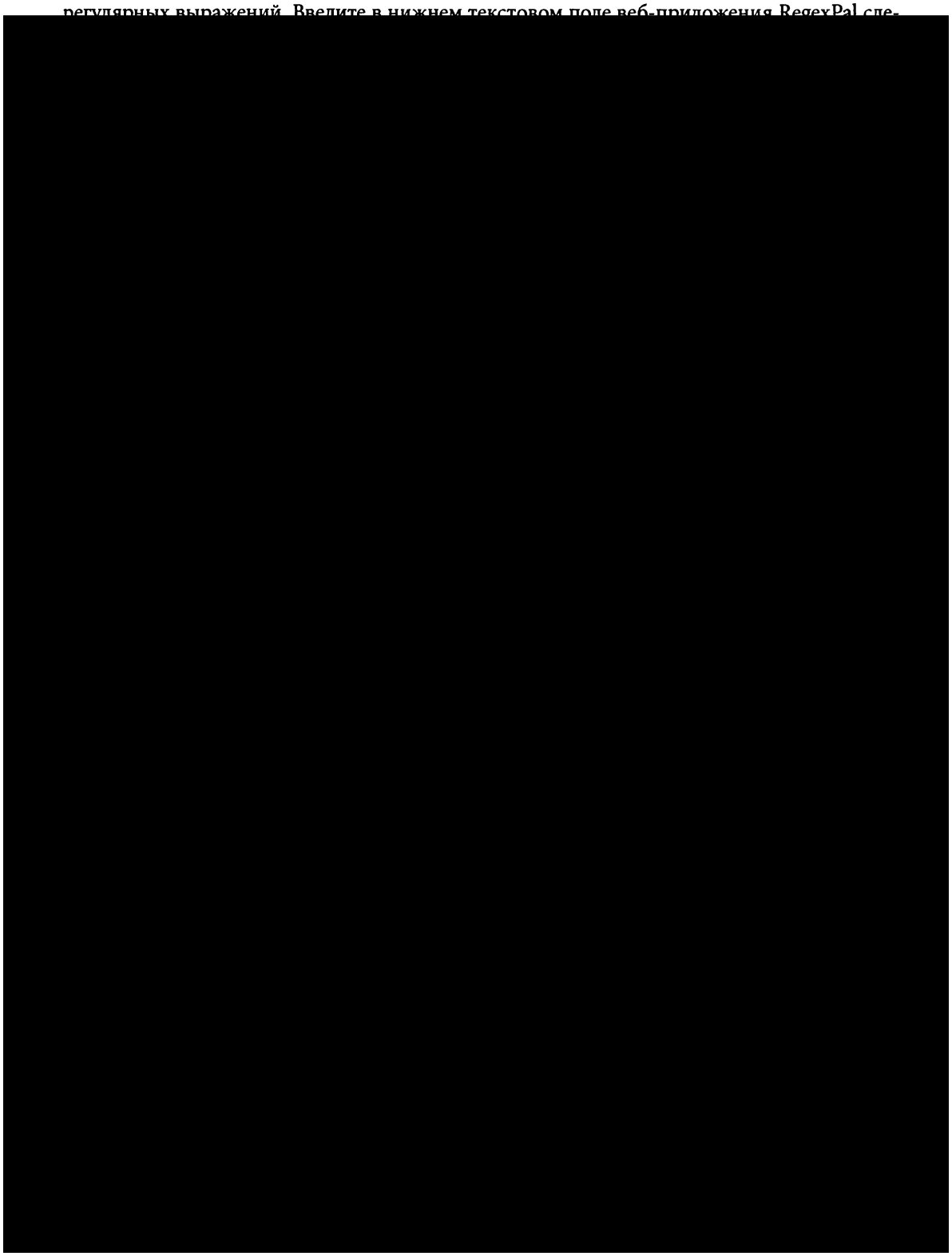
Этому формату соответствуют номера, состоящие из 10 цифр, причем разрешается заключать территориальный код в круглые скобки и использовать дефисы или точки для разделения отдельных групп цифр. (Круглые скобки должны быть обязательно парными; использование одиночных круглых скобок не допускается.)

В главе 10 вы ознакомитесь с более сложным регулярным выражением для поиска телефонных номеров, но для целей текущей главы нам вполне хватит и того выражения, которое представлено выше.



Соответствие телефонному номеру

Мы приступаем к нахождению соответствий шаблону телефонного номера с помощью регулярных выражений. Введите в нижнем текстовом поле веб-приложения RegexPal сле-



В данном случае для поиска подстрок, соответствующих шаблону, мы использовали регулярное выражение в виде *строкового литерала*. Строковый литерал — это буквальное представление строки.

А теперь удалите номер в верхнем окне и введите вместо него одну только цифру `7`. Заметили изменения? Подсвеченными оказались только цифры `7`. Литеральному символу (числу) `7` в регулярном выражении соответствует любое из четырех вхождений цифры `7` в целевом тексте.

Задание соответствия цифрам с помощью символьных классов

Что если вы хотите найти в телефонном номере все цифры за один раз? Иными словами, как задать соответствие каждой из цифр номера, не прибегая к последовательному перебору конкретных значений?

Вновь введите в верхнем поле следующую строку в том виде, как она приведена:

```
[0-9]
```

В результате все числа (точнее, *цифры*) в нижнем поле выделятся попеременно желтым и голубым цветом. Выражение `[0-9]` означает для процессора регулярных выражений следующее: "Найти совпадение с любой цифрой, принадлежащей к диапазону `0-9`".

Квадратные скобки не участвуют в сопоставлении литералов регулярного выражения с целевыми строками, поскольку обрабатываются особым образом как *метасимволы*. Метасимволы имеют специальный смысл в регулярных выражениях и представляют зарезервированные значения. Последовательность символов, заключенная в квадратные скобки, называется *символьным классом* (или *символьным набором*). Символьному классу соответствует любой одиночный символ из числа тех, которые заключены в квадратные скобки.

Можно более точно ограничить набор допустимых цифр, указав их конкретный список:

```
[012789]
```

Введите это регулярное выражение в верхнем текстовом поле. Ему будут соответствовать не любые цифры, а только те, которые входят в список, т.е. `0`, `1`, `2`, `7`, `8` и `9`. И вновь в нижнем поле чередующимися цветами будут выделены все цифры номера.

Для представления любого десятизначного телефонного номера, отдельные группы цифр которого разделены дефисами, можно было бы использовать следующее выражение:

```
[0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
```

Это регулярное выражение вполне функционально, но чересчур громоздко. Гораздо более элегантное решение данной задачи обеспечивают символьные сокращения.

Использование символьных сокращений

Еще один способ представления произвольной одиночной цифры, с которым вы уже успели ознакомиться в начале главы, заключается в использовании метасимвола `\d`, которому, как и регулярному выражению `[0-9]`, соответствует любая арабская цифра. Регулярные выражения подобного типа, обеспечивающие компактную форму записи целых классов символов, называют *символьными сокращениями* или просто *сокращениями*. (Для них существует также другое название — *символьные маски*, но, поскольку последний

термин может вносить некоторую путаницу, я избегаю его использования. Более подробные объяснения будут даны немного позже.)

В качестве регулярного выражения, совпадающего с любым телефонным номером, представленным в описанном ранее формате, можно было бы использовать следующее выражение:

```
\d\d\d-\d\d\d-\d\d\d\d
```

Повторение последовательности символов `\d` три или четыре раза подряд в точности соответствует группам, состоящим из трех или четырех следующих подряд цифр. Дефис используется здесь как литеральный символ, участвующий в сопоставлении шаблона с целевым текстом.

Включение дефисов в регулярное выражение — не единственный способ сопоставления с дефисами в телефонном номере. Для этой цели можно также использовать экранированный символ `D` в верхнем регистре (`\D`), которому соответствует любой символ, *не* являющийся цифрой.

Вот как можно использовать сокращение `\D` вместо литерального дефиса:

```
\d\d\d\D\d\d\d\D\d\d\d\d
```

Можете проверить, что и в этом случае в нижнем текстовом поле выделится весь телефонный номер, включая дефисы.

Соответствие произвольному одиночному символу

Для представления дефисов в регулярном выражении можно было бы также использовать символ точки (`.`):

```
\d\d\d.\d\d\d.\d\d\d\d
```

Здесь точка, по существу, играет роль группового символа, совпадающего с любым одиночным символом (за исключением символа конца строки при определенных условиях). Этому регулярному выражению будут соответствовать телефонные номера, в которых в качестве разделителя может использоваться не только дефис, но и символ процента (`%`):

```
707%827%7019
```

символ вертикальной черты (`|`):

```
707|827|7019
```

и вообще любой другой символ.

Как уже отмечалось, символ точки не всегда будет совпадать с символом новой строки, таким, например, как символ перевода строки (U+000A). И все же существуют способы добиться того, чтобы точке соответствовал также и символ новой строки, о чем будет говориться далее.

Захватывающие группы и обратные ссылки

Сейчас мы составим регулярное выражение, которое совпадает с начальной частью телефонного номера из нашего примера, используя для этой цели *захватывающую группу*, а затем сошлемся на содержимое этой группы с помощью *обратной ссылки*. Создадим

захватывающую группу, заключив сокращение \d в круглые скобки, а для ссылки на содержимое, захваченное этой группой, используем переменную \1:

```
(\d)\d\1
```

Переменная \1 ссылается на содержимое, захваченное (сохраненное) группой. Таким образом, это регулярное выражение обеспечивает совпадение с префиксом 707 телефонного номера. Проанализируем отдельные составляющие этого выражения.

- Группа (\d) совпадает с первой цифрой номера (7) и запоминает ее.
- Сокращение \d совпадает со следующей цифрой номера (0), но не запоминает ее, поскольку не заключено в скобки.
- Переменная \1 ссылается на захваченную цифру (7).

Данное выражение совпадает только с территориальным кодом. Не огорчайтесь, если вам пока что не все до конца понятно. В этой книге вы еще не раз столкнетесь с примерами использования групп.

Теперь мы можем записать регулярное выражение для сопоставления с полным телефонным номером, используя одну группу и несколько раз ссылаясь на ее содержимое:

```
(\d)0\1\D\d\d\1\D\1\d\d\d
```

И все же это выражение еще не выглядит достаточно элегантным. Попытаемся улучшить его.

Использование квантификаторов

Ниже представлен еще один вариант регулярного выражения для сопоставления с телефонным номером, записанный с использованием другого синтаксиса:

```
\d{3}-?\d{3}-?\d{4}
```

Числа в фигурных скобках сообщают процессору регулярных выражений *точное* количество вхождений цифр, которое требуется для совпадения. Фигурные скобки с заключенными в них числами — это одна из разновидностей *квантификаторов*. Сами по себе фигурные скобки считаются метасимволами.

Другой пример квантификатора — вопросительный знак (?), который соответствует повторению предыдущего символа нуль или один раз. В приведенном выше регулярном выражении он следует за символом дефиса, указывая на то, что дефис может отсутствовать, а если он имеется, то не может встречаться более одного раза подряд. Существуют и другие квантификаторы, такие как символ "плюс" (+), соответствующий повторению предыдущего символа один или более раз, и символ "звездочка" (*), соответствующий повторению предыдущего символа нуль или более раз.

С использованием квантификаторов последнее регулярное выражение можно переписать в более компактной форме:

```
(\d{3,4}[.-]?)+
```

"Плюс" означает, что данная величина может встречаться один или несколько раз. Этому регулярному выражению будут соответствовать три или четыре цифры, за которыми следует необязательный дефис или точка, причем вся эта группа, заключенная в круглые скобки, может встречаться один или несколько раз (+).

У вас еще не кружится голова? Надеюсь, нет. Вот посимвольный анализ этого регулярного выражения:

- `(` — открывает захватывающую группу;
- `\` — начальный символ сокращения (экранирует следующий символ);
- `d` — конечный символ сокращения (представляет произвольную одиночную цифру в диапазоне от 0 до 9);
- `{` — открывает квантификатор;
- `3` — минимально допустимое количество вхождений;
- `,` — разделитель;
- `4` — максимально допустимое количество вхождений;
- `}` — закрывает квантификатор;
- `[` — открывает символьный класс;
- `.` — точка (соответствует литеральной точке);
- `-` — дефис (соответствует литеральному дефису);
- `]` — закрывает символьный класс;
- `?` — квантификатор, которому соответствует отсутствие или одиночное вхождение предыдущего символа;
- `)` — закрывает захватывающую группу;
- `+` — квантификатор, которому соответствует одно или несколько вхождений предыдущего символа.

Несмотря на то что это выражение работает, оно не совсем правильное, поскольку ему соответствуют любые группы из 3 или 4 цифр, а не только те, которые вписываются в формат телефонного номера. Однако ошибки учат нас лучше успеха.

Поэтому давайте чуть подправим предыдущее выражение:

```
(\d{3}[.-]?){2}\d{4}
```

Этому регулярному выражению соответствуют две группы по три цифры каждая, причем за любой из этих групп может следовать необязательный дефис, а вся последовательность завершается ровно четырьмя цифрами.

Использование литеральных круглых скобок

Наконец, рассмотрим регулярное выражение, которое допускает, чтобы первая последовательность из трех цифр была заключена в круглые скобки, а кроме того, делает территориальный код необязательным:

```
^(\(\d{3}\)|^\d{3}[.-]?)?\d{3}[.-]?\d{4}$
```

Чтобы убедить вас в том, что интерпретация этого выражения не вызывает трудностей, мы также выполним его посимвольный анализ[1]:

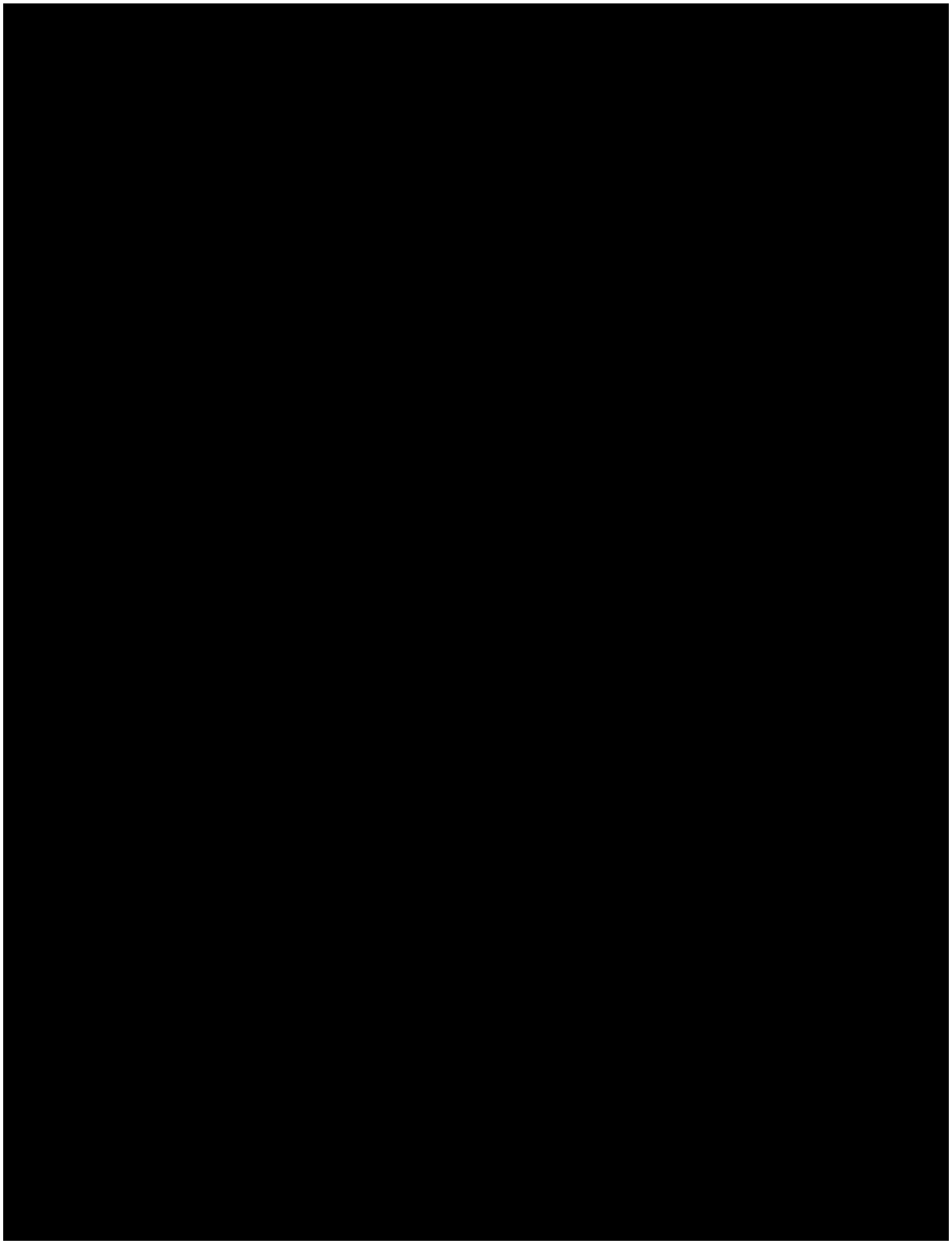
[1] С дискуссией по поводу этого выражения и замечаниями к нему можно ознакомиться по следующему адресу: `http://www.oreilly.com/catalog/errata.csp?isbn=9781449392680`. — *Примеч. ред.*

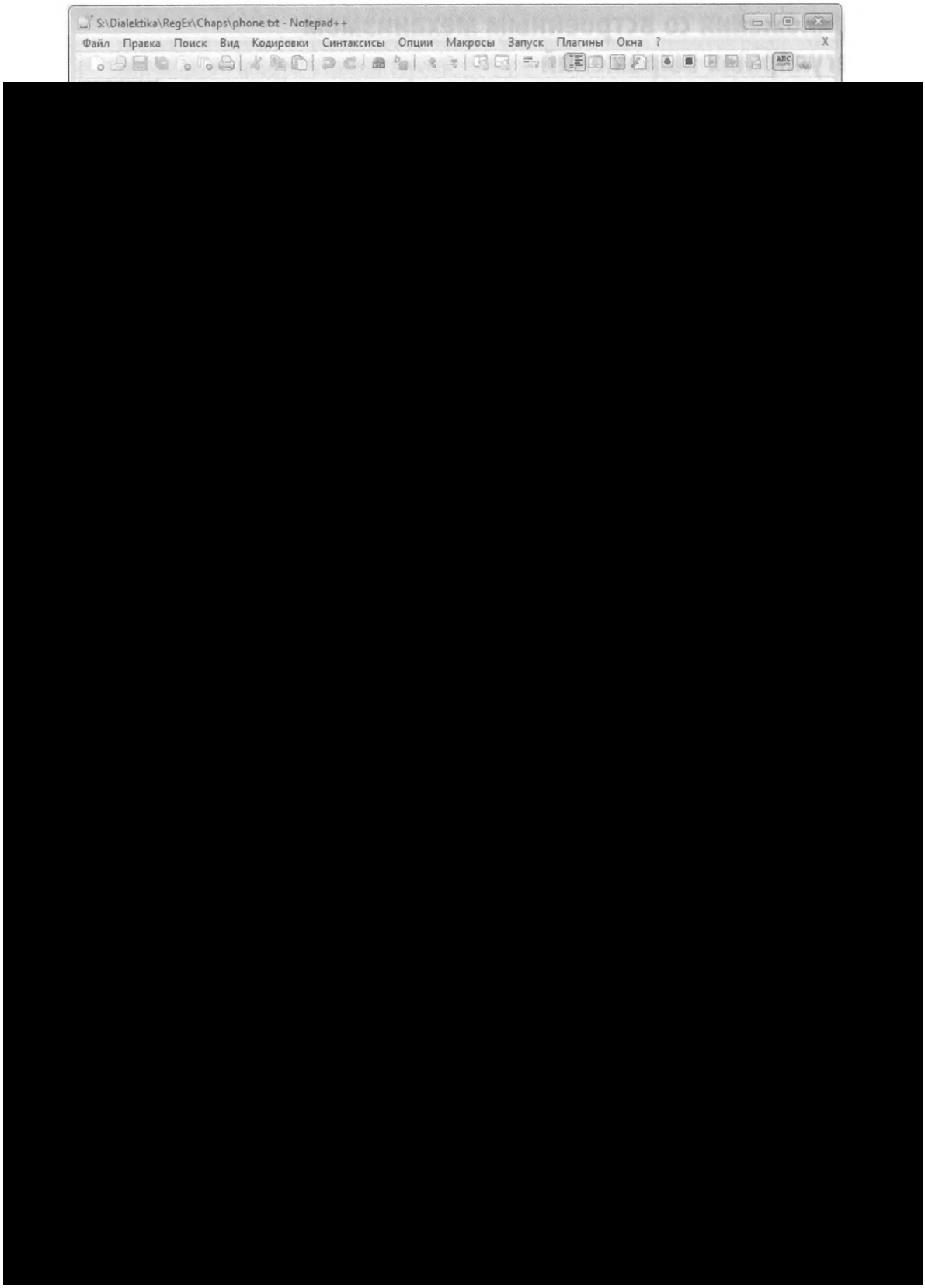
- `^` (символ перевода каретки) — в начале регулярного выражения или после вертикальной черты (`|`) указывает на то, что условием поиска является расположение телефонного номера в начале строки;
- `(` — открывает захватывающую группу;
- `\(` — литеральная открывающая круглая скобка;
- `\d` — представляет произвольную одиночную цифру;
- `{3}` — квантификатор, которому соответствуют ровно три следующие подряд цифры (поскольку он стоит после `\d`);
- `\)` — литеральная закрывающая круглая скобка;
- `|` (вертикальная черта) — разделяет альтернативные варианты. В данном случае она указывает на то, что территориальный код может быть задан как в скобках, так и без скобок;
- `^` — представляет начало строки;
- `\d` — представляет произвольную одиночную цифру;
- `{3}` — квантификатор, которому соответствуют ровно три следующие подряд цифры;
- `[.-]?` — представляет необязательную точку или дефис;
- `)` закрывает захватывающую группу;
- `?` — делает предшествующую группу необязательной, т.е. наличие префикса, задаваемого группой, не является обязательным условием;
- `\d` — представляет произвольную одиночную цифру;
- `{3}` — квантификатор, которому соответствуют ровно три следующие подряд цифры;
- `[.-]?` — представляет еще один необязательный элемент в виде точки или дефиса;
- `\d` — представляет произвольную одиночную цифру;
- `{4}` — квантификатор, которому соответствуют ровно три следующие подряд цифры;
- `$` — соответствие концу строки.

Этому окончательному регулярному выражению соответствует принятый в США и Канаде формат десятизначного телефонного номера, допускающий использование круглых скобок, дефисов и точек. Опробуйте это регулярное выражение на телефонных номерах, записанных с использованием различных форматов, чтобы увидеть, какие из них будут (или не будут) соответствовать данному шаблону.

Включение захватывающей группы в приведенное выше регулярное выражение не является необходимым. Вернее, необходима сама группа, но не ее запоминающий функционал. В подобных случаях лучше использовать незахватывающие группы. Почему это так, вы поймете, когда мы будем пересматривать это выражение в последней главе.

Приложения со встроенным механизмом регулярных выражений




S:\Dialektika\RegEx\Chaps\phone.txt - Notepad++
Файл Правка Поиск Вид Кодировки Синтаксисы Опции Макросы Запуск Плагины Окна ?
X

О чем вы узнали в главе 1

- Что такое регулярные выражения.
- Как использовать RegexPal — простой процессор регулярных выражений.
- Как задать соответствие строковым литералам.
- Как задать соответствие цифрам с помощью символьного класса.
- Как задать соответствие цифрам с помощью символьного сокращения.
- Как задать соответствие нецифровому символу с помощью символьного сокращения.
- Как используются захватывающие группы и обратные ссылки.
- Как задать соответствие точному количеству повторений символа.
- Как задать соответствие необязательному символу (может быть повторен нуль или один раз) или символу, который должен быть повторен один или несколько раз.
- Как задать соответствие тексту, находящемуся в начале или в конце строки.

На заметку

- RegexPal (`http://www.regexpal.com`) — веб-приложение, реализующее механизм регулярных выражений JavaScript. Данная реализация не является наиболее полной и поддерживает не все возможности регулярных выражений. В то же время это приложение предоставляет множество возможностей для начинающих, что делает его весьма простым и удобным в работе инструментом, пригодным для использования в учебных целях.
- Для работы с примерами книги можно использовать браузер Google Chrome (`https://www.google.com/chrome`) или Mozilla Firefox (`http://www.mozilla.org/en-US/firefox/new/`).
- Благодаря чему регулярные выражения способны обеспечивать многовариантные решения? Одним из факторов, способствующих этому, служит то, что регулярные выражения обладают замечательным свойством — *компонуемостью* (composability). (Подробное объяснение этого термина, данное Джеймсом Кларком, можно найти по адресу `http://www.thaiopensource.com/relaxng/design.html#section:5`.) Когда о языке, будь то формальный язык, язык программирования или язык схем, говорят, что он обладает свойством компонуемости, то под этим подразумевается, что можно не только использовать его отдельные атомарные части, но и комбинировать их всевозможными способами с помощью методов, предоставляемых самим языком. Как только вы научитесь работать с отдельными составными частями регулярных выражений, вы сможете находить соответствия шаблонам любой степени сложности.
- Приложение TextMate доступно по адресу `http://www.macromates.com`. Для получения более подробной информации относительно использования регулярных выражений в TextMate посетите сайт `http://manual.macromates.com/en/regular_expressions`.

- Для получения более подробной информации относительно приложения Notepad++ посетите сайт `http://notepad-plus-plus.org`. С документацией, содержащей описание порядка использования регулярных выражений в приложении Notepad++, можно ознакомиться по адресу `http://sourceforge.net/apps/mediawiki/notepad-plus/index.php?title=Regular_Expressions`
- О приложении Oxygen можно прочитать на сайте `http://www.oxygenxml.com`. Информация относительно использования регулярных выражений посредством диалогового окна поиска и замены доступна по адресу `http://www.oxygenxml.com/doc/ug-editor/topics/find-replace-dialog.html`. Информация относительно средства создания регулярных выражений XML Schema доступна по адресу `http://www.oxygenxml.com/doc/ug-ditor/topics/XML-schema-regexp-builder.html`.

ГЛАВА 2

Сопоставление с простыми шаблонами

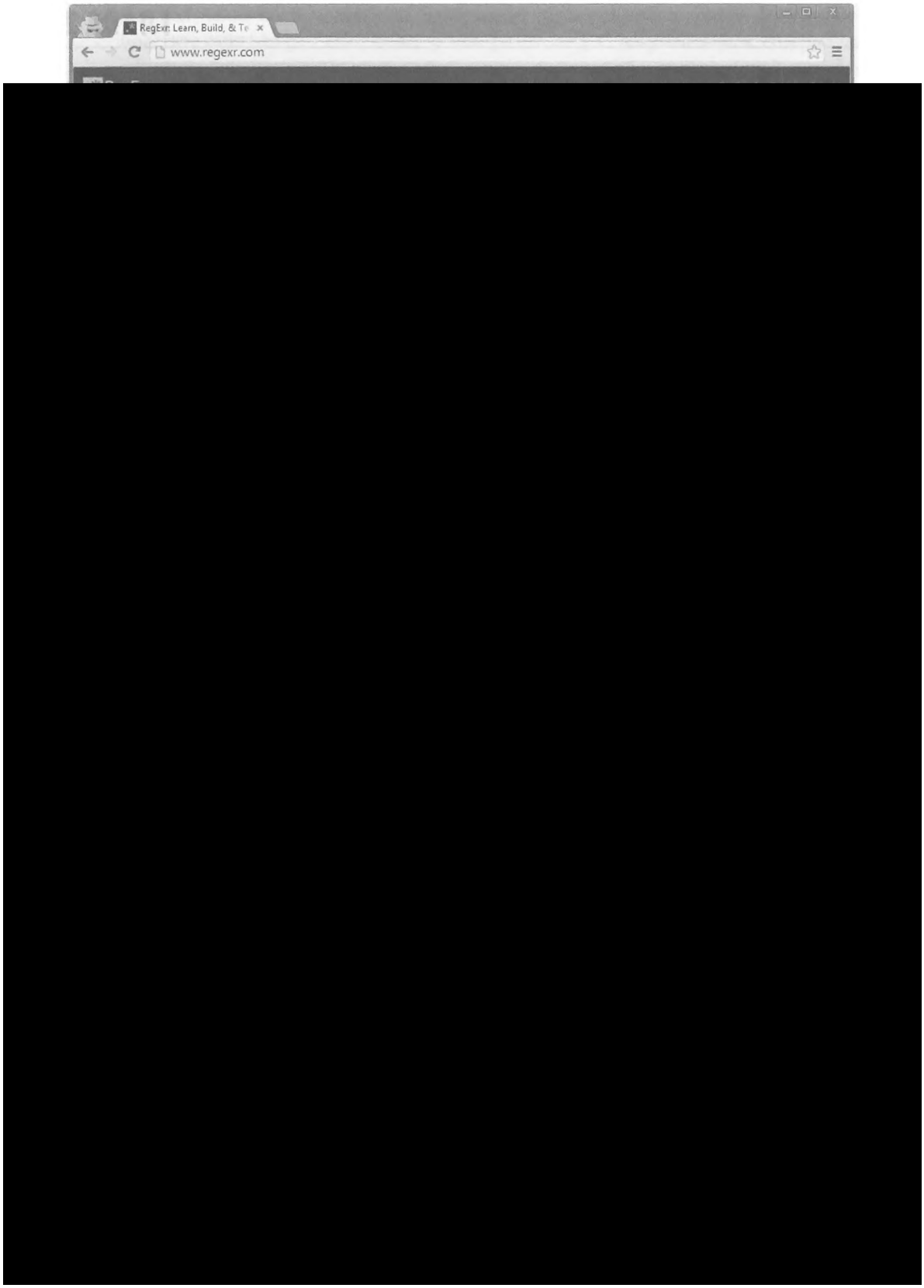
Работа механизма регулярных выражений основана на сопоставлении текста с некоторым шаблоном, нахождении фрагментов, соответствующих этому шаблону, и выполнении над ними определенных действий. В этой главе рассматриваются простейшие способы поиска совпадений текста с шаблоном с помощью следующих элементов:

- строковые литералы;
- цифры;
- буквы;
- произвольные символы.

В первой главе для демонстрации работы регулярных выражений использовалось приложение RegexPal, написанное Стивеном Левитаном. В этой главе мы воспользуемся сайтом RegExr Гранта Скиннера (`www.regexr.com`), как показано на рис. 2.1.

Каждая страница данной книги будет уводить вас все дальше в "дебри" регулярных выражений. Однако вы в любой момент можете сделать кратковременный привал, чтобы разобраться с синтаксисом. Я хочу сказать, что вы должны пытаться опробовать новые возможности сразу же после того, как прочитали о них. Не бойтесь экспериментировать. Ошибайтесь. Старайтесь докопаться до сути и лишь затем вновь устремляйтесь вперед. Не существует никакого другого способа научиться загонять шар прямо в лузу, кроме как *пытаться это сделать*.

Прежде чем мы продолжим, хочу сказать несколько слов о том, какого рода помощь вам может оказать RegExr. Обратите внимание на ссылки Examples (Примеры) и Community (Сообщество) в левой части окна. Первая из них приведет вас к примерам, иллюстрирующим синтаксис регулярных выражений, а вторая — к множеству регулярных выражений, предложенных членами сообщества и снабженных рейтинговыми оценками. Используя эти ссылки, вы найдете массу полезной для себя информации. Дополнительные сведения можно получить, наведя указатель мыши на регулярное выражение или текст в окне RegExr. Именно благодаря возможности использования этих вспомогательных ресурсов RegExr стал одним из моих излюбленных онлайновых тестеров регулярных выражений.


RegExr: Learn, Build, & Te
www.regexr.com

```
and in what manner the Ancyent Marinere came back to his
own Country.

I.

1      It is an ancyent Marinere,
2       And he stoppeth one of three:
3      "By thy long grey beard and thy glittering eye
4       "Now wherefore stoppest me?
```

Скопируйте данный текст и вставьте его в нижнее текстовое поле в окне приложения RegExr. Этот же текст содержится в файле *rime-intro.txt*, который можно найти на сайте Github по следующему адресу:

```
https://github.com/michaeljamesfitzgerald/↳
Introducing-Regular-Expressions
```

Точно такой же файл *rime-intro.txt*, как и файл *rime.txt*, находится в архиве примеров по следующему адресу:

```
http://examples.oreilly.com/9781449392680/examples.zip
```

Соответствие литеральному тексту

Наиболее очевидный и тривиальный способ применения регулярных выражений — это сопоставление строк с одним или более литеральными символами, называемыми *строковыми литералами*, или просто *литералами*.

Поиск соответствий строковым литералам выполняется с использованием обычных литеральных символов. Чувствуется что-то знакомое, не так ли? Это напоминает поиск фрагментов текста в текстовых процессорах или отбор информации по ключевым словам в поисковых системах. Когда вы выполняете поиск текстовой строки путем простого посимвольного сравнения текста с шаблоном, вы фактически осуществляете поиск с помощью строкового литерала.

Например, чтобы найти соответствие слову (строке символов) *Ship*, которое находится в самом начале поэмы, достаточно ввести слово *Ship* в верхнем текстовом поле окна RegExr, и оно автоматически подсветится в тексте, находящемся в нижнем поле. (Вводя слово *Ship*, не забудьте, что оно должно начинаться с прописной буквы S.)

Ну что, получилось? Слово *Ship* должно было выделиться синим цветом. Если этого не произошло, проверьте правильность ввода.

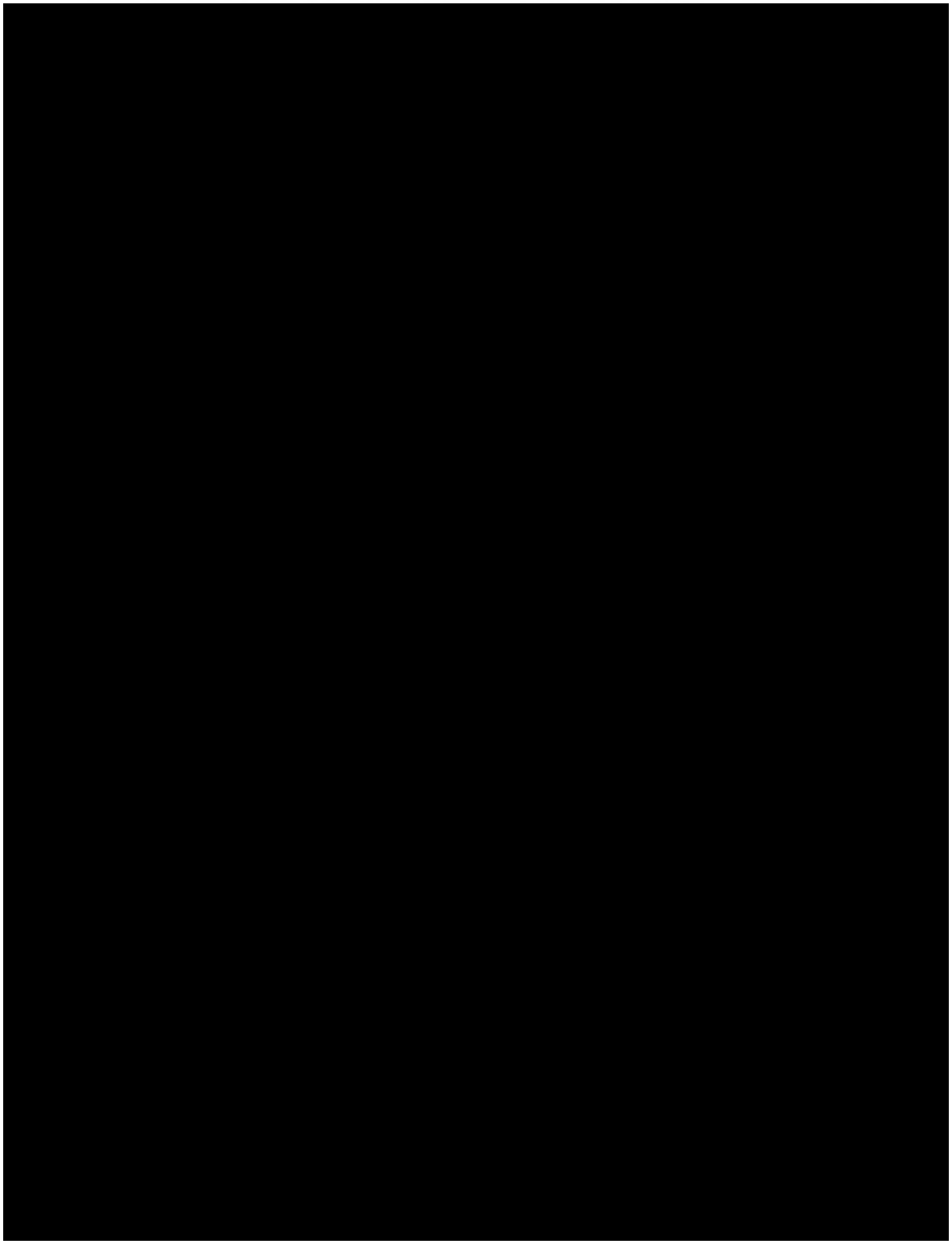
По умолчанию сопоставление строк в RegExr выполняется с учетом регистра символов. Если вы не хотите различать символы нижнего и верхнего регистра, установите флажок ignore case (игнорировать регистр) в верхней части окна. В результате шаблон найдет не только слово *Ship*, но и слово *ship*, если оно также будет присутствовать в целевом тексте.

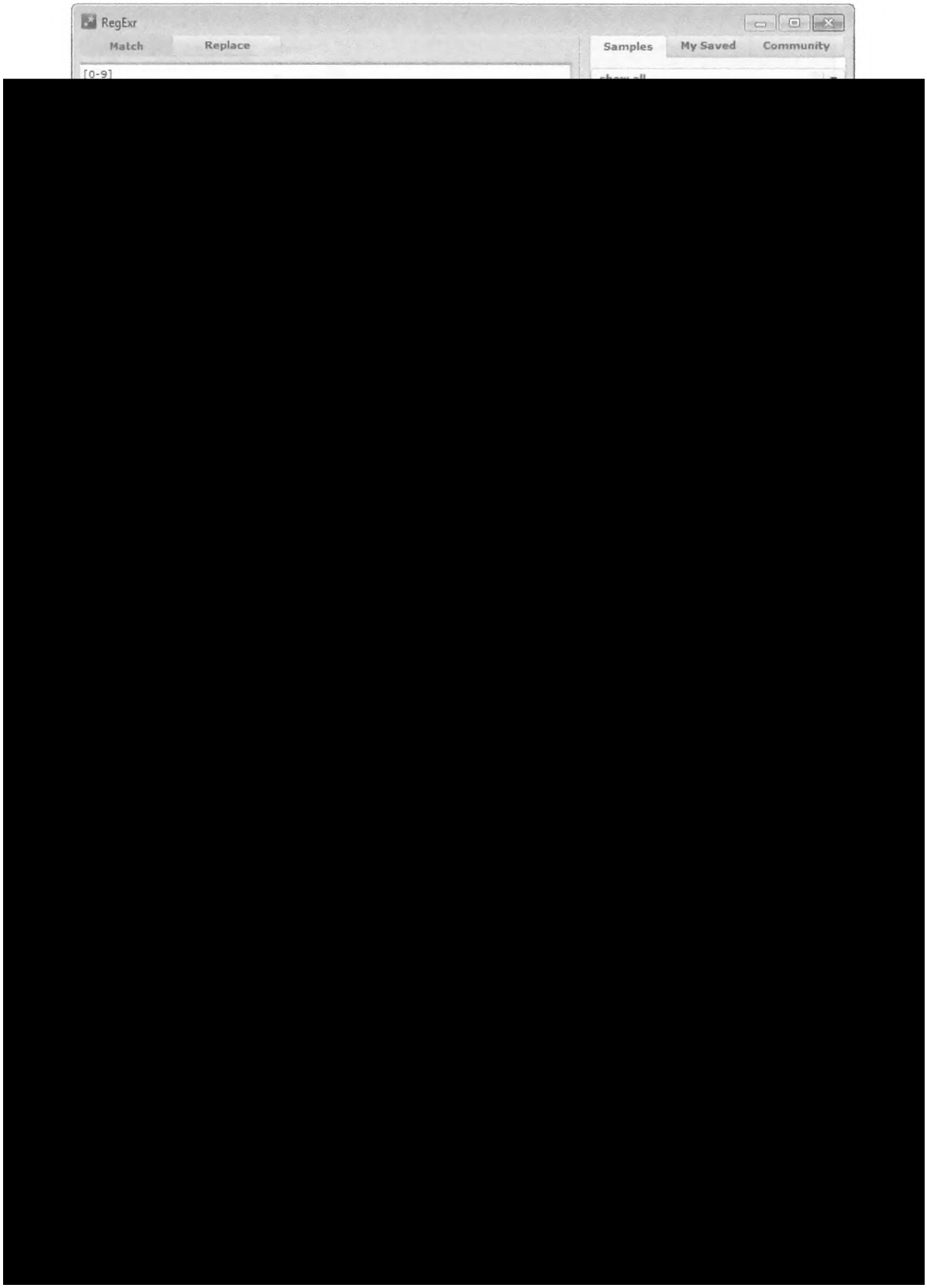
Соответствие цифрам

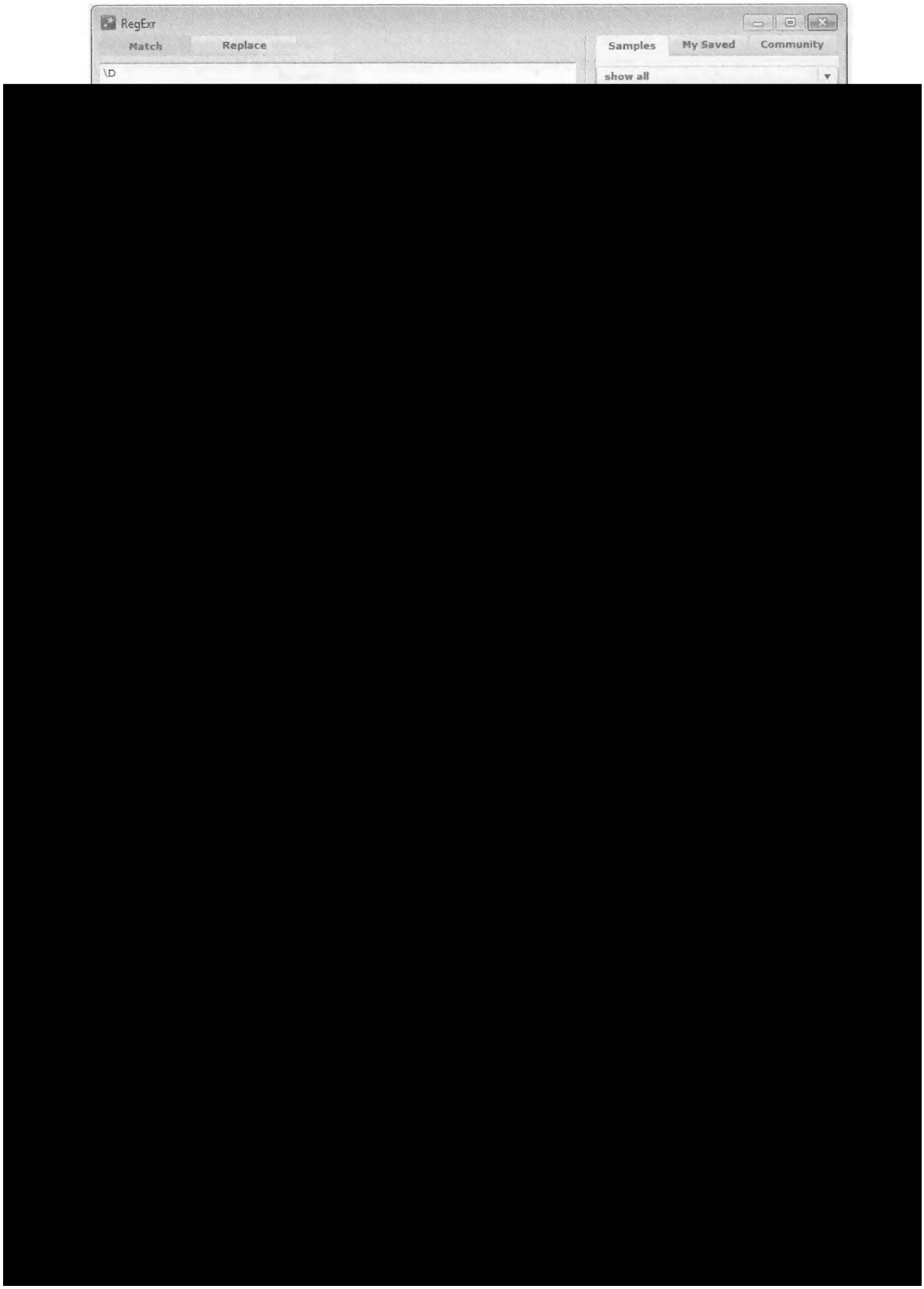
Для поиска соответствий цифрам введите в верхнем текстовом поле окна RegExr следующий метасимвол:

```
\d
```

Поскольку флажок global (глобальный режим) по умолчанию установлен, данному метасимволу будет соответствовать каждая из цифр, встречающихся в тексте (рис. 2.2). При снятом флажке global выражение `\d` найдет только первое вхождение цифрового символа.




RegExr
Match
Replace
Samples
My Saved
Community
[0-9]


RegExr
Match
Replace
\D
Samples
My Saved
Community
show all

Для поиска вхождений *несловарных* символов, т.е. тех, которые не относятся к категории словарных, используется буква `W` в верхнем регистре:

```
\W
```

Это символьное сокращение совпадает с пробельными символами, знаками пунктуации, а также символами других типов, которые не встречаются в данном примере. Приведенное выше сокращение эквивалентно следующему символьному классу:

```
[^_a-zA-Z0-9]
```

Символьные классы обеспечивают заведомо более точный контроль соответствий, но вполне может быть, что вы просто не захотите или не сочтете нужным вручную вводить длинный список символов. В подобных ситуациях срабатывает, как говорят, принцип "минимизации клавиатурного ввода" ("fewest keystrokes win"). В других же случаях для того, чтобы в точности обеспечить получение нужного результата, без указания списка конкретных символов не обойтись. Как именно следует поступить, вы решаете в зависимости от ситуации.

Для сравнения проверьте, как работают в RegExr следующие два символьных класса:

```
[^\w]
```

и

```
[^\W]
```

Уловили разницу?

В табл. 2.1 приведен обширный список метасимволов, используемых в качестве сокращенной формы записи символьных классов. Не каждое из приведенных сокращений работает во всех процессорах регулярных выражений.

Таблица 2.1. Символьные сокращения

Символьное сокращение	Описание
`\a`	Звуковой сигнал
`\b`	Граница слова
`[\b]`	Возврат на шаг (забой)
`\B`	Не граница слова
`\c`*x*	Управляющий символ
`\d`	Цифра
`\D`	Не цифра
`\d` *xxx*	Десятичное значение кода символа
`\f`	Перевод формата
`\r`	Возврат каретки
`\n`	Перевод строки
`\o` *xxx*	Восьмеричное значение кода символа
`\s`	Пробел (пустой символ)
`\S`	Не пробел (непустой символ)

Окончание табл. 2.1

Символьное сокращение	Описание
`\t`	Символ горизонтальной табуляции
`\v`	Символ вертикальной табуляции
`\w`	Словарный символ
`\W`	Не словарный символ
`\0`	Пустой символ (Null)
`\x` *xx*	Шестнадцатеричное значение кода символа
`\u` *xxxx*	Символ в кодировке Unicode

Соответствие пробелам

Для задания соответствия пробелам используется следующее символьное сокращение:

```
\s
```

Введите его в RegExr и посмотрите, какие символы окажутся выделенными в нижнем текстовом поле (рис. 2.5). Тот же результат можно получить с помощью следующего символьного класса:

```
[ \t\n\r]
```

Иными словами, этому классу соответствуют:

- пробелы;
- символы горизонтальной табуляции;
- символы перевода строки;
- символы возврата каретки.

В приложении RegExr выделяются цветом пробелы и символы табуляции, но не символы перевода строки и возврата каретки.

Как нетрудно догадаться, у символьного сокращения `\s` имеется "партнер". Для поиска соответствий непробельным символам используйте следующее сокращение:

```
\S
```

Ему соответствует все, что не является пробелом. Тот же результат можно получить с помощью такого класса:

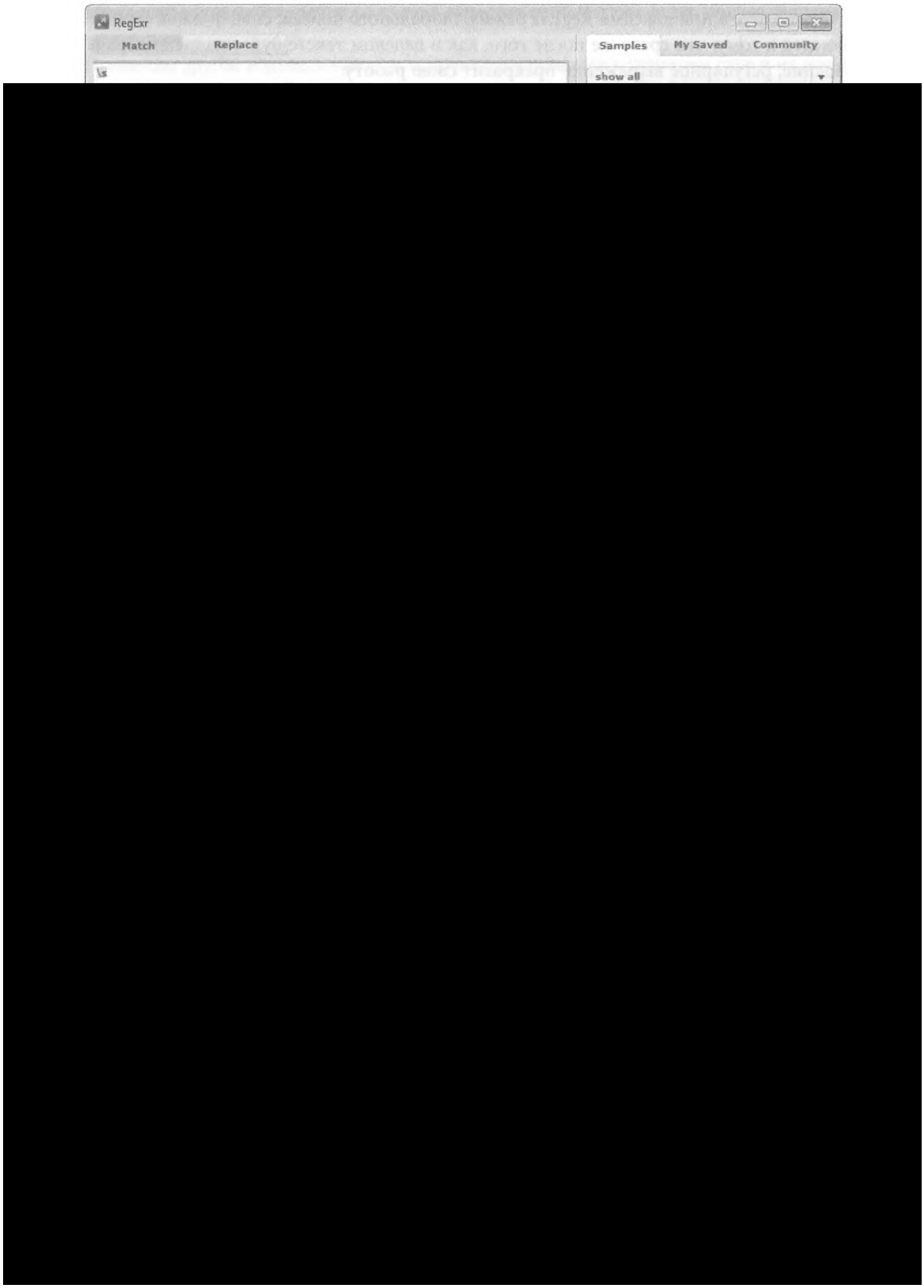
```
[^ \t\n\r]
```

или же такого:

```
[^\s]
```

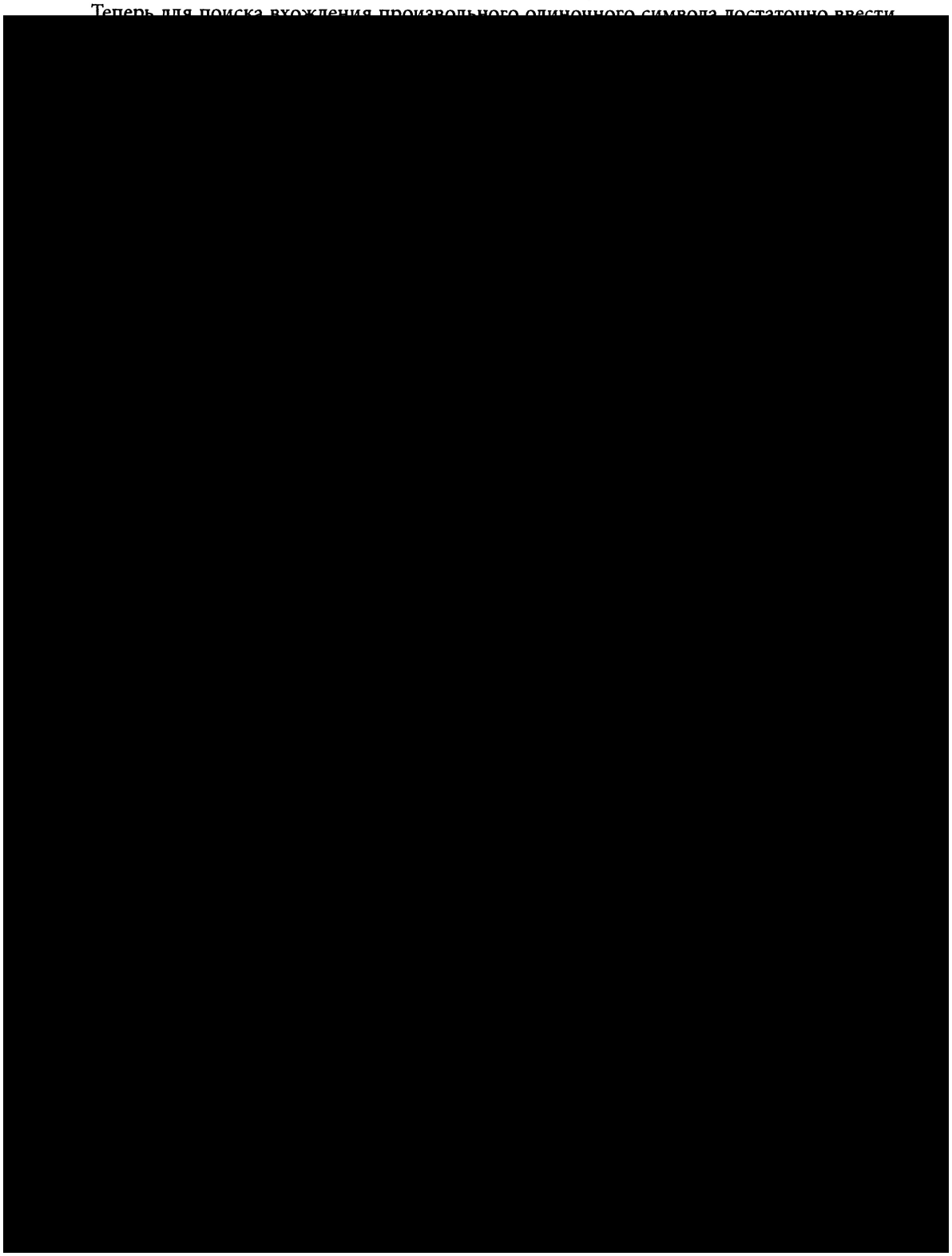
Апробируйте эти выражения в RegExr и проверьте, какие результаты при этом получаются.

Кроме тех символов, которые соответствуют сокращению `\s`, существуют другие, менее известные пробельные символы. Символьные сокращения для этих, а также других редко используемых символов приведены в табл. 2.2.


RegExr
Match
Replace
Samples
My Saved
Community
\s
show all

Отключите в приложении RegExr режим глобального поиска, сняв флажок global. Это приведет к тому, что сразу же после того, как в целевом тексте будет найдено первое совпадение, регулярное выражение прекратит свою работу.

Теперь для поиска вхождения произвольного одиночного символа достаточно ввести



восьми произвольных символов каждая, в которые не попадают лишь последние символы некоторых строк текста.

Применим другой подход, в котором используется привязка к границам слов и их начальным и последним буквам. Введите в верхнем текстовом поле окна RegExr следующий текст:

```
\bA.{5}T\b
```

Это выражение характеризуется чуть более высокой специфичностью. (Попытайтесь трижды произнести вслух слово *специфичность*.) Ему соответствует слово *ANCYENT* — устаревшая форма написания слова *ancient*. Проанализируем, почему так происходит.

- Сокращение `\b` находит границу слова, и ему не соответствует никакая буква.
- Символы `A` и `T` ограничивают последовательность символов.
- Последовательности `.{5}` соответствует последовательность из пяти произвольных символов.
- Сокращению `\b` соответствует другая граница слова.

На самом деле данное регулярное выражение совпало бы как со словом *ANCYENT*, так и со словом *ANCIENT*.

А теперь сравните результаты, которые получаются в случае использования следующего выражения:

```
\b\w{7}\b
```

а также следующего:

```
\b.......\b
```

Наконец, упомяну о поиске соответствий нулю или нескольким произвольным символам с помощью следующей последовательности символов:

```
.*
```

которая означает то же самое, что и последовательность

```
[^\n]
```

или последовательность

```
[^\n\r]
```

Аналогичную роль играет и точка с квантификатором +, имеющим смысл "один или более раз":

```
.+
```

Апробируйте эти выражения в RegExr, и вы увидите, что каждому из них соответствует только первая строка текста (при снятом флажке global). Такое поведение обусловлено тем, что точка, как правило, не совпадает с символами новой строки, такими как символ "перевод строки" (U+000A) или символ "возврат каретки" (U+000D). Если же вы установите флажок dotall (точке соответствуют границы строк), то как `.*`, так и `.+` выберут весь целевой текст, отображаемый в нижнем поле. (При установленном флажке dotall точке соответствуют все символы, включая символы новой строки.)

Объясняется это тем, что указанные квантификаторы — *жадные*; иными словами, они пытаются найти совпадение как можно большей длины. Однако пока что вам не стоит об этом задумываться. В главе 7 о квантификаторах и их "жадности" будет рассказано более подробно.

Разметка текста тегами

Поэма "The Rime of the Ancient Mariner" — всего лишь простой текст. А что если этот текст понадобится отобразить в Интернете? Что если вы захотите разметить его как документ HTML5 не вручную, а с использованием регулярных выражений? Как это сделать?

Более подробно об этом будет рассказано в последующих главах, тогда как в этой главе мы сделаем лишь первые шаги в этом направлении, чтобы в дальнейшем постепенно добавлять новую разметку.

Щелкните в RegExr на вкладке Replace (Замена), включите многострочный режим, установив флажок multiline (многострочный режим), и введите в верхнем текстовом поле такое выражение:

```
(^T.*$)
```

Это выражение совпадет с первой строкой поэмы и запомнит ее в качестве содержимого группы, ограниченной круглыми скобками.

Далее введите во втором сверху поле следующий текст:

```
<h1>$1</h1>
```

Подстановочное регулярное выражение помещает захваченную группу, представленную переменной `$1`, в элемент `h1`. Результат можно наблюдать в текстовом поле в нижней части окна. В большинстве реализаций, включая Perl, для указанной переменной использовался бы стиль `\1`, но приложение RegExr поддерживает только переменные вида `$1`, `$2`, `$3` и т.д. Больше о группах и обратных ссылках вы узнаете в главе 4.

Использование редактора *sed* для разметки текста

Разметку текста можно выполнить из командной строки, используя редактор *sed*. Это потоковый редактор, способный интерпретировать регулярные выражения и преобразовывать текст. Он был разработан в начале 1970-х годов Ли Макмэхоном из Bell Labs. Если вы работаете на компьютере Mac или на компьютере под управлением Linux, значит, этот редактор у вас уже установлен.

Испытайте редактор *sed*, введя в командной строке (например, в окне Terminal на Mac) следующую команду:

```
echo Hello | sed s/Hello/Goodbye/
```

Вот что должно произойти после этого.

- Команда `echo` выводит слово *Hello* на стандартное устройство вывода (каковым обычно является экран), но поскольку вслед за командой `echo` указан символ канала в виде вертикальной черты (`|`), вывод будет передан следующей команде, т.е. редактору *sed*.
- Канал перенаправляет вывод команды `echo` на вход редактора *sed*.
- Затем команда `s` (substitute) редактора *sed* заменяет слово *Hello* словом *Goodbye*, и это слово выводится на экран.

Те читатели, на компьютерах которых редактор *sed* еще не установлен, могут установить его, следуя приведенными в конце главы указаниям, которые относятся к двум версиям *sed*: BSD и GNU.

Когда будете готовы к работе, введите в командной строке следующую команду:

```
sed -n 's/^/<h1>/;s/$/<\/h1>/p;q' rime.txt
```

Вы должны получить следующий вывод:

```
<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

Давайте детально проанализируем эту команду.

- Строка начинается с вызова программы *sed*.
- Опция `-n` подавляет автоматический вывод каждой входной строки, заданный в *sed* по умолчанию; выводятся только те строки, для которых это явно указано командой `p`. Мы используем эту опцию для того, чтобы отобразить только строку, которая подвергается воздействию регулярного выражения, т.е. строку 1.
- Выражение `s/^/<h1>/` помещает открывающий тег `h1` в начало (`\^`) строки.
- Точка с запятой (`;`) служит для разделения команд.
- Выражение `s/$/<\/h1>/` помещает закрывающий тег `h1` в конец (`$`) строки.
- Команда `p` выводит строку, на которую воздействовало регулярное выражение (строка 1).
- Наконец, команда `q` осуществляет выход из программы, поэтому команда *sed* обрабатывает только первую строку.
- Описанные операции применяются к содержимому файла *rime.txt*.

Ниже показан другой возможный способ написания этой команды, в котором используется опция `-e`. Эта опция позволяет объединить команды редактирования, следующие одна за другой:

```
sed -ne 's/^/<h1>/' -e 's/$/<\/h1>/p' -e 'q' rime.txt
```

Что касается меня, то я, конечно же, предпочитаю более экономный способ разделения команд, предполагающий использование точки с запятой. Кроме того, можно собрать все команды в одном файле, аналогичном представленному ниже файлу *h1.sed* (этот файл находится в архиве примеров).

```
#!/usr/bin/sed
s/^/<h1>/
s/$/<\/h1>/
q
```

Чтобы выполнить этот файл, перейдите в каталог (или папку), в котором находится файл *rime.txt*, и введите в командной строке следующую команду:

```
sed -f h1.sed rime.txt
```

Использование Perl для разметки текста

Наконец, я покажу вам, как можно сделать то же самое с помощью Perl — универсального языка программирования, созданного Ларри Уоллом в 1987 году. Этот язык получил широкую известность благодаря предусмотренной в нем мощной поддержке регулярных выражений и широким возможностям обработки текстов.

Чтобы проверить, установлен ли Perl в вашей системе, введите в командной строке следующую команду (нажав в конце клавишу <Enter>):

```
perl -v
```

В результате на экране отобразится номер версии Perl, установленной в вашей системе, или сообщение об ошибке (см. раздел "На заметку").

Для получения того же вывода, что и в примере с редактором *sed*, введите в командной строке следующую команду:

```
perl -ne 'if ($. == 1) { s/^/<h1>/; s/$/<\/h1>/m; print; }'
↳ rime.txt
```

которая должна отобразить уже знакомую вам строку

```
<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

Распишем подробно, что делает каждая из отдельных составляющих приведенной выше команды Perl.

- Команда `perl` вызывает программу Perl.
- Опция `-n` задает цикл по строкам входного файла (*rime.txt*).
- Опция `-e` позволяет передавать код программы из командной строки, а не из файла (аналогично `sed`).
- Оператор `if` проверяет, является ли обрабатываемая строка первой. Переменная `$.` — это специальная переменная Perl, которой соответствует текущая строка.
- Первая команда подстановки `s` находит начало первой строки (`^`) и вставляет вместе нее открывающий тег `h1`.
- Вторая команда подстановки находит конец строки (`$`) и вставляет в этой позиции закрывающий тег `h1`.
- Модификатор, или флаг, `m` (от *multiline* — многострочный) в конце команды подстановки указывает на то, что вы хотите обрабатывать данную строку особым образом, отдельно от других, и поэтому символу `$` соответствует конец строки 1, а не конец файла.
- Наконец, команда выводит результат на стандартное устройство вывода (экран).
- Опять-таки, описанные операции применяются к файлу *rime.txt*.

Кроме того, можно собрать все команды в одном программном файле, аналогичном представленному ниже файлу *h1.pl* (этот файл находится в архиве примеров).

```
#!/usr/bin/perl -n

if ($. == 1) {
s/^/<h1>;
s/$/<\/h1>/m;
print;
}
```

Чтобы выполнить этот файл, перейдите в каталог (или папку), в котором находится файл *rime.txt*, и введите в командной строке следующую команду:

```
perl h1.pl rime.txt
```

В Perl любую задачу всегда можно решить множеством способов. Я вовсе не утверждаю, что рассмотренный нами способ добавления тегов является наиболее эффективным. Это всего лишь один из множества вариантов. Велика вероятность того, что к тому времени, когда вы будете читать книгу, я придумаю другие, более эффективные способы решения задач с помощью Perl (и других инструментов). Надеюсь, что то же самое может быть сказано и о вас.

В следующей главе мы поговорим о границах и о так называемых *условиях с нулевой длиной совпадения* (zero-width assertions).

О чем вы узнали в главе 2

- Как находить совпадения со строковыми литералами.
- Как находить совпадения с цифровыми и нецифровыми символами.
- Что такое *глобальный* режим.
- Как символьные сокращения соотносятся с символьными классами.
- Как находить совпадения с символами, входящими и не входящими в состав слов.
- Как находить совпадения с пробелами.
- Как находить совпадения с произвольным одиночным символом с помощью точки.
- Что такое режим *dotall*.
- Как вставить HTML-разметку в строку текста с помощью RegExr, *sed* и Perl.

На заметку

- Сайт веб-приложения RegExr: `www.regexr.com`. Приложение RegExr создавалось с использованием библиотеки Flex 3 (`http://www.adobe.com/products/flex.html`) на базе движка ActionScript (`http://www.adobe.com/devnet/actionscript.html`), регулярные выражения которого аналогичны тем, которые используются в JavaScript (`https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/RegExp`).
- Git — это быстрая система управления версиями (`http://git-scm.com`). GitHub (`http://github.com`) — это веб-репозиторий проектов, в котором используется Git. Я советую использовать GitHub лишь в том случае, если вы уверенно работаете с Git или какой-либо другой современной системой управления версиями наподобие Subversion или Mercurial.
- HTML5 (`http://www.w3.org/TR/html5/`) — это пятая версия разрабатываемого консорциумом W3C стандарта HTML, который определяет язык разметки документов, публикуемых в Интернете. В течение вот уже нескольких лет этот стандарт существует в виде чернового варианта и регулярно подвергается изменениям, однако воспринимается широкими кругами специалистов как бесспорный наследник стандартов HTML 4.01 и XHTML.
- Редактор текстов *sed* доступен в системах Unix/Linux, включая их версии для Mac (Darwin и BSD). Пользователям Windows он доступен либо

в составе дистрибутивных пакетов, таких как Cygwin (http://www.cygwin.com), либо в виде отдельного приложения, которое можно загрузить по адресу http://gnuwin32.sourceforge.net/packages/sed.htm (текущая версия 4.2.1; см. http://www.gnu.org/software/sed/manual/sed.html).

- Для воспроизведения примеров с Perl вам, возможно, потребуется установить Perl в своей системе. По умолчанию этот язык поставляется вместе с системой Mac OS X Lion и часто уже включен в системы Linux. Если вы являетесь пользователем Windows, то можете получить доступ к Perl, установив соответствующие пакеты Cygwin (www.cygwin.com) или загрузив самый последний пакет Perl на сайте ActiveState (http://www.activestate.com/activeperl/downloads). Инструкции относительно установки Perl можно получить на следующих сайтах: http://learn.perl.org/installing/ или http://www.perl.org/get.html.

Чтобы выяснить, установлен ли Perl в вашей системе, воспользуйтесь приведенной ниже командой. Для этого откройте окно командной строки или интерпретатора команд, например окно Terminal (воспользуйтесь меню Applications/Utilities) на компьютере Mac или окно командной строки Windows (откройте стартовое меню и введите cmd в текстовом поле в нижней части меню). Затем введите следующую команду:

```
perl -v
```

Если Perl действительно установлен в вашей системе, эта команда вернет информацию об установленной версии. На своем компьютере Mac, работающем под управлением OS X Lion, я установил самую последнюю версию Perl (5.16.0 на момент написания книги) путем компиляции исходного кода (http://www.cpan.org/src/5.0/perl-5.16.0.tar.gz). После выполнения приведенной выше команды я получил следующую информацию.

```
This is perl 5, version 16, subversion 0 (v5.16.0) built
for darwin-2level

Copyright 1987-2012, Larry Wall

Perl may be copied only under the terms of either
the Artistic License or the GNU General Public License,
which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists,
should be found on this system using "man perl" or
"perldoc perl". If you have access to the Internet,
point your browser at http://www.perl.org/,
the Perl Home Page.
```

Отсюда видно, что после компиляции и сборки исходного кода как perl, так и perldoc устанавливаются в каталоге /usr/local/bin, который можно добавить в список путей, хранящийся в системной переменной PATH. Более подробную информацию относительно установки или изменения значения переменной PATH можно найти по следующему адресу:

```
http://java.com/en/download/help/path.xml
```

ГЛАВА 3

Границы

В этой главе мы сосредоточимся на позиционных привязках. Привязки соответствуют не каким-либо символам, а условным границам, разделяющим символы, т.е. определенным позициям между символами, в связи с чем их называют *условиями с нулевой длиной совпадения* (zero-width assertions). Некоторые границы, такие как ^ и $, называют *якорными метасимволами* или просто *якорями*.

В этой главе будут рассмотрены следующие границы:

- начало и конец логической или физической строки;
- границы слов (два вида);
- начало и конец подстроки;
- границы строковых литералов.

На первых порах моим инструментом вновь будет приложение RegExr. Кроме того, я буду использовать тот же текст, что и до этого: первые 12 строк содержимого файла *rime.txt*. Откройте в браузере веб-приложение RegExr (`http://www.regexr.com/v1/`) или воспользуйтесь его настольной версией и скопируйте первые 12 строк файла *rime.txt* из архива кода в нижнее текстовое поле (рис. 3.1).

Начало и конец строки

Как вы уже не раз видели, для поиска совпадений с началом логической или физической строки используется метасимвол ^ (U+005E), или *циркумфлекс* ("крышка"):

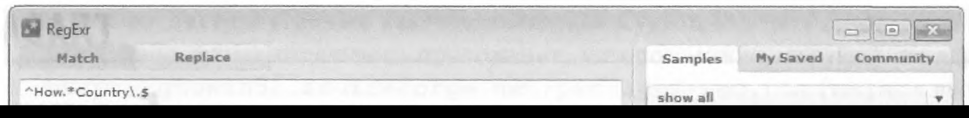
```
^
```

В зависимости от контекста метасимвол ^ будет совпадать с началом логической или физической строки, а иногда и целого документа. Конкретные свойства контекста зависят от приложения и используемых опций.

Вы уже знаете, что для поиска совпадений с концом строки используется знак доллара:

```
$
```

Убедитесь в том, что в приложении RegExr установлены флажки multiline (многострочный режим) и dotall (точке соответствуют границы строк). При открытии приложения RegExr флажок global (глобальный режим) установлен по умолчанию, но для данного примера безразлично, установлен он или нет. При снятом флажке global весь целевой текст считается одной строкой, а при снятом флажке dotall точке (`.`) будут соответствовать все символы, кроме символов конца строки (`\n` и `\r`).


RegExr
Match
Replace
Samples
My Saved
Community
^How.*Country\.$
show all

Вновь установите флажок dotall, и последнему выражению станет соответствовать весь текст. Для совпадения с оставшейся частью текста включать `\?$` в регулярное выражение не потребовалось.

Позиции, являющиеся и не являющиеся границами слов

Вы уже видели несколько случаев использования метасимвола `\b`. Он обозначает границу слова, т.е. позицию между словом и пробелом. Введите следующее выражение:

```
\bTHE\b
```

и оно совпадет с двумя вхождениями слова *THE* в первой строке (при условии, что установлен флажок global). Как и метасимволы `^` и `$`, метасимвол `\b` относится к категории условий с нулевой длиной совпадения. Вы можете включать его в выражения для поиска таких, например, объектов, как пробел или начало строки, но то, с чем он совпадает, на самом деле не представляет собой никакого символа. Вы обратили внимание на то, что пробелы, окружающие второе слово *THE*, остались невыделенными? Так произошло потому, что они не оказываются частью найденного соответствия. Возможно, вам трудно уловить суть сказанного с первого раза, но после некоторой практики вам все станет понятно.

Также существуют позиционные привязки, отличные от границ слов, к которым, например, относятся позиции между буквами или цифрами в слове. Введите в верхнем текстовом поле следующее выражение:

```
\Be\B
```

и посмотрите, с чем оно совпадает (рис. 3.2). Вы увидите, что ему соответствуют те буквы *e* нижнего регистра, которые окружены другими буквами или символами, не являющимися границами слов. Поскольку это условие с нулевой длиной совпадения, оно не ищет совпадения с окружающими символами, но способно распознавать те литералы *e*, которые окружены границами, не являющимися границами слов.

В некоторых приложениях границы слов указываются иначе. Например, для указания начала слова используется такая последовательность:

```
\<
```

а для указания конца слова — такая:

```
\>
```

Это пример устаревшего синтаксиса, который не применяется в современных приложениях, работающих с регулярными выражениями. В некоторых случаях он оказывается полезным, поскольку, в противоположность последовательности `\b`, позволяет различать начало и конец слова.

Если в вашей системе установлен редактор текстов *vi* или *vim*, воспользуйтесь им, чтобы проверить, как работает описанный синтаксис. Для этого следуйте приведенным ниже простым инструкциям. Вы легко их выполните, даже если никогда прежде не работали с редактором *vim*. Откройте окно командной строки или интерпретатора команд, перейдите в каталог, в котором находится файл с текстом поэмы, и выполните следующую команду:

```
vim rime.txt
```


RegExr
Match
Replace
Samples
My Saved
Community
\Be\B
show all

в командной строке. Проверьте, как работает эта утилита, введя в командной строке такую команду:

```
grep -Eoc '\<(THE|The|the\>' rime txt
```

Опция `-E` указывает на то, что вы хотите использовать расширенный вариант регулярных выражений (ERE), а не базовый (BRE), принятый в *grep* по умолчанию. Опция `o` означает, что в качестве результата будет отображаться только та часть строки, которая совпадает с шаблоном, тогда как опция `c` означает, что возвращаться должен только счетчик совпадений. Шаблону, заключенному в одинарные кавычки, будет соответствовать любое из следующих целых слов: *THE*, *The* и *the*. Поиск целых слов обеспечивается символами < и >.

Данная команда возвратит следующий результат:

```
259
```

представляющий общее количество найденных слов.

С другой стороны, если исключить символы < и > из шаблона, результат будет совершенно иной. Выполните следующую команду:

```
grep -Eoc '\(THE|The|the\' rime txt
```

и вы получите другое число:

```
327
```

Почему? Потому что новому шаблону соответствуют не только указанные целые слова, но и *любая* последовательность, содержащая *любое* такое слово. Надеюсь, теперь вам стало понятно, какие удобства обеспечивает использование символов < и >.

Другие якорные привязки

Существует метасимвол, аналогичный якорю ^, которому соответствует начало входной строки:

```
\A
```

Это сокращение доступно не во всех реализациях регулярных выражений, но по крайней мере в Perl и PCRE (Perl Compatible Regular Expressions) им можно пользоваться. Для поиска совпадений с концом входной строки служит дополняющее его сокращение:

```
\Z
```

или, в некоторых контекстах, такое сокращение:

```
\z
```

Утилита *pcregrep* — это аналог утилиты *grep* для библиотеки PCRE. (О том, где ее можно получить, см. в разделе "На заметку".) Чтобы проверить, как работает эта утилита, выполните следующую команду:

```
pcregrep -c '\A\s*(THE|The|the)' rime.txt
```

которая вернет значение счетчика вхождений (`-c`) слова *the* (в трех вариантах написания), равное 108, при условии, что это слово встречается в начале строки и ему предшествуют нуль или более пробелов. Затем введите следующую команду:

```
pcregrep -n '(MARINERE|Marinere)(.)?\Z' rime.txt
```

Эта команда выполнит поиск слов *MARINERE* и *Marinere*, которые находятся в конце строки и за которыми может следовать необязательный произвольный символ, в данном случае являющийся либо знаком препинания, либо буквой *S*. (Скобки, окружающие точку, не играют никакой существенной роли.)

В результате выполнения этой команды вы должны получить следующий вывод.

```
9:      It is an ancyent Marinere,
37:        The bright-eyed Marinere.
63:        The bright-eyed Marinere.
105:      "God save thee, ancyent Marinere!
282:      "I fear thee, ancyent Marinere!
702:      He loves to talk with Marineres
```

Опция `-n` команды *pcregrep* задает вывод номеров строк. Опции утилиты *pcregrep* в своем большинстве аналогичны опциям утилиты *grep*. Их полный перечень можно получить с помощью такой команды:

```
pcregrep --help
```

Задание группы символов как литералов

Для указания начала и конца группы символов, которые должны интерпретироваться как литералы, используются метасимволы

```
\Q
```

и

```
\E
```

Чтобы увидеть, как это работает, введите в нижнем текстовом поле RegExr следующие метасимволы:

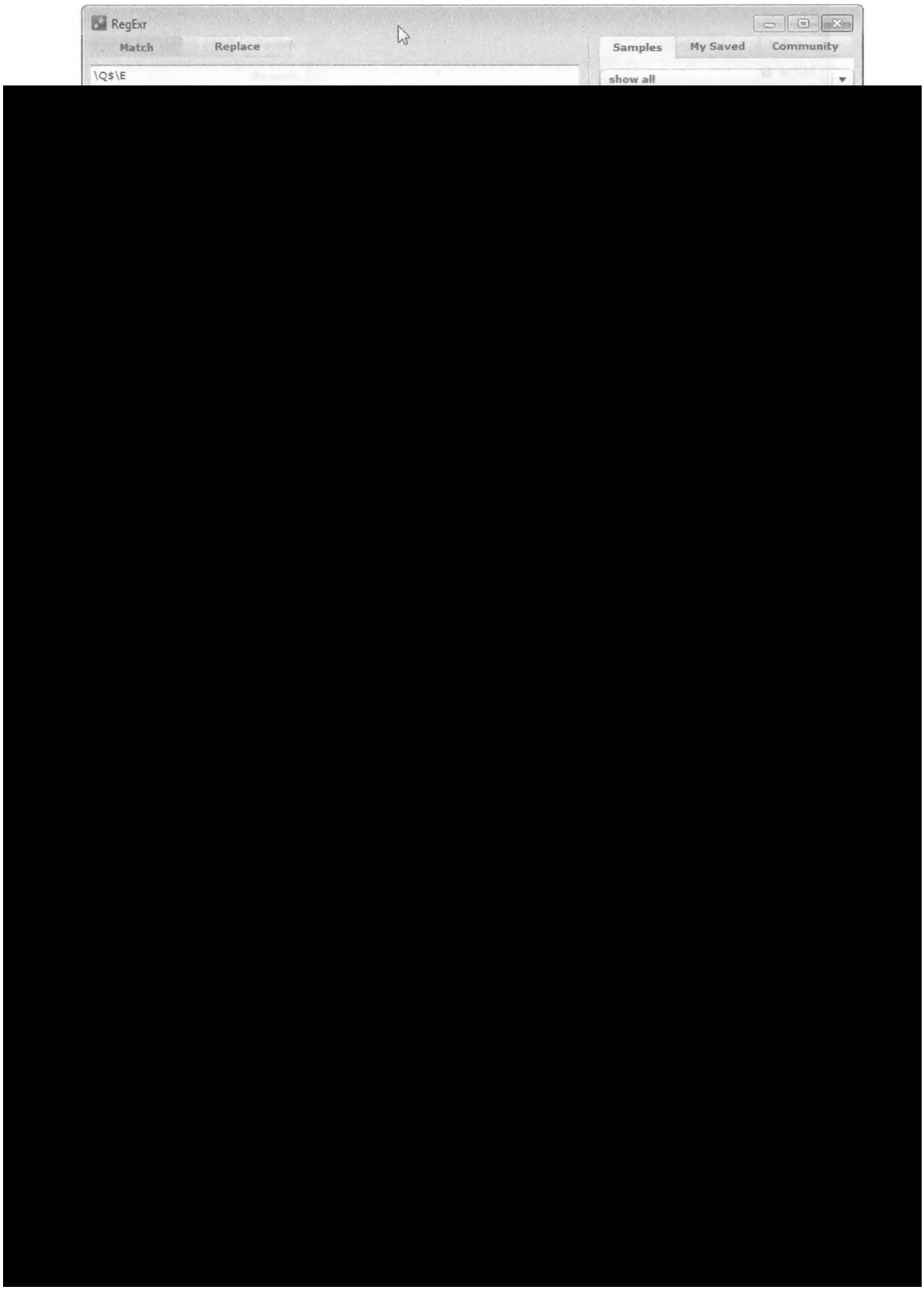
```
.^$*+?|(){}[]\-
```

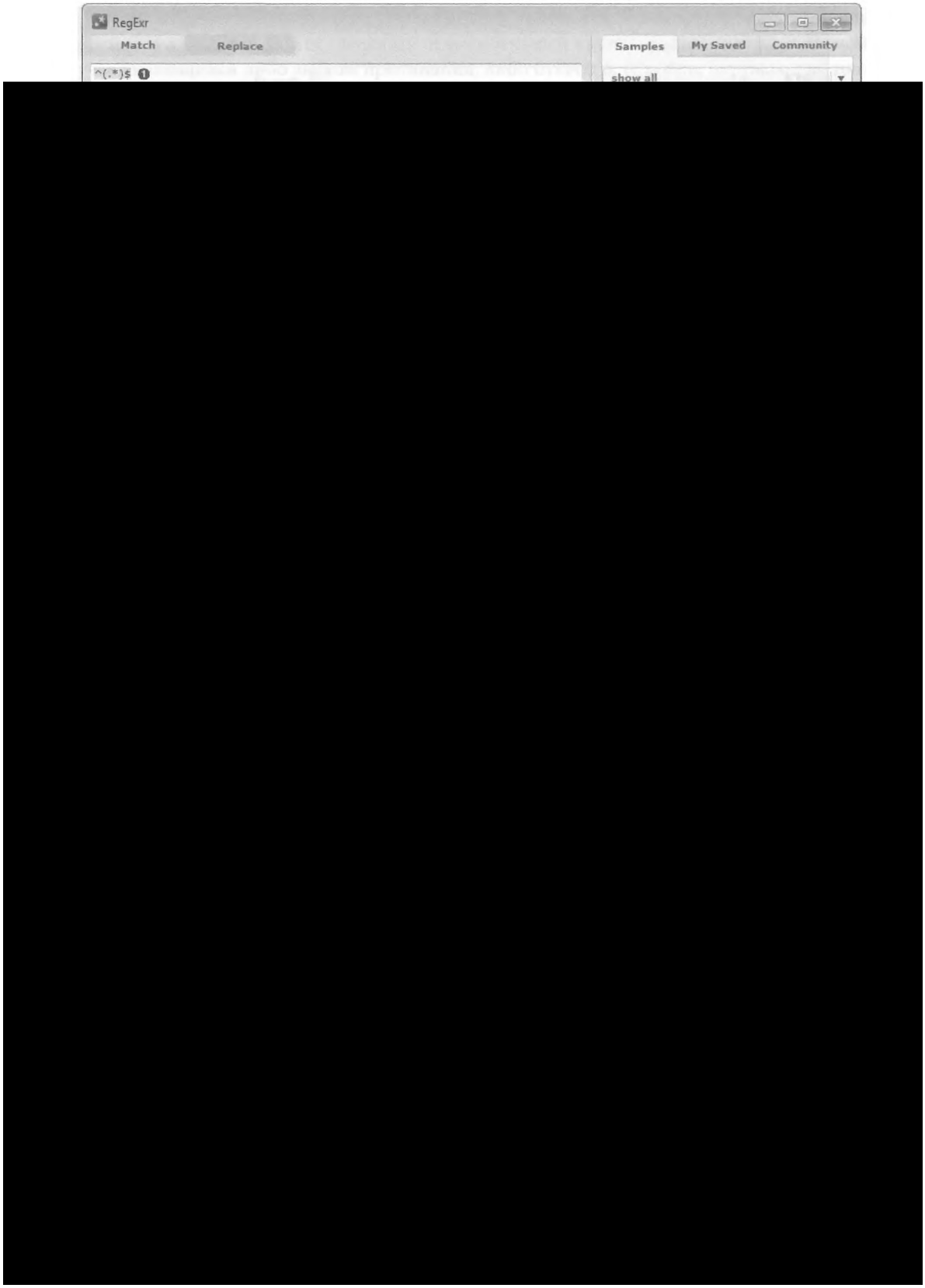
В регулярных выражениях, используемых для кодирования шаблонов, все эти 15 метасимволов играют роль специальных символов. (Дефис интерпретируется как специальный символ, если он используется для указания диапазона в квадратных скобках, определяющих символьный класс. Во всех остальных случаях он интерпретируется как обычный символ.)

Если вы хотите найти совпадение с любым из этих символов, введя его в верхнем текстовом поле RegExr, то ничего не произойдет. Почему? Потому что RegExr думает (если приложение вообще способно думать), что вы вводите регулярное выражение, а не литеральные символы. А теперь введите в верхнем текстовом поле следующий шаблон:

```
\Q$\E
```

и вы увидите, что в нижнем текстовом поле выделится символ `$`, поскольку все, что заключено между `\Q` и `\E`, интерпретируется как набор литеральных символов (рис. 3.3). (Вспомните, что обычно для того, чтобы превратить метасимвол в литерал, перед ним надо ставить обратную косую черту.)


RegExr
Match
Replace
Samples
My Saved
Community
\Q$\E
show all


RegExr
Match
Replace
Samples
My Saved
Community
^(.*)$
show all

обратной косой черты перед кавычками *экранируют* их, чтобы они интерпретировались как литеральные символы, а не как часть команды.

Вот как должен выглядеть вывод при правильном выполнении этой команды в *sed*.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>The Rime of the Ancyent Mariner (1798)</title>
</head>
<body>
<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

Точно такие же команды *sed* хранятся в файле *top.sed* в архиве примеров. Вы можете выполнить их из файла с помощью следующей команды:

```
sed -f top.sed rime.txt
```

Вы должны получить вывод, аналогичный предыдущему. Если это требуется, можно перенаправить вывод в файл:

```
sed -f top.sed rime.txt > temp
```

Эта команда не только отображает результат на экране, но и сохраняет вывод в файле *temp* за счет той части команды, которая отвечает за перенаправление вывода (`> temp`).

Добавление тегов с помощью Perl

Попытаемся сделать то же самое с помощью Perl. Для этого достаточно выполнить следующую команду.

```
perl -ne 'print "<!DOCTYPE html>\
<html lang=\"en\">\
<head><title>Rime</title></head>\
<body>\
" if $. == 1;
s/^/<h1>/;s/$/<\/h1>/m;print;exit;' rime.txt
```

Сравните эту команду с аналогичной командой *sed*. Похожи ли они? Чем они отличаются? Команда *sed* выглядит немного проще, но, с моей точки зрения, команда Perl более мощная.

Проанализируем подробно, как это все работает.

- Переменная `$.`, значение которой проверяется в операторе `if`, представляет текущую строку. Возвратом значения `true` оператор `if` подтверждает, что текущая строка является строкой 1.
- Найдя строку 1 с помощью оператора `if`, Perl выводит директиву типа документа и несколько HTML-тегов. Как и в случае *sed*, кавычки необходимо экранировать.
- Первая подстановка вставляет открывающий тег `h1` в начало строки, а вторая — закрывающий тег `h1` в конец строки.
- Символ `m` в конце второй подстановки означает, что используется модификатор `multiline`. Это делается для того, чтобы команда распознавала конец первой строки. Если не использовать этот модификатор, символу `$` будет соответствовать конец файла.

- Команда `print` выводит результат выполнения подстановки.
- Команда `exit` осуществляет немедленный выход из Perl. Без нее, в силу наличия опции `-n`, этот сценарий обрабатывал бы в цикле каждую строку, что в данном случае нам не нужно.

Поскольку ручной ввод этой команды довольно утомителен, я поместил в файл *top.pl*, который вы также найдете в архиве примеров, следующий код Perl.

```
#!/usr/bin/perl -n

if ($ == 1) {
print "<!DOCTYPE html>\
<html lang=\"en\">\
<head>\
<title>The Rime of the Ancyent Mariner (1798)</title>\
</head>\
<body>\
";
s/^/<h1>/;
s/$/<\/h1>/m;
print;
exit;
}
```

Этот код можно выполнить с помощью следующей команды:

```
perl top.pl rime.txt
```

Полученный вывод будет совпадать с предыдущим, хотя он и формируется немного иначе. (Как и в случае *sed*, для перенаправления вывода можно использовать символ >.)

О чем вы узнали в главе 3

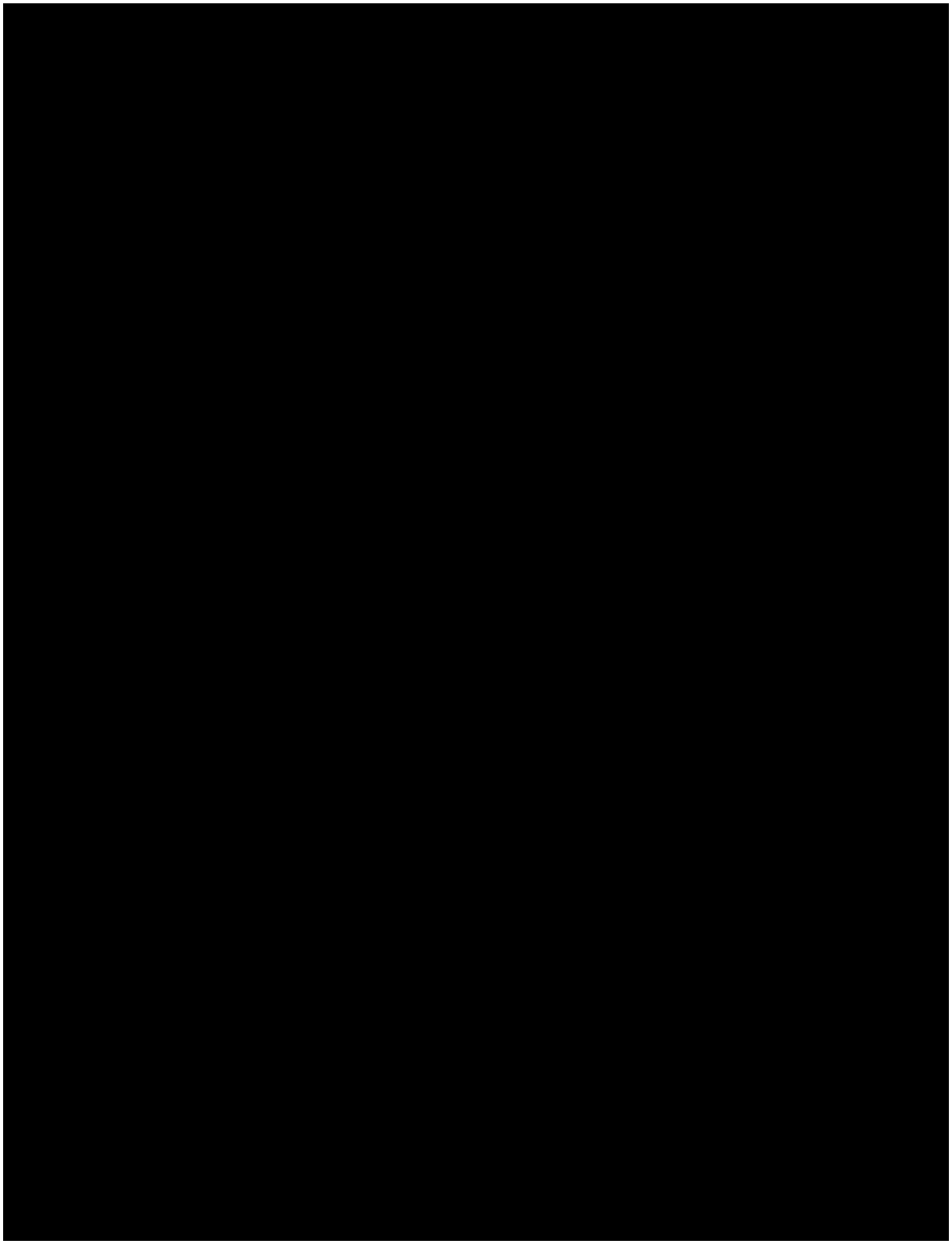
- Как использовать якорные привязки в начале и в конце строки с помощью метасимволов `\^` и `$`.
- Как использовать границы слов и другие границы.
- Как находить начало и конец подстроки с помощью метасимволов `\A` и `\Z` (или `\z`).
- Как обозначать строки символов как литералы с помощью метасимволов `\Q` и `\E`.
- Как добавить HTML-теги в документ с помощью RegExr, *sed* и Perl.

На заметку

- Редактор *vi* (сокращение от *visual*) — редактор текстов со встроенным механизмом регулярных выражений, разработанный сотрудником компании Sun Биллом Джоем в 1976 году. Редактор *vim* (сокращение от ***vi improved***) — усовершенствованный текстовый редактор, первоначально разработанный Брамом Моленаром на основе редактора *vi* (`http://www.vim.org`). Раннюю статью Билла Джоя и Марка Хортона, содержащую описание редактора *vi*, можно прочитать здесь: `http://docs.freebsd.org/44doc/usd/12.vi/paper.html`. Впервые

я использовал его в 1983 году и с тех пор не расстаюсь с ним практически ни на один день. В отношении выполнения больших объемов работы за короткое время редактору *vi* нет равных. А кроме того, его возможности настолько широки, что я не перестаю удивляться тому, как много нового для себя я постоянно открываю несмотря на то, что мое знакомство с ним длится уже более тридцати лет.

- Утилита *grep* — средство командной строки Unix, предназначенное для поиска и вывода строк с использованием регулярных выражений. Поговаривают, что на идею разработки этой утилиты, предложенной в 1973 году, ее создателя Кена Томпсона натолкнула одна из команд редактора *ed*: `g/re/p` (**g**lobal/**r**egular expression/**p**rint). Впоследствии она была вытеснена, хотя и не полностью, утилитой *egrep* (или *grep -E*), использующей расширенные регулярные выражения (ERE) с такими дополнительными метасимволами, как `|`, `+`,`?`, `(` и `)`. Утилита *fgrep* (*grep -F*) предназначена для выполнения операций поиска в файлах с использованием литеральных строк; метасимволы наподобие `$`, `*` и `|` никакого особого смысла в ней не имеют. Утилита *grep* доступна в Linux, а также в Mac OS X Darwin. Ее также можно получить в составе дистрибутивного пакета Cygwin GNU (`http://www.cygwin.com`) или загрузить по адресу `http://gnuwin32.sourceforge.net/packages/grep.htm`.
- PCRE (`http://www.pcre.org`), или Perl Compatible Regular Expressions, — библиотека функций языка C (8- и 16-разрядная версии) для работы с регулярными выражениями, совместимыми с Perl 5, которая также включает некоторые возможности других реализаций. Средство *pcregrep* — 8-разрядная утилита наподобие *grep*, позволяющая использовать возможности библиотеки PCRE в командной строке. Утилиту *pcregrep* для компьютеров Mac можно получить на сайте Macports (`http://www.macports.org`), выполнив команду `sudo port install pcre`. (На компьютере должна быть предварительно установлена интегрированная среда разработки Xcode; см. `https://developer.apple.com/technologies/tools/`. На сайте требуется пройти процедуру регистрации пользователя.)



ГЛАВА 4

Альтернативы, группы и обратные ссылки

Вы уже видели, как работают группы. Создание группы путем заключения текста в круглые скобки упрощает выполнение ряда операций, перечисленных ниже:

- чередование, т.е. выбор одного из нескольких возможных шаблонов;
- создание подшаблонов;
- захват (запоминание) групп для последующих обращений к ним с помощью обратных ссылок;
- применение операций к групповому шаблону, например квантификатору;
- использование групп без функции захвата;
- атомарные группы (дополнительная возможность).

В примерах этой главы наряду с полным текстом поэмы "The Rime of the Ancient Mariner" (файл *rime.txt*) используется ряд дополнительных текстов. Нашим основным инструментов будет настольная версия приложения RegExr, написанная с использованием технологии Adobe AIR (об установке приложения см. в главе 2), однако будут привлекаться и другие средства, такие как редактор *sed*.

Чередование

Термин *чередование* (alteration) означает возможность выбора альтернативных вариантов (альтернатив) шаблона при поиске совпадений. Предположим, требуется определить, сколько раз артикль *the* встречается в тексте поэмы "The Rime of the Ancient Mariner". Проблема заключается в том, что в поэме артикль может встречаться в различных формах: *THE*, *The* и *the*. Альтернативы позволяют справиться с этой проблемой.

Откройте настольное приложение RegExr, дважды щелкнув на его значке, и скопируйте в него текст поэмы из файла *rime.txt*, находящегося в архиве примеров.

Введите в верхнем текстовом поле такой шаблон:

```
(the|The|THE)
```

и вы увидите, как в расположенном под ним поле с текстом поэмы выделятся все вхождения артикля *the* (рис. 4.1). Для просмотра скрытой части текста воспользуйтесь полосой прокрутки.


RegExr
Match
Replace
Samples
My Saved
Community
(the|The|THE)
show all

Окончание табл. 4.1

Опция	Описание	Поддержка
`(?s)`	Обрабатывать текст как одну строку	PCRE, Perl, Java
`(?u)`	Обрабатывать шаблоны как строки Unicode	Java
`(?U)`	Делает модификаторы "нежадными" по умолчанию	PCRE
`(?x)`	Игнорировать пробельные символы и комментарии	PCRE, Perl, Java
`(?-...)`	Сброс или отключение опций	PCRE

* См. раздел "Named Subpatterns" на странице `http://www.pcre.org/pcre.txt`.

Далее мы рассмотрим применение альтернатив в *grep*. Кстати, опции, приведенные в табл. 4.1, в *grep* не работают, поэтому мы будем использовать исходный шаблон, содержащий перечисление альтернатив. Для подсчета количества строк, в которых встречается артикль *the*, причем независимо от регистра символов и того, сколько именно раз шаблон встречается в строке, используйте такую команду:

```
grep -Ec "(the|The|THE)" rime.txt
```

что должно привести к следующему результату:

```
327
```

Однако это еще не вся история, поэтому не расслабляйтесь.
Ниже приведен подробный анализ того, как работает данная команда.

- Опция `-E` означает, что вы хотите использовать расширенные регулярные выражения (ERE), а не базовые (BRE). Это позволяет избавиться от необходимости экранировать скобки и вертикальную черту (`\(THE\|The\|the\)`), что надо было бы сделать в случае использования BRE.
- Опция `-c` указывает на необходимость вывода количества строк, в которых обнаружены совпадения (а не собственно количества совпадений).
- Скобки объединяют варианты выбора, или альтернативы, заданные в виде *the*, *The* и *THE*, в одну группу.
- Символ вертикальной черты разделяет альтернативы, обработка которых осуществляется слева направо.

Чтобы получить фактическое количество вхождений артикля в тексте поэмы, необходимо использовать следующую команду:

```
grep -Eo "(the|The|THE)" rime.txt | wc -l
```

возвращающую каждое совпадение в виде отдельной строки, что приводит к следующему результату:

```
412
```

Проанализируем эту команду.

- Опция `-o` указывает на то, что отображать необходимо лишь ту часть строки, которая совпадает с шаблоном, хотя это и не очевидно по той причине, что канал (`|`) перенаправляет вывод команде `wc`.
- В данном контексте вывод команды `grep` перенаправляется в поток ввода команды `wc`. Команда `wc` — это команда подсчета слов, опция `-l` которой задает подсчет количества входных строк.

Откуда взялась столь большая разница в значениях: 327 и 412? Это произошло потому, что опция `-c` задает лишь подсчет строк, в которых встречаются совпадения с шаблоном, но ведь в одной строке может встретиться несколько совпадений. Если в команде `wc -l` дополнительно использовать опцию `-o`, то каждое вхождение искомого слова в любой из его форм будет появляться на отдельной строке и учитываться при подсчете, что и приводит к получению большего значения.

Выполним аналогичный поиск совпадений с помощью Perl, используя следующую команду:

```
perl -ne 'print if /(the|The|THE)/' rime.txt
```

Эту команду можно оптимизировать за счет применения опции `(?i)`, делающей ненужным использование списка альтернатив:

```
perl -ne 'print if /(?i)the/' rime.txt
```

Но и последнюю команду можно дополнительно улучшить, добавив модификатор `i` вслед за последним разделителем шаблона:

```
perl -ne 'print if /the/i' rime.txt
```

Результат останется тем же. Однако чем проще, тем лучше. Список дополнительных *модификаторов* (называемых также *флагами*) приведен в табл. 4.2. Одновременно у вас появляется возможность сравнить (разумеется, с учетом различий в синтаксисе) эти модификаторы с опциями, приведенными в табл. 4.1.

Таблица 4.2. Модификаторы (флаги) Perl*

Модификатор	Описание
`a`	Поиск соответствий для сокращений \d, \s, \w и классов POSIX только в диапазоне символов ASCII
`c`	Не сбрасывать текущую позицию поиска при неудачном исходе сопоставления
`d`	Использовать собственные правила платформы, заданные по умолчанию
`g`	Глобальное сопоставление, т.е. поиск всех вхождений шаблона
`i`	Игнорировать регистр при сопоставлении
`l`	Использовать правила текущей локали
`m`	Обрабатывать исходный текст как многострочный
`p`	Сохранять строку, которая совпала
`s`	Обрабатывать исходный текст как единую строку

Окончание табл. 4.2

Модификатор	Описание
u	Использовать правила Unicode при сопоставлении
x	Игнорировать пробельные символы и комментарии

* См. http://perldoc.perl.org/perlre.html#Modifiers

Подшаблоны

Когда говорят о *подшаблонах* в регулярных выражениях, то под этим термином чаще всего подразумевают группу или группы, входящие в другую группу. Подшаблон — это шаблон в шаблоне. Часто, хотя и не всегда, совпадение с подшаблоном проверяется лишь в том случае, если найдено совпадение для предшествующего ему шаблона. Подшаблоны можно конструировать множеством способов, но нас интересуют в первую очередь те из них, которые определяются с помощью круглых скобок.

В некотором смысле вы уже познакомились с подшаблонами, когда работали со следующим шаблоном:

```
(the|The|THE)
```

Здесь мы имеем дело с тремя подшаблонами. Первый из них — *the*, второй — *The*, третий — *THE*, но в данном случае поиск совпадений для второго подшаблона осуществляется независимо от поиска совпадений для первого подшаблона. (Поиск совпадений начинается с крайнего слева шаблона.)

А вот пример, в котором работа одного подшаблона (подшаблонов) зависит от результатов работы предыдущего:

```
(t|T)h(eir|e)
```

Этому выражению будут соответствовать подстроки, начинающиеся с одной из букв `t` или `T`, за которой следует буква `h`, за которой, в свою очередь, следует либо последовательность букв `eir`, либо буква `e`. Таким образом, данный шаблон совпадет с любым из следующих слов:

- *the*
- *The*
- *their*
- *Their*

В данном случае второй подшаблон — `(e|eir)` — зависит от первого — `(t|T)`.

Использовать круглые скобки в подшаблонах необязательно. Вот пример определения подшаблонов с помощью классов:

```
\b[tT]h[ceinry]*\b
```

Кроме слов *the* и *The* данному шаблону будут соответствовать также слова *thee*, *thy*, *thence* и много других. Указание двух границ слов (`\b`) означает, что шаблону будут соответствовать только целые слова, а не подстроки, входящие в состав других слов.

Проанализируем, как работает этот шаблон.

- Метасимволу `\b` соответствует начало слова.
- Выражение `[tT]` — это символьный класс, которому соответствует либо буква `t` нижнего регистра, либо буква `T` верхнего регистра.
- Далее шаблон находит (или пытается найти) букву `h` нижнего регистра.
- Второй (и последний) подшаблон также записан в виде символьного класса `[ceinry]` с последующим квантификатором, которому соответствует нуль или несколько символов.
- Наконец, шаблон оканчивается еще одной границей слова `\b`.

Хочу обратить ваше внимание на одну интересную особенность, характеризующую нынешнее состояние дел в области регулярных выражений. Будучи, как правило, строгой, терминология регулярных выражений в некоторых случаях довольно размыта. Пытаясь дать определения *подшаблона* и некоторых других терминов, используемых в данной книге, я просмотрел множество источников, стремясь привести их к общему знаменателю. Подозреваю, кое-кто может утверждать, что символьный класс нельзя причислять к подшаблонам. Но, поскольку символьные классы могут функционировать как шаблоны, я считаю, что для применения к ним термина "подшаблон" есть все основания.

Захватывающие группы и обратные ссылки

Если весь шаблон или некоторая часть его содержимого заключается в круглые скобки, образуя группу, то содержимое этой группы захватывается и временно сохраняется в памяти. Впоследствии на сохраненное содержимое можно ссылаться с помощью обратных ссылок вида

```
\1
```

или

```
$1
```

где переменные `\1` или `$1` ссылаются на первую захваченную группу, `\2` или `$2` — на вторую и т.д. Редактор *sed* воспринимает лишь ссылки вида `\1`, тогда как Perl воспринимает ссылки обоих типов.

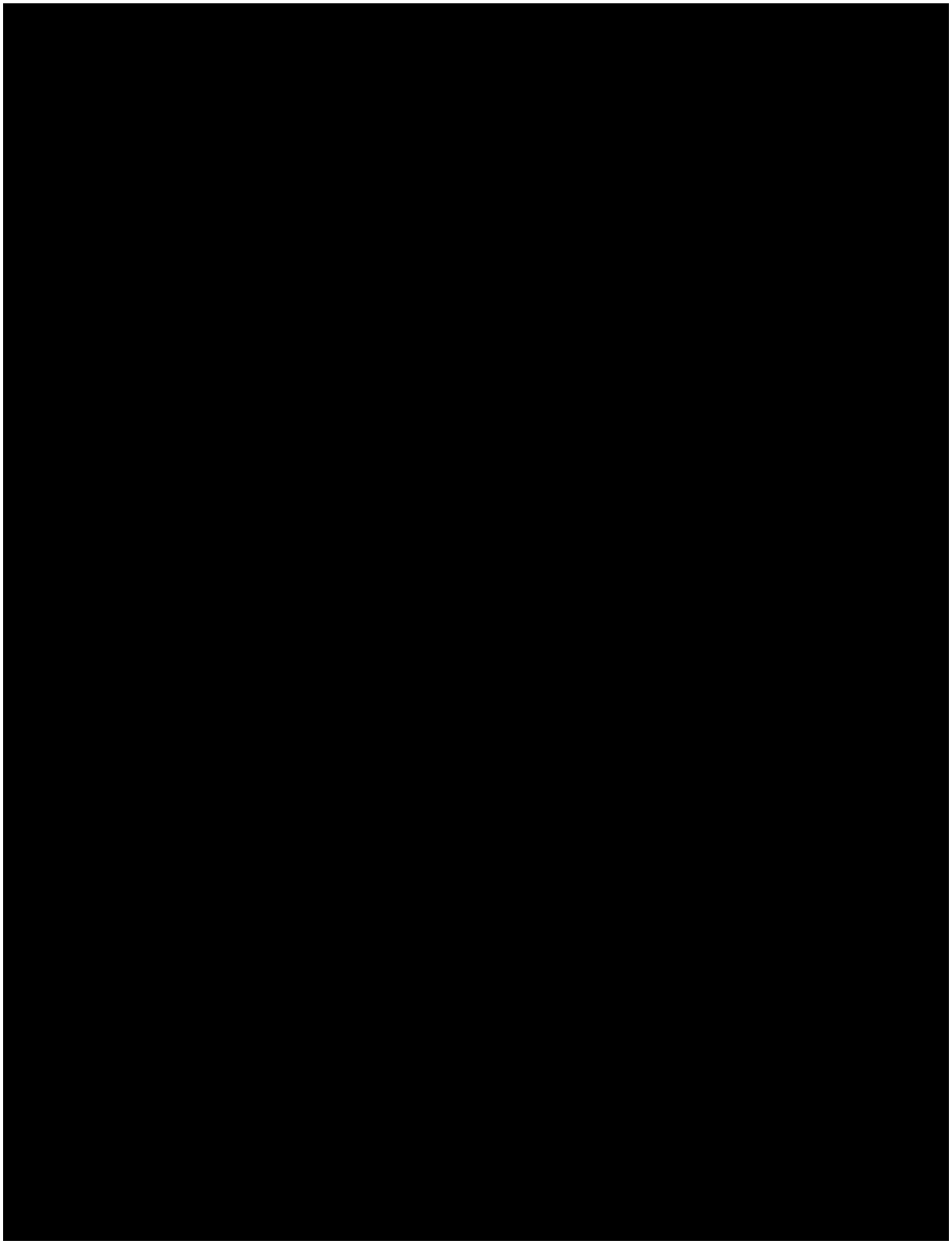
Первоначально в редакторе *sed* разрешалось использовать только ссылки в диапазоне от `\1` до `\9`, но в настоящее время это ограничение, по всей видимости, снято.

С подобным вы уже встречались в предыдущих главах, однако я все же приведу соответствующий пример. Его суть заключается в перестановке слов в одной из строк поэмы, за что я заранее приношу свои извинения ее автору Сэмюэлу Тейлору Кольриджу. Щелкнув в окне приложения RegExr на вкладке Replace, введите в верхнем текстовом поле следующий шаблон:

```
(It is) (an ancyent Marinere)
```

Прокрутите обрабатываемый текст (верхняя текстовая область) вниз, пока не увидите подсвеченную строку, и введите во втором текстовом поле следующий текст:

```
$2 $1
```



- Та же команда подстановки направляет в вывод измененную строку, полученную путем перестановки местами частей исходной строки, доступных по обратным ссылкам `\1` и `\2`.
- Символ `p` в конце команды подстановки означает вывод строки на печать.

Аналогичная команда Perl выглядит следующим образом:

```
perl -ne 'print if s/(It is) (an ancyent Marinere)/\2 \1/'↪
 rime.txt
```

Заметьте, что в этой команде используется синтаксис обратных ссылок в стиле `\1`. Разумеется, в Perl с равным правом можно использовать синтаксис вида `$1`:

```
perl -ne 'print if s/(It is) (an ancyent Marinere)/$2 $1/'↪
 rime.txt
```

Мне нравится та простота, с какой Perl позволяет вывести выделенную строку на печать. Скажу несколько слов о полученном выводе:

```
an ancyent Marinere It is,
```

Преобразование нарушило общепринятые правила использования прописных и строчных букв, но Perl позволяет исправить это с помощью директив `\u` и `\l`:

```
perl -ne 'print if s/(It is) (an ancyent Marinere)/\u$2↪
 \l$1/' rime.txt
```

Теперь результат выглядит гораздо лучше:

```
An ancyent Marinere it is,
```

Объясню, как мы этого добились:

- символ `\u` ничему не соответствует и преобразует следующий символ в верхний регистр;
- символ `\l` ничему не соответствует и преобразует следующий символ в нижний регистр;
- директива `\U` (здесь не используется) преобразует все символы следующей строки в верхний регистр;
- директива `\L` (здесь не используется) преобразует все символы следующей строки в нижний регистр.

Эти директивы действуют до тех пор, пока не будут отменены (например, `\l` отменяет действие `\U`). Поэкспериментируйте с ними самостоятельно, чтобы проверить, как они работают.

Именованные группы

Именованные группы — это захватывающие группы, которым присвоены имена. Доступ к сохраненному содержимому таких групп может осуществляться с использованием имен, а не целых чисел. Их использование демонстрируется ниже на примере Perl.

```
perl -ne 'print if s/(?<one>It is) (?<two>an ancyent Marinere)/↪
\u$+{two}\l$+{one}/' rime.txt
```

В этой команде именованные группы создаются и используются следующим образом.

- Присвоение группам имен `one` и `two` осуществляется путем дописывания последовательностей `?<one>` и `?<two>` в начале содержимого соответствующей группы в круглых скобках.
- Последовательность `$+{one}` ссылается на группу `one`, а последовательность `$+{two}` — на группу `two`.

Допускается повторное использование именованных групп в том шаблоне, в котором они были поименованы. Сейчас поясню, что имеется в виду. Предположим, выполняется поиск строки, содержащей подстроку в виде шести следующих подряд нулей:

```
000000
```

Это тривиальный пример, но для наших целей его будет вполне достаточно. Присвоим имя группе, состоящей из трех нулей, с помощью следующего шаблона (имя `z` выбрано произвольно):

```
(?<z>0{3})
```

Далее эту группу можно использовать примерно так:

```
(?<z>0{3})\k<z>
```

так:

```
(?<z>0{3})\k'z'
```

или так:

```
(?<z>0{3})\g{z}
```

Апробируйте эти выражения в RegExr, и вы убедитесь в том, что все они отлично работают. Ряд других синтаксических конструкций, предназначенных для работы с именованными группами, приведен в табл. 4.3.

Таблица 4.3. Синтаксис именованных групп

Синтаксис	Описание
`(?<`*имя*`>...)`	Именованная группа
`(?`*имя*`...)`	Другая именованная группа
`(?P<`*имя*`>...)`	Именованная группа в Python
`\k<`*имя*`>`	Ссылка на группу по ее имени в Perl
`\k'`*имя*`'`	Ссылка на группу по ее имени в Perl
`\g{`*имя*`}`	Ссылка на группу по ее имени в Perl
`\k{`*имя*`}`	Ссылка на группу по ее имени в .NET
`(?P=`*имя*`)`	Ссылка на группу по ее имени в Python

Незахватывающие группы

Существуют также *незахватывающие* группы, содержимое которых не сохраняется в памяти. Иногда это оказывается преимуществом, особенно в тех случаях, когда вы не собираетесь ссылаться на группу. Отсутствие необходимости сохранения данных в памяти может обеспечивать более высокую производительность, хотя при выполнении простых примеров, приведенных в данной книге, об этом вряд ли стоит задумываться.

Помните, что собой представляла первая из групп, которые обсуждались в этой главе? Вот как она выглядела:

```
(the|The|THE)
```

Никакой необходимости в использовании обратных ссылок здесь нет, поэтому можно создать незахватывающую группу такого вида:

```
(?:the|The|THE)
```

По аналогии с примером, приведенным в самом начале главы, в это выражение можно было бы добавить опцию, делающую шаблон нечувствительным к регистру (хотя эта опция фактически устраняет необходимость в использовании группы):

```
(?i)(?:the)
```

Эту же группу можно записать в таком виде:

```
(?:(?i)the)
```

а еще лучше в таком:

```
(?i:the)
```

Букву `i`, устанавливающую данную опцию, можно записывать между вопросительным знаком и двоеточием, что и было сделано в последнем случае.

Атомарные группы

Атомарные группы — это разновидность незахватывающих групп. В случае использования движка регулярных выражений, работающего в режиме поиска с возвратом, атомарная группа отключает этот режим, но не для всего регулярного выражения, а лишь для той его части, которая заключена в данной группе. Соответствующий синтаксис выглядит примерно так:

```
(?>the)
```

В каких ситуациях имеет смысл использовать атомарные группы? К числу операций, способных существенно замедлять работу механизма регулярных выражений, относится поиск с возвратом. Это связано с тем, что такой поиск требует перебора всех возможностей, на что уходит много времени и расходуются значительные вычислительные ресурсы. Иногда операции такого рода становятся основными пожирателями времени. Существует даже термин — *катастрофический поиск с возвратом.*

Можете либо вообще отказаться от поиска с возвратом, используя движки наподобие re2 (`http://code.google.com/p/re2/`), либо отключать этот режим для некоторых частей регулярных выражений, создавая атомарные группировки.

Основная цель, которую я преследую в книге, — это ознакомление читателя с регулярными выражениями. Вопросов производительности я почти не касаюсь, но именно с ними, по моему мнению, наиболее тесно связано использование атомарных группировок.

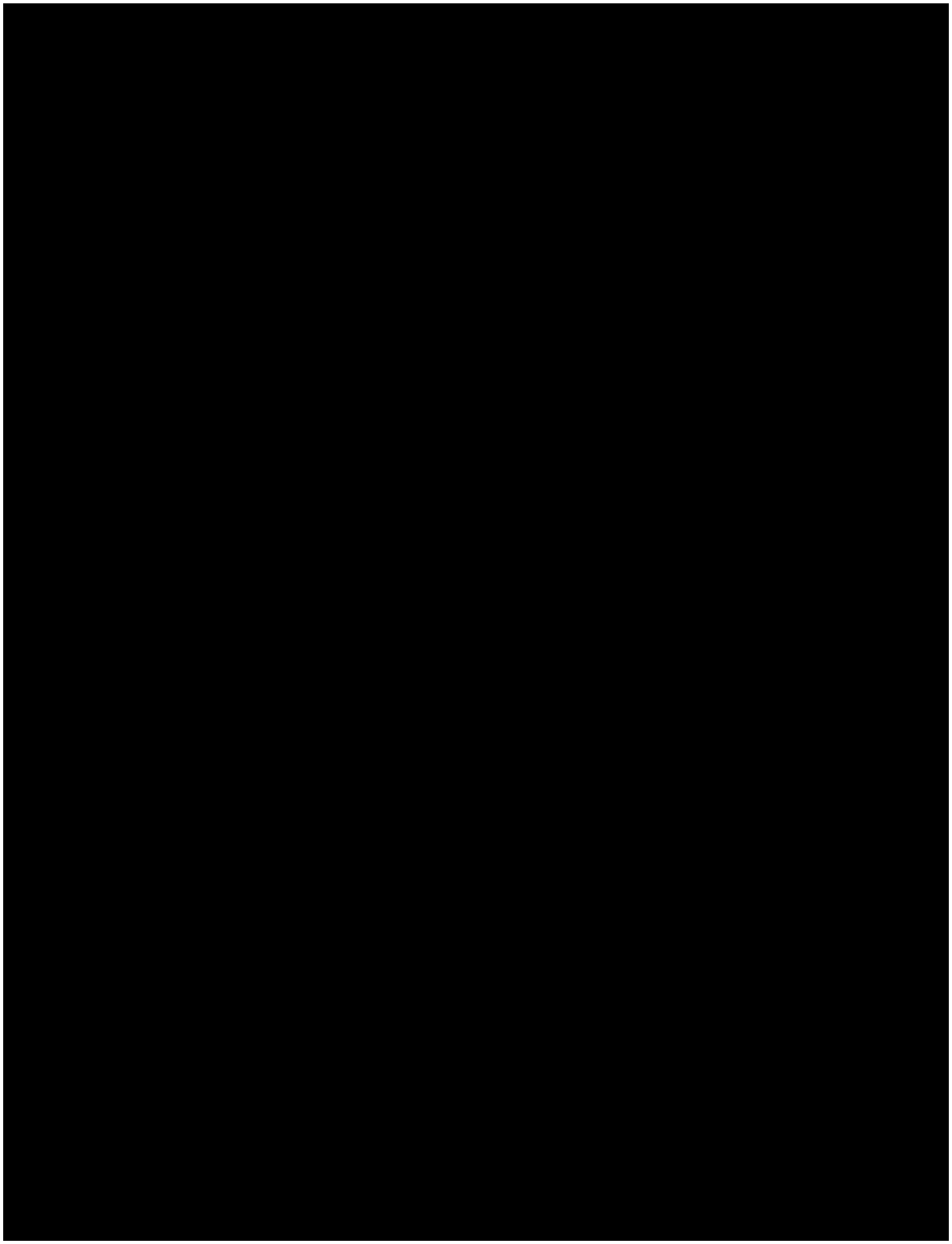
В главе 5 речь пойдет о символьных классах.

О чем вы узнали в главе 4

- Каким образом альтернативы обеспечивают возможность выбора между двумя и более шаблонами.
- Какие существуют модификаторы опций и как они используются в шаблоне.
- Какие виды подшаблонов существуют.
- Использование захватывающих групп и обратных ссылок.
- Использование именованных групп и ссылок на них.
- Использование незахватывающих групп.
- Что такое атомарная группа.

На заметку

- Adobe Air — кроссплатформенная среда выполнения, позволяющая использовать HTML, JavaScript, Flash и ActionScript для создания веб-приложений, способных выполняться в качестве автономных клиентских приложений без использования браузера. Более подробную информацию по этому вопросу можно получить по адресу `http://www.adobe.com/products/air.html`.
- Python (`http://www.python.org`) — высокоуровневый язык программирования, имеющий собственную реализацию механизма регулярных выражений (`http://docs/python.org/library/re.html`).
- .NET (`http://www.microsoft.com/net`) — программная платформа для компьютеров, работающих под управлением Windows, в которой также реализован механизм регулярных выражений (`http://msdn.microsoft.com/en-us/library/hs600312.aspx`).
- Более полное описание атомарных групп приведено на сайтах `http://www.regularexpressions.info/atomic.html` и `http://stackoverflow.com/questions/6488944/atomic-group-and-non-capturing-group`.



ГЛАВА 5

Символьные классы

В этой главе речь пойдет о символьных классах, также известных как *скобочные выражения*. Символьные классы упрощают сопоставление с одиночными символами или последовательностями символов. Для многих классов предусмотрены сокращенные формы записи, называемые *сокращениями*. Например, сокращение `\d` эквивалентно следующему классу:

```
[0-9]
```

которому соответствует любая одиночная цифра из диапазона 0–9. Классы способны обеспечивать именно ту степень специфичности, которая вам необходима. В этом смысле они более универсальны, чем символьные аббревиатуры.

Для выполнения приведенных ниже примеров можете использовать любой удобный для вас процессор регулярных выражений. Однако приведенные ниже описания примеров относятся к веб-приложению Rubular, используемому совместно с браузером Firefox, и настольному приложению Reggy.

Введите в области веб-страницы, предназначенной для размещения целевого текста, следующую строку.

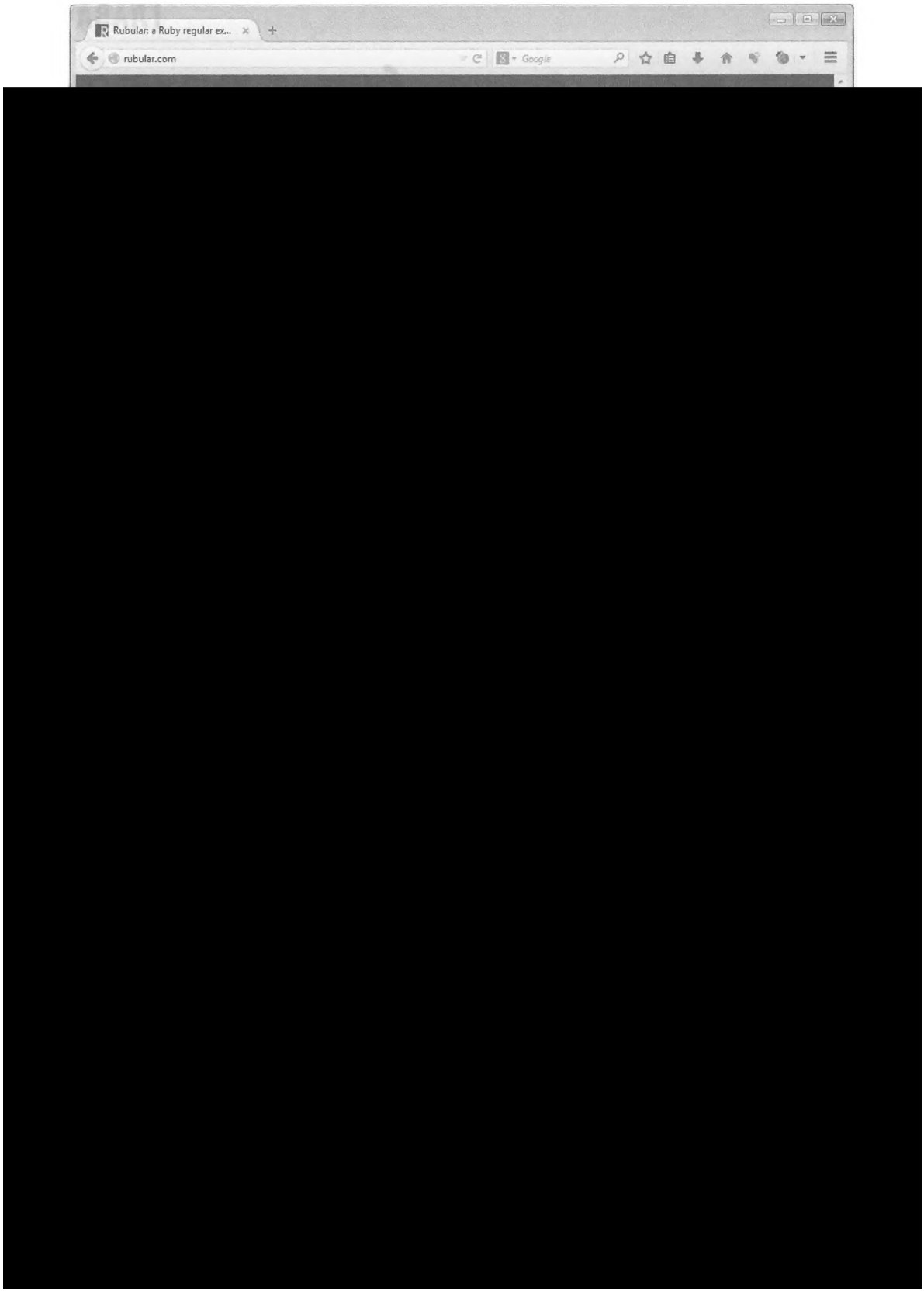
```
! " # $ % & ' ( ) * + , - . /
0       1       2       3       4       5       6       7
8       9
: ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _ '
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

Вам вовсе не обязательно вводить все вручную. Этот текст хранится в файле *ascii-graphic.txt*, включенном в архив примеров.

Мы начнем с использования символьного класса для поиска определенных букв латинского алфавита, а именно гласных:

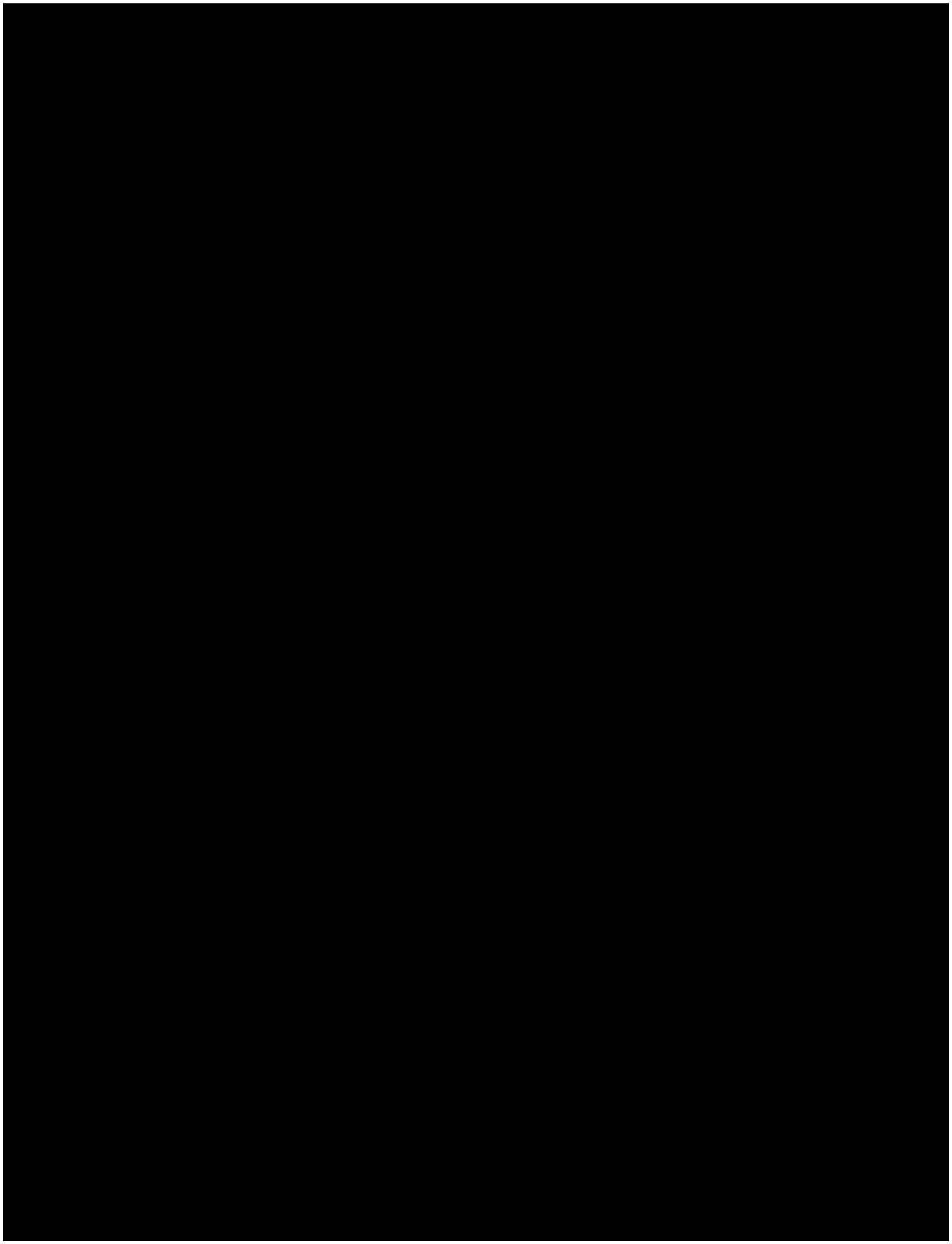
```
[aeiou]
```

Эти гласные должны выделиться в расположенной ниже поля ввода текстовой области (рис. 5.1). А как бы вы организовали поиск гласных, записанных в верхнем регистре, или всех гласных независимо от регистра?

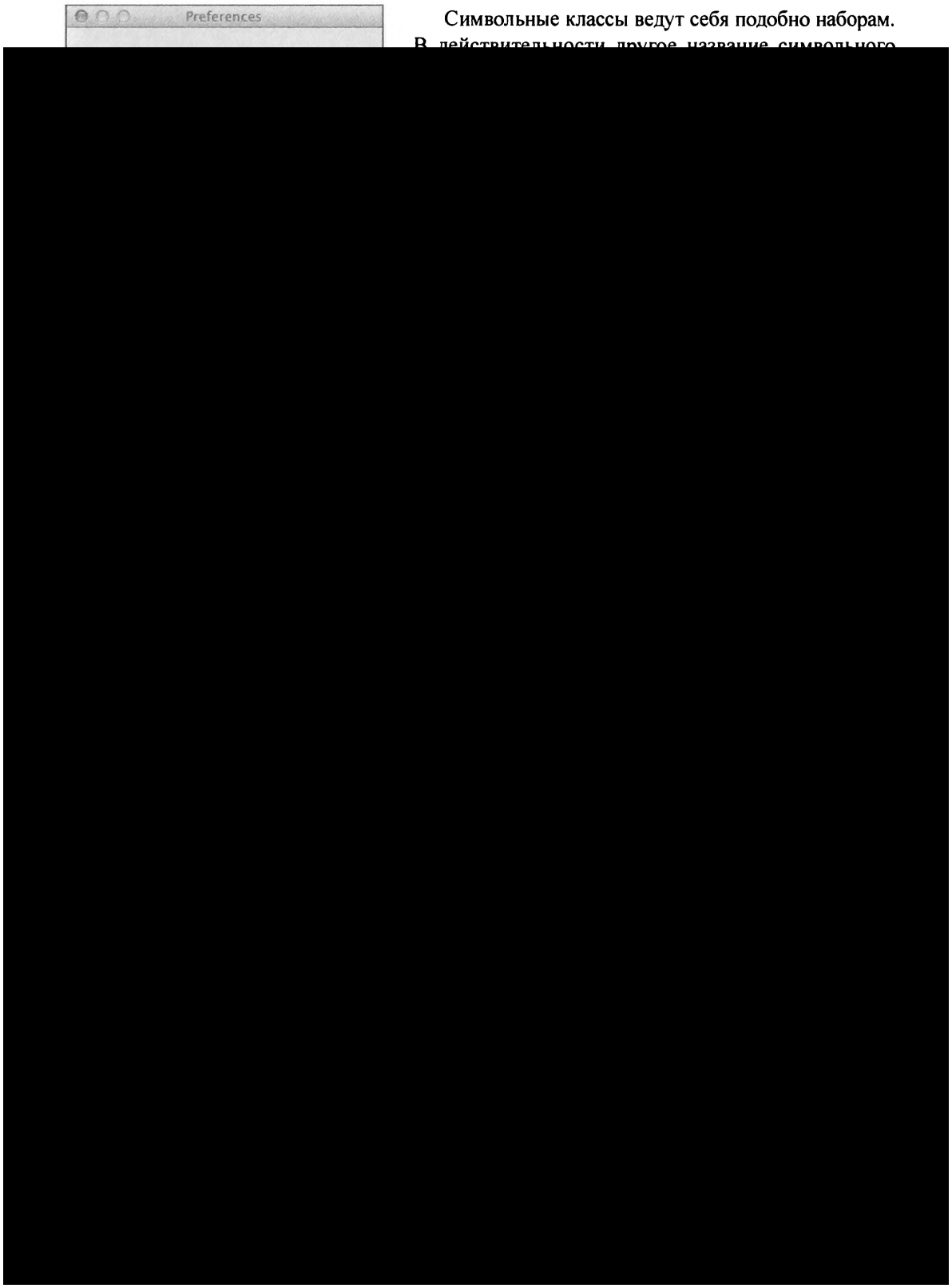

Rubular: a Ruby regular ex...
rubular.com
Google

В классах разрешается использовать символьные сокращения. Например, для сопоставления с пробельными и словарными символами можно создать следующий класс:

```
[\w\s]
```

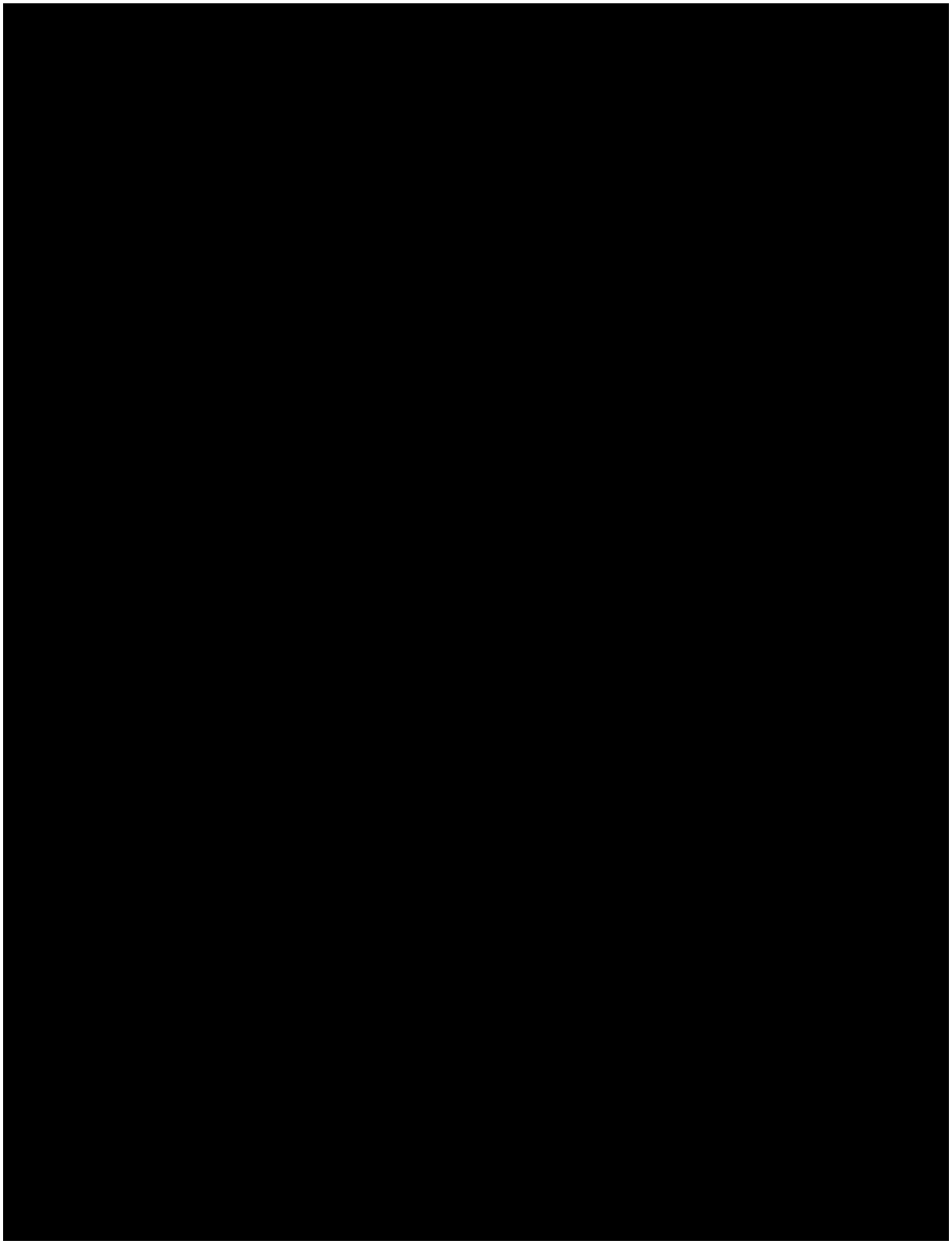


Объединение и разность



Символьные классы ведут себя подобно наборам.

Для поиска соответствий символам, которые имеются только в одном из двух наборов (а по сути — символам, принадлежащим к разности наборов), можно использовать выражения наподобие следующего:



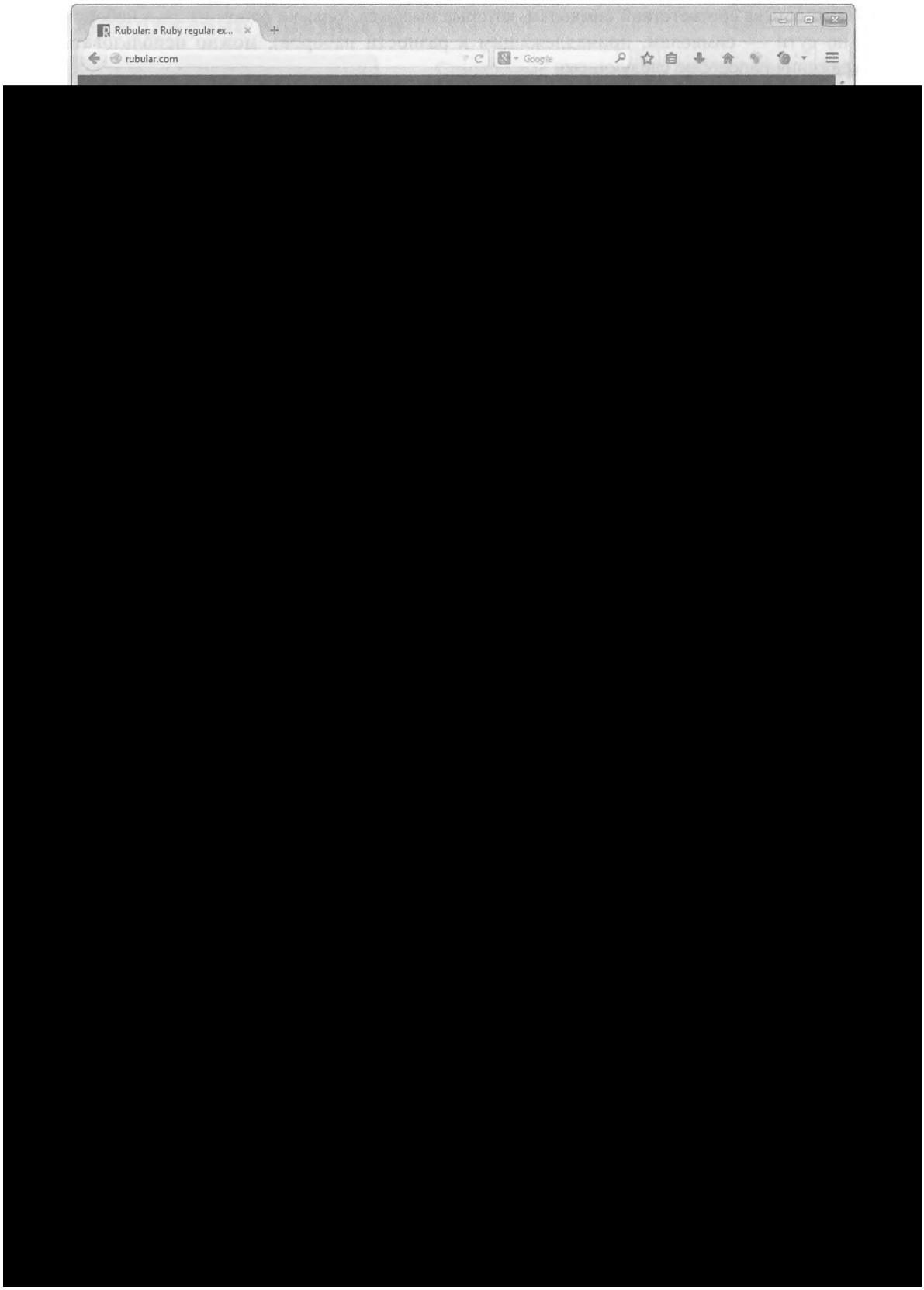

Rubular: a Ruby regular ex...
rubular.com
Google

Таблица 5.1. Символьные классы POSIX

Класс символов	Описание
`[[:alnum:]]`	Алфавитно-цифровые символы (буквы и цифры)
`[[:alpha:]]`	Алфавитные символы (буквы)
`[[:ascii:]]`	ASCII-символы (все 128)
`[[:blank:]]`	Пробельные символы
`[[:ctrl:]]`	Управляющие символы
`[[:digit:]]`	Цифры
`[[:graph:]]`	Графические символы
`[[:lower:]]`	Буквы нижнего регистра
`[[:print:]]`	Печатные символы
`[[:punct:]]`	Знаки пунктуации
`[[:space:]]`	Пробел
`[[:upper:]]`	Буквы верхнего регистра
`[[:word:]]`	Словарные символы
`[[:xdigit:]]`	Шестнадцатеричные цифры

Следующая глава посвящена сопоставлению с символами Unicode и другими символами.

О чем вы узнали в главе 5

- Как создать символьный класс или набор с помощью скобочного выражения.
- Как создать один или несколько диапазонов в символьном классе.
- Как выполнить сопоставление с четными числами в диапазоне от 0 до 99.
- Как выполнить сопоставление с шестнадцатеричным числом.
- Как использовать символьные сокращения в символьном классе.
- Как инвертировать символьный класс.
- Как выполнить объединение и вычитание символьных классов.
- Что такое символьные классы POSIX.

На заметку

- Настольное приложение Reggy для компьютеров Mac можно бесплатно загрузить на сайте `http://www.reggyapp.com`. Найденные в тексте совпадения с шаблоном

выделяются изменением цвета. По умолчанию для этого используется синий цвет, но его можно заменить другим цветом на вкладке Preferences, доступ к которой возможен через меню Reggy.

- Приложение Rubular (`http://www.rubular.com`) — написанный Майклом Ловитом на языке Ruby онлайновый редактор регулярных выражений, поддерживающий версии Ruby 1.8.7 и 1.9.2.
- О четных числах, одним из которых является нуль, можно прочитать на сайте `http://mathworld.wolfram.com/EvenNumber.html`.
- Реализация регулярных выражений в Java (1.6) задокументирована на сайте `http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html`.
- Подробнее об организации IEEE и поддерживаемом ею семействе стандартов POSIX можно прочитать на сайте `http://www.ieee.org`.

ГЛАВА 6

Сопоставление с символами Unicode и другими символами

Сейчас у вас будет возможность заняться поиском совпадений с символами или диапазонами символов, выходящими за пределы стандартного набора ASCII. Стандарт ASCII (American Standard Code for Information Interchange) определяет кодировку символов, представляющую цифры, буквы латинского алфавита от A до Z в верхнем и нижнем регистре, знаки препинания, непечатные знаки и ряд других символов. Кодировка ASCII используется в течение многих лет — набор из 128 символов на основе латинского алфавита был стандартизирован еще в 1968 году. Это произошло задолго до появления персональных компьютеров, мыши и Интернета, но мне до сих пор приходится регулярно заглядывать в таблицу ASCII.

Вспоминаю, как много лет назад, когда моя профессиональная деятельность только начиналась, один из моих коллег-инженеров постоянно держал в своем портмоне таблицу кодов ASCII, чтобы она всегда была под рукой.

Поэтому я ни в коем случае не буду отрицать значения кодировки ASCII, хотя она и кажется устаревшей в наши дни, особенно после появления стандарта Unicode (`http://www.unicode.org`), текущая версия которого представляет свыше 100000 символов. В то же время Unicode не отбрасывает полностью ASCII; он включает ASCII в свою кодовую таблицу Basic Latin (`http://www.unicode.org/charts/PDF/U0000.pdf`).

Текст, с которым мы будем работать, находится в файле *voltaire.txt* из архива примеров и представляет собой высказывание Вольтера (1694–1778), французского философа эпохи Просвещения:

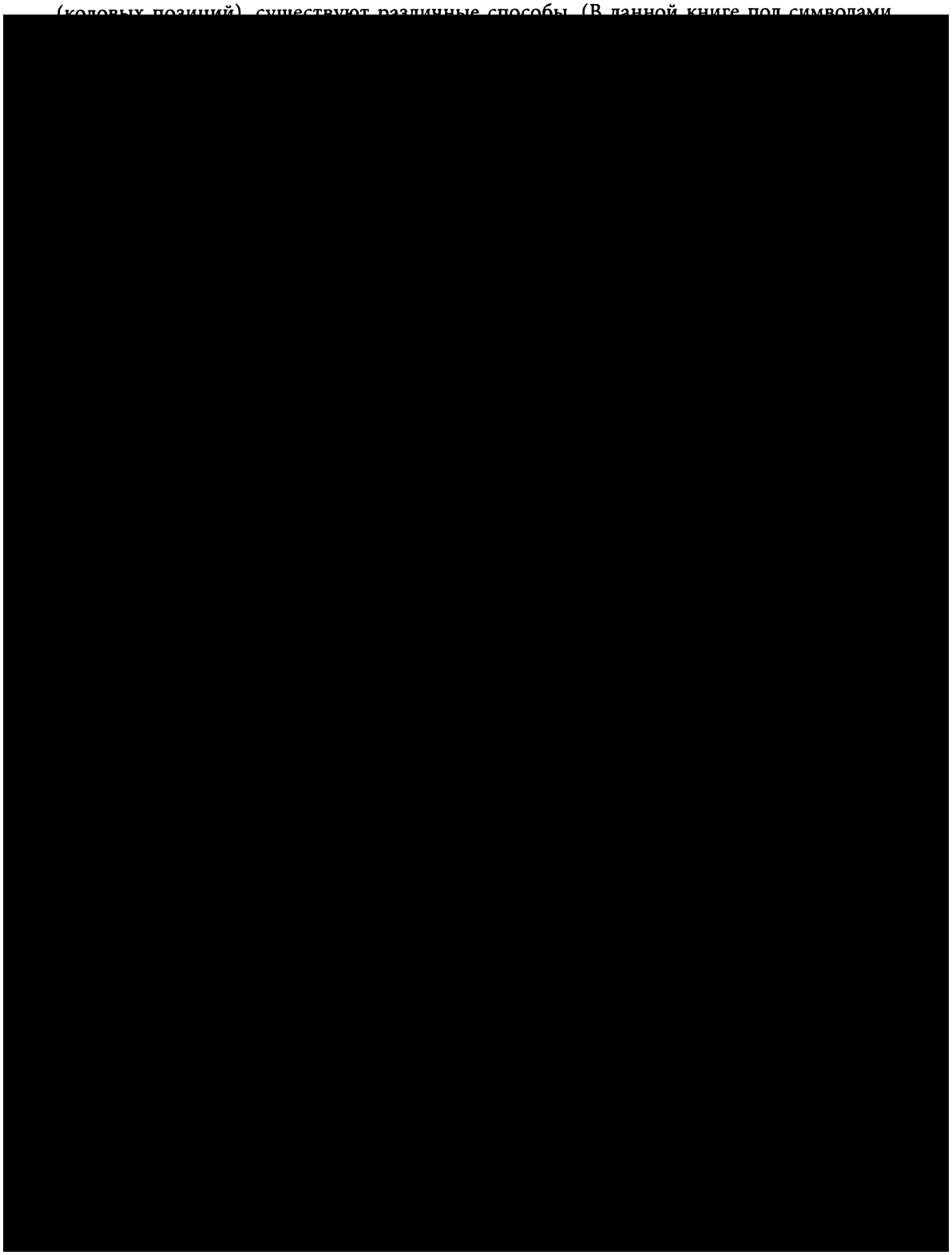
> Qu'est-ce que la tolérance? C'est l'apanage de l'humanité. Nous sommes tous pétris de faiblesses et d'erreurs; pardonnons-nous réciproquement nos sottises, c'est la première loi de la nature.

В переводе на русский язык оно звучит так:

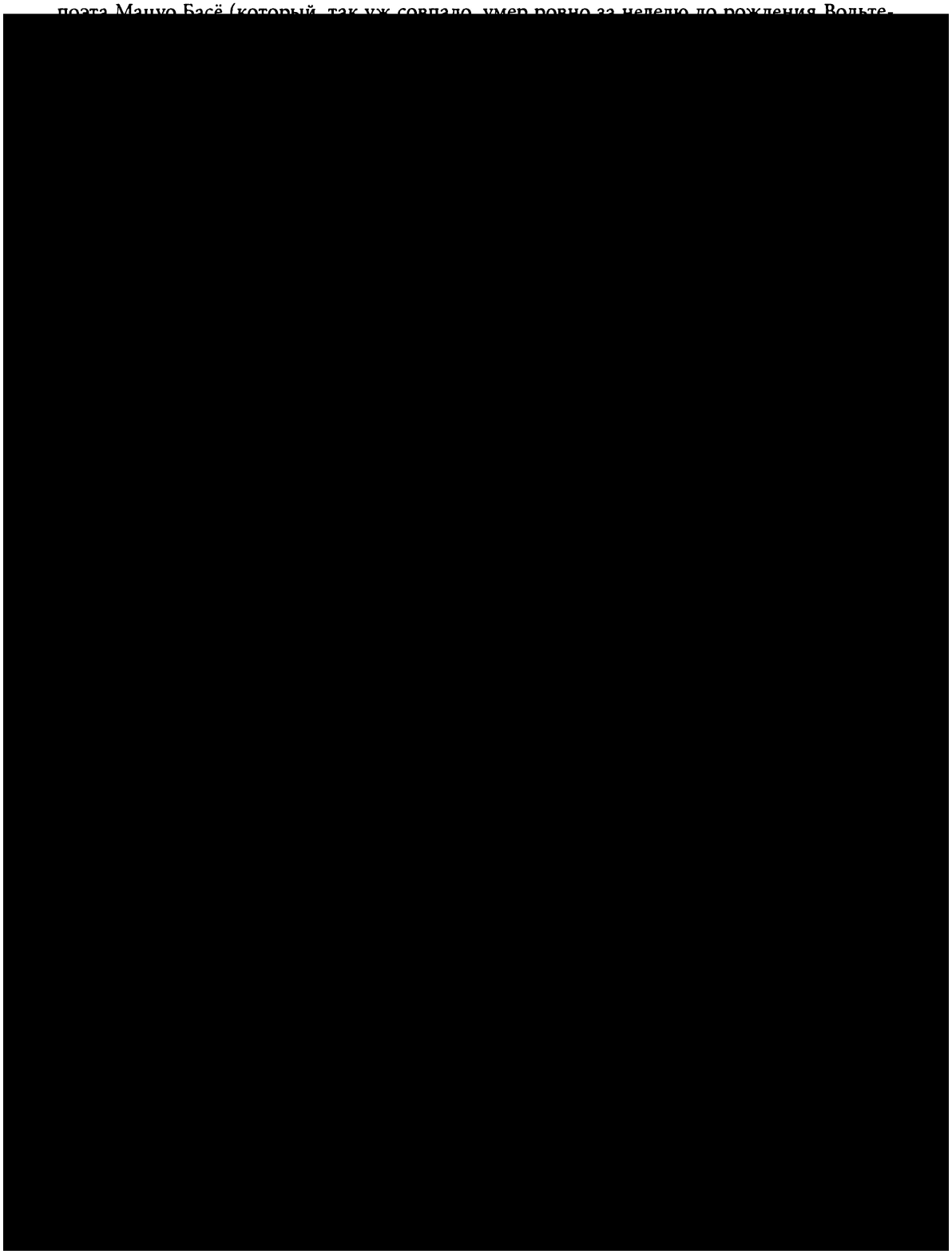
> Что такое терпимость? Это достояние человечества. Все мы слабы и полны заблуждений; взаимно прощать друг другу наши глупости является первым естественным законом.

Сопоставление с символами Unicode

Для указания символов Unicode, известных также под названием *кодовых точек* (кодовых позиций), существуют различные способы. (В данной книге под символами



и использует несколько отличающийся синтаксис. Скопируйте содержимое файла *basho.txt* в текстовую область Target String (Тестируемая строка). В этом файле хранится отрывок из поэмы "Старый пруд", считающейся самым великим произведением японского поэта Мацуо Басё (который, так уж совпало, умер ровно за неделю до рождения Вольте-



Продолжите выполнение данного примера, попытавшись найти соответствие длинному тире (—) с помощью следующего выражения:

```
\u2014
```

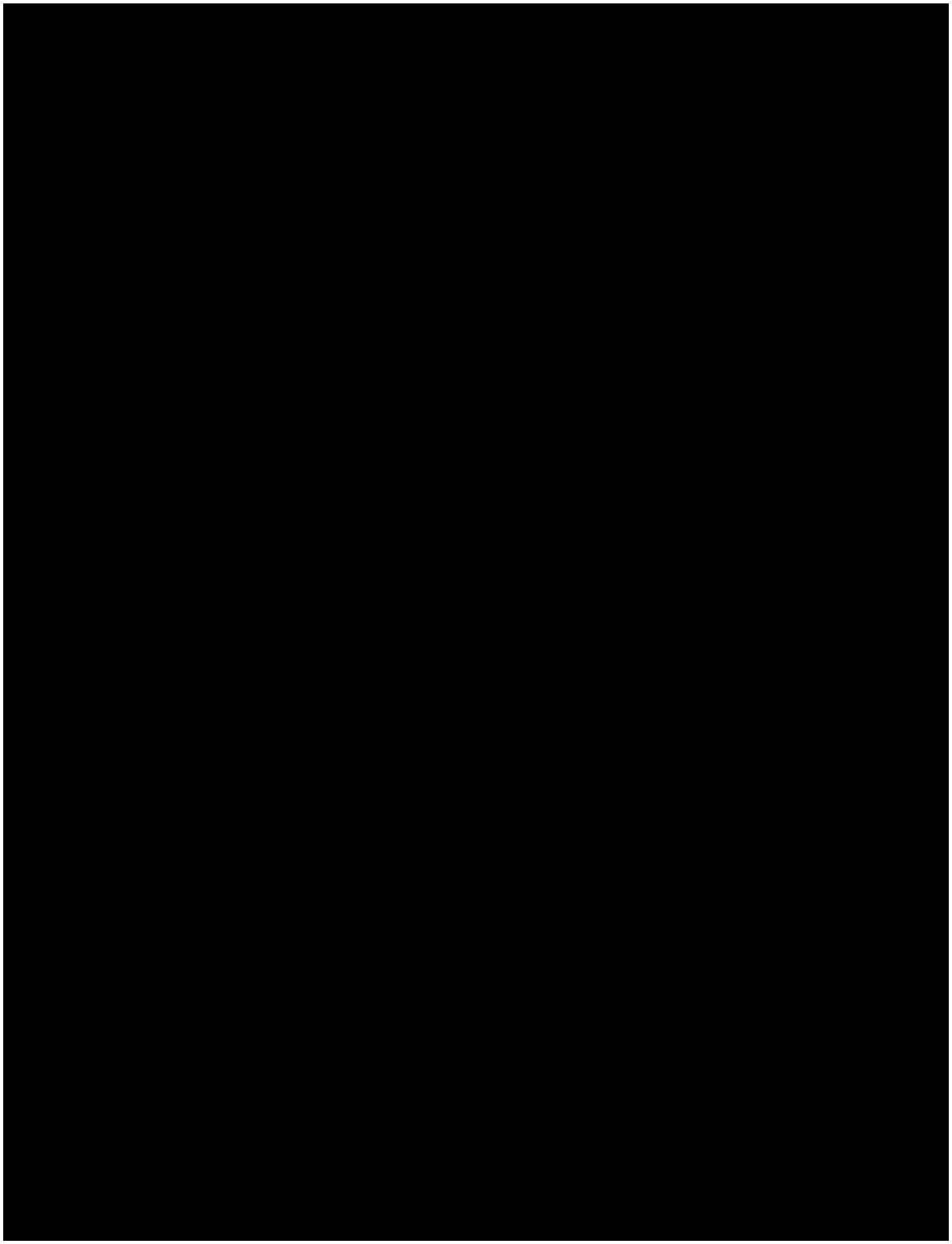


Таблица 6.1. Сопоставление с символами Unicode в редакторе vim

Первый символ	Максимальное количество знаков кода	Максимальное значение
x или X	2	255 (FF)
u	4	65535 (FFFF)
U	8	2147483647 (7FFFFFFF)

Поиск соответствий восьмеричным кодам символов

В шаблонах можно использовать восьмеричные коды символов, которые могут содержать цифры от 0 до 7. При этом числовому коду должен предшествовать символ обратной косой черты (\), а при работе с редактором *vim* — дополнительно знак процента и o (\%o). В последнем случае дополнительный знак процента требуется также при использовании кодировки Unicode[1].

Например, выражение

```
\351
```

или

```
\%o351
```

приводит к тому же результату, что и выражение

```
\u00e9
```

или

```
\%u00e9
```

Проведите эксперимент с текстом Вольтера в приложении RegexPal. Вы обнаружите, что коду `\351` также соответствует буква *é*, но при этом он короче своего шестнадцатеричного аналога.

Поиск соответствий свойствам символов Unicode

В некоторых реализациях регулярных выражений, например в Perl, при сопоставлении текста с шаблоном допускается использовать свойства символов Unicode. К числу таких свойств относится, например, принадлежность символа к буквам, цифрам или знакам пунктуации.

Сейчас я познакомлю вас с *ack* — инструментом командной строки, написанным для Perl, который во многом напоминает *grep* (`http://betterthangrep.com`). Маловероятно, что он установлен в вашей системе, поэтому вы должны загрузить и установить его самостоятельно (см. раздел "На заметку").

Для работы с этим инструментом я выбрал в качестве примера отрывок из оды "К радости" ("An die Freude"), написанной (разумеется, на немецком языке) Фридрихом Шиллером в 1785 году.

[1] См. также обсуждение по адресу `http://www.oreilly.com/catalog/errata.csp?isbn=9781449392680`. — *Примеч. ред.*

An die Freude.

Freude, schöner Götterfunken,
Tochter aus Elisium,
Wir betreten feuertrunken
Himmlische, dein Heiligthum.
Deine Zauber binden wieder,
was der Mode Schwerd getheilt;
Bettler werden Fürstenbrüder,
wo dein sanfter Flügel weilt.

Seid umschlungen, Millionen!
Diesen Kuß der ganzen Welt!
Brüder, überm Sternenzelt
muß ein lieber Vater wohnen.

В этом отрывке есть несколько интересных символов, выходящих за пределы области кодов ASCII. Для выполнения операций поиска в этом тексте мы будем использовать свойства. (Если вас интересует перевод данного фрагмента, можете найти его в Википедии: `https://ru.wikipedia.org/wiki/Ода_к_радости`)

Используя *ack* в командной строке, вы, в частности, сможете задать поиск символов, обладающих тем свойством (**p**roperty), что они являются буквами (**L**etter):

```
ack '\pL' schiller.txt
```

Эта команда выделит в тексте все буквы. Если вас интересуют буквы нижнего (**l**ower) регистра, воспользуйтесь следующей командой:

```
ack '\p{Ll}' schiller.txt
```

Фигурные скобки обязательны. В случае букв верхнего (**u**pper) регистра команда будет выглядеть так:

```
ack '\p{Lu}' schiller.txt
```

Для указания символов, не обладающих заданным свойством, используется буква *P* верхнего регистра:

```
ack '\PL' schiller.txt
```

Эта команда выделит в тексте все символы, не являющиеся буквами.

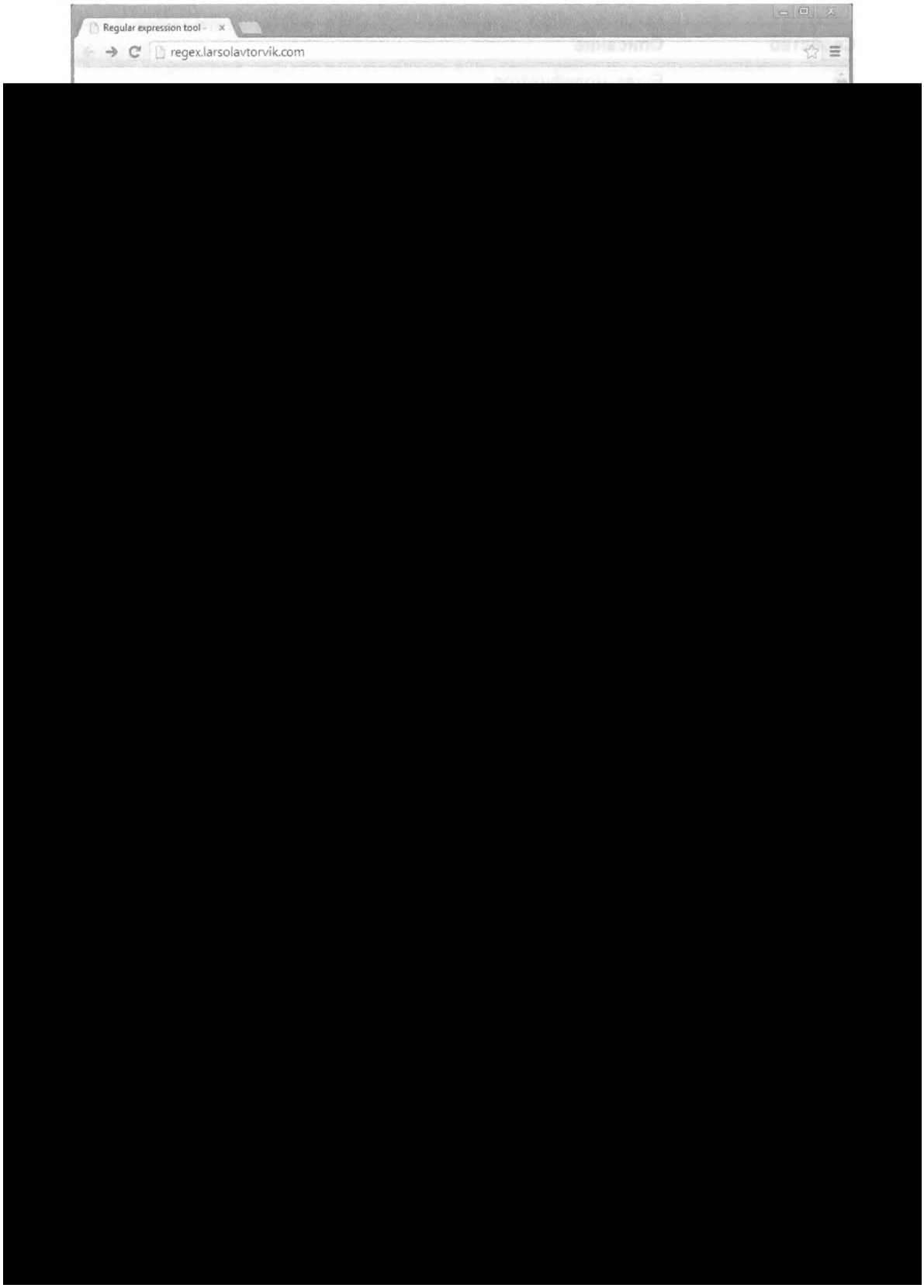
Следующая команда выделит все символы, не являющиеся буквами нижнего регистра:

```
ack '\P{Ll}' schiller.txt
```

А эта команда выделит все символы, не являющиеся буквами верхнего регистра:

```
ack '\P{Lu}' schiller.txt
```

Можете повторить эти операции в другом онлайновом тестере, перейдя в браузере по адресу `http://regex.larsolavtorvik.com`. На рис. 6.4 показан шиллеровский текст, в котором аналогичным образом были выделены все буквы нижнего регистра.


Regular expression tool
regex.larsolavtorvik.com

Окончание табл. 6.2

Свойство	Описание
Lm	Буква-модификатор
Lo	Другая буква
Lt	Прописная буква
Lu	Буква верхнего регистра
L&	Ll, Lu или Lt
M	Знак
Mc	Внутристрочный знак
Me	Заключающий знак
Mn	Внестрочный знак
N	Цифра
Nd	Десятичное число
Nl	Буквенная цифра
No	Другая цифра
P	Знак препинания
Pc	Соединительный знак препинания
Pd	Тире
Pe	Закрывающий знак препинания
Pf	Завершающий знак препинания
Pi	Начальный знак препинания
Po	Другой знак препинания
Ps	Открывающий знак препинания
S	Символ
Sc	Символ валюты
Sk	Символ-модификатор
Sm	Математический символ
So	Другой символ
Z	Разделитель
Zl	Разделитель строк
Zp	Разделитель абзацев
Zs	Разделительный пробел

Поиск соответствий управляющим символам

Какие выражения следует использовать для сопоставления с управляющими символами? Необходимость в поиске управляющих символов возникает не так уж часто, но знание того, как это делается, вам не помешает. В архиве примеров находится файл *ascii.txt*, состоящий из 128 строк, которые содержат все ASCII-символы, по одному в каждой строке (отсюда и 128 строк). Поэтому в случае успешного поиска вы обычно будете получать одну строку. Этот файл отлично подходит для целей тестирования.

При поиске строк или управляющих символов в файле *ascii.txt* с помощью утилиты *grep* или *ack* обе они могут интерпретировать этот файл как двоичный. В таком случае при обнаружении совпадения каждая из них может всего лишь вывести сообщение "Binary file ascii.txt matches" ("Обнаружено совпадение в двоичном файле ascii.txt"), и на этом все закончится.

В регулярных выражениях управляющие символы можно указывать в следующем виде:

```
\cx
```

где *x* — интересующий вас управляющий символ.

Предположим, вы хотите найти в файле пустой символ (NULL). В Perl это можно сделать с помощью следующей команды:

```
perl -n -e 'print if /\c@/' ascii.txt
```

Результат должен быть таким:

```
0. Null
```

Поскольку пустой символ — непечатный, вы не сможете увидеть его, но он есть в файле.

Если вы откроете файл *ascii.txt* не в *vim*, а в каком-то другом редакторе, то он, вероятно, удалит управляющие символы из файла, поэтому я не рекомендую вам этого делать.

Для поиска пустого символа можно также использовать *escape*-последовательность `\0`. Проверьте это, выполнив следующую команду:

```
perl -n -e 'print if /\0/' ascii.txt
```

Продолжая в том же духе, найдите символ BEL (звуковой сигнал) с помощью такой команды:

```
perl -n -e 'print if /\cG/' ascii.txt
```

Эта команда вернет следующую строку:

```
7. Bell
```

Этот же результат можно получить, используя аббревиатуру `\a`:

```
perl -n -e 'print if /\a/' ascii.txt
```

Для поиска символа ESCAPE (альтернативный регистр №2, AP2) используйте такую команду:

```
perl -n -e 'print if /\c[/' ascii.txt
```

приводящую к следующему результату:

```
27. Escape
```

или эквивалентную команду, в которой используется аббревиатура \e:

```
perl -n -e 'print if /\e/' ascii.txt
```

А как насчет символа BACKSPACE (возврат на шаг)? Проверьте:

```
perl -n -e 'print if /\cH/' ascii.txt
```

Эта команда выдаст следующий результат:

```
8. Backspace
```

То же самое можно получить, используя скобочное выражение:

```
perl -n -e 'print if /[\b]/' ascii.txt
```

А как интерпретировалась бы последовательность символов \b, не используй мы скобок? Совершенно верно: как граница слова (см. главу 2).

Скобки изменяют способ обработки последовательности процессором. В данном случае благодаря им Perl воспринимает эту последовательность как символ BACKSPACE.

В табл. 6.3 сведены все коды, которые мы использовали в этой главе для сопоставления с различными символами.

Таблица 6.3. Коды для сопоставления с символами Unicode и другими символами

Код	Описание
\uxxxx	Unicode (4 позиции)
\xxx	Unicode (2 позиции)
\Uxxxxxxxx	Unicode (8 позиций)
\xxx	Восьмеричный код
\cx	Управляющий символ
\0	Пустой символ
\a	Звуковой сигнал
\e	Escape
[\b\]	Возврат на шаг (BACKSPACE)

На этом данная глава завершается. В следующей главе вы поближе познакомитесь с квантификаторами.

О чем вы узнали в главе 6

- Как выполнить сопоставление с любым символом Unicode, используя выражения `\uxxxx` и `\xxx`.
- Как выполнить сопоставление с любым символом Unicode в редакторе *vim*, используя выражения `\%xxx`, `\%Xxx`, `\%uxxxx` и `\%Uxxxx`.
- Как выполнить сопоставление с символами, коды которых находятся в пределах диапазона 0–255, используя восьмеричный формат `\000`.
- Как использовать свойства символов Unicode с помощью последовательности `\o{x}`.
- Как выполнить сопоставление с управляющими символами, используя последовательности `\e` и `\cH`.
- Дополнительные сведения об использовании Perl в командной строке (дополнительные однострочные команды Perl).

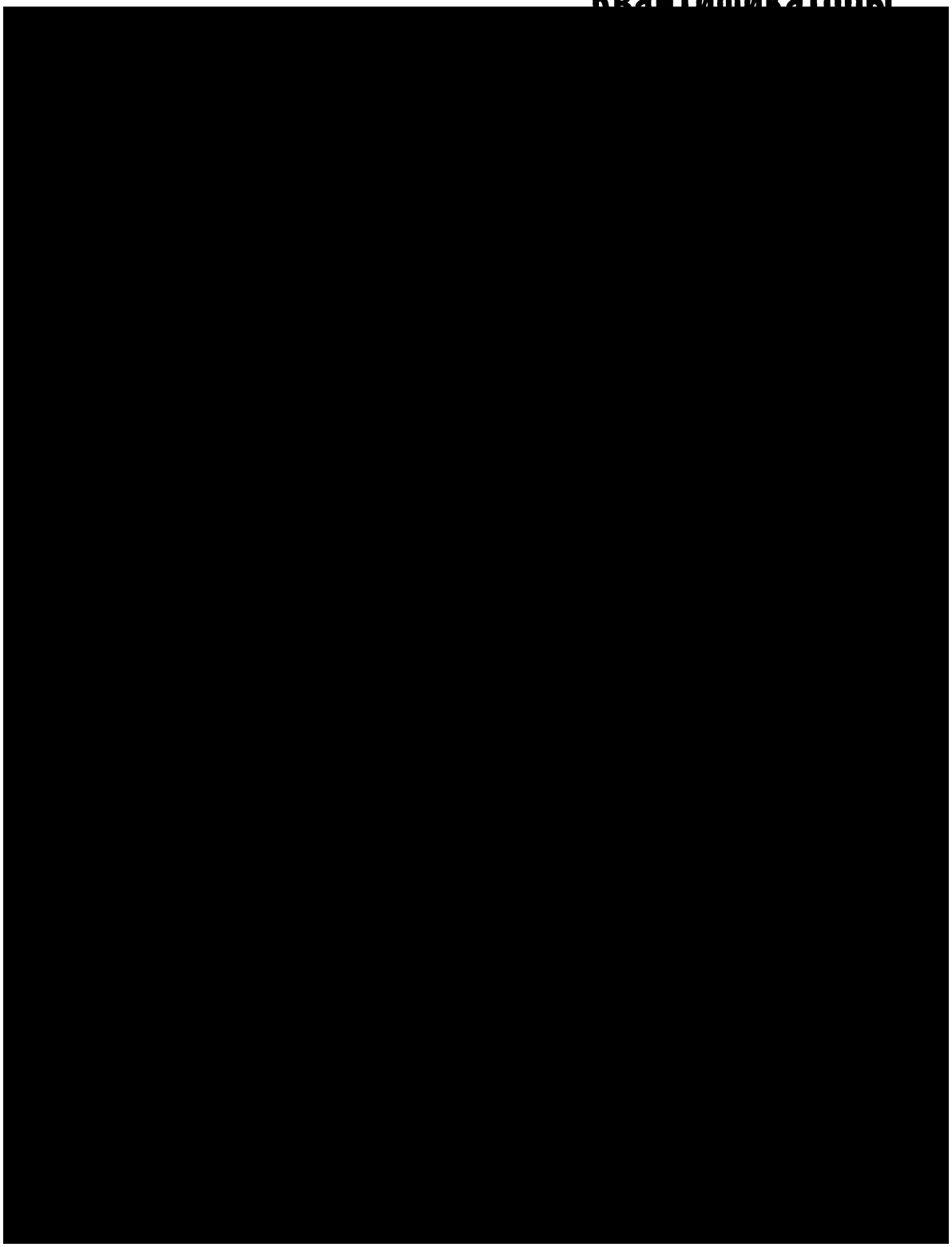
На заметку

- Для ввода управляющих символов в файл *ascii.txt* я использовал редактор *vim* (`http://www.vim.org`). В *vim* для этого надо сначала нажать комбинацию клавиш <Ctrl+V>, а затем комбинацию, соответствующую управляющей последовательности для данного символа, например <Ctrl+C> для символа конца текста (END OF TEXT). Для ввода двухзначных шестнадцатеричных кодов символов я также использовал комбинацию клавиш <Ctrl+V> с последующим вводом *x* и кода символа. Для ввода управляющих символов можно также использовать диграфы — пары букв, которые воспринимаются системами текстового поиска не как пара отдельных символов, а как единое целое. Чтобы увидеть список возможных кодов, введите в *vim* команду `:digraph`. Чтобы ввести диграф, необходимо нажать комбинацию клавиш <Ctrl+K> в режиме Insert (Вставка), а затем ввести двухбуквенное обозначение диграфа (например, *NU* для ввода пустого символа).
- Regex Hero (`http://regexhero.net/tester`) — реализация регулярных выражений в браузере для платформы .NET, написанная Стивом Уортамом. Это платный продукт, но вы можете бесплатно протестировать его и, если он вам понравится, приобрести по умеренной цене (существуют две версии продукта: стандартная и профессиональная).
- Редактор *vim* (`http://www.vim.org`) — усовершенствованная версия редактора *vi*, написанного Биллом Джоем. Основным разработчиком редактора *vim* был Брам Молинар. Непосвященным редактор *vim* покажется архаичным, но, как я уже отмечал, он обладает невероятно мощными возможностями.
- Утилита *ack* (`http://betterthangrep.com`) написана на языке Perl. Она работает приблизительно так же, как и утилита *grep*, и имеет ряд собственных опций командной строки, но превосходит *grep* во многих отношениях. Например, в ней используются регулярные выражения Perl, а не базовые регулярные

выражения *grep* (без опции `-E`). Инструкции по ее установке вы найдете по адресу `http://betterthangrep.com/install//`. Я следовал инструкциям, описанным в разделе "Install the ack executable", но при этом не использовал утилиту *curl*, а просто загрузил *ack*, воспользовавшись предоставленной там ссылкой, и скопировал сценарий в каталог */usr/bin* как на своем компьютере Mac, так и на компьютере, работающем под управлением Windows 7 с установленной программой Cygwin (`http://www.cygwin.com`).

ГЛАВА 7

Квантификаторы



Жадный, ленивый и сверхжадный поиск

Несмотря на то что прилагательные, стоящие в заголовке, отражают не самые лучшие человеческие качества, они представляют интересные свойства квантификаторов, которые вы обязательно должны понимать, если хотите работать с регулярными выражениями на профессиональном уровне.

Обычный квантификатор изначально является "жадным" (greedy). Жадный квантификатор в первую очередь стремится обнаружить совпадение со всей строкой. Он захватывает столько, сколько может, целую строку, пытаясь обнаружить совпадение. Если первая попытка оказывается неудачной, он возвращается на один символ и повторяет попытку. Этот процесс называют *бектрекингом* или *поиском с возвратом*. Пошаговый возврат на один символ продолжается до тех пор, пока не будет обнаружено совпадение или не исчерпаются все символы. Кроме того, жадный квантификатор сохраняет информацию о каждом своем действии и поэтому потребляет наибольшее количество ресурсов по сравнению с двумя другими разновидностями квантификаторов, к рассмотрению которых мы переходим.

Ленивый (lazy) квантификатор (другое название — *нежадный*) действует иначе. Он пытается обнаружить совпадение, начиная с самого первого символа тестируемой строки. Далее он просматривает всю строку по одному символу за раз в поиске совпадения. Наконец, он пытается найти совпадение со всей строкой. Чтобы превратить обычный квантификатор в ленивый, перед ним необходимо записать вопросительный знак (?). Ленивый квантификатор "пережевывает" строку небольшими порциями.

Сверхжадный (possessive) квантификатор пытается найти совпадение, захватывая целиком всю строку, но делает это в рамках одной попытки, не прибегая к поиску с возвратом. Чтобы превратить обычный квантификатор в сверхжадный, перед ним необходимо записать знак "плюс" (+). Сверхжадный квантификатор не тратит время на "пережевывание" строки, а "глотает" ее целиком и только после этого пытается понять, а что же именно он "съел". Перейдем к более подробному рассмотрению каждого из вышеперечисленных типов квантификаторов по отдельности.

Сопоставление с использованием квантификаторов *, + и ?

Вы уже перенесли в Reggy треугольник из цифр? Тогда мы можем приступить к тестированию. Прежде всего, используем *звезду Клини* — символ "звездочка" (*), названный так в честь Стивена Клини, разработавшего формальную теорию регулярных выражений. Если записать звездочку вслед за точкой:

```
.*
```

то это выражение, будучи жадным, совпадет со всеми символами (в данном случае — цифрами) тестируемого текста. Как вам уже известно из предыдущих глав, последовательности `.*` соответствует любой одиночный символ, повторяющийся нуль или более раз. Таким образом, в результате применения этого выражения к тексту в нижней области окна выделенным окажется весь текст. В одном из ранних руководств по регулярным выражениям о звезде Клини сказано следующее:

> Регулярному выражению, за которым следует символ "*" [звезда Клини], соответствует любое количество (включая нулевое) следующих подряд вхождений текста, соответствующего данному регулярному выражению.

Если вы попробуете применить выражение

```
9
```

то увидите, что выделится весь нижний ряд девяток. Однако в результате применения выражения

```
9.*
```

выделится не только ряд девяток, но и ряд нулей под ним. Почему это произошло? Потому что флажок Multiline (в нижней части окна приложения) установлен, и поэтому символы новой строки также будут соответствовать точке, чего не будет происходить при снятом флажке.

Для поиска совпадений с одной или несколькими девятками введите такое выражение:

```
9+
```

Чем это выражение отличается от предыдущего? Основное отличие состоит в том, что квантификатор + ищет по крайней мере одно вхождение цифры *9*, тогда как квантификатор * ищет нуль или более вхождений.

Следующему выражению соответствует нуль или одно (необязательное) вхождение цифры *9*:

```
9?
```

Этому выражению соответствует лишь первое вхождение цифры *9*. Она указана как необязательная во второй позиции, но поскольку она уже существует в проверяемом тексте в первой позиции, она включается в соответствие и выделяется цветом. Если вы используете следующее выражение:

```
99?
```

то выделится как первая, так и вторая цифра *9*.

В табл. 7.1 приведены базовые квантификаторы вместе с описанием их функций. Все эти квантификаторы по умолчанию *жадные*, т.е. они пытаются найти совпадение с максимально возможным количеством символов с первой попытки.

Таблица 7.1. Базовые квантификаторы

Синтаксис	Описание
?	Предыдущий символ или группа символов может встретиться в данной позиции нуль или один раз
+	Предыдущий символ или группа символов может встретиться в данной позиции один или более раз
*	Предыдущий символ или группа символов может встретиться в данной позиции нуль или более раз

Соответствие заданному количеству повторений символа

Фигурные скобки позволяют создавать шаблоны, которым соответствует определенное количество повторений символа, попадающее в указанный диапазон значений. Если поведение этих квантификаторов не изменено с помощью модификаторов, они ведут себя как жадные. Например, выражению

```
7{1}
```

будет соответствовать первое вхождение цифры *7*. Если вы хотите, чтобы этому выражению соответствовало *одно или более* вхождение цифры 7, в него необходимо добавить запятую:

```
7{1,}
```

Вероятно, вы уже догадались, что выражения

```
7+
```

и

```
7{1,}
```

фактически означают одно и то же. Точно так же одинаковый смысл имеют выражения

```
7*
```

и

```
7{0,}
```

То же самое остается справедливым и в отношении пары выражений

```
7?
```

и

```
7{0,1}
```

Для указания некоторого диапазона допустимого количества повторений, когда цифра, символ или группа может встречаться не менее *m* раз, но не более *m* раз, используйте выражения следующего типа:

```
7{3,5}
```

Данному выражению будут соответствовать три, четыре или пять следующих подряд цифр 7.

Как вы имели возможность убедиться, синтаксис, предполагающий использование диапазонов, задаваемых с помощью фигурных скобок, является наиболее гибким. Сводка элементов этого синтаксиса приведена в табл. 7.2.

Таблица 7.2. Синтаксис фигурных скобок

Синтаксис	Описание
`{n}`	Соответствует ровно n следующим подряд вхождениям предыдущего символа или группы
`{n,}`	Соответствует n или более следующим подряд вхождениям предыдущего символа или группы

Окончание табл. 7.2

Синтаксис	Описание
`{m,n}`	Соответствует не менее чем n и не более чем m следующим подряд вхождениям предыдущего символа или группы
`{0,1}`	То же самое, что и `?` (отсутствие или однократное вхождение предыдущего символа или группы)
`{1,0}`	То же самое, что и + (одно или более следующих подряд вхождений предыдущего символа или группы)
`{0,}`	То же самое, что и `\*` (отсутствие или любое количество следующих подряд вхождений предыдущего символа или группы)

Ленивые квантификаторы

Давайте избавимся от жадности и предадимся лени. Вам будет гораздо легче во всем разобраться, если мы обратимся к конкретным примерам. Прежде всего убедитесь в том, что в приложении Reggy установлен флажок Match All, и попытайтесь найти совпадение с нулевым или единичным количеством повторений цифры 5, используя одиночный вопросительный знак (`?`):

```
5?
```

Выделится первая цифра *5*. Добавьте дополнительный знак `?`, чтобы сделать квантификатор ленивым:

```
5??
```

В этом случае ни один символ не выделяется. Причина заключается в том, что данный шаблон — ленивый, и ничто не вынуждает его выделить первую цифру *5*. По самой своей сути *ленивые* выражения стремятся совпасть с как можно меньшим допустимым количеством символов. На то они и лентяи!

Если вы попытаетесь использовать следующее выражение:

```
5*?
```

то убедитесь в том, что ни одна из цифр не будет выделена, поскольку символ `?` позволяет шаблону считать соответствие найденным, даже если цифра *5* в него не включена, что и произошло.

Проверьте, к чему приведет следующее выражение:

```
5+?
```

Увидели? Ленивец оторвался от дивана и нашел одну цифру *5*. Это было все, что от него требовалось, чтобы выполнить рабочую норму.

Разнообразим пример, используя диапазонный поиск *m,n*:

```
5{2,5}?
```

Это выражение совпадет всего лишь с двумя цифрами `5`, чего и следовало ожидать от ленивого поиска.

Полный перечень ленивых квантификаторов приведен в табл. 7.3. В каких случаях полезны ленивые выражения? Используйте их тогда, когда требуется найти минимально допустимое количество повторений символа, а не максимально возможное.

Таблица 7.3. Ленивые квантификаторы

Синтаксис	Описание
`??`	Ленивый поиск, при котором предыдущий символ или группа может встретиться в данной позиции нуль или один раз
`+?`	Ленивый поиск, при котором предыдущий символ или группа может встретиться в данной позиции один или более раз
`*?`	Ленивый поиск, при котором предыдущий символ или группа может встретиться в данной позиции нуль или более раз
`{n}?`	Ленивый поиск ровно *n* повторений предыдущего символа или группы
`{n,}?`	Ленивый поиск *n* или более повторений предыдущего символа или группы
`{m,n}?`	Ленивый поиск не менее чем *n* и не более чем *m* повторений предыдущего символа или группы

Сверхжадные квантификаторы

Сверхжадный поиск, как и жадный, пытается обнаружить совпадение с как можно большим количеством символов, однако, в отличие от него, не использует перебор с возвратом (бектрекинг). Сверхжадное выражение не отдает ничего из того, с чем ему удается совпасть. Оно исключительно эгоистично. Именно поэтому его и называют *сверхжадным*. Заняв боевую стойку, оно не отдает ни пяди завоеванной территории. В сверхжадных квантификаторах положительно то, что, благодаря выполнению поиска без возврата, они работают быстрее жадных квантификаторов, а если даже и не находят совпадений, то это также обнаруживается очень быстро.

Честно говоря, приведенные в книге примеры вряд ли позволят вам ощутить в полной мере различия в поведении жадных, ленивых и сверхжадных квантификаторов. Но, по мере того как вы будете приобретать опыт и вам придется задумываться о производительности, знание этих различий будет иметь для вас важное значение.

Чтобы это понять, сравните работу двух выражений, в одном из которых литеральный нуль указан в начале, а в другом — в конце выражения. Убедитесь в том, что в приложении Reggy установлен флажок Match All, и введите следующее выражение (начальный нуль):

```
0.*+
```

Что при этом происходит? Выделяется вся строка с нулями. Совпадение было найдено. Создается впечатление, будто сверхжадный поиск приводит к тому же результату,

что и жадный, однако есть одна тонкость: он не сопровождается перебором с возвратом. Сейчас мы это докажем. Введите следующее выражение (конечный нуль):

```
.*+0
```

Совпадения отсутствуют. Объясняется это тем, что не выполнялся поиск с возвратом. Это выражение "проглотило" весь ввод, даже не подумав о том, чтобы оглянуться назад. Оно растратило свое наследство из-за разгульного образа жизни и не смогло найти конечный нуль. Оно просто не знает, где его искать. Если же вы уберете знак +, то получаемое при этом следующее выражение сможет найти все нули, поскольку оно будет делать это уже в режиме жадного сопоставления:

```
.*0
```

Сверхжадные квантификаторы имеет смысл использовать в тех случаях, когда вам известно, что собой представляет тестируемый текст, и вы знаете, где следует искать совпадения. В подобных ситуациях аппетит квантификаторов этого типа вас не может смутить, поскольку он обеспечивает более высокую производительность операций поиска. Перечень сверхжадных квантификаторов приведен в табл. 7.4.

Таблица 7.4. Сверхжадные квантификаторы

Синтаксис	Описание
`?+`	Сверхжадный поиск, при котором предыдущий символ или группа может встретиться в данной позиции нуль или один раз
`++`	Сверхжадный поиск, при котором предыдущий символ или группа может встретиться в данной позиции один или более раз
`*+`	Сверхжадный поиск, при котором предыдущий символ или группа может встретиться в данной позиции нуль или более раз
`{n}+`	Сверхжадный поиск ровно n повторений предыдущего символа или группы
`{n,}+`	Сверхжадный поиск n или более повторений предыдущего символа или группы
`{m,n}+`	Сверхжадный поиск не менее чем n и не более чем m повторений предыдущего символа или группы

Следующая глава посвящена обсуждению различных видов контекстной проверки текста.

О чем вы узнали в главе 7

- Чем различаются между собой жадный, ленивый и сверхжадный квантификаторы.
- Как найти одно или более вхождений символа (+).
- Как найти необязательный символ (отсутствие или однократное вхождение символа, ?).

- Как найти отсутствие или многократное вхождение символа (`\*`).
- Как используются квантификаторы `{m,n}`.
- Как используются жадные, ленивые (нежадные) и сверхжадные квантификаторы.

На заметку

Приведенная в начале главы цитата взята из следующего источника:

- Ken Thompson, *QED Text Editor* (Murray Hill, NJ, Bell Labs, 1970), p. 3 (`http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf`).

ГЛАВА 8

Группы проверки

Группы проверки (lookarounds), называемые также *группами просмотра*, — это группы без захвата, которые сопоставляют текст с шаблоном с учетом того, что именно они находят впереди или позади шаблона. Группы проверки также считаются *условиями с нулевой длиной совпадения* (zero-width assertions).

К группам проверки относятся:

- группы с положительной опережающей проверкой;
- группы с отрицательной опережающей проверкой;
- группы с положительной ретроспективной проверкой;
- группы с отрицательной ретроспективной проверкой.

В этой главе будет продемонстрировано, как работает каждый из перечисленных видов проверки. Мы начнем с использования настольного приложения RegExr, но затем переключимся на Perl и *ack* (утилите *grep* о группах проверки ничего не известно). Нашим рабочим текстом по-прежнему будет уже набившая вам оскомину поэма Кольриджа.

Положительная опережающая проверка

Предположим, вы хотите найти все вхождения слова *ancyent*, за которым следует слово *marinere* (я использую эти слова в том архаичном написании, в каком они встречаются в файле).

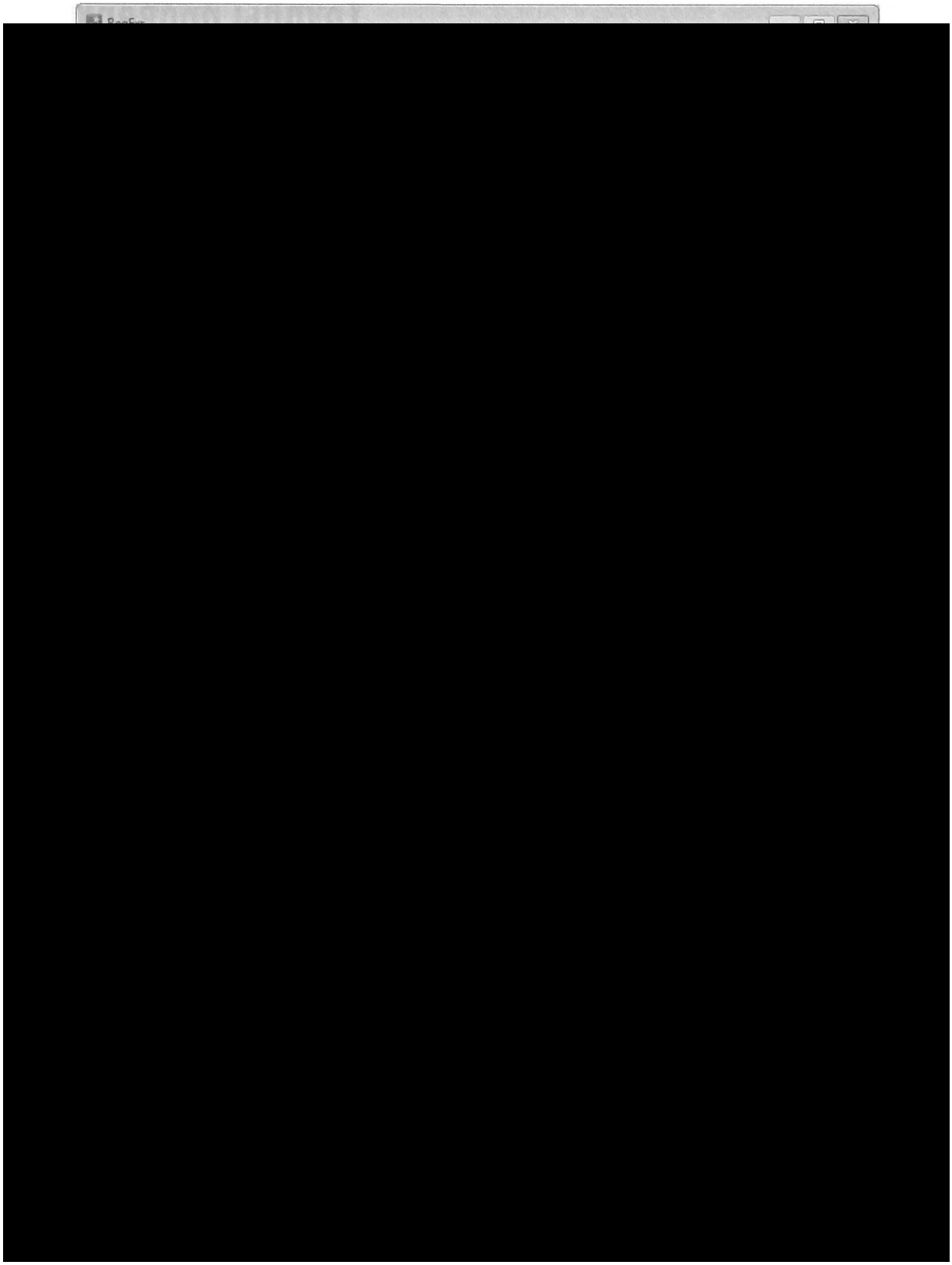
Прежде всего посмотрим, как все это работает в приложении RegExr. Введите в верхнее текстовое поле следующий нечувствительный к регистру букв (опция *i*) шаблон:

```
(?i)ancyent (?=marinere)
```

В приложении RegExr можно задать нечувствительность шаблона к регистру букв другим способом. Для этого достаточно установить флажок ignoreCase, щелкнув на квадратике рядом с надписью. Можете использовать тот способ, который для вас наиболее удобен.

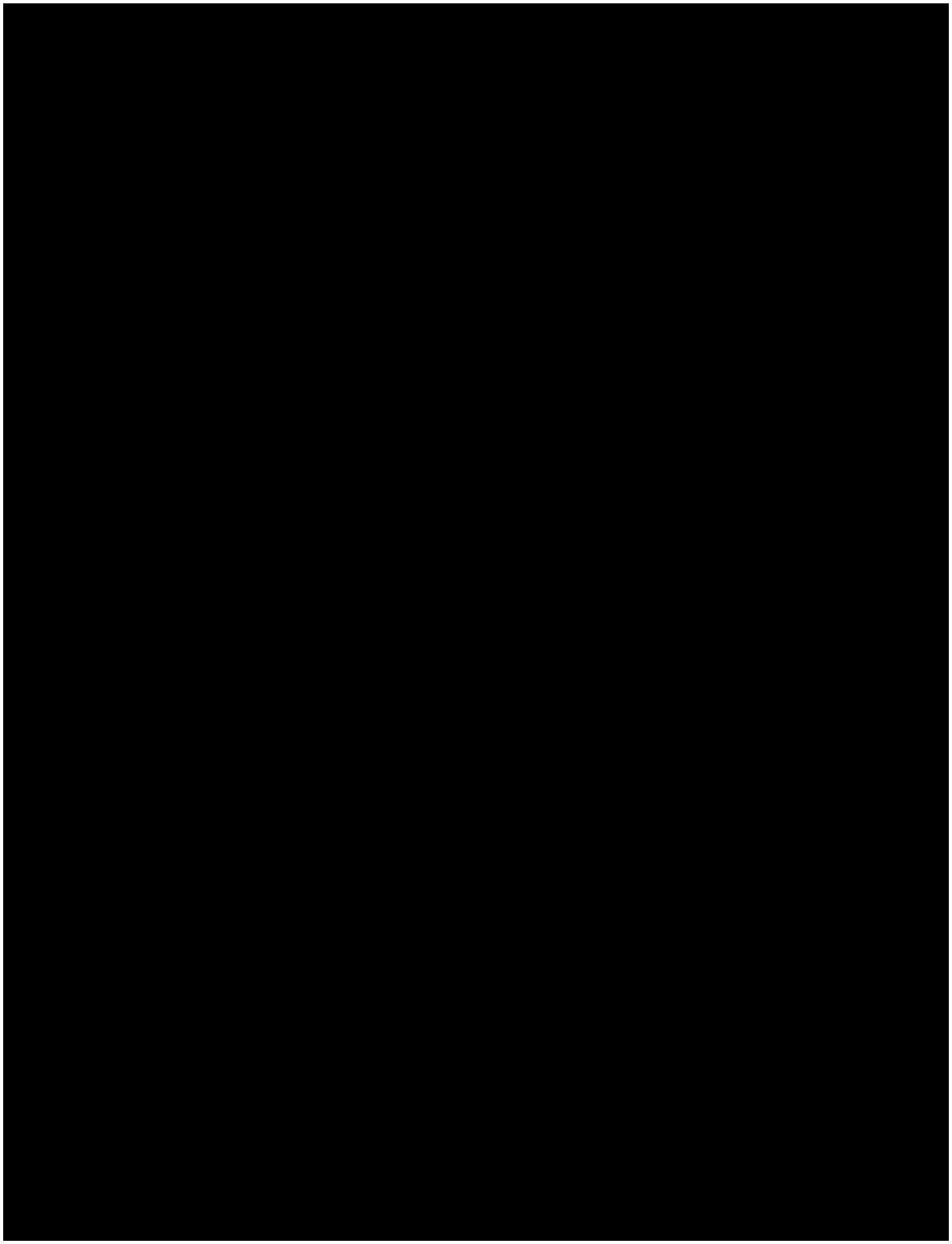
Поскольку выбран нечувствительный к регистру режим (`?i`), вы не должны беспокоиться о том, какой регистр букв использовать в шаблоне. Ваша цель — найти строки, в которых имеется слово *ancyent*, за которым следует слово *marinere*. Результат выделится цветом в текстовой области ниже шаблона (рис. 8.1), но выделена будет лишь первая

часть шаблона (*ancyent*), тогда как шаблон опережающей проверки (*Marinere*) останется невыделенным.



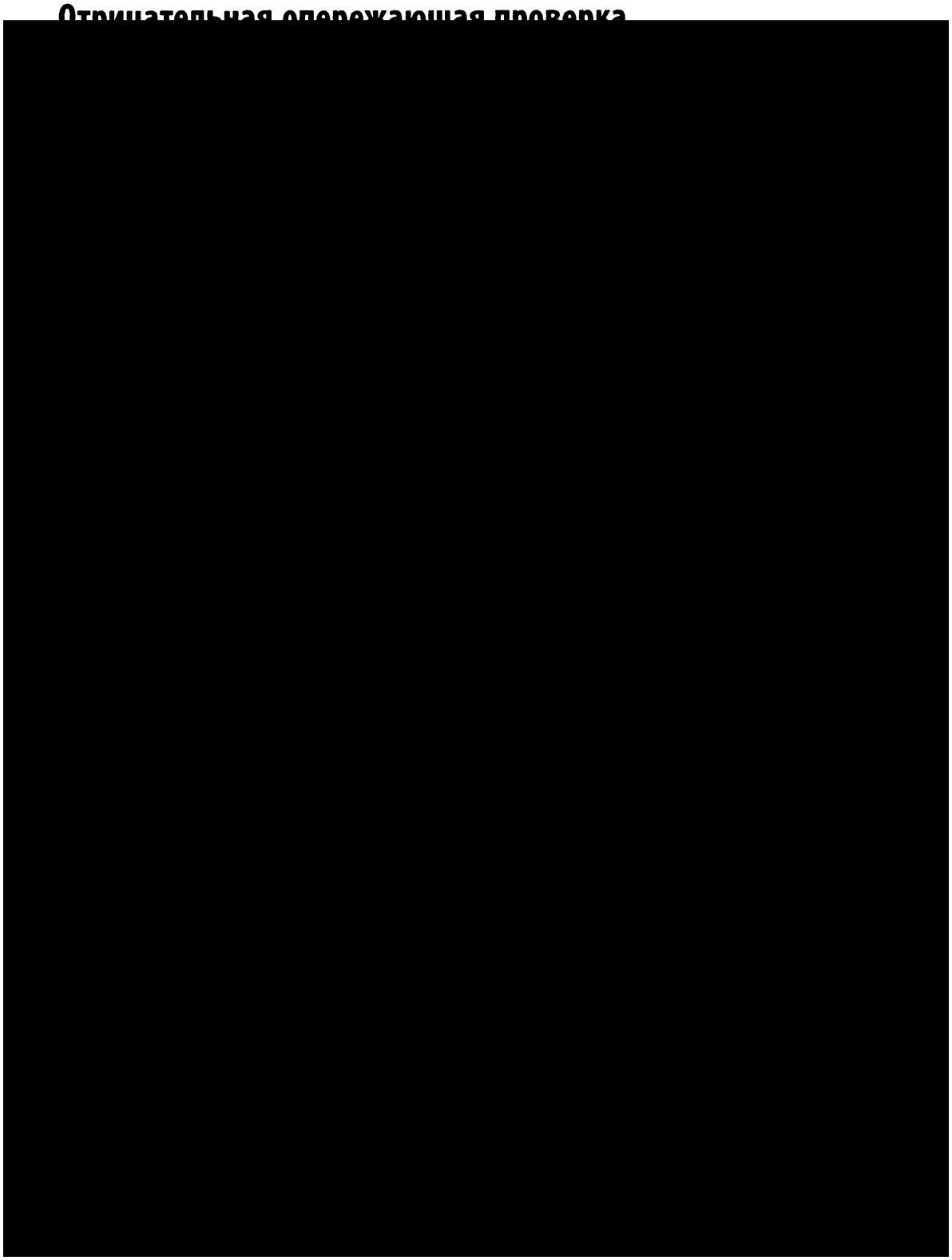
следующее за словом *ancyent*, с буквы *m*, независимо от ее регистра? Это можно сделать с помощью следующей команды:

```
perl -ne 'print if /(?1)ancyent (?=m)/' rime.txt
```



Это в некотором смысле обходной путь, отключающий выделение найденных совпадений, однако он работает.

Отрицательная опережающая проверка



В Perl аналогичную отрицательную опережающую проверку можно выполнить с помощью следующей команды:

```
perl -ne 'print if /(?i)ancyent (?!marinere)/' rime.txt
```

Она приводит к такому результату.

```
And thus spake on that ancyent man,
And thus spake on that ancyent Man,
```

Тот же результат может быть получен с помощью утилиты *ack*:

```
ack -i 'ancyent (?!marinere)' rime.txt
```

Положительная ретроспективная проверка

При *положительной ретроспективной проверке* (positive lookbehind; другое название — *просмотр назад*) целью сопоставления текста с шаблоном является нахождение искомого текста, которому предшествует определенный текст. Вот пример соответствующего синтаксиса:

```
(?i)(?<=ancyent) marinere
```

Положительной ретроспективной проверке соответствует знак "меньше чем" (<), сам вид которого подсказывает направление просмотра текста при проверке выполнения заданного условия. Используйте это выражение в RegExr, и вы сразу же увидите отличия в результате по сравнению с предыдущими примерами: вместо слова *ancyent* выделится слово *marinere*. Почему так происходит? Потому что левая часть шаблона представляет проверяемое условие и не включается в захватываемую часть результата сопоставления.

Чтобы сделать то же самое в Perl, используйте следующую команду:

```
perl -ne 'print if /(?i)(?<=ancyent) marinere/' rime.txt
```

А вот так это можно сделать с помощью утилиты *ack*:

```
ack -i '(?<=ancyent) marinere' rime.txt
```

Отрицательная ретроспективная проверка

Наконец, также возможна *отрицательная ретроспективная проверка* (negative lookbehind; другое название — *просмотр назад с отрицанием*). Догадываетесь, как она работает?

В этом случае условие поиска заключается в том, чтобы искомому тексту *не* предшествовал некоторый иной текст, а в качестве напоминания о направлении поиска вновь используется знак "меньше чем" (<).

Используйте в RegExr следующее выражение и посмотрите, какой при этом получается результат:

```
(?i)(?<!ancyent) marinere
```

Для просмотра всего текста воспользуйтесь полосой прокрутки.

Проделайте то же самое в Perl:

```
perl -ne 'print if /(?i)(?<!ancyent) marinere/' rime.txt
```

В полученном результате полностью отсутствуют какие-либо следы слова *ancyent*.

```
The Marinere hath his will.
The bright-eyed Marinere.
The bright-eyed Marinere.
The Marineres gave it biscuit-worms,
Came to the Marinere's hollo!
Came to the Marinere's hollo!
The Marineres all 'gan work the ropes,
The Marineres all return'd to work
The Marineres all 'gan pull the ropes,
"When the Marinere's trance is abated."
He loves to talk with Marineres
The Marinere, whose eye is bright,
```

Наконец, используйте утилиту *ack*:

```
ack -i '(?<!ancyent) marinere' rime.txt
```

На этом мы заканчиваем краткое рассмотрение одного из мощных средств современных регулярных выражений — проверки выполнения условий поиска в прямом и обратном направлении по отношению к потоку текста.

В следующей главе мы рассмотрим развернутый пример разметки документа тегами HTML5 с помощью редактора *sed* и Perl.

О чем вы узнали в главе 8

- Как выполняется положительная и отрицательная опережающая проверка.
- Как выполняется положительная и отрицательная ретроспективная проверка.

На заметку

См. стр. 88–97 в книге Джеффри Фридла *Регулярные выражения, 3-е издание* (Символ-Плюс, 2008 г.).

ГЛАВА 9

Разметка документа тегами HTML5

В этой главе представлено поэтапное описание процесса разметки простого текстового документа тегами HTML5 с использованием регулярных выражений, включая и те, которые обсуждались в самом начале книги.

Если бы речь шла обо мне, то я предпочел бы использовать для этого утилиту AsciiDoc. Однако, с учетом учебного характера книги, мы притворимся, что такой вещи, как AsciiDoc, просто не существует (какой позор!). Мы будем медленно продвигаться вперед, используя те немногие инструменты, которые имеются у нас под рукой, а именно: редактор *sed*, Perl и собственную изобретательность.

В качестве текстового документа мы по-прежнему будем использовать поэму Кольриджа, текст которой хранится в файле *rime.txt*.

Приведенные в этой главе сценарии отлично работают с файлом *rime.txt*, поскольку структура этого файла нам хорошо известна. В случае их применения к произвольным текстовым файлам результаты будут менее предсказуемыми, однако эта глава позволит вам подготовиться к обработке более сложных текстовых структур.

Сопоставление с тегами

Прежде чем мы приступим к добавлению тегов в текст поэмы, обсудим, какой способ наиболее пригоден для поиска тегов HTML или XML. Поиск можно выполнять по начальным тегам (например, `<html>`), по конечным тегам (например, `</html>`) и еще рядом других способов, но я нашел способ, отличающийся повышенной надежностью. Соответствующий шаблон находит начальные теги как при наличии атрибутов, так и в их отсутствие:

```
<[_a-zA-Z][^>]*>
```

Вот как работает это выражение.

- Первый символ представляет открывающую угловую скобку (<).
- Имена элементов в XML-документе могут начинаться с символа подчеркивания (_) или буквы, принадлежащей к диапазону ASCII-символов, безразлично в каком регистре — верхнем или нижнем (см. раздел "На заметку").

- За начальным символом имени может следовать любое количество символов (в том числе и нулевое), отличных от закрывающей угловой скобки (>).
- Выражение завершается закрывающей угловой скобкой.

Воспользуемся утилитой *grep*. Применим ее к содержащемуся в архиве примеров DITA-файлу *lorem.dita*:

```
grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita
```

В результате выполнения этой команды получим следующий результат.

```
<topic id="lorem">
<title>
<body>
<p>
<p>
<ul>
<li>
<li>
<li>
<p>
<p>
```

Чтобы находить совпадения как с начальным, так и с конечным тегом, достаточно дополнить шаблон прямой косой чертой с последующим вопросительным знаком. Вопросительный знак делает прямую косую черту необязательной:

```
</?[_a-zA-Z][^>]*>
```

В этом примере я и далее буду работать только с открывающим тегом. Для улучшения внешнего вида выходной информации *grep* я часто перенаправляю ее на вход других инструментов, используя канал.

```
grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita | sort | uniq | sed↵
's/^<//;s/ id=\".*
\"//;s/>
$//'
```

Это дает нам следующий отсортированный список имен XML-тегов.

```
body
li
p
p
title
topic
ul
```

В последней главе книги мы продвинемся еще дальше. А пока используем то, что нам уже известно, но применим новые приемы.

Преобразование простого текста с помощью редактора *sed*

Добавим дополнительную разметку в самом начале текста, хранящегося в файле *rime.txt*. Мы сделаем это с помощью команды вставки (`i\`). Перейдя в каталог, в котором находится файл *rime.txt*, введите в командной строке следующую команду.

```
sed '1 i\
<!DOCTYPE html>\
<html lang="en">\
<head>\
<title>The Rime of the Ancyent Marinere (1798)</title>\
<meta charset="utf-8"/>\
</head>\
<body>\
q' rime.txt
```

После того как вы нажмете клавишу <Enter> или <Return>, вы должны увидеть примерно следующий вывод с тегами в самом начале.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>The Rime of the Ancyent Marinere (1798)</title>
<meta charset="utf-8"/>
</head>
<body>
THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.
```

Команда, которую вы только что ввели, фактически не изменила файл; она всего лишь обеспечила вывод строк на экран. О том, как записать изменения в файл, будет рассказано позднее.

Замена текста с помощью редактора *sed*

В приведенном ниже примере редактор *sed* находит первую строку и захватывает все ее содержимое в группу с использованием экранированных скобок `\(` и `\)`. Экранирование скобок в захватывающих группах является обязательным требованием, если только вы не используете опцию `-E` (к этому вопросу мы еще вернемся).

Начало строки обозначается символом `^`, а конец — символом `$`. Обратная ссылка `\1` извлекает сохраненное группой содержимое в элемент `title`, который выводится с отступом в один пробел.

А теперь выполните следующую команду:

```
sed '1s/^\(.*\)$/ <title>\1<\/title>/;q' rime.txt
```

Результирующая строка будет выглядеть так:

```
<title>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.
</title>
```

Теперь видоизменим команду.

```
sed -E '1s/^(.*)$/<!DOCTYPE html>\
<html lang="en">\
<head>\
 <title>\1<\/title>\
<\/head>\
<body>\
<h1>\1<\/h1>\
/;q' rime.txt
```

Обсудим отдельные составляющие этой команды.

- Опция `-E` означает использование расширенных выражений, или ERE (которые не требуют экранирования скобок и т.п.)
- Команда подстановки (`s`) запоминает строку 1 в захватывающей группе (`\^(.\*)$`), что обеспечивает возможность повторного использования текста посредством ссылки `\1`.
- Далее создаются HTML-теги, причем символы новой строки экранируются символами обратной косой черты (`\`).
- После этого сохраненный текст вставляется в элементы `title` и `h1` посредством ссылки `\1`.
- Вывод на экран оставшейся части текста поэмы блокируется посредством команды `q`.

Результат выполнения этой команды должен быть следующим.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.
</title>
</head>
<body>
<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

Обработка римских цифр в редакторе *sed*

Поэма разделена на семь частей, пронумерованных римскими цифрами. Кроме того, имеется заголовок "ARGUMENT". Следующая команда *sed* предназначена для захвата этого заголовка и римских цифр и окружения их тегами `h2`:

```
sed -En 's/^(ARGUMENT\.|I{0,3}V?I{0,2}\.)$/<h2>\1<\/h2>/p'↵
 rime.txt
```

Вот результат ее работы.

```
<h2>ARGUMENT\.</h2>
<h2>I.</h2
<h2>II.</h2
<h2>III.</h2
<h2>IV.</h2
<h2>V.</h2
<h2>VI.</h2
<h2>VII.</h2
```

Рассмотрим более подробно отдельные составляющие этой команды.

- Опция `-E` включает режим расширенных регулярных выражений, а опция `-n` подавляет вывод каждой строки, заданный в *sed* по умолчанию.
- Команда подстановки (`s`) захватывает заголовок и семь римских цифр в верхнем регистре в диапазоне от I до VII, располагаемых в отдельных строках с точкой после цифры.
- Затем команда `s` извлекает каждую строку захваченного текста и помещает ее в отдельный элемент `h2`.
- Флаг `p` в конце команды подстановки выводит результат на экран.

Обработка отдельного абзаца в редакторе *sed*

Следующая команда находит абзац в строке 5:

```
sed -En '5s/^([A-Z].*)$/<p>\1<\/p>/p' rime.txt
```

и помещает его в элемент `p`.

```
<p>How a Ship having passed the Line was driven by Storms
to the cold Country towards the South Pole; and how from
thence she made her course to the tropical Latitude of the
Great Pacific Ocean; and of the strange things that befell;
and in what manner the Ancyent Marinere came back to his
own Country.<p>
```

Я понимаю, что пока что мы продвигаемся очень мелкими шагами, но наберитесь терпения, и вскоре я сведу все в единое целое.

Обработка строк поэмы в редакторе *sed*

Следующий шаг заключается в разметке строк поэмы с помощью показанной ниже команды.

```
sed -E '9s/^[ ]*(.*)/ <p>\1<br\/>/;↳

10,832s/^([ ]{5,7}.*)/\1<br\/>/;↳
833s/^(.*)/\1<\/p>/' rime.txt
```

Здесь команды редактора *sed* жестко привязаны к конкретным номерам строк. В общем случае такой подход неприемлем, но если вы знаете, с чем имеете дело, то он работает довольно неплохо.

- Команда `s` захватывает строку 9 (первая строка стихотворной части поэмы) и после присоединения к ней нескольких пробелов вставляет начальный тег `p` и присоединяет тег `br` (разрыв строки) к концу строки.
- В диапазоне номеров строк от 10 до 832 к концу каждой строки, начинающейся с 5–7 пробелов, присоединяется тег `br`.
- К строке 833, последней строке поэмы, вместо тега `br` команда `s` присоединяет конечный тег `p`.

Вот как выглядит часть результирующей разметки.

```
<p>It is an ancyent Marinere,<br/>
     And he stoppeth one of three:<br/>
  "By thy long grey beard and thy glittering eye<br/>
     "Now wherefore stoppest me?<br/>

  "The Bridegroom's doors are open'd wide<br/>
    "And I am next of kin;<br/>
  "The Guests are met, the Feast is set,--<br/>
    "May'st hear the merry din.--<br/>
```

Кроме того, вы можете создать промежутки между стихами, заменив пустые строки тегами `br`:

```
sed -E 's/^$/<br\/>/' rime.txt
```

Вот как выглядит результат.

```
     He prayeth best who loveth best,
       All things both great and small:
     For the dear God, who loveth us,
       He made and loveth all.
<br/>
     The Marinere, whose eye is bright,
       Whose beard with age is hoar,
     Is gone; and now the wedding-guest
       Turn'd from the bridegroom's door.
<br/>
     He went, like one that hath been stunn'd
       And is of sense forlorn:
     A sadder and a wiser man
       He rose the morrow morn.
```

Я мог бы бесконечно долго заниматься приукрашиванием внешнего вида текста, расставляя теги и пробелы в нужных местах. Рекомендую вам самостоятельно попрактиковаться в этом.

Добавление тегов

Добавим в конце поэмы некоторые дополнительные теги. В приведенной ниже команде символ `$` находит последнюю строку (конец файла), а с помощью команды `\a` в этом месте добавляются закрывающие теги `body` и `html`.

```
sed '$ a\
<\/body>\
<\/html>\
' rime.txt
```

После применения этой команды последние строки файла будут выглядеть так.

```
He went, like one that hath been stunn'd
  And is of sense forlorn:
A sadder and a wiser man
  He rose the morrow morn.
</body>
</html>
```

Ну что же, редактор *sed* потрудился на славу!

А можно ли выполнить все описанные до этого изменения текста поэмы за один раз? Вам известно, что именно надо делать. Вы уже прошли через это. Осталось лишь собрать все команды в один файл и использовать в редакторе *sed* опцию `-f`.

Использование командного файла в редакторе *sed*

В этом примере представлен файл *html.sed*, в котором собраны все описанные ранее команды редактора *sed*, применявшиеся для добавления HTML-тегов в текст поэмы, и пара других команд. Этот файл мы используем для преобразования файла *rime.txt* в документ HTML с помощью редактора *sed*. Нумерация поможет вам проследить за тем, что происходит в данном сценарии.

```
#!/usr/bin/sed ❶

1s/^(.*)$/<!DOCTYPE html>\ ❷
<html lang="en">\
<head>\
<title>\1<\/title>\
<\/head>\
<body>\
<h1>\1<\/h1>\
/
s/^(ARGUMENT|I{0,3}V?I{0,2})\.$/<h2>\1<\/h2>/ ❸
5s/^([A-Z].*)$/<p>\1<\/p>/ ❹
9s/^[ ]*(.*)/ <p>\1<br\/>/ ❺
10,832s/^([ ]{5,7}.*)/\1<br\/>/ ❻
833s/^(.*)/\1<\/p>/ ❼
13,$s/^$/<br\/>/ ❽
$ a\ ❾
<\/body>\
<\/html>\
```

❶ Первая пара символов (`#!` — *магическая строка*) указывает на то, что данный файл является сценарием; оставшаяся часть первой строки задает местоположение соответствующего командного интерпретатора (*sed*).

❷ Заменить (`s`) строку 1 последующими тегами. Символ обратной косой черты (`\`) указывает на то, что текст, который вы хотите добавить, продолжается на следующей строке, и поэтому необходимо вставить символ новой строки. Вставить заголовок поэмы, взятый из строки 1 с помощью переменной `\1`, в качестве содержимого элементов `title` и `h1`.

❸ Окружить заголовки и римские цифры тегами `h2`.

❹ Обработать строку 5, заключив вводный абзац в элемент `p`.

❺ Обработать строку 9, добавив открывающий тег `p` в начале строки и тег `br` в конце строки.

❻ Обработать строки с номерами от 9 до 832, добавив тег `br` в конце каждой строки, начинающейся с определенного количества пробелов.

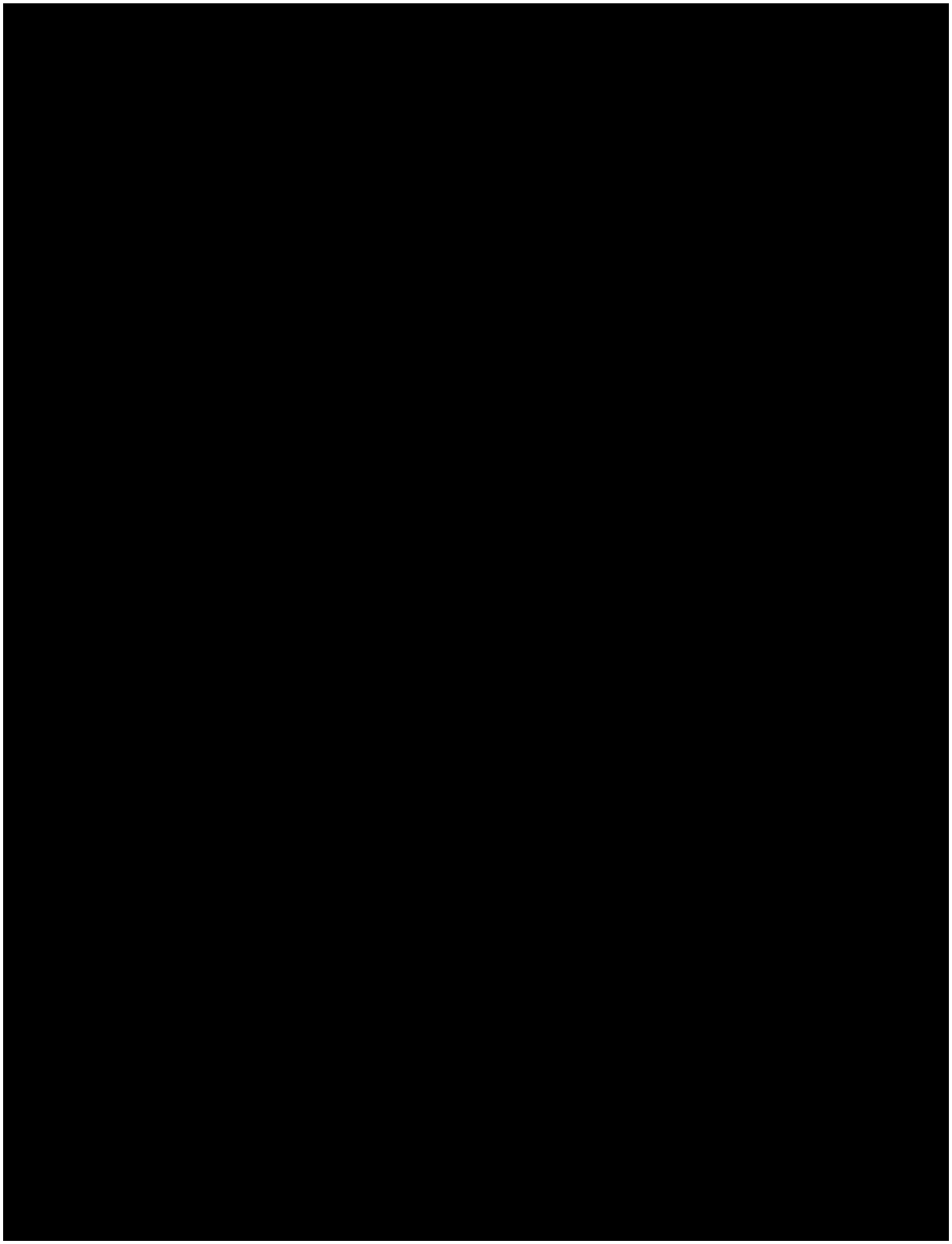
❼ В конце текста поэмы добавить закрывающий тег `p`.

❽ После строки 13 заменить каждую пустую строку тегом `br`.

❾ Добавить несколько тегов в конце (`$`) документа.

Чтобы применить этот командный файл к файлу *rime.txt*, введите следующую строку и нажмите клавишу <Enter> or <Return>:

```
sed -E -f html.sed rime.txt
```



Если текущей строкой (`$.`) является строка 1, назначьте всю строку (`$_`) переменной `$title` и выведите ее.

```
perl -ne 'if ($. == 1) {chomp($title = $_); print↩
"<h1>" . $title . "</h1>" .
"\n";};'
    rime.txt
```

В случае успешного выполнения этой команды результат должен быть таким:

```
<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

Ниже приведено описание работы этой команды Perl.

- Проверить с помощью функции `$`, является ли строка 1 текущей.
- Захватить строку (`$_`) и назначить ее переменной `$title`. Когда функция `chomp` захватывает строку, она удаляет из нее концевой символ новой строки.
- Вывести переменную `$title` в составе элемента `h1`, добавив в конце символ новой строки (`\n`).

- Для получения более подробной информации о таких встроенных функциях Perl, как `$.`, выполните команду `perldoc -v $.` (обычно *perldoc* устанавливается одновременно с Perl). Если это не сработает, обратитесь к разделу "На заметку".

Чтобы добавить в начало файла разметку, включая тег `h1`, выполните следующую команду.

```
perl -ne 'if ($. == 1) {chomp($title = $_)};
print "<!DOCTYPE html>\
<html xmlns=\"http://www.w3.org/1999/xhtml\">\
 <head>\
  <title>$title</title>\
  <meta charset=\"utf-8\"/>\
 </head>\
<body>\
<h1>$title</h1>\n" if $. == 1; exit' rime.txt
```

В результате вы получите следующий вывод.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.
  </title>
  <meta charset=\"utf-8\"/>
 </head>
<body>
<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

Функция `print` выводит последующие теги, и за каждой строкой следует символ `\`, который вводит символ новой строки в выходной поток. Содержимое переменной `$title` помещается в элементы `title` и `h1`.

Обработка римских цифр с помощью Perl

Чтобы снабдить тегами заголовок и римские цифры, которыми обозначаются разделы, выполните следующую команду:

```
perl -ne 'print if s/^(ARGUMENT\.|I{0,3}V?I{0,2}\.)↳
$/<h2>\1<\/h2>/;' rime.txt
```

Вот ее вывод.

```
<h2>ARGUMENT.</h2>
<h2>I.</h2>
<h2>II.</h2>
<h2>III.</h2>
<h2>IV.</h2>
<h2>V.</h2>
<h2>VI.</h2>
<h2>VII.</h2>
```

Команда подстановки (`s`) захватывает заголовок `ARGUMENT` и семь римских цифр от I до VII в верхнем регистре, каждый раз дополняя захваченный текст точкой в конце и заключая его в элемент `h2`.

Обработка отдельного абзаца с помощью Perl

Проверив, что номер строки равен 5, заключите вступительный абзац в элемент `p` с помощью следующей команды:

```
perl -ne 'if ($. == 5) {s/^([A-Z].*)↳
$/<p>$1<\/p>/;print;}' rime.txt
```

Результат ее применения должен быть таким.

```
<p>How a Ship having passed the Line was driven by Storms
to the cold Country towards the South Pole; and how from
thence she made her course to the tropical Latitude of the
Great Pacific Ocean; and of the strange things that befell;
and in what manner the Ancyent Marinere came back to his
own Country.</p>
```

Обработка строк поэмы с помощью Perl

Приведенная ниже команда вставляет открывающий тег `p` в начале первой строки поэмы, а тег `br` — в конце этой строки.

```
perl -ne 'if ($. == 9) {s/^[ ]*(.*)/ <p>$1<br\/>/;↳
print;}' rime.txt
```

В результате мы получаем такую строку:

```
<p>It is an ancyent Marinere,<br/>
```

Следующая команда Perl преобразует строки с номерами от 10 до 832, добавляя тег `br` в конце каждой строки:

```
perl -ne 'if (10..832) { s/^([ ]{5,7}.*)/$1<br\/>/;↳
print;}' rime.txt
```

Ниже в качестве примера приведена часть преобразованного текста.

```
Farewell, farewell! but this I tell<br/>
  To thee, thou wedding-guest!<br/>
He prayeth well who loveth well<br/>
  Both man and bird and beast.<br/>
```

Добавим закрывающий тег p в конце последней строки поэмы:

```
perl -ne 'if ($. == 833) {s/^(.*)/$1<\/p>/; print;}' rime.txt
```

Вот как выглядит результат:

```
He rose the morrow morn.</p>
```

Замените пустые строки в конце каждой строфы тегами br:

```
perl -ne 'if (9..eof) {s/^$/<br\/>/; print;}' rime.txt
```

что приведет к показанному ниже результату.

```
<br/>
     He prayeth best who loveth best,
       All things both great and small:
     For the dear God, who loveth us,
       He made and loveth all.
<br/>
     The Marinere, whose eye is bright,
       Whose beard with age is hoar,
     Is gone; and now the wedding-guest
       Turn'd from the bridegroom's door.
<br/>
```

И в завершение выведем пару тегов по достижении конца файла:

```
perl -ne 'if (eof) {print "</body>\n</html>\n"};' rime.txt
```

Гораздо легче организовать работу всего кода, объединив его в одном файле, к чему мы сейчас и переходим.

Использование командного файла в Perl

Ниже приведен листинг файла *html.pl*, который содержит команды, преобразующие файл *rime.txt* в HTML-документ с помощью Perl. Выноски помогут вам проследить за тем, что происходит в данном сценарии.

```
#!/usr/bin/perl -p ❶

if ($. == 1) { ❷
chomp($title = $_);
}
print "<!DOCTYPE html>\ ❸
<html xmlns=\"http://www.w3.org/1999/xhtml\">\
 <head>\
  <title>$title</title>\
  <meta charset=\"utf-8\"/>\
 </head>\
<body>\
```

```
<h1>$title</h1>\n" if $. == 1;
s/^(ARGUMENT|I{0,3}V?I{0,2})\.$/<h2>$1<\/h2>/; ❹
if ($. == 5) { ❺
 s/^([A-Z].*)$/<p>$1<\/p>/;
}
if ($. == 9) { ❻
 s/^[ ]*(.*)/ <p>$1<br\/>/;
}
if (10..832) { ❼
 s/^([ ]{5,7}.*)/$1<br\/>/;
}
if (9..eof) { ❽
 s/^$/<br\/>/;
}
if ($. == 833) { ❾
 s/^(.*)$/$1<\/p>\n <\/body>\n<\/html>\n/;
}
```

❶ Первая пара символов (#! — *магическая строка*) указывает интерпретатору команд на то, что данный файл является программой; оставшаяся часть первой строки задает местоположение программы, которую вы хотите выполнить.

❷ Если текущая строка (`$.`) является строкой 1, назначить всю строку (`$_`) переменной `$title`, попутно удаляя последний символ строки (символ новой строки) с помощью функции `chomp`.

❸ Вывести тип документа и некоторые HTML-теги в начале документа в строке 1 и повторно использовать значение переменной `$title` в нескольких местах.

❹ Поместить заголовок `ARGUMENT` и римские цифры в элементы `h2`.

❺ Окружить вступительный абзац тегами `p`.

❻ Вставить открывающий тег `p` в начале первой строки стиха, а тег `br` — в конце этой строки.

❼ Вставить тег `br` в конце каждой строки, за исключением последней.

❽ Заменить каждую пустую строку после строки 9 тегом `br`.

❾ Вставить в конце последней строки закрывающие теги `p`, `body` и `html`.

Чтобы выполнить данную программу, введите следующую команду:

```
perl html.pl rime.txt
```

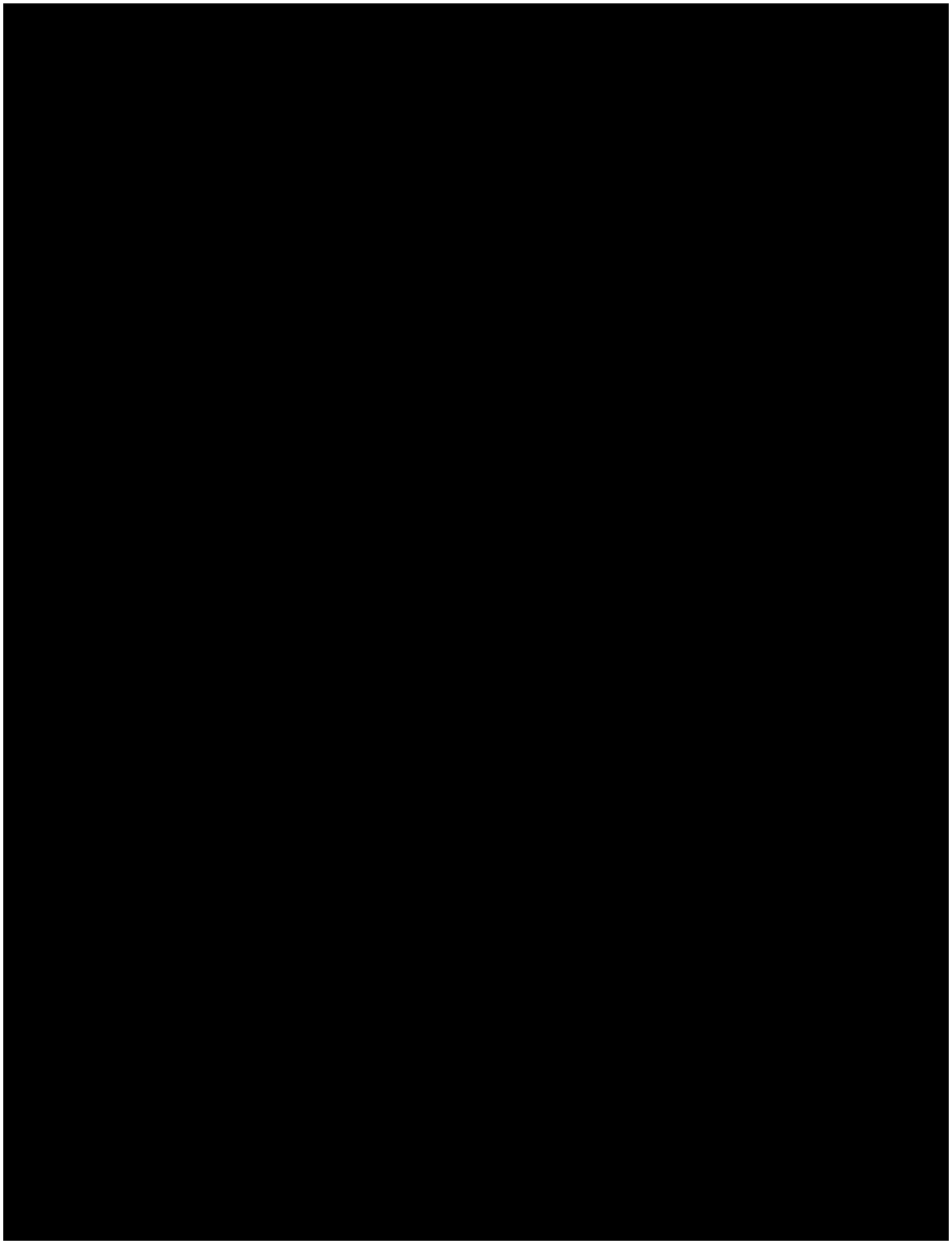
Вы также можете воспользоваться возможностью перенаправления для сохранения вывода в файле с помощью выражения `a >`. Следующая глава завершает данное краткое руководство по регулярным выражениям.

О чем вы узнали в главе 9

- Как использовать редактор *sed* в командной строке.
- Как вставить, заменить и присоединить текст (теги) с помощью редактора *sed*.
- Как использовать Perl для выполнения тех же операций.

На заметку

- Утилита AsciiDoc (`http://www.methods.co.nz/asciidoc`), написанная Стюартом Рэкхемом, позволяет конвертировать обычные текстовые файлы в форматы HTML, PDF, ePUB, DocBook и *man* с помощью процессора Python. Синтаксис для текстовых файлов аналогичен синтаксису Wiki и Markdown и обеспечивает гораздо более быструю подготовку документов по сравнению с ручным кодированием тегов HTML или XML.
- Символ подчеркивания может применяться только в документах XML, но не HTML. Кроме того, диапазон символов, использование которых в именах допускается стандартом XML, намного шире того, который представлен набором ASCII. Более подробно об использовании символов в именах XML можно прочитать по адресу `http://www.w3.org/TR/REC-xml/#sec-common-syn`.
- Если команда `perldoc` не сработает, в вашем распоряжении есть альтернативные варианты. Прежде всего, можете прочитать онлайновую документацию на сайте `http://perldoc.perl.org`. (Например, для того чтобы узнать больше о переменной `$.`, откройте страницу `http://perldoc.perl.org/perlvar.html#Variables-related-to-filehandles`.) Если вы работаете на компьютере Mac, попробуйте использовать версию `perldoc5.12`. Если вы установили Perl из ActiveState, то эта версия должна находиться в каталоге `/usr/local/ActivePerl-5.XX/bin`. В случае установки путем компиляции исходного кода как `perl`, так и `perldoc` устанавливаются в каталоге `/usr/local/bin`. Чтобы `perl` и `perldoc` запускались, добавьте путь `/usr/local/bin` в переменную PATH. Более подробную информацию относительно установки значений переменной PATH можно получить по адресу `http://java.com/en/download/help/path.xml`.



ГЛАВА 10

Конец начала

"UNIX не был создан для того, чтобы не дать кому-то делать глупости, ведь это помешало бы умным людям делать умные вещи".

— Дуглас Гвин

Примите поздравления в связи с тем, что у вас хватило терпения дочитать книгу до этой страницы. Теперь вы уже далеко не новичок в области регулярных выражений. Вам был представлен синтаксис наиболее часто используемых регулярных выражений, что открыло перед вами много новых возможностей.

Знание регулярных выражений сэкономило мне массу времени. Позвольте привести конкретный пример.

По роду работы мне часто приходится использовать XSLT, и во многих случаях я должен анализировать теги, встречающиеся в группе XML-файлов.

Кое-что из этого я постарался продемонстрировать вам в последней главе, а сейчас прошу взглянуть на однострочную команду, которая выбирает список тегов из файла *lorem.dita* и преобразует его в простую таблицу стилей XSLT.

```
grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita | sort | uniq | sed↳
 '1 i\
<xsl:stylsheet version="2.0" xmlns:xsl=
  "http://www.w3.org/1999/XSL/Transform">\

; s/^</\
<xsl:template match="/;s/ id=\".*\"//;s/>$/">\
 <xsl:apply-templates\/>\
<\/xsl:template>/;$ a\
\
</xsl:stylesheet>\
'
```

Я знаю, что этот сценарий выглядит сплошной эквилибристикой, но, после того как вы внимательно его изучите, ваше мнение изменится. Я даже не буду объяснять, что именно делает эта команда, поскольку уверен, что теперь вы и сами вполне можете в этом разобраться.

Вот как выглядит вывод этой команды.

```
<xsl:stylsheet version="2.0" xmlns:xsl=↳
 "http://www.w3.org/1999/XSL/Transform">

<xsl:template match="body">
 <xsl:apply-templates/>
</xsl:template>
```

```
<xsl:template match="li">
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="p">
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="title">
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="topic">
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="ul">
 <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```

Это всего лишь начало. Конечно же, над такой простой таблицей стилей нужно еще немало потрудиться, прежде чем она сможет делать что-нибудь полезное, но зато она избавит вас от необходимости вручную вводить большие объемы данных.

Я признаю, что было бы проще поместить все вышеприведенные команды *sed* в файл. В действительности я так и сделал. Соответствующий файл *xslt.sed* вы найдете в архиве примеров. Вот его содержимое.

```
#!/usr/bin/sed

1 i\
<xsl:stylsheet version="2.0" xmlns:xsl=↳
 "http://www.w3.org/1999/XSL/Transform">\

s/^</\
<xsl:template match="/;s/ id=\".*\"//;s/>$/">\
<xsl:apply-templates\/>\
<\/xsl:template>/;$ a\
\
</xsl:stylesheet>\
```

Ниже показано, как запустить этот файл на выполнение.

```
grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita | sort |↳
     uniq | sed -f xslt.sed
```

Что дальше

Несмотря на то что сейчас вы уже достаточно хорошо подготовлены для работы с регулярными выражениями, вам еще предстоит многому научиться. Хочу предложить ряд других книг, которые вам стоило бы прочитать.

Мои советы основаны на личном опыте и наблюдениях и не продиктованы никакими рекламными соображениями. Я делаю это совершенно искренне, потому что чтение перечисленных ниже книг действительно пойдет вам на пользу.

Книга Джеффри Фридла *Регулярные выражения, 3-е издание* (Символ-Плюс, 2008 г.) — источник, к которому обращаются многие программисты в поиске компетентного рассмотрения регулярных выражений. Если вы собираетесь серьезно работать с регулярными выражениями, то эта содержательная и хорошо написанная книга обязательно должна стоять на вашей книжной полке.

Книга Яна Гойвертса и Стивена Левитана *Регулярные выражения. Сборник рецептов, 2-е издание* (Символ-Плюс, 2015 г.) — другой замечательный труд, особенно если речь идет о сравнении различных реализаций. Эту книгу вам также стоит иметь у себя.

Книга Тони Стаблбайна *Regular Expression Pocket Reference: Regular Expressions for Perl, Ruby, PHP, Python, C, Java and .Net* (O'Reilly, 2007) — это 128-страничное руководство, которое, несмотря на сравнительно давний срок выхода в свет, и по сей день остается популярным.

Книга Эндрю Уотта *Beginning Regular Expressions* (Wrox, 2005) была высоко оценена программистами. Особенно полезным я считаю также онлайновое руководство по редактору *sed*, написанное Брюсом Барнеттом (`http://www.grymoire.com/Unix/Sed.html`). В нем демонстрируется использование целого ряда нетривиальных средств, о которых мною не было сказано ни слова.

Инструменты, реализации и библиотеки

Perl

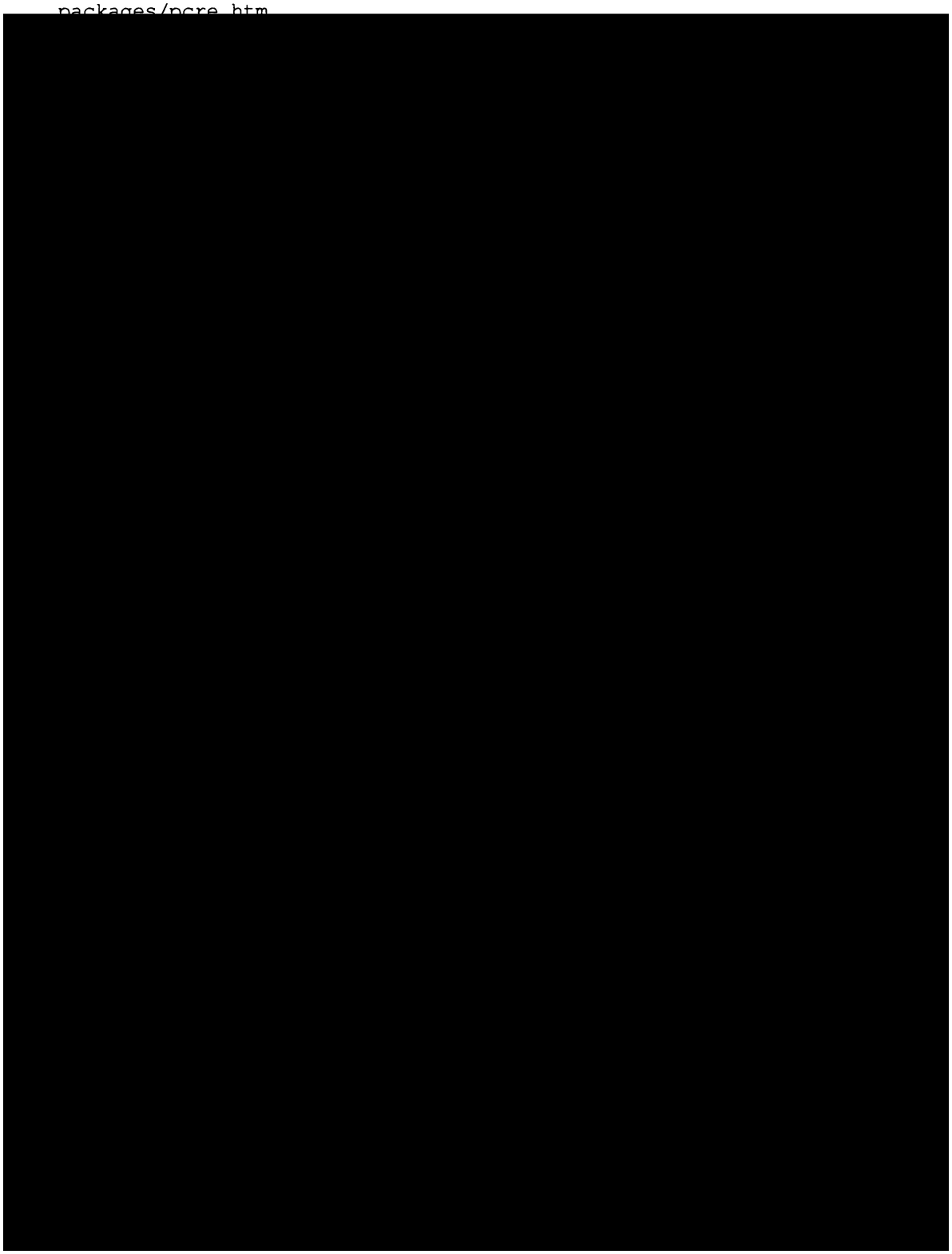
Perl — популярный универсальный язык программирования. Для обработки текстов с помощью регулярных выражений многие предпочитают использовать именно Perl, а не какой-либо другой язык. Вероятно, он уже установлен на вашей машине, но при необходимости вы можете найти описание процедуры установки по адресу `http://www.perl.org/get.html`. Информацию о регулярных выражениях Perl можно получить здесь: `http://perldoc.perl.org/perlre.html`. Не поймите меня превратно. Существует множество других языков, которые прекрасно справляются с регулярными выражениями, но наличие Perl в вашем арсенале инструментов всегда себя окупит.

PCRE

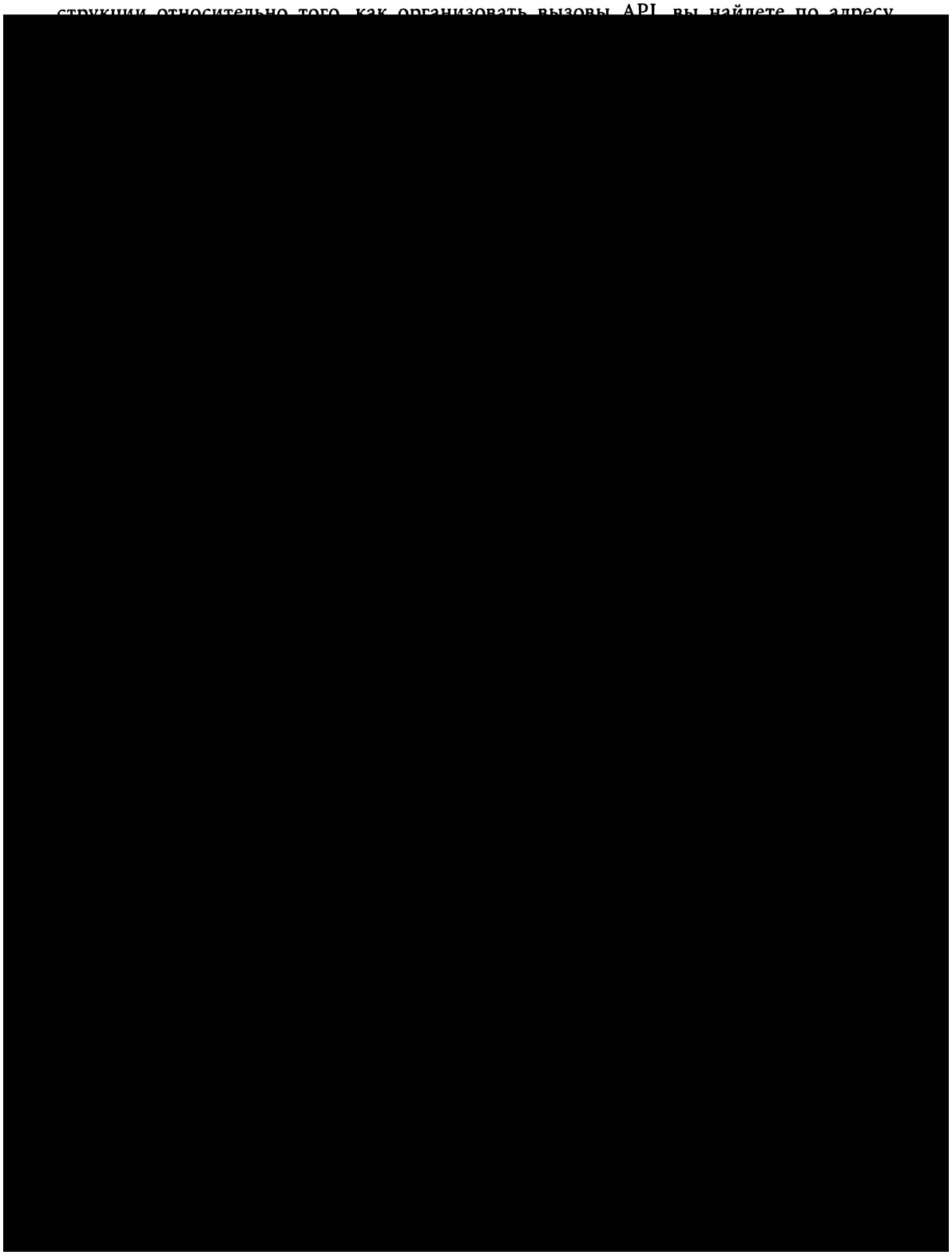
Perl Compatible Regular Expressions, или PCRE (`http://www.pcre.org`), — это библиотека регулярных выражений, написанная на языке C (8- и 16-разрядная версии). Она состоит главным образом из функций, которые могут вызываться в любой среде разработки C или в любом другом языке, допускающем использование библиотек C. Как говорит само ее название, данная библиотека совместима с регулярными выражениями Perl 5 и включает некоторые средства из других реализаций регулярных выражений. Библиотека PCRE используется в текстовом редакторе Notepad++.

Утилита *pcregrep* — это 8-разрядный инструмент, подобный *grep*, позволяющий использовать возможности библиотеки PCRE в командной строке. Вы работали с ним в главе 3. Информацию о его загрузке (с сайта `ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/`) можно найти по адресу `http://www.pcre.org`. Его версию для компьютеров Mac можно загрузить на сайте Macports (`http://www.macports.org`), выполнив команду `sudo port install pcre` (требуется предварительная установка интегрированной среды разработки Xcode; для получения дополнительной информации обратитесь

на сайт `https://developer.apple.com/technologies/tools/`, вход на который возможен после прохождения процедуры регистрации). Для установки на платформе Windows (в виде двоичных файлов) перейдите по адресу `http://gnuwin32.sourceforge.net/packages/pcre.htm`



ссылками (более подробно об этом см. http://code.google.com/p/re2). Библиотека доступна в виде пакета SPAN для Perl и в тех случаях, когда требуются обратные ссылки, позволяет обращаться к функциям собственной библиотеки Perl. Инструкции относительно того, как организовать вызовы API, вы найдете по адресу



- `^` — проверка совпадения с началом строки.
- `\(?` — литеральная открывающая круглая скобка, не являющаяся обязательной (`?`).
- `(?:\d{3})` — группа без захвата, которой соответствует последовательность из трех цифр.
- `\)?` — необязательная закрывающая круглая скобка.
- `[-.]?` — допускает включение необязательного дефиса или точки.
- `(?:\d{3})` — еще одна группа без захвата, которой соответствует последовательность из трех цифр.
- `[-.]?` — допускает включение необязательного дефиса или точки.
- `(?:\d{4})` — еще одна группа без захвата, которой соответствует последовательность из четырех цифр.
- `$` — соответствие концу строки или текста.

Это выражение — адаптированный вариант аналогичного выражения, приведенного в книге Гойвертса и Левитана *Регулярные выражения. Сборник рецептов*. Его можно и далее улучшать, однако я оставляю эту задачу для вас, поскольку теперь вам по силам справиться с ней самостоятельно.

Сопоставление с адресами электронной почты

Наконец, я покажу еще одно регулярное выражение, которое представляет адрес электронной почты:

```
^([\w-.!#$%&;'*+-/=?^_'{|}~]+)@((?:\w+\.)+)(?:[a-zA-Z]{2,4})$
```

Это адаптация аналогичного выражения из приложения RegExr Гранта Скиннера. Мне бы очень хотелось, чтобы вы сами объяснили, что означает здесь каждый символ в контексте регулярного выражения, и попытались внести дальнейшие улучшения. Я уверен, что вы сможете это сделать.

Благодарю вас за то, что потратили время на чтение моей книги. Мне было приятно наше общение. С основными понятиями регулярных выражений вы теперь хорошо знакомы. Вы уже не член клуба новичков. Надеюсь, что вам удалось освоить регулярные выражения и вы узнали многое из того, что пригодится вам в дальнейшем.

О чем вы узнали в главе 10

- Как извлечь из документа список XML-элементов и преобразовать список в таблицу стилей XSLT.
- Где искать дополнительные ресурсы для изучения регулярных выражений.
- Известные инструменты, реализации и библиотеки, предназначенные для работы с регулярными выражениями.
- Как создать надежный шаблон для сопоставления с телефонными номерами в формате, принятом в США и Канаде.

ПРИЛОЖЕНИЕ А

Справочник по регулярным выражениям

Данное приложение представляет собой краткий справочник по регулярным выражениям.

Регулярные выражения в QED

Редактор QED (сокращение от "Quick Editor") первоначально был написан для операционной системы Berkeley Time-Sharing System, выполнявшейся на компьютерах Scientific Data Systems SDS 940. Его версия, переработанная Кеном Томпсоном для операционной системы Compatible Time-Sharing System, эксплуатировавшейся в MIT, предоставила одну из первых (если не самую первую) реализацию регулярных выражений для практических вычислений. В табл. А.1, взятой со стр. 3 и 4 служебного руководства Bell Labs от 1970 года, описаны возможности встроенных средств редактора QED для работы с регулярными выражениями. Как это ни поразительно, но большая часть синтаксиса тех времен сохранилась и по сей день, спустя более чем сорок лет.

Таблица А.1. Регулярные выражения редактора QED

Средство	Описание
литерал	"a) Обычный символ (литерал) — регулярное выражение, буквально совпадающее с вхождением данного символа"
^	"b) ^ — регулярное выражение, которому соответствует пустой символ в начале строки"
$	"c) $ — регулярное выражение, которому соответствует символ <nl> [новая строка] (обычно находящийся в конце строки)"
.	"d) . — регулярное выражение, которому соответствует любой одиночный символ, за исключением символа <nl> [новая строка]"
[*<строка>*]	"e) [*<строка>*] — регулярное выражение, которому соответствует любой из символов, перечисленных в *строке*, и никакой другой символ"
[^*<строка>*]	"f) [^*<строка>*] — регулярное выражение, которому соответствует любой символ, кроме символа <nl> [новая строка] и символов, перечисленных в *строке*"

Окончание табл. А.1

Средство	Описание
*	"g) Регулярное выражение, за которым следует символ *, — это регулярное выражение, которому соответствует любое (включая и нулевое) число смежных повторений текста, соответствующего исходному регулярному выражению"
	"h) Два смежных регулярных выражения образуют регулярное выражение, которому соответствуют смежные вхождения текстов, соответствующих исходным регулярным выражениям"
|	"i) Два регулярных выражения, разделенные символом |, образуют регулярное выражение, которому соответствует любой из текстов, соответствующих исходным регулярным выражениям"
()	"j) Регулярное выражение, заключенное в круглые скобки, представляет собой регулярное выражение, которому соответствует тот же текст, что и соответствующий исходному регулярному выражению. Круглые скобки используются для изменения порядка вычисления выражений, подразумеваемого в пп. g, h и i: выражению a(b|c)d будет соответствовать текст **abd** или **acd**, а выражению ab|cd — **ab** или **cd**"
{}	"k) Если *<regexp>* — регулярное выражение, то и {*<regexp>*}x — регулярное выражение, где x — произвольный символ). Этому выражению соответствует то же самое, что и выражению *<regexp>*; оно имеет некоторые побочные эффекты, о которых сказано в описании команды Substitute." [Команда Substitute формировалась в виде (.,.S/*<regexp>*/*<строка>*) (см. стр. 13 руководства) аналогично тому, как это все еще используется в программах наподобие **sed** и в Perl]
\E	"l) Если *<rexname>* — имя регулярного выражения, поименованного командой E (см. ниже), то \E*<rexname>* — регулярное выражение, которому соответствует то же самое, что и регулярному выражению, указанному в команде E. Дальнейшее обсуждение содержится в описании команды E" [Команда \E позволяла присваивать имя регулярному выражению и повторно использовать его по этому имени]
	"m) Обособленное пустое выражение эквивалентно последнему встретившемуся регулярному выражению. Пустое регулярное выражение изначально является неопределенным; оно также становится неопределенным после использования ошибочного регулярного выражения и после использования команды E"
	"n) Ничто больше, кроме вышеописанного, не является регулярным выражением"
	"o) Текст, распространяющийся более чем на одну строку, не соответствует никакому регулярному выражению"

Метасимволы

В регулярных выражениях используются 14 метасимволов, каждый из которых имеет специальное назначение (табл. А.2). Если любой из этих символов необходимо использовать как литерал, его следует экранировать, поставив перед ним символ обратной косой черты. Например, экранированный символ доллара выглядит как \$, а экранированный символ обратной косой черты — как \\.

Таблица А.2. Метасимволы регулярных выражений

Метасимвол	Название	Кодовая точка	Назначение
.	Точка	U+002E	Совпадает с любым одиночным символом, кроме символа новой строки
\	Обратная косая черта	U+005C	Экранирует символ
|	Вертикальная линия	U+007C	Альтернатива (или)
^	Знак вставки (циркумфлекс)	U+005E	Якорь начала строки
$	Знак доллара	U+0024	Якорь конца строки
?	Вопросительный знак	U+003F	Квантификатор "0 или 1"
*	Звездочка	U+002A	Квантификатор "0 или более"
+	Плюс	U+002B	Квантификатор "1 или более"
[	Левая квадратная скобка	U+005B	Открывает символьный класс
]	Правая квадратная скобка	U+005D	Закрывает символьный класс
{	Левая фигурная скобка	U+007B	Открывает квантификатор или блок
}	Правая фигурная скобка	U+007D	Закрывает квантификатор или блок
(	Левая круглая скобка	U+0028	Открывает группу
)	Правая круглая скобка	U+0029	Закрывает группу

Специальные символы

В табл. А.3 приведены специальные символы (символьные сокращения), используемые в регулярных выражениях.

Таблица А.3. Специальные символы

Специальный символ	Описание
\a	Предупреждение
\b	Граница слова
[\b]	Возврат на шаг (забой)
\B	Не граница слова
\cx	Управляющий символ
\d	Цифра
\D	Не цифра
\dxxx	Десятичное значение кода символа
\f	Прогон страницы

Окончание табл. A.3

Специальный символ	Описание
\r	Возврат каретки
\n	Перевод строки
\oxxx	Восьмеричное значение кода символа
\s	Пробел (пустой символ)
\S	Не пробел (непустой символ)
\t	Горизонтальная табуляции
\v	Вертикальная табуляции
\w	Словарный символ
\W	Не словарный символ
\0	NULL
\xxx	Шестнадцатеричное значение кода символа
\uxxxx	Символ в кодировке Unicode

Пробельные символы

В табл. A.4 приведены условные символы, представляющие пробел.

Таблица A.4. Пробельные символы

Условный символ	Описание
\f	Прогон страницы
\h	Горизонтальный пробел
\H	Не горизонтальный пробел
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальный пробел
\V	Не вертикальный пробел

Пробельные символы Unicode

В табл. A.5 приведены пробельные символы в кодировке Unicode.

Таблица A.5. Пробельные символы в кодировке Unicode

Аббревиатура или псевдоним	Название (пояснение)	Кодовая точка Unicode	Регулярное выражение
HT	Горизонтальная табуляция	U+0009	\u0009 или \t
LF	Перевод строки	U+000A	\u000A или \n

Окончание табл. А.5

Специальный символ	Описание	Специальный символ	Описание
VT	Вертикальная табуляция	U+000B	\u000B или \v
FF	Прогон страницы	U+000C	\u000C или \f
CR	Возврат каретки	U+000D	\u000d или \r
SP	Межсловный пробел	U+0020	\u0020 или \s*
NEL	Следующая строка	U+0085	\u0085
NBSP	Неразрывный межсловный пробел	U+00A0	\u00A0
—	Знак пробела в алфавите Ogham	U+1680	\u1680
MVS	Разделитель гласных монгольского алфавита	U+180E	\u180E
BOM	Метка порядка байтов	U+FEFF	\ufeff
NQSP	Полукруглая (полукегельная) шпация	U+2000	\u2000
MQSP, Mutton Quad	Круглая (кегельная) шпация	U+2001	\u2001
ENSP, Nut	Полукруглая шпация	U+2002	\u2002
EMSP, Mutton	Круглая шпация	U+2003	\u2003
3MSP, Thick space	Третная шпация	U+2004	\u2004
4MSP, Mid space	Четвертная шпация	U+2005	\u2005
6/MSP	Шестерная шпация	U+2006	\u2006
FSP	Пробел, равный по ширине цифрам	U+2007	\u2007
PSP	Пунктуационный пробел	U+2008	\u2008
THSP	Тонкий пробел	U+2009	\u2009
HSP	Волосяной пробел	U+200A	\u200A
ZWSP	Пробел нулевой ширины	U+200B	\u200B
LSEP	Разделитель строк	U+2028	\u2028
PSEP	Разделитель абзацев	U+2029	\u2029
NNBSP	Неразрывный узкий пробел	U+202F	\u202F
MMSP	Средний математический пробел	U+205F	\u205f
IDSP	Идеографический пробел	U+3000	\u3000

* Также совпадает с вхождениями других пробельных символов.

Управляющие символы

В табл. А.6 показан способ представления управляющих символов в регулярных выражениях.

Таблица А.6. Сопоставление с управляющими символами

Управляющий символ	Значение Unicode	Аббревиатура	Название
`c@`*	U+0000	NUL	Пустой символ
`\cA`	U+0001	SOH	Начало заголовка
`\cB`	U+0002	STX	Начало текста
`\cC`	U+0003	ETX	Конец текста
`\cD`	U+0004	EOT	Конец передачи
`\cE`	U+0005	ENQ	Запрос
`\cF`	U+0006	ACK	Подтверждение
`\cG`	U+0007	BEL	Звуковой сигнал
`\cH`	U+0008	BS	Возврат на шаг
`\cI`	U+0009	HT	Горизонтальная табуляция
`\cJ`	U+000A	LF	Перевод строки (новая строка, конец строки)
`\cK`	U+000B	VT	Вертикальная табуляция
`\cL`	U+000C	FF	Прогон страницы
`\cM`	U+000D	CR	Возврат каретки
`\cN`	U+000E	SO	Режим национальных символов
`\cO`	U+000F	SI	Обычный режим ASCII
`\cP`	U+0010	DLE	Освобождение канала данных
`\cQ`	U+0011	DC1	1-й код управления устройством
`\cR`	U+0012	DC2	2-й код управления устройством
`\cS`	U+0013	DC3	3-й код управления устройством
`\cT`	U+0014	DC4	4-й код управления устройством
`\cU`	U+0015	NAK	Отрицательное подтверждение
`\cV`	U+0016	SYN	Пустой символ для синхронного режима передачи
`\cW`	U+0017	ETB	Конец блока передаваемых данных
`\cX`	U+0018	CAN	Отмена
`\cY`	U+0019	EM	Конец носителя
`\cZ`	U+001A	SUB	Символ замены
`\c[`	U+001B	ESC	Альтернативный регистр № 2 (AP2)

Окончание табл. A.6

Управляющий символ	Значение Unicode	Аббревиатура	Название
\c\	U+001C	FS	Разделитель данных № 4 (разделитель файлов)
\c]	U+001D	GS	Разделитель данных № 3 (разделитель групп)
\c^	U+001E	RS	Разделитель данных № 2 (разделитель записей)
\c_	U+001F	US	Разделитель данных № 1 (разделитель полей)

* Может использоваться верхний или нижний регистр. Например, выражения \cA и \ca эквивалентны. Однако реализации Java требуют использования верхнего регистра.

Свойства символов

В табл. A.7 приведены имена свойств символов, используемые в командах \p{*свойство*} и \P{*свойство*}.

Таблица A.7. Свойства символов*

Свойство	Описание
C	Другое
Cc	Управляющий символ
Cf	Формат
Cn	Не присвоено
Co	Частное использование
Cs	Суррогат
L	Буква
Ll	Буква нижнего регистра
Lm	Буква-модификатор
Lo	Другая буква
Lt	Прописная буква
Lu	Буква верхнего регистра
L&	Ll, Lu или Lt
M	Знак
Mc	Внутристрочный знак
Me	Заключающий знак
Mn	Внестрочный знак
N	Цифра
Nd	Десятичная цифра

Свойство	Описание
Nl	Буквенная цифра
No	Другая цифра
P	Знак препинания
Pc	Соединительный знак препинания
Pd	Тире
Pe	Закрывающий знак препинания
Pf	Завершающий знак препинания
Pi	Начальный знак препинания
Po	Другой знак препинания
Ps	Открывающий знак препинания
S	Символ
Sc	Символ валюты
Sk	Символ-модификатор
Sm	Математический символ
So	Другой символ
Z	Разделитель
Zl	Разделитель строк
Zp	Разделитель абзацев
Zs	Разделительный пробел

* См. pcresyntax(3) на странице `http://www.pcre.org/pcre.txt`.

Имена шрифтов в свойствах символов

В табл. А.8 представлены имена шрифтов, используемые в командах `\p{`*свойства*`}` и `\P{`*свойства*`}`.

Таблица А.8. Имена шрифтов*

Arabic (Arab)	Glagolitic (Glag)	Lepcha (Lepc)	Samaritan (Samr)
Armenian (Armn)	Gothic (Goth)	Limbu (Limb)	Saurashtra (Saur)
Avestan (Avst)	Greek (Grek)	Linear B (Linb)	Shavian (Shaw)
Balinese (Bali)	Gujarati (Gujr)	Lisu (Lisu)	Sinhala (Sinh)
Bamum (Bamu)	Gurmukhi (Guru)	Lycian (Lyci)	Sundanese (Sund)
Bengali (Beng)	Han (Hani)	Lydian (Lydi)	Syloti Nagri (Sylo)
Bopomofo (Bopo)	Hangul (Hang)	Malayalam (Mlym)	Syriac (Syrc)
Braille (Brai)	Hanunoo (Hano)	Meetei Mayek (Mtei)	Tagalog (Tglg)
Buginese (Bugi)	Hebrew (Hebr)	Mongolian (Mong)	Tagbanwa (Tagb)
Buhid (Buhd)	Hiragana (Hira)	Myanmar (Mymr)	Tai Le (Tale)
Canadian Aboriginal (Cans)	Hrkt: Katakana or Hiragana)	New Tai Lue (Talu)	Tai Tham (Lana)

Carian (Cari)	Imperial Aramaic (Armi)	Nko (Nkoo)	Tai Viet (Tavt)
Cham (None)	Inherited (Zinh/Qaai)	Ogham (Ogam)	Tamil (Taml)
Cherokee (Cher)	Inscriptional Pahlavi (Phli)	Ol Chiki (Olck)	Telugu (Telu)
Common (Zyyy)	Inscriptional Parthian (Prti)	Old Italic (Ital)	Thaana (Thaa)
Coptic (Copt/Qaac)	Javanese (Java)	Old Persian (Xpeo)	Thai (None)
Cuneiform (Xsux)	Kaithi (Kthi)	Old South Arabian (Sarb)	Tibetan (Tibt)
Cypriot (Cprt)	Kannada (Knda)	Old Turkic (Orkh)	Tifinagh (Tfng)
Cyrillic (Cyrl)	Katakana (Kana)	Oriya (Orya)	Ugaritic (Ugar)
Deseret (Dsrt)	Kayah Li (Kali)	Osmanya (Osma)	Unknown (Zzzz)
Devanagari (Deva)	Kharoshthi (Khar)	Phags Pa (Phag)	Vai (Vaii)
Egyptian Hieroglyphs (Egyp)	Khmer (Khmr)	Phoenician (Phnx)	Yi (Yiii)
Ethiopic (Ethi)	Lao (Laoo)	Rejang (Rjng)	Tagalog (Tglg)
Georgian (Geor)	Latin (Latn)	Runic (Runr)	Tagbanwa (Tagb)

* См. pcresyntax(3) на странице `http://www.pcre.org/pcre.txt` или `http://ruby.runpaint.org/regexps#properties`.

Символьные классы POSIX

В табл. A.9. приведен список символьных классов POSIX

Таблица A.9. Символьные классы POSIX

Символьный класс	Описание
`[[:alnum:]]`	Алфавитно-цифровые символы (буквы и цифры)
`[[:alpha:]]`	Алфавитные символы (буквы)
`[[:ascii:]]`	ASCII-символы (все 128)
`[[:blank:]]`	Пробельные символы
`[[:ctrl:]]`	Управляющие символы
`[[:digit:]]`	Цифры
`[[:graph:]]`	Графические символы
`[[:lower:]]`	Буквы нижнего регистра
`[[:print:]]`	Печатные символы
`[[:punct:]]`	Знаки пунктуации
`[[:space:]]`	Пробел
`[[:upper:]]`	Буквы верхнего регистра
`[[:word:]]`	Словарные символы
`[[:xdigit:]]`	Шестнадцатеричные цифры

Опции и модификаторы

В табл. A.10 и A.11 приведен список опций и модификаторов.

Таблица A.10. Опции в регулярных выражениях

Опция	Описание	Поддержка
`(?d)`	Строки Unix	Java
`(?i)`	Игнорировать регистр	PCRE, Perl, Java
`(?J)`	Разрешить дублирование имен	PCRE*
`(?m)`	Многострочный режим	PCRE, Perl, Java
`(?s)`	Однострочный режим (dotall)	PCRE, Perl, Java
`(?u)`	Кодировка Unicode	Java
`(?U)`	Ленивый поиск по умолчанию	PCRE
`(?x)`	Игнорировать пробелы, комментарии	PCRE, Perl, Java
`(?-...)`	Сброс или отключение всех опций	PCRE

* См. раздел "Named Subpatterns" на странице `http://www.pcre.org/pcre.txt`.

Таблица A.11. Модификаторы (флаги) Perl*

Модификатор	Описание
`a`	Искать совпадения для `\d`, `\s`, `\w` и классов POSIX только в диапазоне ASCII-символов
`c`	Сохранять текущую позицию после неудачного сопоставления
`d`	Использовать собственные правила платформы, заданные по умолчанию
`g`	Глобальный поиск
`i`	Игнорировать регистр при сопоставлении
`l`	Использовать правила текущей локали
`m`	Независимая обработка логических строк
`p`	Сохранять совпавшую строку
`s`	Обрабатывать логические строки как одну физическую строку
`u`	Использовать правила Unicode при сопоставлении
`x`	Игнорировать пробелы и комментарии

* См. `http://perldoc.perl.org/perlre.html#Modifiers`.

Таблица кодов ASCII и представление ASCII-символов в регулярных выражениях

В табл. A.12 представлена таблица кодов ASCII и указаны способы представления ASCII-символов для использования в регулярных выражениях

Таблица А.12. Таблица кодов ASCII

Двоичный	Восьмеричный	Десятичный	Шестнадцатеричный	Символ	Клавиши	Регулярное выражение	Название
00000000	0	0	0	NUL	^@	\c@	Пустой символ
00000001	1	1	1	SOH	^A	\cA	Начало заголовка
00000010	2	2	2	STX	^B	\cB	Начало текста
00000011	3	3	3	ETX	^C	\cC	Конец текста
00000100	4	4	4	EOT	^D	\cD	Конец передачи
00000101	5	5	5	ENQ	^E	\cE	Запрос
00000110	6	6	6	ACK	^F	\cF	Подтверждение
00000111	7	7	7	BEL	^G	\a, \cG	Звуковой сигнал
00001000	10	8	8	BS	^H	[\b], \cH	Возврат на шаг
00001001	11	9	9	HT	^I	\t, \cI	Горизонтальная табуляция
00001010	12	10	0A	LF	^J	\n, \cJ	Перевод строки
00001011	13	11	0B	VT	^K	\v, \cK	Вертикальная табуляция
00001100	14	12	0C	FF	^L	\f, \cL	Подача бумаги
00001101	15	13	0D	CR	^M	\r, \cM	Возврат каретки
00001110	16	14	0E	SO	^N	\cN	Режим национальных символов
00001111	17	15	0F	SI	^O	\cO	Обычный режим ASCII
00010000	20	16	10	DLE	^P	\cP	Освобождение канала данных
00010001	21	17	11	DC1	^Q	\cQ	1-й код управления устройством (XON)
00010010	22	18	12	DC2	^R	\cR	2-й код управления устройством
00010011	23	19	13	DC3	^S	\cS	3-й код управления устройством (XOFF)
00010100	24	20	14	DC4	^T	\cT	4-й код управления устройством
00010101	25	21	15	NAK	^U	\cU	Отрицательное подтверждение
00010110	26	22	16	SYN	^V	\cV	Пустой символ для синхронного режима передачи
00010111	27	23	17	ETB	^W	\cW	Конец блока передаваемых данных
00011000	30	24	18	CAN	^X	\cX	Отмена
00011001	31	25	19	EM	^Y	\cY	Конец носителя
00011010	32	26	1A	SUB	^Z	\cZ	Символ замены
00011011	33	27	1B	ESC	^[	\e, \c[	Альтернативный регистр № 2 (AP2)
00011100	34	28	1C	FS	^|	\c|	Разделитель файлов

Двоичный	Восьмеричный	Десятичный	Шестнадцатеричный	Символ	Клавиши	Регулярное выражение	Название
00011101	35	29	1D	GS	^]	\c]	Разделитель групп
00011110	36	30	1E	RS	^^	\c^	Разделитель записей
00011111	37	31	1F	US	^_	\c_	Разделитель полей
00100000	40	32	20	SP	SP	\s, [ ]	Пробел
00100001	341	33	21	!	!	!	Восклицательный знак
00100010	42	34	22	"	"	"	Кавычки
00100011	43	35	23	#	#	#	Знак номера
00100100	44	36	24	$	$	\$	Знак доллара
00100101	45	37	25	%	%	%	Знак процента
00100110	46	38	26	&	&	&	Амперсанд
00100111	47	39	27	'	'	'	Апостроф
00101000	50	40	28	(	(	(, \(	Левая круглая скобка
00101001	51	41	29	)	)	), \)	Правая круглая скобка
00101010	52	42	2A	*	*	*	Звездочка
00101011	53	43	2B	+	+	+	Плюс
00101100	54	44	2C	"	"	"	Запятая
00101101	55	45	2D	-	-	-	Дефис, минус
00101110	56	46	2E	.	.	\., [.]	Точка
00101111	57	47	2F	/	/	/	Косая черта
00110000	60	48	30	0	0	\d, [0]	Цифра 0
00110001	61	49	31	1	1	\d, [1]	Цифра 1
00110010	62	50	32	2	2	\d, [2]	Цифра 2
00110011	63	51	33	3	3	\d, [3]	Цифра 3
00110100	64	52	34	4	4	\d, [4]	Цифра 4
00110101	65	53	35	5	5	\d, [5]	Цифра 5
00110110	66	54	36	6	6	\d, [6]	Цифра 6
00110111	67	55	37	7	7	\d, [7]	Цифра 7
00111000	70	56	38	8	8	\d, [8]	Цифра 8
00111001	71	57	39	9	9	\d, [9]	Цифра 9
00111010	72	58	3A	:	:	:	Двоеточие

Двоичный	Восьмеричный	Десятичный	Шестнадцатеричный	Символ	Клавиши	Регулярное выражение	Название
00111011	73	59	3B	;	;	;	Точка с запятой
00111100	74	60	3C	<	<	<	Знак "меньше чем"
00111101	75	61	3D	=	=	=	Знак равенства
00111110	76	62	3E	>	>	>	Знак "больше чем"
00111111	77	63	3F	?	?	?	Вопросительный знак
01000000	100	64	40	@	@	@	Коммерческий знак "эт"
01000001	101	65	41	A	A	\w, [A]	Латинская заглавная буква A
01000010	102	66	42	B	B	\w, [B]	Латинская заглавная буква B
01000011	103	67	43	C	C	\w, [C]	Латинская заглавная буква C
01000100	104	68	44	D	D	\w, [D]	Латинская заглавная буква D
01000101	105	69	45	E	E	\w, [E]	Латинская заглавная буква E
01000110	106	70	46	F	F	\w, [F]	Латинская заглавная буква F
01000111	107	71	47	G	G	\w, [G]	Латинская заглавная буква G
01001000	110	72	48	H	H	\w, [H]	Латинская заглавная буква H
01001001	111	73	49	I	I	\w, [I]	Латинская заглавная буква I
01001010	112	74	4A	J	J	\w, [J]	Латинская заглавная буква J
01001011	113	75	4B	K	K	\w, [K]	Латинская заглавная буква K
01001100	114	76	4C	L	L	\w, [L]	Латинская заглавная буква L
01001101	115	77	4D	M	M	\w, [M]	Латинская заглавная буква M
01001110	116	78	4E	N	N	\w, [N]	Латинская заглавная буква N
01001111	117	79	4F	O	O	\w, [O]	Латинская заглавная буква O
01010000	120	80	50	P	P	\w, [P]	Латинская заглавная буква P
01010001	121	81	51	Q	Q	\w, [Q]	Латинская заглавная буква Q
01010010	122	82	52	R	R	\w, [R]	Латинская заглавная буква R
01010011	123	83	53	S	S	\w, [S]	Латинская заглавная буква S
01010100	124	84	54	T	T	\w, [T]	Латинская заглавная буква T
01010101	125	85	55	U	U	\w, [U]	Латинская заглавная буква U
01010110	126	86	56	V	V	\w, [V]	Латинская заглавная буква V
01010111	127	87	57	W	W	\w, [W]	Латинская заглавная буква W
01011000	130	88	58	X	X	\w, [X]	Латинская заглавная буква X

Двоичный	Восьмеричный	Десятичный	Шестнадцатеричный	Символ	Клавиши	Регулярное выражение	Название
01011001	131	89	59	Y	Y	\w, [Y]	Латинская заглавная буква Y
01011010	132	90	5A	Z	Z	\w, [Z]	Латинская заглавная буква Z
01011011	133	91	5B	[	[	\[	Левая квадратная скобка
01011100	134	92	5C	\	\	\	Обратная косая черта
01011101	135	93	5D	]	]	\]	Правая квадратная скобка
01011110	136	94	5E	^	^	^, [^]	Циркумфлекс
01011111	137	95	5F	_	_	_, [_]	Символ подчеркивания
00100000	140	96	60	`	`	\`	Гравис
01100001	141	97	61	a	a	\w, [a]	Латинская строчная буква a
01100010	142	98	62	b	b	\w, [b]	Латинская строчная буква b
01100011	143	99	63	c	c	\w, [c]	Латинская строчная буква c
01100100	144	100	64	d	d	\w, [d]	Латинская строчная буква d
01100101	145	101	65	e	e	\w, [e]	Латинская строчная буква e
01100110	146	102	66	f	f	\w, [f]	Латинская строчная буква f
01100111	147	103	67	g	g	\w, [g]	Латинская строчная буква g
01101000	150	104	68	h	h	\w, [h]	Латинская строчная буква h
01101001	151	105	69	i	i	\w, [i]	Латинская строчная буква i
01101010	152	106	6A	j	j	\w, [j]	Латинская строчная буква j
01101011	153	107	6B	k	k	\w, [k]	Латинская строчная буква k
01101100	164	108	6C	l	l	\w, [l]	Латинская строчная буква l
01101101	155	109	6D	m	m	\w, [m]	Латинская строчная буква m
01101110	156	110	6E	n	n	\w, [n]	Латинская строчная буква n
01101111	157	111	6F	o	o	\w, [o]	Латинская строчная буква o
01110000	160	112	70	p	p	\w, [p]	Латинская строчная буква p
01110001	161	113	71	q	q	\w, [q]	Латинская строчная буква q
01110010	162	114	72	r	r	\w, [r]	Латинская строчная буква r
01110011	163	115	73	s	s	\w, [s]	Латинская строчная буква s
01110100	164	116	74	t	t	\w, [t]	Латинская строчная буква t
01110101	165	117	75	u	u	\w, [u]	Латинская строчная буква u
01110110	166	118	76	v	v	\w, [v]	Латинская строчная буква v

Окончание табл. A.12

Двоичный	Восьмеричный	Десятичный	Шестнадцатеричный	Символ	Клавиши	Регулярное выражение	Название
01110111	167	119	77	w	w	\w, [w]	Латинская строчная буква **w**
01111000	170	120	78	x	x	\w, [x]	Латинская строчная буква **x**
01111001	171	121	79	y	y	\w, [y]	Латинская строчная буква **y**
01111010	172	122	7A	z	z	\w, [z]	Латинская строчная буква **z**
01111011	173	123	7B	{	{	{	Левая фигурная скобка
01111100	174	124	7C	|	|	|	Вертикальная линия
01111101	175	125	7D	}	}	}	Правая фигурная скобка
01111110	176	126	7E	~	~	\~	Тильда
01111111	177	127	7F	DEL	^?	\c?	Удаление

На заметку

Руководство по редактору QED, написанное Кентом Томпсоном и Деннисом Ритчи, можно найти по адресу `http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf`.

Глоссарий

ASCII
American Standard Code for Information Interchange (Американский стандартный код для обмена информацией) — кодировочная таблица, содержащая 128 символов, которая была разработана в 1960-х годах и включает буквы английского алфавита (латиницу), цифры, знаки препинания, непечатные и другие символы. См. также *Unicode*.

BRE
См. *базовые регулярные выражения*.

ERE
См. *расширенные регулярные выражения*.

ed
Текстовый редактор, созданный Кеном Томпсоном в 1971 году для системы UNIX, в котором был реализован механизм регулярных выражений; предшественник редакторов *sed* и *vi*.

grep
Утилита командной строки UNIX, предназначенная для поиска и вывода строк с помощью регулярных выражений. Говорят, что на идею разработки этой утилиты, предложенной в 1973 году, ее создателя Кена Томпсона натолкнула одна из команд редактора *ed*: `g/re/p` (**g**lobal/**re**gular **e**xpression/**p**rint). Впоследствии эта утилита была вытеснена, хотя и не полностью, утилитой *egrep* (или `grep -E`), в которой вместо базовых регулярных выражений (BRE) используются расширенные регулярные выражения (ERE) с такими дополнительными метасимволами, как `|`, `+`,`?`, `(` и `)`. Утилита *fgrep* (`grep -F`) предназначена для выполнения операций поиска в файлах с использованием литеральных строк, и символы наподобие `$`, `*` или `|` не имеют в ней никакого специального назначения. См. также *базовые регулярные выражения*, *расширенные регулярные выражения*.

POSIX
Portable **O**perating **S**ystem **I**nterface for Un**i**x (переносимый интерфейс операционных систем UNIX) — семейство стандартов, созданное IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и электронике). Последняя версия стандарта POSIX для регулярных выражений: POSIX. 1-2008 (`http://standards.ieee.org/ findstds/standard/1003.1-2008.html`).

Perl

Созданный Ларри Уоллом в 1987 году универсальный язык программирования, известный своими мощными возможностями обработки текстов с использованием регулярных выражений. См. `http://www.perl.org`.

sed

Потоковый текстовый редактор для системы UNIX, способный воспринимать регулярные выражения и преобразовывать текст. Первоначально был написан Ли Макмэхоном из Bell Labs в начале 1970-х годов. Вот пример использования этого редактора: `sed -n 's/this/that/g\' file.ext > new.ext`. Включение режима расширенных регулярных выражений осуществляется с помощью следующей команды: `sed -E`. См. также *расширенные регулярные выражения*.

vi

Текстовый редактор, разработанный Биллом Джоем в 1976 году для системы UNIX, в котором используются регулярные выражения.

vim

Текстовый редактор *vim* (`http://www.vim.org`) — это усовершенствованная версия редактора *vi*, главным разработчиком которой являлся Брам Моленар. На протяжении своего обычного рабочего дня мне приходится использовать до семи различных текстовых редакторов, но чаще всего я работаю с *vim*. Если бы меня должны были забросить на необитаемый остров и при этом разрешили взять с собой только один какой-нибудь редактор, то я без всяческих колебаний выбрал бы именно *vim*.

Unicode

Unicode — стандарт кодирования символов, позволяющий представить знаки почти всех письменных языков. Каждому символу в кодировке Unicode соответствует определенная числовая кодовая точка. Стандарт Unicode представляет свыше 100000 символов. В регулярных выражениях символы Unicode могут записываться в виде `\uxxxx` или `\x{xxxx}`, где *x* представляет шестнадцатеричную цифру в диапазоне `0-9` или `A-F` (допускается запись `a-f`) с использованием от одной до четырех позиций. Например, код `\u00E9` представляет символ é (латинская буква e с акутом). См. также `http://www.unicode.org`.

Атом (atom)

См. *метасимвол*.

Атомарная группа (atomic group)

Группировка, отключающая поиск с возвратом в тех случаях, когда для указанного в ней регулярного выражения (`?>...`) не удается найти совпадение. См. также *поиск с возвратом*, *группы*.

Базовые регулярные выражения (basic regular expressions, BRD)

Ранняя реализация механизма регулярных выражений, лишенная ряда современных возможностей, которая в настоящее время считается устаревшей. Чтобы некоторые символы могли функционировать как метасимволы в этой реализации, их приходится экранировать (например, `\{` и `\}`). См. также *расширенные регулярные выражения*.

Буферное хранилище (hold buffer)

См. *пространство хранения*.

Ветвь (branch)

Результат конкатенации отдельных частей регулярного выражения в терминологии POSIX.1. См. также *POSIX*.

Восьмеричная запись символов (octal characters)

Для представления символов в регулярных выражениях можно использовать восьмеричную нотацию. В этом случае символ записывается в виде `\xxx`, где `x` — это цифра в диапазоне `1-7`; возможно использование одной или двух позиций. Например, `\351` представляет символ `é` — латинскую строчную букву `e` со знаком акута.

Группа (group)

Группа — это объединение атомарных регулярных выражений, заключенных в пару круглых скобок `(` и `)`. В некоторых приложениях, таких как *grep* или *sed* (без опции `-E`), каждой из этих скобок должна предшествовать обратная косая черта: `\)` или `\(`. Группы бывают захватывающие (с запоминанием) и незахватывающие (без запоминания). Содержимое захватывающей группы временно сохраняется в памяти и впоследствии может быть использовано повторно. Содержимое незахватывающей группы не сохраняется, что исключает возможность его повторного использования. В случае атомарных групп поиск с возвратом не выполняется. См. также *атомарная группа*.

Жадный поиск (greedy match)

При жадном поиске первоначально захватывается как можно большая часть строки, а затем последовательно выполняются попытки возврата на один символ для обнаружения совпадения. См. *поиск с возвратом*, *ленивый поиск*, *сверхжадный поиск*.

Захватывающая группа (capturing group)

См. *группа*.

Инверсия (negation)

Изменяет условия соответствия регулярному выражению на противоположные. Например, регулярному выражению `[^2-7]`, в котором перечисление символов класса начинается с символа "циркумфлекс" (`^`), соответствует любая одиночная цифра, кроме `2`, `3`, `4`, `5`, `6` и `7`, т.е. любая из цифр `0`, `1`, `8`, `9`.

Катастрофический поиск с возвратом (catastrophic backtracking)

См. *поиск с возвратом*.

Квантификатор (quantifier)

Определяет допустимое количество повторений регулярного выражения в условии поиска. В одной из форм записи квантификатор представляется заключенными в пару фигурных скобок одним или двумя целыми числами, разделенными запятой. Например, запись `{3}` означает, что выражение может встретиться ровно три раза (при работе со старыми инструментальными средствами, в которых используются базовые регулярные выражения, фигурные скобки необходимо экранировать символами обратной косой черты: `\{3\}`). Другими квантификаторами являются `?` (соответствует повторению предыдущего выражения нуль или один раз), `+` (соответствует повторению предыдущего выражения один или более раз) и `*` (соответствует повторению предыдущего выражения нуль или более раз). Другие названия квантификатора: *ограничение*, *модификатор*. Обычные квантификаторы жадные. Существуют также

ленивые (например, `{3}?`) и сверхжадные (например, `{3}+`). См. также *базовые регулярные выражения*, *жадный поиск*, *ленивый поиск*, *сверхжадный поиск*.

Кодовая точка (code point)
См. *Unicode*.

Компонуемость (composability)
"Язык схемы, или язык описания структуры документа (а по существу — язык программирования), предоставляет ряд атомарных объектов и ряд методов их компоновки. Методы компоновки позволяют объединять атомарные объекты в составные, которые, в свою очередь, могут компоноваться в другие составные объекты. Компонуемость языка — это возможность единообразного применения методов компоновки к различным объектам языка, как атомарным, так и составным... Компонуемость характеризует степень легкости изучения и использования языка. Кроме того, повышение компонуемости языка обычно приводит к улучшению отношения "сложность/мощность": при одинаковой степени сложности двух языков тот из них, который обладает более высокой компонуемостью, одновременно обладает и более высокой мощностью". — Джеймс Кларк, "The Design of RELAX NG," `http://www.thaiopensource.com/relaxng/ design.html#section:5`.

Ленивый поиск (lazy match)
При ленивом поиске попытки найти совпадение осуществляются путем захвата проверяемой строки по одному символу за один раз без использования механизма возврата. См. также *поиск с возвратом*, *жадный поиск*, *сверхжадный поиск*.

Литерал (literal)
См. *строковый литерал*.

Метасимвол (metacharacter)
Символ, имеющий специальное значение в регулярном выражении. К метасимволам относятся следующие символы (здесь запятые являются разделителями): `.`, `\`, `\|`, `*`, `+`, `?`, `~`, `$`, `[`, `]`, `(`, `)`, `{`, `}`. Другое название метасимволов — атомы.

Модификатор (modifier)
Символ, который помещается перед шаблоном поиска для изменения правил сопоставления. Например, модификатор `i` означает игнорирование регистра. Другое название модификаторов — флаги.

Незахватывающая группа (non-capturing group)
Группа, заключенная в круглые скобки, содержимое которой не запоминается для последующего использования. Соответствующий синтаксис: `(?:pattern)`. См. также *группа*.

Обратная ссылка (backreference)
Позволяет ссылаться на текст, сохраненный предыдущим регулярным выражением в круглых скобках: обратные ссылки записываются в виде нумерованных переменных `\1`, `\2` и т.д.

Ограничение (bound)
См. *квантификатор*

Ограничение количества вхождений (occurrence constraint)
См. *квантификатор.*

Опережающая проверка (lookahead)
Опережающая проверка бывает двух видов: положительная и отрицательная. Для положительной опережающей проверки используется синтаксис `(?= regex)`, где `regex` — произвольное регулярное выражение. Это условие истинно, если в текущей позиции находится текст, совпадающий с `regex`. Соответственно, для регулярного выражения `regex1 (?= regex2)` совпадение считается достигнутым только в том случае, если вслед за найденным вхождением `regex1` находится вхождение `regex2`. Для отрицательной опережающей проверки используется синтаксис `(?! regex)`. Это условие истинно, если в текущей позиции *не* находится текст, совпадающий с `regex`. Соответственно, для регулярного выражения `regex1 (?! regex2)` совпадение считается достигнутым только в том случае, если вслед за найденным вхождением `regex1` не обнаружено вхождения `regex2`.

Опции (options)
Позволяют изменять правила сопоставления. Например, опция `(?i)` включает режим нечувствительности к регистру. Опции аналогичны модификаторам, но используют другой синтаксис. См. также *модификатор.*

Отрицательная опережающая проверка (negative lookahead)
См. *опережающая проверка.*

Отрицательная ретроспективная проверка (negative lookbehind)
См. *ретроспективная проверка.*

Поиск с возвратом (backtracking)
Попытки нахождения совпадения посредством последовательного возврата на один символ. Используется только жадными шаблонами (в отличие от ленивых и сверхжадных шаблонов). Поиск с возвратом, в процессе которого количество попыток, предпринимаемых процессором регулярных выражений для обнаружения совпадений, исчисляется тысячами, что сопряжено с потреблением значительных вычислительных ресурсов, называют *катастрофическим.* Одним из способов, позволяющих избежать перехода поиска в катастрофический режим, является использование атомарных группировок. См. также *атомарная группировка*, *жадный поиск*, *ленивый поиск*, *сверхжадный поиск.*

Поиск соответствия (matching)
Регулярное выражение может осуществлять поиск соответствия заданному шаблону в тексте, а затем, в зависимости от приложения, инициировать получение соответствующего результата.

Положительная опережающая проверка (positive lookahead)
См. *опережающая проверка.*

Положительная ретроспективная проверка (positive lookbehind)
См. *ретроспективная проверка.*

Группа проверки (lookaround)

См. *опережающая проверка*, *ретроспективная проверка*.

Пространство хранения (hold space)
Используется редактором *sed* для сохранения одной или нескольких строк с целью их дальнейшего использования. Другое название — *буферное хранилище*. См. также *пространство шаблона*.

Пространство шаблона (pattern space)
Обычно программа *sed* обрабатывает по одной входной строке за один раз. Текущая входная строка сохраняется в пространстве шаблона. Другое его название — *рабочий буфер*. См. также *пространство хранения*.

Рабочий буфер (work buffer)
См. *пространство шаблона*.

Расширенные регулярные выражения (extended regular expressions, ERE)
Расширенные регулярные выражения отличаются от базовых наличием дополнительной функциональности, такой как альтернативные шаблоны (`|`) или квантификаторы наподобие `?` и `+`, которые работают с *egrep* (extended grep — расширенная утилита *grep*). Дополнительные возможности описаны в стандарте IEEE POSIX 1003.2-1992. Опция `-E` утилиты *grep* означает, что вы хотите использовать не базовые регулярные выражения, а расширенные. См. также *альтернативные шаблоны*, *базовые регулярные выражения*, *grep*.

Регулярные выражения (regular expressions)
Особым образом закодированные строки символов, которые могут использоваться приложениями и утилитами для поиска других строк или наборов строк. Регулярные выражения были впервые описаны математиком Стивеном Клини (1909–1994) в статье, посвященной формальной теории языков, которая была опубликована в его книге *Metamathematics* в 1952 году. Одной из первых компьютерных программ, в которых начали использоваться регулярные выражения, был текстовый редактор QED, разработанный Кеном Томпсоном и др. для компьютера GE-635, работавшего под управлением операционной системы General Electric Time Sharing System (GE-TSS). В начале 1970-х годов регулярные выражения были включены в операционную систему UNIX, развернутую в AT&T Bell Labs.

Ретроспективная проверка (lookbehind)
Ретроспективная проверка бывает двух видов: положительная и отрицательная. Для положительной ретроспективной проверки используется синтаксис `(?<= regex)`. Это условие истинно, если перед текущей позицией находится текст, совпадающий с `regex`. Соответственно, для регулярного выражения `regex1 (?<= regex2)` совпадение считается достигнутым только в том случае, если вхождению `regex1` непосредственно предшествует вхождение `regex2`. В то же время, в отличие от опережающей проверки, регулярное выражение `regex`, исходя из соображений производительности, не может быть произвольным и должно соответствовать определенному количеству символов. Следовательно, `regex` не может содержать квантификаторы переменной длины и альтернативы, которым соответствует текст разной длины. Для отрицательной опережающей проверки используется синтаксис `(?<! regex)`. Это условие истинно, если перед текущей позицией *не* находится текст, совпадающий

с *regex*. Соответственно, для регулярного выражения *regex1* (?!< *regex2*) совпадение считается достигнутым только в том случае, если найденному вхождению *regex1 не* предшествует непосредственно вхождение *regex2*.

Сверхжадный поиск (possessive match)
При сверхжадном поиске делается попытка найти совпадение путем захвата сразу всей строки. Поиск с возвратом не выполняется. См. также *поиск с возвратом*, *жадный поиск*, *ленивый поиск*.

Символьный класс (character class)
Набор символов, заключенный в квадратные скобки. Например, `[a-bA-B0-9]` — это класс символов, представляющий произвольную одиночную букву нижнего или верхнего регистра или цифру из таблицы ASCII.

Скобочное выражение (bracketed expression)
Регулярное выражение, заключенное в квадратные скобки. Например, выражение `[a-f]` представляет диапазон букв нижнего регистра от `a` до `f`. См. также *символьный класс*.

Строковый литерал (string literal)
Строка символов, интерпретируемая буквально. Пример: строковый литерал `It is an ancyent Marinere` в отличие от чего-нибудь наподобие `[Ii]t[ ]is[ ].*nere`.

Условия с нулевой длиной совпадения (zero-width assertions)
Метасимволы, которым соответствуют не определенные литеральные символы (или их последовательности), а позиции между символами, отвечающие определенным условиям (чем и объясняется название). В качестве примера можно привести метасимволы `^` and `$`, совпадающие, соответственно, с началом и концом строки.

Флаг (flag)
См. *модификатор*.

Чередование (alternation)
Регулярное выражение, представляющее собой список регулярных выражений (альтернатив), разделенных символами вертикальной черты (`|`), имеющими смысл логического "ИЛИ". Иными словами, в таких регулярных выражениях используется первый (в порядке записи слева направо) совпавший шаблон, после чего оставшиеся шаблоны пропускаются. В некоторых приложениях, таких как *grep* или *sed*, в которых используются базовые регулярные выражения (BRE), символу `|` должен предшествовать символ обратной косой черты (`\|`). См. также *базовые регулярные выражения*.

Шестнадцатеричное представление (hexadecimal)
Число, представленное в шестнадцатеричной системе исчисления, в которой используются цифры в диапазоне `0-9` и буквы в диапазоне `A-F` (или `a-f`). Например, десятичное число 15 представляется в шестнадцатеричной системе как F, а число 16 — как 10.

Экранирование символа (character escape)

Предварение символа обратной косой чертой. Примеры: `\t` (горизонтальная табуляция), `\v` (вертикальная табуляция), `\f` (прогон страницы).

Элемент (piece)
В терминологии POSIX.1 — часть регулярного выражения, которая может объединяться с другими частями для образования ветвей. См. также *POSIX*.

Якорь (anchor)
Позиционная привязка. Указывает определенную позицию в строке. Например, символу "крышка" (`^`) соответствует начало строки, символу доллара (`$`) — конец строки.

Предметный указатель

Программирование

Регулярные выражения: основы

Если вы программист, не имеющий опыта работы с регулярными выражениями, то данная книга — как раз то, что нужно для первого знакомства с ними. Многочисленные примеры, приведенные