O'REILLY®

3-е издание

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

ПИТЕР®

Джеффри Фридл

Mastering Regular Expressions

Third Edition

Jeffrey E.F. Friedl

O'REILLY®

Джеффри Фридл

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

3-е издание

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск
2018

ББК 32.973.2-018.1
УДК 004.43
Ф88

Фридл Дж.

Ф88 Регулярные выражения. 3-е изд. — СПб.: Питер, 2018. — 608 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0646-2

Вы никогда не пользовались регулярными выражениями? Откройте мир regex и станьте профессионалом, способным эффективно работать с данными в Java, JavaScript, C, C++, C#, Perl, Python, Ruby, PHP и других языках программирования.

Международный бестселлер знакомит с фундаментальными основами регулярных выражений, функциональными возможностями языков программирования и позволяет оптимизировать работу с информацией. Вы научитесь самостоятельно конструировать регулярные выражения и использовать приведенные в книге примеры для быстрого решения самых актуальных задач.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-0596528126 англ.

ISBN 978-5-4461-0646-2

Краткое содержание

Оглавление

МОЕЙ Фумиэ 文枝

За смирение.
За то, что терпела меня все эти годы,
пока я работал над книгой.

Предисловие

Эта книга посвящена регулярным выражениям — мощному средству обработки текстов. С ее помощью вы научитесь использовать регулярные выражения на практике и извлекать максимум пользы из тех программ и языков программирования, в которых они поддерживаются. Большая часть документации, в которой упоминаются регулярные выражения, не дает даже отдаленного представления об их мощи, а данное издание поможет вам овладеть регулярными выражениями действительно на *мастерском* уровне.

Регулярные выражения поддерживаются многими программами (редакторами, системными утилитами, ядрами баз данных и т. д.), но их возможности в полной мере проявляются в языках программирования, в том числе Java и Jscript, Visual Basic и VBScript, JavaScript и ECMAScript, C, C++, C#, elisp, Perl, Python, Tcl, Ruby, PHP, sed и awk. Регулярные выражения занимают центральное место во многих программах, написанных на этих языках.

Поддержка регулярных выражений в столь разнородных приложениях объясняется тем, что они обладают исключительно богатыми возможностями. На низком уровне регулярное выражение описывает некий фрагмент текста. Им можно воспользоваться для проверки данных, введенных пользователем, или, например, для фильтрации больших объемов данных. На более высоком уровне регулярные выражения позволяют управлять данными. Вы полностью контролируете свои данные и заставляете их работать на себя.

Почему я написал эту книгу

Я завершил работу над первым изданием книги в конце 1996 года. Книга была написана потому, что она была действительно нужна. Хорошей документации по регулярным выражениям не существовало, поэтому большая часть их возможностей оставалась неиспользуемой. Впрочем, документации по регулярным выражениям хватало, но она фокусировалась на работе на «низком уровне». Если показать кому-нибудь алфавит, трудно ожидать, что он сразу заговорит на новом языке.

Пять с половиной лет, прошедшие между публикациями первого и второго изданий этой книги, отмечены ростом популярности интернета и (вряд ли случайным)

значительным расширением области применения регулярных выражений. Практически во всех языках и программах поддержка регулярных выражений стала более мощной и выразительной. Perl, Python, Tcl, Java и Visual Basic — во всех этих языках были созданы новые средства для работы с регулярными выражениями. Появились и завоевали популярность новые языки с поддержкой регулярных выражений (такие, как Ruby, PHP и C#). Все это время основные решаемые книгой вопросы — как правильно понимать регулярные выражения и как извлечь из них максимальную практическую пользу — оставались важными и актуальными.

Но со временем первое издание стало морально устаревать. Оно нуждалось в обновлении, которое позволило бы отразить новые языки и возможности, а также возрастающую роль регулярных выражений в современном Интернете. Второе издание книги вышло в 2002 году, когда появились принципиально новые версии `java.util.regex`, .NET Framework от Microsoft и Perl 5.8. Все они полностью были описаны во втором издании. Единственное, о чем я сожалею, — это то, что недостаточно внимания уделил языку PHP. В течение всех четырех лет после выхода второго издания книги значение языка PHP устойчиво возрастало, поэтому я считаю своим долгом исправить этот недостаток.

В данном, третьем, издании языку PHP уделено углубленное внимание в первых главах; кроме того, появилась новая объемная глава, полностью посвященная регулярным выражениям в PHP и методам наиболее эффективного их использования. Помимо того, в этом издании глава, посвященная языку Java, была переписана и расширена.

Для кого написана эта книга

Книга представляет интерес для всех, кто мог бы использовать регулярные выражения в своей работе. Если вы еще не представляете, насколько богатыми возможностями обладают регулярные выражения, для вас откроется целый новый мир. Книга расширит ваш кругозор, даже если вы считаете себя экспертом в области регулярных выражений. После выхода первого издания я получил немало сообщений по электронной почте типа «Я считал, что умею пользоваться регулярными выражениями, пока не прочитал эту книгу. *Теперь* я действительно умею».

Программисты, занимающиеся обработкой текста (например, веб-программированием), найдут здесь многочисленные технические подробности, рекомендации, советы, а самое главное — *осознают* новые возможности, которые можно немедленно применить на практике. Столь подробного и скрупулезного изложения материала вы просто не найдете в других источниках.

Регулярные выражения — это абстрактная концепция, по-разному реализуемая в разных программах (которых гораздо больше, чем рассмотрено в этой книге).

Если вы поймете общую концепцию регулярных выражений, освоить конкретную реализацию будет не так уж трудно. Этот принцип положен в основу всей книги, поэтому большая часть изложенных сведений не ограничена конкретными программами и языками, использованными в примерах.

Как читать эту книгу

Эта книга может стать учебником, справочником или просто рассказом — все зависит от того, как к ней подойти. Читатели, знакомые с регулярными выражениями, обычно рассматривают книгу как подробный справочник и сразу переходят к разделу, посвященному их любимой программе. Я не рекомендую так поступать.

Чтобы извлечь максимум пользы из этой книги, сначала прочитайте первые шесть глав как рассказ. По своему опыту знаю, что слежение за развитием мысли способствует более полному пониманию материала, при этом подобные вещи лучше усваивать при последовательном чтении, а не пытаться запоминать по списку.

Повесть, рассказанная в первых шести главах, формирует основу для прочтения остальных четырех глав, где описываются характерные особенности работы с регулярными выражениями в языках Perl, Java, .NET и PHP. Я не скупился на перекрестные ссылки и постарался сделать их максимально полезными.

До полного знакомства с материалом книги вряд ли можно работать с ней как со справочником. Конечно, вы можете заглянуть в одну из сводных таблиц (например, в таблицу на с. 128) и решить, что в ней содержится вся необходимая информация. Однако огромное количество полезных сведений находится не в таблице, а в сопровождающем ее тексте. После знакомства с материалом вы начнете ориентироваться в рассматриваемых вопросах и поймете, какие сведения можно просто запомнить, а к каким придется возвращаться снова и снова.

Структура книги

Десять глав этой книги условно делятся на три логические части:

Вводная часть

В главе 1 представлены основные концепции регулярных выражений.

В главе 2 рассматривается применение регулярных выражений при обработке текста.

В главе 3 приводится обзор диалектов регулярных выражений, а также некоторые исторические сведения.

Подробное описание

В главе 4 подробно рассмотрен механизм обработки регулярных выражений.

В главе 5 проанализированы некоторые последствия и практические применения материала главы 4.

В главе 6 обсуждаются проблемы эффективности.

Информация по инструментам

В главе 7 подробно описан диалект регулярных выражений Perl.

В главе 8 рассматривается пакет `java.util.regex` для работы с регулярными выражениями в языке Java.

В главе 9 описан нейтральный по отношению к языкам пакет для работы с регулярными выражениями на платформе .NET.

В главе 10 рассматривается семейство функций preg, предназначенных для работы с регулярными выражениями в языке PHP.

Вводная часть книги дает новичкам представление о рассматриваемой теме. Более опытные читатели могут пропустить начальные главы, хотя я настоятельно рекомендую прочитать главу 3 даже самым закаленным ветеранам.

- Глава 1 «Знакомство с регулярными выражениями» написана для стопроцентного новичка. Читатель познакомится с концепцией регулярных выражений на примере распространенной программы *egrep*. Я постарался изложить свое видение того, как *мыслить* регулярными выражениями, закладывая тем самым надежную основу для понимания нетривиального материала следующих глав. Даже читателям, имеющим опыт работы с регулярными выражениями, стоит просмотреть первую главу.

- В главе 2 «Дополнительные примеры» рассматривается практическая обработка текста в языках программирования, обладающих поддержкой регулярных выражений. Дополнительные примеры помогут лучше разобраться в сложном материале следующих глав и продемонстрируют некоторые важные принципы мышления, используемые при построении сложных регулярных выражений. Чтобы читатель лучше представил, как «мыслить регулярными выражениями», в этой главе будет рассмотрена нетривиальная задача и показаны пути ее решения в двух разных программах с поддержкой регулярных выражений.

- В главе 3 «Регулярные выражения: возможности и диалекты» приведен обзор всевозможных диалектов регулярных выражений, встречающихся в современных программах. Эволюция регулярных выражений проходила довольно бурно, поэтому многие диалекты, распространенные в наши дни, заметно отличаются

друг от друга. В этой главе описана история, а также процесс эволюции регулярных выражений и тех программ, в которых они используются. В конце главы приведен краткий «Путеводитель по серьезным главам». Это своего рода «дорожная карта», при помощи которой вы сможете извлечь максимум пользы из непростого материала следующих глав.

Подробное описание

Разобравшись с основными принципами, мы переходим к поиску ответов на вопросы «как?» и «почему?». Полностью освоив этот материал, вы сможете применять полученные знания везде, где регулярные выражения приносят пользу.

- Глава 4 «Механика обработки регулярных выражений» начинает изложение основного материала этой книги. В ней важные принципы внутренней работы механизма регулярных выражений рассматриваются с практической точки зрения. Тот, кто разберется во всех тонкостях процесса обработки регулярных выражений, пройдет большую часть пути к вершинам мастерства.

- В главе 5 «Практические приемы построения регулярных выражений» изложенный материл выводится на более высокий уровень практического применения. Мы рассмотрим ряд распространенных (но иногда довольно нетривиальных) проблем с целью совершенствования и углубления ваших познаний в области регулярных выражений.

- В главе 6 «Построение эффективных регулярных выражений» рассматриваются специфические аспекты механизмов регулярных выражений, реализованных во многих языках программирования. Руководствуясь материалом, подробно изложенным в главах 4 и 5, вы научитесь использовать сильные стороны каждого механизма и узнаете, как обходить их недостатки.

Информация по инструментам

Если вы хорошо усвоили материал глав 4, 5 и 6, то разобраться в специфике любой конкретной реализации будет не слишком сложно. Впрочем, я посвятил отдельную главу каждой из четырех популярных систем.

- Глава 7 «Perl» посвящена языку Perl — вероятно, самому популярному из всех современных языков программирования с поддержкой регулярных выражений. В языке Perl существует всего четыре оператора для работы с регулярными выражениями, однако из-за бесчисленных режимов, особых случаев и т. д. перед программистом открываются широчайшие возможности, в которых кроются многочисленные ловушки. Богатство возможностей, позволяющее быстро

перейти от концепции к программе, превращается в «минное поле» для начинающих программистов. Надеюсь, подробное изложение материала этой главы поможет вам преодолеть все трудности.

- В главе 8 «Java» подробно рассматривается пакет для работы с регулярными выражениями `java.util.regex`, ставший стандартной частью языка Java начиная с версии 1.4. Основное внимание в этой главе уделяется Java 1.5, но при этом отмечаются отличия от версий 1.4.2 и 1.6.

- Глава 9 «.NET» содержит информацию о работе с библиотекой регулярных выражений .NET (данная информация не предоставлена компанией Microsoft). Вы найдете в ней все необходимое для полноценного применения регулярных выражений в VB.NET, C#, C++, JScript, VBScript, ECMAScript и во всех остальных языках .NET.

- Глава 10 «PHP» представляет собой краткое введение в многочисленные реализации регулярных выражений, встроенных в язык PHP, а также содержит полное описание прикладного интерфейса семейства функций preg, входящих в состав библиотеки PCRE.

Условные обозначения

При подробном описании сложных операций с текстом необходима точность (как, впрочем, и при выполнении этих операций). Всего один лишний или недостающий пробел может иметь колоссальные последствия, поэтому я буду использовать в книге некоторые условные обозначения.

- Регулярные выражения обычно выглядят так: ⌜`this`⌟. Обратите внимание на тонкие «уголки» — они означают, что перед вами регулярное выражение. Литеральный текст (например, тот, который вы ищете) обычно заключается в кавычки: '`this`'. Иногда, если это гарантированно не приведет к недоразумениям, я буду опускать уголки или кавычки. Фрагменты программного кода и скриншоты всегда представляются в естественном виде, поэтому уголки и кавычки в них не используются.

- В литеральном тексте и регулярных выражениях используются разные многоточия. Например, `[…]` — квадратные скобки с произвольным содержимым, а `[. . .]` — последовательность из трех точек.

- Без специальных обозначений довольно трудно сказать, сколькими пробелами разделены буквы в строке «a b». По этой причине пробелы, присутствующие в регулярном выражении (а иногда и в литеральном тексте), изображаются символом '•'. При такой записи становится очевидно, что строка '`a`••••`b`' содержит ровно четыре пробела.

- Я также буду использовать визуальные обозначения для символов табуляции, новой строки и перевода каретки:

•	пробел
TAB	символ табуляции
NL	символ новой строки
CR	символ возврата курсора

- Время от времени я использую подчеркивание или темный фон для выделения частей литерального текста или регулярного выражения. Например:

  ...поскольку ⌜cat⌟ совпадает с фрагментом 'It•indicates•your•cat •is...', а не со словом 'cat', то мы получаем...

В данном случае подчеркиванием выделяется фактически совпавший текст. Можно привести другой пример, в котором подчеркиванием выделяется результат добавления к рассматриваемому выражению:

...чтобы использовать это выражение, можно заключить ⌜Subject|Date⌟ в круглые скобки и добавить двоеточие с пробелом. Результат выглядит так: ⌜(Subject|Date):•⌟.

- В этой книге содержится огромный объем информации и множество примеров, поэтому в ней размещено более 1200 перекрестных ссылок, которые помогут вам эффективно работать с материалом. В тексте книги ссылки выглядят так: «☞ 123», что означает «смотрите страницу 123». В книге вам часто будет встречаться примерно такое описание: «...подробности — в табл. 8.2 (☞ 000)».

Упражнения

Время от времени — особенно в начальных главах — я помещаю контрольные вопросы, напрямую связанные с обсуждаемыми концепциями. Не стоит полагать, что они просто занимают место в книге; я действительно хочу, чтобы вы отвечали на них, прежде чем двигаться дальше. Пожалуйста, не ленитесь. Чтобы вы серьезнее относились к этим задачам, я сократил их количество. Кроме того, по ним можно оценить степень понимания материала. Если для решения задачи вам потребовалось больше нескольких секунд, вероятно, лучше вернуться к соответствующему разделу и перечитать его заново, прежде чем продолжать чтение.

Я постарался по возможности упростить поиск ответов. Все, что от вас требуется, — перевернуть страницу. Ответы на вопросы, помеченные знаком ❖, обычно находятся на следующей странице. Пока вы размышляете над вопросом, ответы не видны, но добраться до них проще простого.

Ссылки, программный код, ошибки и контакты

Наученный горьким опытом, когда ссылки на ресурсы в интернете изменяются быстрее, чем книга выходит в свет, вместо приложения со списком URL я дам всего лишь одну ссылку:

http://regex.info/

Здесь вы найдете ссылки на ресурсы, посвященные регулярным выражениям, все фрагменты программного кода из книги и многое другое. Здесь же вы найдете список обнаруженных в книге ошибок (наличие которых весьма маловероятно :-)).

Если вы найдете ошибку в книге или просто захотите отправить мне какие-либо замечания, пишите на адрес *jfriedl@regex.info*.

Обратиться к издателю можно по адресу:

O'Reilly Media, Inc.

1005 Gravenstein Highway North Sebastopol, CA 95472

(800) 998-9938 (в США и Канаде)

(707) 829-0515 (международный/местный)

(707) 829-0104 (факс)

bookquestions@oreilly.com

За дополнительной информацией о книгах, конференциях, центре ресурсов и O'Reilly Network обращайтесь на сайт издательства:

http://www.oreilly.com

Личные комментарии и благодарности

Работа над первым изданием книги оказалась сложнейшей задачей, решение которой заняло два с половиной года и потребовало помощи со стороны многих людей. Вся эта эпопея до такой степени отразилась на моем здоровье и душевном состоянии, что я поклялся никогда не браться впредь за что-либо подобное.

Я должен поблагодарить многих людей, которые помогли мне отказаться от этого решения. Прежде всего, это моя жена Фумиэ; без ее поддержки и понимания у меня не хватило бы ни сил, ни решимости взяться за такую сложную задачу, как написание и выпуск этой книги.

Во время сбора материала и работы над книгой многие люди помогали мне разобраться в незнакомых языках и системах. Еще больше помощников занималось рецензированием и правкой черновых вариантов книги.

Я особенно благодарен своему брату Стивену Фридлу (Stephen Friedl) за его глубокие и подробные комментарии, заметно улучшившие книгу. (Кроме того, что он прекрасный рецензент, он еще и замечательный писатель, известный по колонке «Tech Tips» на сайте *http:// www.unixwiz.net/*.)

Отдельное спасибо Заку Гренту (Zak Greant), Яну Морзе (Ian Morse), Филиппу Хейзелу (Philip Hazel), Стюарту Джиллу (Stuart Gill), Уильяму Ф. Мэтону (William F. Maton) и моему редактору Энди Ораму (Andy Oram).

Особой благодарности за достоверную информацию о Java заслуживают Майк «madbot» Макклоски (Mike «madbot» McCloskey) (ранее работавший в Sun Microsystems, а ныне — в Google), Марк Рейнольд (Mak Reinhold) и доктор Клифф Клик (Dr Cliff Click), сотрудники Sun Microsystems. Информация о .NET была предоставлена Дэвидом Гутиэрресом (David Gutierrez), Китом Джорджем (Kit George) и Райаном Байнгтоном (Ryan Byington). Сведения о PHP были предоставлены Андреем Змиевским (Andrei Zmievski) из Yahoo!.

Хочу поблагодарить доктора Кена Лунде (Ken Lunde) из Adobe Systems, который создал специальные символы и шрифты для оформления книги[1]. Японские иероглифы взяты из гарнитуры Heisei Mincho W3, а корейские — из гарнитуры Munhwa Министерства культуры и спорта Южной Кореи. Именно Кен когда-то сформулировал главный принцип моей работы: «Ты занимаешься исследованиями, чтобы избавить от этого своих читателей».

За помощь в настройке сервера *http://regex.info* я хочу поблагодарить Джеффри Папена (Jeffrey Papen) и компанию Peak Web Hosting (*http://www.PeakWebhosting.com/*).

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты *comp@piter.com* (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства *http://www.piter.com* вы найдете подробную информацию о наших книгах.

[1] Для английского издания. — *Примеч. ред.*

1 Знакомство с регулярными выражениями

Представьте ситуацию: вам поручено проверить страницы веб-сервера и найти в них повторяющиеся слова (например, «this this»). Эта проблема довольно часто возникает в документах, подвергающихся постоянному редактированию. Ваше решение должно:

- обеспечивать проверку произвольного количества файлов; сообщать о каждой строке каждого файла, содержащей повторяющиеся слова; выделять (при помощи стандартных Escape-последовательностей ANSI) каждое повторяющееся слово и выводить имя исходного файла в каждой строке отчета;
- учитывать возможные разрывы строк и обнаруживать ситуации, когда слово, находящееся в конце одной строки, повторяется в начале следующей;
- находить повторяющиеся слова, несмотря на различия в регистре символов (например, '`The the`…') и в количестве *пропусков* (пробелы, символы табуляции, переводы строки и т. п.) между словами;
- находить повторяющиеся слова, разделенные тегами HTML. Теги HTML применяются при разметке текста в веб-страницах, например, для выделения слов жирным шрифтом: '…`it is <B>very</B> very important`…'.

Ничего себе! Однако описанная задача вполне реальна, а для реальных задач нужны реальные решения. В какой-то момент при подготовке текста этой книги я воспользовался подобным инструментом и был просто поражен количеством обнаруженных повторений. Существует немало языков программирования, позволяющих решить эту задачу, но какой бы язык вы ни использовали, поддержка им регулярных выражений существенно упростит вашу работу.

Регулярные выражения — мощное, гибкое и эффективное средство обработки текстов. Универсальные шаблоны регулярных выражений сами по себе напоминают миниатюрный язык программирования, предназначенный для описания и разбора текста. При дополнительной поддержке со стороны конкретной утилиты или языка программирования регулярные выражения способны вставлять, удалять, выделять

и выполнять самые невероятные операции с текстовыми данными любого вида. Они бывают очень простыми, вроде команды поиска в текстовом редакторе, или очень сложными, как специализированные языки обработки текстов. Из этой книги вы узнаете, как повысить эффективность своей работы при помощи регулярных выражений. Вы научитесь *мыслить* регулярными выражениями, и это позволит вам в полной мере использовать их выдающиеся возможности.

Во многих популярных современных языках программа поиска повторяющихся слов занимает всего несколько строк. Всего одна команда поиска/замены находит и выделяет повторяющиеся слова во всем документе. Другая команда удаляет из отчета все строки, не содержащие повторяющихся слов (и оставляет только те строки, которые включаются в отчет). Наконец, третья команда выводит в начале каждой строки имя файла, к которому относится эта строка. В следующей главе будут приведены примеры на языках Perl и Java.

Язык (Perl, Java, VB.NET и т. д.) обеспечивает периферийную поддержку, но подлинная сила исходит от регулярных выражений. Укротив эту силу для своих целей, вы научитесь писать регулярные выражения, которые отыскивают нужный текст и обходят то, что вас не интересует. После этого остается объединить готовые выражения со вспомогательными конструкциями языка, чтобы выполнить с текстом нужную операцию (добавить коды выделения, удалить текст, изменить его и т. д.).

Решение реальных задач

Регулярные выражения откроют перед вами возможности, о которых вы, вероятно, даже не подозревали. Ежедневно я неоднократно использую их для решения разных задач — и простых, и сложных (и если бы не регулярные выражения, многие простые задачи оказались бы довольно сложными).

Конечно, эффектные примеры, открывающие путь к решению серьезных проблем, наглядно демонстрируют достоинства регулярных выражений. Менее очевиден тот факт, что регулярные выражения используются в повседневной работе для решения «неинтересных» задач — «неинтересных» в том смысле, что программисты вряд ли станут обсуждать их с коллегами в курилке, но без решения этих задач вы не сможете нормально работать.

Приведу простой пример. Однажды мне потребовалось проверить множество файлов (точнее, 70 с лишним файлов с текстом этой книги) и убедиться в том, что в каждом файле строка '`SetSize`' встречается ровно столько же раз, как и строка '`ResetSize`'. Задача усложнялась тем, что регистр символов при подсчете не учитывался (т. е. строки '`setSIZE`' и'`SetSize`' считаются эквивалентными). Конечно, просматривать 32 000 строк текста вручную было бы, по меньшей мере, неразумно. Даже использование стандартных команд поиска в редакторе потребует воистину

титанических усилий, учитывая количество файлов и возможные различия в регистре символов.

На помощь приходят регулярные выражения! Я ввожу всего одну короткую команду, которая проверяет все файлы и выдает всю необходимую информацию. Общие затраты времени — секунд 15 на ввод команды и еще 2 секунды на проверку данных. Потрясающе! (Если вам интересно, как выглядит эта команда, загляните на с. 64.)

Другой пример: однажды я помогал своему другу решить проблемы с электронной почтой на удаленном компьютере, и он захотел, чтобы я отправил ему список всех сообщений из его почтового ящика. Конечно, можно было загрузить копию всего файла с сообщениями в текстовый редактор и вручную удалить в каждом сообщении все строки, кроме заголовка, оставив что-то вроде краткого содержания. Но даже если бы файл не был таким большим и если бы не медленное подсоединение по телефонной линии, эта работа была бы долгой и монотонной. Наконец, читать чужую почту попросту неэтично.

И снова меня выручили регулярные выражения! Я ввел простую команду (используя стандартную утилиту поиска *egrep*, описанную ниже в этой главе) для вывода строк `From:` и `Subject:` каждого сообщения. Чтобы точно указать *egrep*, какие строки должны (или не должны) присутствовать в выходных данных, я воспользовался регулярным выражением `^(From|Subject):`.

Получив список, друг попросил меня отправить одно конкретное сообщение, состоящее из 5000 строк! И в этом случае на извлечение одного сообщения в текстовом редакторе потребовалось бы слишком много времени. Вместо этого я воспользовался другой утилитой (*sed*) и при помощи регулярных выражений точно описал, какая часть текста в файле меня интересует. Это позволило легко и быстро извлечь и отправить нужное сообщение.

Применение регулярных выражений позволило нам обоим сэкономить массу времени и сил. Пусть оно не было особенно «интересным», но в любом случае это интереснее, чем часами просиживать за текстовым редактором. Если бы я не знал о существовании регулярных выражений, мне бы и в голову не пришло, что существует другой выход. Эта характерная история показывает, что при помощи регулярных выражений и тех утилит, в которых они поддерживаются, можно делать совершенно неожиданные вещи.

Научившись пользоваться регулярными выражениями, вы будете удивляться, как же раньше обходились без них[1].

Умение работать с регулярными выражениями — воистину бесценный навык. Информация, приведенная в этой книге, поможет вам этим навыком овладеть.

[1] Если у вас имеется TiVo, то это чувство должно быть вам знакомо!

Надеюсь, мне также удастся убедить вас в том, что потраченные усилия не пропадут зря.

Регулярные выражения как язык

Тем, кому еще не приходилось работать с регулярными выражениями, строка `⌜^(From|Subject):⌟` из предыдущего примера покажется непонятной. На самом деле никакого волшебства здесь нет — как нет его и в выступлениях циркового фокусника. Просто фокусник знает что-то простое, что *не кажется* простым или естественным его неискушенным зрителям. Стоит научиться держать карту так, чтобы рука казалась пустой, и немного потренироваться — и вы тоже сможете «показывать фокусы». Регулярные выражения также можно сравнить с иностранным языком — когда вы начинаете изучать язык, он перестает казаться белибердой.

Аналогия с файловыми шаблонами

Поскольку вы взялись за эту книгу, вероятно, вы хотя бы отчасти представляете себе, что такое «регулярное выражение». Но даже если и не представляете, общий принцип их работы вам наверняка знаком.

Как известно, каждому файлу присваивается конкретное имя (например, *report.txt*). Однако любой пользователь UNIX или DOS/Windows знает, что для выборки нескольких файлов можно воспользоваться шаблоном вида «`*.txt`». В подобных шаблонах (называемых *файловыми глобами*, или *групповыми символами* (*wildcards*)) используются символы, имеющие особый смысл. Звездочка (*) означает «любая последовательность символов», а вопросительный знак (?) — «один произвольный символ». Итак, шаблон «`*.txt`» начинается с `⌜*⌟` и заканчивается строковым литералом `⌜.txt⌟`. Полученный в результате шаблон означает «Выбрать все файлы, имена которых начинаются с любой последовательности символов и заканчиваются символами `.txt`».

Во многих системах существуют дополнительные специальные символы, но в общем случае выразительные возможности файловых шаблонов ограничены. Впрочем, это вряд ли можно считать недостатком, поскольку файловые шаблоны применяются в относительно узкой области — только при работе с именами файлов.

С другой стороны, при работе с текстом вообще возникает гораздо больше проблем. Проза и поэзия, листинги программ, отчеты, HTML, кодовые таблицы, списки слов... при желании продолжайте сами.

Конечно, для узкоспециализированной задачи (например, «выбор файлов») можно разработать специализированную схему или инструмент. Но за долгие годы был

выработан обобщенный язык текстовых шаблонов, обладающий достаточной мощью и выразительностью для применения в разных областях. Каждая программа по-своему реализует и использует эти шаблоны, но в общем случае этот мощный язык и сами шаблоны называются *регулярными выражениями*.

Аналогия с языками

Регулярное выражение состоит из двух типов символов. Специальные символы (вроде * в файловых шаблонах) называются *метасимволами*. Все остальные символы, т. е. обычный текст, называются *литералами*. Регулярные выражения отличаются от файловых шаблонов в первую очередь гораздо большими выразительными возможностями своих метасимволов. В файловых шаблонах используется малое количество метасимволов, предназначенных для ограниченных целей, «язык» же регулярных выражений содержит богатый и впечатляющий набор метасимволов для опытных пользователей.

Регулярные выражения можно рассматривать как самостоятельный язык, в котором литералы выполняют функции слов, а метасимволы — функции грамматических элементов. Слова по определенным правилам объединяются с грамматическими элементами и создают конструкции, выражающие некоторую мысль. Скажем, в примере с электронной почтой я воспользовался регулярным выражением `^(From|Subject):` для поиска строк, начинающихся с '`From:`' или '`Subject:`'. Метасимволы в этом выражении подчеркнуты, а их смысл будет рассмотрен ниже.

На первый взгляд регулярные выражения (как и любой другой незнакомый язык) производят устрашающее впечатление. Они выглядят как магические заклинания, понятные лишь немногим избранным и абсолютно недоступные простым смертным. Но подобно тому, как строка 正規表現は簡単だよ！[1] вскоре становится понятной для изучающего японский язык, регулярное выражение в команде

```
s!<emphasis>([0-9]+(\.[0-9])+){3})</emphasis>!<inet>$1</inet>!
```

вскоре станет абсолютно понятным и для вас.

---

[1] «В регулярных выражениях нет ничего сложного!» Довольно забавный комментарий к этому: в главе 3 говорится, что термин *регулярные выражения* был заимствован из формальной алгебры. Когда люди, не знакомые с концепцией регулярных выражений, спрашивают меня, о чем эта книга, то ответ «о регулярных выражениях» вызывает у них недоумение. Для среднего японца слово 正規表現 несет ничуть не больше смысла, однако мой ответ на японском вызывает нечто большее, чем просто удивление. Оказывается в японском языке слово «регулярный» звучит очень похоже на другое, более распространенное слово, которое в медицинской терминологии можно обозначить как «репродуктивные органы». Только представьте себе, о чем думает мой собеседник, пока я не объясню в чем дело!

Приведенный пример взят из сценария Perl, использованного моим редактором при правке авторского варианта рукописи. Автор ошибочно использовал теги `<emphasis>` для выделения IP-адресов (которые выглядят как набор чисел, разделенных точками, например `209.204.146.22`). В этой строке команда Perl, предназначенная для поиска/замены текста, в сочетании с регулярным выражением

```
s/<emphasis>([0-9]+(\.[0-9])+){3})</emphasis>/<inet>$1</inet>/gi;
```

заменяет их тегами `<inet>`, оставляя прочие теги `<emphasis>` без изменения. В следующих главах вы узнаете, как конструируются подобные выражения, и сможете использовать их в своих целях в контексте приложения или языка программирования.

Цель этой книги

Вряд ли лично *вам* когда-нибудь придется заменять теги `<emphasis>` тегами `<inet>`, но похожие задачи типа «найти *то-то* и заменить *тем-то*» возникают довольно часто. Эта книга написана не для того, чтобы снабдить вас готовыми решениями конкретных проблем. Она научит вас *мыслить* категориями регулярных выражений, чтобы вы могли успешно справиться с любой задачей такого типа.

Регулярные выражения как особый склад ума

Как вы вскоре узнаете, большие регулярные выражения строятся из маленьких «кирпичей». Сами по себе эти «кирпичи» просты, но в сочетании друг с другом они образуют бесконечное множество комбинаций. Чтобы научиться правильно объединять их для достижения желаемой цели, вам потребуется некоторый опыт, поэтому в этой главе я хочу в общих чертах представить вам некоторые концепции регулярных выражений. Не углубляясь в подробности, этот обзор закладывает основу для всего материала книги и готовит почву для важных вопросов, которые лучше обсудить до того, как мы займемся самими регулярными выражениями.

Хотя некоторые примеры выглядят глупыми (потому что они действительно глупые), они хорошо соответствуют тем задачам, которые вам предстоит решать, — пусть вы даже этого еще не понимаете. Если что-то из сказанного покажется непонятным, не огорчайтесь. Постарайтесь уловить общий смысл. Именно с этой целью написана данная глава.

Для читателей, имеющих опыт работы с регулярными выражениями

Читатели, знакомые с регулярными выражениями, вряд ли найдут в этом обзоре что-нибудь принципиально новое, но, пожалуйста, хотя бы просмотрите его. Даже

если вы знаете смысл некоторых метасимволов, какие-то особенности или аспекты применения регулярных выражений окажутся новыми и для вас.

Подобно различиям между хорошим исполнением музыкального произведения и *сотворением музыки* существуют различия между пониманием регулярных выражений и их *настоящим пониманием*. Возможно, информация, представленная в некоторых уроках, окажется знакомой, но будет представлена под новым углом зрения, и это окажется первым шагом на пути к *настоящему пониманию*.

Поиск в текстовых файлах: egrep

Одним из простейших применений регулярных выражений является поиск текста — во многих текстовых редакторах и текстовых процессорах предусмотрена возможность поиска по шаблонам регулярных выражений. Еще более простым примером является утилита *egrep*. При запуске программе *egrep* передается регулярное выражение и список просматриваемых файлов. Утилита сопоставляет регулярное выражение с каждой строкой файла и выводит только те строки, в которых было найдено совпадение. Бесплатные версии *egrep* существуют во многих системах, включая DOS, MacOS, Windows, UNIX и т. д.

Вернемся к примеру с электронной почтой. Команда, использованная для построения оглавления по файлу почтового ящика, показана на рис. 1.1. Утилита *egrep* интерпретирует первый аргумент строки как регулярное выражение, а остальные аргументы — как имена просматриваемых файлов. Обратите внимание: апострофы, присутствующие на рис. 1.1, не входят в регулярное выражение, но их присутствия требует мой командный интерпретатор[1]. При использовании *egrep* я почти всегда заключаю регулярные выражения в апострофы. Какие символы имеют особый смысл, в каком контексте (регулярного выражения или программы) и в каком порядке они интерпретируются — все эти вопросы особенно важны при работе с регулярными выражениями в полноценных языках программирования. Мы начнем рассматривать эту тему, начиная со следующей главы.

---

[1] Командным интерпретатором называется часть операционной системы, которая обрабатывает введенные команды и запускает указанные в них программы. В том интерпретаторе, которым я пользуюсь, апострофы предназначаются для группировки аргументов. Они говорят о том, что командный интерпретатор не должен обращать внимания на заключенные в них символы. Если бы апострофы отсутствовали, командный интерпретатор мог бы решить, что символ '*' из регулярного выражения в действительности является частью файлового шаблона, который должен интерпретировать именно *он*, а не *egrep*. Конечно, такого быть не должно, поэтому я при помощи апострофов «скрываю» метасимволы от командного интерпретатора. Пользователи интерпретатора COMMAND.COM и CMD.EXE системы Windows вместо апострофов используют кавычки.

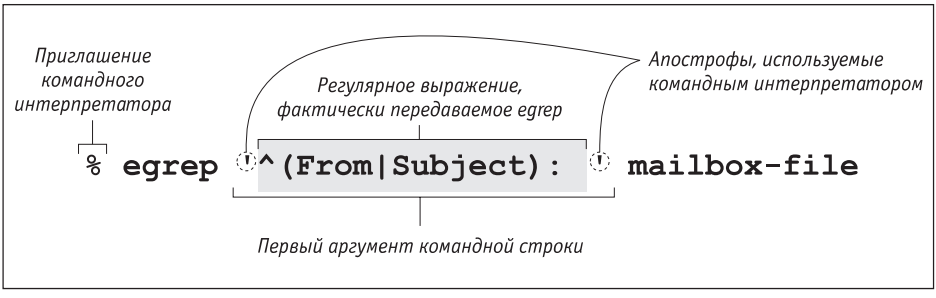


Рис. 1.1. Запуск egrep из командной строки

Вскоре мы проанализируем основные компоненты регулярных выражений, и вы узнаете, что означает каждый из них, но и сейчас нетрудно догадаться, что некоторые символы интерпретируются особым образом. В нашем примере круглые скобки, символы ⌜^⌟ и ⌜|⌟ относятся к категории метасимволов регулярных выражений и используются в сочетании с другими символами для достижения желаемого результата.

С другой стороны, если в регулярном выражении не используется ни один из десятка с лишним метасимволов, поддерживаемых *egrep*, оно фактически превращается в средство «простого поиска текста». Например, при поиске выражения ⌜`cat`⌟ будут найдены и выведены все строки файла, содержащие три стоящих подряд буквы `c·a·t`. Например, среди них будут выведены строки, в которых встречается слово `vacation`.

Даже если в строке нет отдельного *слова* `cat`, последовательность `c·a·t` в слове `vacation` все равно считается успешно найденной. Необходимо только наличие указанных символов, и поскольку символы присутствуют, *egrep* выводит всю строку. Ключевым моментом здесь является то, что поиск осуществляется не на уровне «слов» — *egrep* различает байты и строки файла, но обычно не имеет ни малейшего представления о языках, предложениях, абзацах или других концепциях высокого уровня.

Метасимволы egrep

Начнем с рассмотрения некоторых метасимволов, используемых *egrep* при работе с регулярными выражениями. Существует несколько типов метасимволов, выполняющих разные функции. Мы в общих чертах познакомимся с ними в нескольких примерах, а подробные описания и многочисленные примеры будут приведены в последующих главах.

Прежде чем читать дальше, пожалуйста, просмотрите список условных обозначений в предисловии. В книге используются некоторые нестандартные обозначения, и смысл каких-то из них на первых порах может оказаться неочевидным.

Начало и конец строки

Вероятно, простейшими метасимволами являются ⌜^⌟ (крышка, циркумфлекс) и ⌜$⌟ (доллар), представляющие соответственно начало и конец проверяемой строки. Как говорилось выше, регулярное выражение ⌜cat⌟ находит последовательность символов c·a·t в любом месте строки, но для выражения ⌜^cat⌟ совпадение происходит лишь в том случае, если символы c·a·t находятся в начале строки — ⌜^⌟ фактически *привязывает* совпадение (остальной части регулярного выражения) к началу строки. Аналогично, выражение ⌜cat$⌟ находит символы c·a·t только в том случае, если они находятся в конце строки — например, если строка завершается словом scat.

Развивайте в себе привычку буквально интерпретировать регулярные выражения. Например, не надо думать:

> ⌜^cat⌟ совпадает, если строка начинается с cat.

Думать нужно так:

> ⌜^cat⌟ совпадает, если мы находимся в начале строки, после которой сразу же следует символ c, потом сразу следует символ a, и потом сразу следует символ t.

Фактически это обозначает одно и то же, но побуквенная интерпретация позволит лучше понять суть нового выражения, когда оно вам встретится. Как бы вы прочитали выражение ⌜^cat$⌟, ⌜^$⌟ и даже простейшее ⌜^⌟? ❖ Проверить правильность своей интерпретации вы сможете, прочитав ответ во врезке на следующей странице.

Специфика символов ^ и $ заключается в том, что они совпадают с определенной *позицией* в строке, а не с символами текста. Конечно, большая часть метасимволов предназначена для поиска именно текста. Помимо символов-литералов в регулярных выражениях также можно использовать метасимволы, описанные в нескольких следующих разделах.

Символьные классы

Совпадение с одним символом из нескольких возможных

Допустим, необходимо найти строку «grey», которая также может быть записана в виде «gray». При помощи конструкции ⌜[…]⌟, называемой *символьным классом* (*character class*), можно перечислить символы, которые могут находиться в данной позиции текста. Выражение ⌜e⌟ совпадает только с буквой e, выражение ⌜a⌟ совпадает только с буквой a, но регулярное выражение ⌜ea⌟ совпадает с любой из этих букв. Таким образом, выражение ⌜gr[ea]y⌟ означает: «Найти символ g, за которым следует r, за которым следует e или a, и все это завершается символом y». В орфографии я не силен и поэтому всегда использую такие регулярные выражения для поиска

ИНТЕРПРЕТАЦИЯ ⌜^CAT$⌟, ⌜^$⌟ И ⌜^⌟

❖ *Ответ на вопрос со с. 33.*

⌜^cat$⌟ **Буквально**: совпадает, если у строки есть начало (конечно, оно есть у любой строки), за которым сразу же следуют символы `c·a·t`, после чего немедленно следует конец строки.

Фактически: строка состоит только из слова `cat` — никаких дополнительных слов, пробелов, знаков препинания... просто «cat» и ничего больше.

⌜^$⌟ **Буквально**: совпадает, если у строки есть начало, после которого немедленно следует конец строки.

Фактически: пустая строка (не содержащая никаких символов, даже пробелов).

⌜^⌟ **Буквально**: совпадает, если у строки есть начало.

Фактически: *бессмысленно*! Начало есть у любой строки, поэтому совпадают все строки — даже пустые!

правильных вариантов написания в списках слов. В частности, я нередко использую выражение ⌜`sep[ea]r[ea]te`⌟, потому что никак не могу запомнить, как же правильно пишется это слово — «seperate», «separate», «separete» или как-нибудь еще.

Вне символьных классов предполагается, что перечисляемые литералы (например, ⌜`g`⌟ и ⌜`r`⌟ в выражении ⌜`gr[ae]y`⌟) следуют непосредственно друг за другом — «найти совпадение для ⌜`g`⌟, за которым следует совпадение для ⌜`r`⌟ и т. д.). Внутри символьного класса дело обстоит совершенно иначе. Содержимое класса определяет список символов, с которыми совпадает данный символ регулярного выражения, поэтому здесь подразумевается связка «или».

Еще один пример — возможная смена регистра в первой букве слова: ⌜`[Ss]mith`⌟. Не забывайте, что результат совпадения будет также включать строки, в которых последовательность `smith` (или `Smith`) находится внутри другого слова — например, `blacksmith`. Я не хочу снова и снова напоминать об этом, но у новичков нередко возникают проблемы. Некоторые решения проблемы «внутренних» слов будут рассмотрены ниже, после того как мы рассмотрим еще несколько метасимволов.

Количество символов в классе может быть любым. Например, класс ⌜`[123456]`⌟ совпадает с любой из перечисленных цифр. Этот класс входит в выражение ⌜`<H[123456]>`⌟, совпадающее с последовательностями вида `<H1>`, `<H2>`, `<H3>` и т. д. Такое выражение может использоваться при поиске заголовков HTML.

В контексте символьного класса метасимвол *символьного класса* '-' (*дефис*) обозначает интервал символов; так, выражение ⌜<H[1-6]>⌟ эквивалентно предыдущему примеру. Классы ⌜[0-9]⌟ и ⌜[a-z]⌟ обычно используются соответственно для поиска цифр и символов нижнего регистра.

Символьный класс может содержать несколько интервалов, поэтому класс ⌜[0123456789abcdefABCDEF]⌟ записывается в виде ⌜[0-9a-fA-F]⌟ (или ⌜[A-Fa-f0-9]⌟, поскольку порядок перечисления роли не играет). Такое выражение пригодится при обработке шестнадцатеричных чисел. Интервалы также можно объединять с литералами: выражение ⌜[0-9A-Z_!.?]⌟ совпадает с цифрой, символом верхнего регистра, символом подчеркивания, восклицательным знаком, точкой или вопросительным знаком.

Обратите внимание: дефис выполняет функции метасимвола только внутри символьного класса — в остальных случаях он совпадает с обычным дефисом. Более того, даже в символьных классах дефис не всегда интерпретируется как метасимвол. Если дефис является первым символом, указанным в классе, он заведомо не может определять интервал и поэтому интерпретируется как литерал. Аналогично, вопросительный знак и точка в конце класса считаются метасимволами в контексте обычных регулярных выражений, но не в контексте класса. Таким образом, в классе ⌜[0-9A-Z_!.?]⌟ специальное значение имеют только два дефиса.

> Символьные классы можно рассматривать как своеобразный мини-язык. Набор поддерживаемых метасимволов (и их интерпретация) различны внутри класса и за его пределами.

Далее мы рассмотрим примеры этого.

Инвертированные символьные классы

Если вместо ⌜[…]⌟ используется запись ⌜[^…]⌟, класс совпадает с любыми символами, не входящими в приведенный список. Например, ⌜[^1-6]⌟ совпадает с символом, *не* принадлежащим интервалу от 1 до 6. Префикс ^ в каком-то смысле «инвертирует» список, вместо того чтобы перечислять символы, принадлежащие классу, вы перечисляете символы, не входящие в него.

Возможно, вы заметили, что для инвертирования классов используется тот же символ ^, который отмечает начало строки. Символ действительно тот же, но смысл у него совсем другой. Например, слово «крыша» в зависимости от контекста может иметь совершенно разный смысл; то же самое можно сказать и о метасимволах. Мы уже встречались с одним примером множественной интерпретации — дефисом. Дефис интерпретируется как определитель интервалов только в символьном классе (и то если он не находится в первой позиции). За пределами символьного класса символ ^ выполняет привязку позиции к началу строки, внутри класса он является

метасимволом класса, но лишь в том случае, если следует сразу же после открывающей скобки (в противном случае он интерпретируется как обычный символ). Не пугайтесь, это самые сложные примеры множественной интерпретации символов; в остальных случаях дело обстоит проще.

Рассмотрим другой пример. Как известно, в английском языке за буквой `q` практически всегда следует `u`. Давайте поищем экзотические слова, в которых за буквой `q` следует какой-нибудь другой символ — в переводе на язык регулярных выражений это выглядит как `q[^u]`. Я применил это выражение к своему списку слов. Как и следовало ожидать, таких слов оказалось немного! Более того, о существовании некоторых из найденных слов я вообще не подозревал.

Вот как это выглядело:

```
% egrep 'q[^u]' word.list
Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum%
```

В списке нет слов «Qantas» (австралийская авиакомпания) и «Iraq». Хотя оба слова присутствуют в файле *word.list*, ни одно из них не попало в результаты поиска. Почему? ❖ Подумайте, а затем проверьте свои предположения, прочитав ответ на с. 38.

Помните: инвертированный символьный класс означает «совпадение с символами, не входящими в список», а не «несовпадение с символами, входящими в список». На первый взгляд кажется, что это одно и то же, однако пример со словом `Iraq` демонстрирует отличия между этими двумя трактовками. Инвертированный класс удобно рассматривать как сокращенную форму записи для обычного класса, включающего все возможные символы, кроме перечисленных.

Один произвольный символ

Метасимвол `.` (точка) представляет собой сокращенную форму записи для символьного класса, совпадающего с любым символом. Он применяется в тех случаях, когда в некоторых позициях регулярного выражения могут находиться произвольные символы. Допустим, надо найти дату, которая может быть записана в формате `19/03/76`, `19-03-76` или даже `19.03.76`. Конечно, можно сконструировать регулярное выражение, в котором между числами указываются все допустимые символы-разделители ('/', '-' и '.'), например `19[-./]03[-./]76`. Однако возможен и другой вариант — просто ввести выражение `19.03.76`.

В приведенном примере имеется ряд неочевидных аспектов. В выражении ⌜19[-./]03[-./]76⌟ точки *не* являются метасимволами, поскольку они находятся внутри символьного класса (не забывайте: состав и интерпретация метасимволов различаются внутри класса и за его пределами). Дефисы в данном случае тоже интерпретируются как литералы, поскольку они следуют сразу же после [или [^. Если бы дефисы не стояли на первых местах (например, ⌜[.-/]⌟), они интерпретировались бы как интервальные метасимволы, что в данном случае привело бы к ошибке.

В выражении ⌜19.03.76⌟ точки являются метасимволами, совпадающими с любым символом, в том числе и с ожидаемыми нами '/', '-' и '.'. Тем не менее необходимо учитывать, что каждая точка может совпадать с абсолютно любым символом, поэтому совпадение обнаруживается, например, в строке '`lottery numbers: 19 203319 7639`'.

Итак, выражение ⌜19[-./]03[-./]76⌟ точнее определяет область допустимых совпадений, но его труднее читать и записывать. Выражение ⌜19.03.76⌟ легко понять, но оно дает неоднозначный результат. Какой вариант использовать? Все зависит от того, что вам известно об искомых данных и насколько точным должен быть поиск. При построении регулярных выражений часто приходится идти на компромисс с точностью за счет знания текста. Например, если вы уверены, что в вашем тексте выражение ⌜19.03.76⌟ наверняка не вызовет нежелательных совпадений, будет вполне логично воспользоваться именно этим вариантом. Знание целевого текста — важный фактор, обеспечивающий эффективное использование регулярных выражений.

Выбор

Одно из нескольких подвыражений

Очень удобный метасимвол ⌜|⌟ означает «или». Он позволяет объединить несколько регулярных выражений в одно, совпадающее с любым из выражений-компонентов. Например, ⌜Bob⌟ и ⌜Robert⌟ — два разных выражения, а ⌜Bob|Robert⌟ — одно выражение, совпадающее с любой из этих строк. Подвыражения, объединенные этим способом, называются *альтернативами* (*alternatives*).

Вернемся к примеру ⌜gr[ea]y⌟. Любопытная подробность: выражение также можно записать в виде ⌜grey|gray⌟ и даже ⌜gr(a|e)y⌟. В последнем варианте круглые скобки отделяют конструкцию выбора от остального выражения (и, кстати говоря, тоже являются метасимволами). Конструкция вида ⌜gr[a|e]y⌟ нам *не* подойдет — в символьном классе символ '|' является обычным символом, как ⌜a⌟ или ⌜e⌟.

В выражении ⌜gr(a|e)y⌟ круглые скобки обязательны, поскольку без них ⌜gra|ey⌟ будет означать «⌜gra⌟ или ⌜ey⌟» — совсем не то, что нам нужно. Конструкция выбора действует только внутри круглых скобок. Рассмотрим другой пример:

ОТВЕТ НА ВОПРОС

❖ *Ответ на вопрос со с. 36.*

Почему ⌜q[^u]⌟ не совпадает со словами 'Qantas' или 'Iraq'?

`Qantas` не совпадает, поскольку в регулярном выражении указан символ `q` в нижнем регистре, а в слове «Qantas» он относится к верхнему регистру. Если использовать выражение ⌜`Q[^u]`⌟, будет найдено это слово, но зато пропущены все остальные. Выражение ⌜`[Qq][^u]`⌟ обнаружило бы все слова.

В примере со словом `Iraq` кроется подвох. В регулярном выражении указан символ `q`, *за которым следует символ*, отличный от `u`, что вполне может соответствовать символу перевода строки. Проверяемый текст обычно завершается символом перевода строки, но поскольку перед проверкой *egrep* удаляет эти символы (простите, забыл упомянуть об этом!), после `q` нет вообще никаких данных. Там нет никакого символа, отличного от `u`, который можно было бы принять за совпадение.

Не огорчайтесь из-за того, что вопрос оказался неожиданно сложным[1]. Уверяю вас: если бы программа *egrep* не удаляла символы перевода строк (как это делают некоторые другие программы) или если бы за словом `Iraq` следовали пробелы, другие слова или еще что-нибудь, строка была бы успешно обнаружена. Необходимо хорошо разбираться в тонкостях работы каждой программы, но пока из этого примера необходимо вынести одно: *символьному классу, даже инвертированному, в тексте обязательно должен соответствовать какой-нибудь символ.*

⌜`(First]|1st)•[Ss]treet`⌟[2]. Вообще говоря, поскольку ⌜`First`⌟ и ⌜`1st`⌟ заканчиваются строкой ⌜`st`⌟, выражение можно сократить до ⌜`Fir|1)st•[Ss]treet`⌟. Стоит иметь в виду, что читать его будет сложнее, а также что выражения ⌜`(first|1st)`⌟ и ⌜`(fir|1)st`⌟ фактически дают тот же результат.

В следующем примере мы рассмотрим несколько вариантов написания моего имени. Сравните следующие три выражения, которые означают фактически одно и то же:

⌜`Jeffrey|Jeffery`⌟
⌜`Jeff(rey|ery)`⌟
⌜`Jeff(re|er)y`⌟

[1] Однажды в четвертом классе на уроке орфографии учительница спросила меня, как пишется слово «miss», а я всегда был первый в классе по этому предмету. Я ответил: «`m·i·s·s`». Мисс Смит со счастливой улыбкой сказала, что это неправильно, а правильно будет: «`M·i·s·s`» — с заглавной буквы `M` и что сначала я должен был попросить пример предложения. Это был сильный удар по детской психике. После этого случая я невзлюбил мисс Смит и так и не научился писать грамотно.

[2] Напомню, что символ '•' используется для обозначения пропуска.

Чтобы выражения также учитывали вариант написания, принятый в Великобритании, они принимают следующий вид:

```
(Geoff|Jeff)(rey|ery)
(Geo|Je)ff(rey|ery)
(Geo|Je)ff(re|er)y
```

Следует заметить, что эти три варианта эквивалентны более длинной (но более понятной) записи `Jeffrey|Geoffery|Jeffery|Geoffrey`. Все это разные способы для определения одних и тех же условий совпадения.

Впрочем, сравнение `gr[ea]y` с `gr(a|e)y` слегка отвлекло нас от основной темы. Будьте внимательны и не путайте конструкцию выбора с символьными классами. Символьный класс представляет *один символ* целевого текста. В конструкциях выбора каждая альтернатива может являться полноценным регулярным выражением, совпадающим с произвольным количеством символов. Символьные классы, можно считать, обладают собственным мини-языком (и, в частности, собственными представлениями о метасимволах), тогда как конструкция выбора является частью «основного» языка регулярных выражений. Как вы сможете убедиться, обе конструкции в высшей степени полезны.

Кроме того, будьте внимательны при использовании знаков `^` и `$` в выражениях с конструкциями выбора. Сравните два выражения: `^From|Subject|Date:•` и `^(From|Subject|Date):•`. Они напоминают рассмотренный выше пример с электронной почтой, но имеют разный смысл (а значит, и разную степень полезности). Первое выражение состоит из трех простых альтернатив; оно означает «`^From`, или `Subject`, или `Date:•`» и потому особой пользы не приносит. Нам нужно, чтобы префикс `^` и суффикс `:•` относились к каждой из альтернатив. Для этого конструкция выбора «ограничивается» круглыми скобками:

```
^(From|Subject|Date):•
```

Действие выбора ограничивается круглыми скобками, поэтому приведенное выражение буквально означает: «начало строки, затем одна из подстрок `From`, `Subject` или `Date` и затем `:•`». Оно совпадает в следующих трех случаях:

1) начало строки, символы `F·r·o·m`, а затем '`:•`' *или*

2) начало строки, символы `S·u·b·j·e·c·t`, а затем '`:•`' *или*

3) начало строки, символы `D·a·t·e`, а затем '`:•`'.

Проще говоря, совпадение происходит в каждой строке, которая начинается либо с '`From:•`', либо с '`Subject:•`', либо с '`Date:•`', — именно то, что нам нужно для получения списка сообщений из файла электронной почты.

Пример:

```
% egrep '^(From<Subject<Date): ' mailbox
From: elvis@tabloid.org (The King)
Subject: be seein' ya around
Date: Mon, 23 Oct 2006 11:04:13
From: The Prez <president@whitehouse.gov>
Date: Wed, 25 Oct 2006 8:36:24
Subject: now, about your vote…
⋮
```

Игнорирование различий в регистре символов

Рассматривая пример с почтовым ящиком, будет уместно представить концепцию поиска совпадения без *учета регистра*. Поля в заголовках сообщений электронной почты обычно начинаются с прописной буквы (например, «Subject» или «From»), однако стандарт в действительности допускает произвольное сочетание регистра символов, поэтому поля «DATE» и «from» тоже допустимы. К сожалению, регулярное выражение из предыдущего раздела с такими полями не совпадет.

Одно из возможных решений — заменить `⌜From⌟` выражением `⌜[Ff][Rr][Oo][Mm]⌟`, совпадающим с любым вариантом написания слова «from», но подобная конструкция выглядит по меньшей мере громоздко. К счастью, мы можем приказать *egrep* полностью игнорировать регистр при сравнениях, т. е. выполнять поиск совпадений *без учета регистра*. Данная возможность не является частью языка регулярных выражений; это вспомогательное средство, поддерживаемое многими языками. Режим поиска без учета регистра активизируется запуском *egrep* с ключом `-i`. Ключ размещается в командной строке перед регулярным выражением:

```
% egrep -i '^(From;Subject;Date): ' mailbox
```

В этом случае будут найдены все те же строки, что и раньше, но к ним добавятся строки вида:

```
SUBJECT: MAKE MONEY FAST
```

Я довольно часто использую ключ `-i` и рекомендую его читателям. Ниже мы встретимся с другими вспомогательными средствами подобного рода.

Границы слов

Одна из распространенных проблем, возникающих при поиске, заключается в том, что регулярное выражение случайно совпадает с последовательностью символов, входящих в другое слово. Я уже упоминал об этом в примерах с `cat`, `gray` и `Smith`.

Хотя я также говорил о том, что *egrep* обычно не воспринимает концепции слов, в некоторых версиях *egrep* реализована их ограниченная поддержка — а именно возможность привязки к границе слова (началу или концу).

Вы можете использовать странноватые *метапоследовательности* `\<` и `\>`, если они поддерживаются вашей версией *egrep*. Они представляют собой аналоги `^` и `$` на уровне слов и обозначают позицию, находящуюся соответственно в начале и в конце слова. Как и якоря `^` и `$`, эти метапоследовательности обеспечивают привязку других частей регулярного выражения к определенной позиции и не соответствуют конкретным символам текста. Регулярное выражение `\<cat\>` буквально означает «начало слова, за которым немедленно следуют символы `c·a·t`, а затем идет конец слова». Проще говоря, это означает «найти отдельное слово `cat`». При желании можно воспользоваться выражениями `\<cat` или `cat\>` для поиска слов, начинающихся и заканчивающихся символами `cat`.

Обратите внимание: сами по себе `<` и `>` метасимволами не являются — особый смысл приобретает только *комбинация* «обратный слэш + символ». Именно поэтому я и назвал их «метапоследовательностями». Причем здесь важна их особая интерпретация, а не количество символов, поэтому в большей части книги я буду считать эти два «мета»-термина синонимами.

Помните о том, что не все версии *egrep* поддерживают метасимволы границ слов, но даже при наличии такой поддержки они не начинают волшебным образом понимать английский язык. «Началом слова» просто считается та позиция, с которой начинается последовательность алфавитно-цифровых символов; «концом слова» считается позиция, в которой эта последовательность завершается. На рис. 1.2 приведен пример строки с разметкой этих позиций.

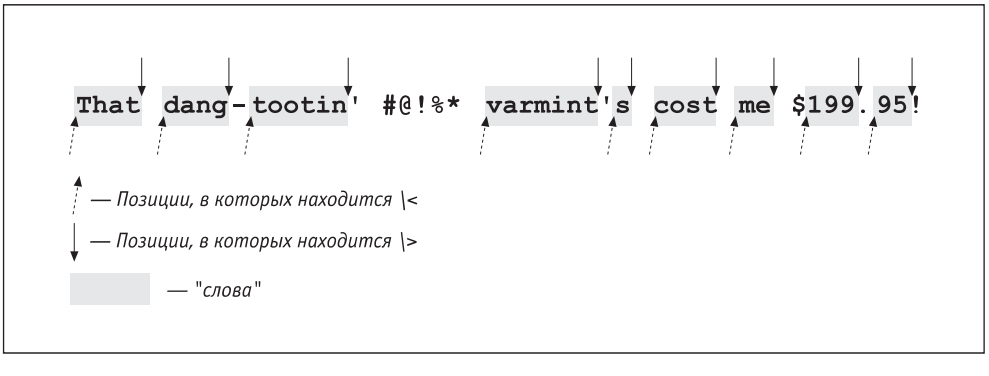


Рис. 1.2. Позиции начала и конца слов

Начала слов (как их опознает *egrep*) отмечены стрелками, направленными вверх; концы слов отмечены стрелками, направленными вниз. Как видите, «начало и ко-

нец слова» правильнее было бы называть «началом и концом алфавитно-цифровой последовательности», но это получается слишком длинно.

В двух словах

В табл. 1.1 перечислены все метасимволы, упоминавшиеся до настоящего момента.

Таблица 1.1. Сводка упоминавшихся метасимволов

Метасимвол	Название	Смысл
`.`	Точка	Один любой символ
`[...]`	Символьный класс	Любой символ в списке
`[^...]`	Инвертированный символьный класс	Любой символ, не входящий в список
`^`	Циркумфлекс	Позиция в начале строки
`$`	Доллар	Позиция в конце строки
`\<`	Обратный слэш + знак меньше	Позиция в начале слова*
`\>`	Обратный слэш + знак больше	Позиция в конце слова*
`|`	Вертикальная черта	Любое из разделяемых выражений
`(...)`	Круглые скобки	Ограничивает действие `|`, а также используется в других случаях
* Не поддерживается некоторыми версиями *egrep*.		

Также следует запомнить несколько важных фактов.

- В символьных классах существуют особые правила, определяющие, какие символы являются или не являются метасимволами (а также их точную интерпретацию). Например, точка считается метасимволом за пределами класса, но не внутри него. И наоборот, дефис (в общем случае) является метасимволом внутри класса, но не за его пределами. Более того, символ `^` имеет один смысл за пределами класса, другой смысл — внутри класса сразу же после открывающей скобки `[` и третий — в любой другой позиции класса.

- Не путайте конструкцию выбора с символьным классом. Класс `[abc]` и конструкция выбора `(a|b|c)` фактически означают одно и то же, но эта эквивалентность относится только к конкретному примеру. Символьный класс совпадает ровно с одним символом, какой длины ни был бы список допустимых символов.

- С другой стороны, конструкция выбора может содержать альтернативы произвольной длины, текстуально не связанные друг с другом: `\<(1,000,0001|million`

`|thousand•thou)\>`⌋. В отличие от символьных классов, конструкции выбора не могут инвертироваться.

- Инвертированный символьный класс предоставляет дополнительное удобство для записи символьного класса, который соответствует любым символам, отсутствующим в списке. Таким образом, класс ⌈`[^x]`⌋ означает «соответствие любому символу, отличному от `x`», а вовсе не «соответствие считается обнаруженным, если отсутствует символ `x`». Различие между этими формулировками едва уловимо, но оно имеет существенное значение. Во втором случае под такую формулировку подпадает даже пустая строка, тогда как класс ⌈`[^x]`⌋ определенно не соответствует ей.
- Удобный ключ `-i` позволяет выполнять поиск соответствий без учета регистра символов (☞ 40).

Все, что рассматривалось до сих пор, безусловно несет немалую пользу, но истинная мощь заключается в *необязательных* и *вычисляемых* элементах, которые рассматриваются ниже.

Необязательные элементы

Рассмотрим пример поиска слова `color` или `colour`. Поскольку эти два слова отличаются только символом `u`, для их поиска можно использовать выражение ⌈`colou?r`⌋. Метасимвол ⌈`?`⌋ (*знак вопроса*) означает, что предшествующий ему символ является *необязательным*. Он помещается сразу же вслед за символом, который может появиться в этом месте в выражении, но его отсутствие не должно влиять на успешный результат поиска.

В отличие от других метасимволов, обсуждавшихся до сих пор, знак вопроса воздействует только на предшествующий ему элемент выражения. Таким образом, выражение ⌈`colou?r`⌋ интерпретируется как «⌈`c`⌋, за которым следует ⌈`o`⌋, затем ⌈`l`⌋, затем ⌈`o`⌋, затем ⌈`u?`⌋ и затем ⌈`r`⌋».

В части ⌈`u?`⌋ выражение всегда будет иметь истинное значение: в одних случаях эта часть будет соответствовать имеющемуся символу `u`, в других случаях — его отсутствию. Необязательный элемент, отмеченный метасимволом `?`, всегда будет давать успешный результат. Нельзя сказать, что любое регулярное выражение, содержащее `?`, всегда будет обнаруживать соответствие. Например, применительно к слову '`semicolon`' первые два выражения — ⌈`colo`⌋ и ⌈`u?`⌋ — будут обнаруживать соответствие (совпадает `colo` и отсутствует `u`, соответственно). Однако выражение ⌈`r`⌋ в конце будет терпеть неудачу, что не позволит считать слово `semicolon` соответствующим выражению ⌈`colou?r`⌋.

Рассмотрим другой пример — поиск даты, представляющей 4 июля (July), где название месяца «July» может быть записано как `July` или `Jul`, а число может быть записано как `fourth`, `4th` или просто `4`. Безусловно, можно было бы использовать выражение ⌜`(July|Jul)•(fourth|4th|4)`⌟, однако мы попробуем выразить то же самое другим способом.

Прежде всего, часть ⌜`(July|Jul)`⌟ можно сократить до ⌜`(July?)`⌟. Как видите, эти два выражения фактически эквивалентны. Ликвидация метасимвола ⌜`|`⌟ позволяет убрать круглые скобки, ставшие ненужными. Строго говоря, наличие круглых скобок не повлияет на конечный результат, однако выражение ⌜`July?`⌟ без скобок воспринимается немного проще. В результате мы получили выражение ⌜`July?•(fourth|4th|4)`⌟.

Переходим ко второй половине выражения. ⌜`4th|4`⌟ можно сократить до ⌜`4(th)?`⌟. Как видите, ⌜`?`⌟ может присоединяться и к выражениям в круглых скобках. Подвыражение внутри скобок может быть сколь угодно сложным, но «снаружи» оно воспринимается как единое целое. Группировка для ⌜`?`⌟ (и других аналогичных метасимволов, рассматриваемых ниже) является одним из главных применений круглых скобок.

Итак, наше выражение принимает вид ⌜`July?•(fourth |4(th)?`⌟. Хотя оно содержит довольно много метасимволов и даже вложенные круглые скобки, расшифровать и понять его не так уж трудно. Мы довольно долго обсуждали два простых примера, но при этом затронули многие побочные вопросы, которые значительно (хотя, возможно, и на подсознательном уровне) расширяют понимание регулярных выражений. Кроме того, вы убедились, что одну и ту же задачу можно решить несколькими способами. По мере изложения материала (и обретения опыта работы с регулярными выражениями) вы будете открывать новые возможности для проявления своих творческих наклонностей при поиске оптимального решения сложной задачи. Построение регулярных выражений имеет мало общего с сухой и формальной теорией, оно сродни искусству.

Другие квантификаторы: повторение

У вопросительного знака имеются родственники: ⌜`+`⌟ (*плюс*) и ⌜`*`⌟ (*звездочка*). Метасимвол ⌜`+`⌟ обозначает «один или несколько экземпляров непосредственно предшествующего элемента», а ⌜`*`⌟ — «любое количество экземпляров элемента (в том числе и нулевое)». Иначе говоря, ⌜`…*`⌟ означает «найти столько экземпляров, сколько это возможно, но при необходимости обойтись и без них». Конструкция ⌜`…+`⌟ имеет похожий смысл (она также пытается найти как можно большее число экземпляров указанного элемента), но при отсутствии хотя бы одного экземпляра сопоставление завершается неудачей. Три метасимвола, которые могут совпадать с переменным

количеством экземпляров элемента, ⌜?⌟, ⌜+⌟ и ⌜*⌟, называются *квантификаторами*, поскольку они определяют количество элементов, на которые воздействуют.

Совпадение для конструкции ⌜…*⌟, как и для ⌜…?⌟, существует всегда. Вопрос лишь в том, какой текст будет (и будет ли вообще) содержаться в совпадении. Сравните с конструкцией ⌜…+⌟, требующей наличия хотя бы одного экземпляра искомого текста.

Например, выражение ⌜•?⌟ допускает не более одного необязательного пробела, тогда как ⌜•*⌟ — *произвольное число* необязательных пробелов. Применение квантификатора позволит сделать наш пример со с. 34 — `<H[1-6]>` более гибким. В спецификации HTML[1] говорится, что непосредственно перед закрывающей угловой скобкой > допускаются пробелы — например, `<H3•>` или `<H4•••>`. Вставляя ⌜•*⌟ в ту позицию регулярного выражения, где могут находиться (а могут и отсутствовать) пробелы, мы получаем ⌜`<H[1-6]•*`⌟. Выражение по-прежнему совпадает с `<H1>`, поскольку наличие пробелов необязательно, но при этом также подходит и для других вариантов.

Сделаем шаг вперед и попробуем организовать поиск тегов HTML вида `<HR•SIZE=14>`, этот тег означает, что на экране рисуется горизонтальная линия толщиной 14 пикселов. Как и в примере `<H3>`, перед закрывающей угловой скобкой могут стоять необязательные пробелы. Кроме того, пробелы могут находиться и по обе стороны знака `=`. Наконец, минимум один пробел должен разделять `HR` и `SIZE`, хотя их может быть и больше. В последнем случае можно применить выражение ⌜•*⌟, но мы воспользуемся ⌜•+⌟. Плюс разрешает дополнительные пробелы, но требует обязательного присутствия хотя бы одного пробела. Результат эквивалентен ⌜••*⌟, но выглядит более логично. В итоге мы приходим к выражению ⌜`<HR•+SIZE•*=•*14•*>`⌟.

При всей гибкости по отношению к пробелам наше выражение по-прежнему жестко фиксирует толщину линии, заданную в теге. Теперь вместо поиска тегов с одной конкретной толщиной линии (например, `14`) мы хотим найти все варианты. Для этого ⌜`14`⌟ заменяется выражением для поиска обобщенного числа из одной или нескольких цифр. Цифра определяется выражением ⌜`[09]`⌟, а чтобы отыскать «одну или несколько цифр», достаточно добавить символ `+`, поэтому в результате ⌜`14`⌟ заменяется ⌜`[0-9]+`⌟. Символьный класс является отдельным «элементом», применение к которому метасимволов `+`, `?` и т. д. не требует круглых скобок.

Полученное выражение ⌜`<HR•+SIZE•*=•*[0-9]+•*>`⌟ выглядит довольно громоздко даже при том, что я выделил метасимволы жирным шрифтом и использовал «видимый» символ пробела '•'. К счастью, *egrep* поддерживает ключ игнорирования

[1] Если вы не знаете языка HTML, не расстраивайтесь. Я использую его, чтобы примеры выглядели более реально, но при этом привожу всю необходимую информацию. Читатели, знакомые с задачей разбора тегов HTML, наверняка увидят некоторые важные обстоятельства, которые пока не упоминаются.

регистра символов `-i` (☞ 39), поэтому нам не пришлось использовать `[Hh][Rr]` вместо `HR`. «Без украшений» выражение `<HR +SIZE *= *[0-9]+ *>` вызывает еще большее замешательство. Пример усложняется и тем, что большинство звездочек и плюсов относится к пробелам, а наш глаз не привык выделять пробелы в строке. При чтении регулярных выражений вам придется бороться с этой привычкой, поскольку пробел является таким же обычным символом, как, например, `j` или `4` (как будет показано ниже, в некоторых программах поддерживается специальный режим, в котором пропуски игнорируются, но в *egrep* такого режима не существует).

Продолжим усовершенствование хорошего примера и внесем в него еще одно изменение. Будем считать, что атрибут размера является необязательным, поэтому для проведения линии стандартной толщины может использоваться простой тег `<HR>` (как и ранее, перед `>` могут находиться дополнительные пробелы). Как модифицировать наше регулярное выражение так, чтобы оно совпадало с любым из этих типов? Главное — понять, что его часть с размером является *необязательной* (это подсказка). ❖ Проверьте свой ответ на с. 48.

Внимательно проанализируйте окончательное выражение (на врезке с ответом), чтобы понять, чем различаются метасимволы `?`, `*` и `+` и что они означают на практике. Смысл этих метасимволов перечисляется в табл. 1.2.

Обратите внимание: у каждого квантификатора существует минимальное количество элементов текста, которые он обязательно должен найти (в некоторых случаях минимальное количество равно нулю). Максимальное количество элементов не ограничено.

Таблица 1.2. Сводка квантификаторов

Квантификатор	Необходимый минимум	Максимальное количество	Смысл
`?`	Нет	1	Допускается один экземпляр; не требуется ни один («*один необязательно*»)
`*`	Нет	Не ограничено	Допускается неограниченное количество; не требуется ни один («*любое количество необязательно*»)
`+`	1	Не ограничено	Требуется один экземпляр; допускается неограниченное количество («*хотя бы один и более*»)

Определение интервалов количества экземпляров

В некоторых версиях *egrep* поддерживается метапоследовательность для явного определения минимального и максимального количества совпадений: `…{min,`

max}⌋. Эта конструкция называется интервальным квантификатором. Например, выражение ⌈…{3,12}⌋ совпадает до 12 раз, если это возможно, но может ограничиться и тремя совпадениями. Регулярное выражение ⌈[a-zA-Z]{1,5}⌋ может использоваться для поиска обозначений биржевых котировок (от одной до пяти букв). Запись {0,1} эквивалентна метасимволу ?.

Интервальный квантификатор поддерживается пока не всеми версиями *egrep*. Зато он поддерживается множеством других инструментов, о которых речь пойдет в главе 3 при рассмотрении широкого спектра метасимволов, используемых в настоящее время.

Круглые скобки и обратные ссылки

До настоящего момента мы встречались с двумя применениями круглых скобок: ограничение области действия в конструкции выбора ⌈|⌋ и группировка символов для применения квантификаторов (например, ? и *). Я бы хотел упомянуть еще одно специализированное применение круглых скобок, которое поддерживается лишь некоторыми версиями *egrep* (в том числе и популярной GNU-версией), но часто встречается в других программных средствах.

Во многих диалектах регулярных выражений круглые скобки могут «запоминать» текст, который совпал с находящимся в них подвыражением. Эта возможность будет использована в частичном решении проблемы повторяющихся слов, описанной в начале главы. Если вам известно конкретное повторяющееся слово, его можно включить в регулярное выражение, например ⌈the•the⌋. Правда, в этом случае также будут найдены строки типа the•theory, но проблема легко решается, если ваша версия *egrep* поддерживает метапоследовательности для обозначения границ слов, упоминавшиеся на с. 41: ⌈\<the•the\>⌋. Вместо одного пробела даже можно использовать ⌈•+⌋, чтобы выражение стало более гибким.

Тем не менее проверить все возможные пары слов попросту невозможно. Хотелось бы найти одно «обобщенное» слово, а потом сказать: «А теперь поищи то же самое». Если ваша версия *egrep* поддерживает механизм *обратных ссылок* (*backreferencing*), такая возможность существует. Обратные ссылки позволяют искать новый текст, который совпадает с другим текстом, совпавшим с предшествующей частью выражения, причем на момент написания выражения этот текст неизвестен.

Начнем с выражения ⌈\<the•+the\>⌋ и заменим первое ⌈the⌋ регулярным выражением для обозначения обобщенного слова — ⌈[A-Za-z]+⌋. Затем по соображениям, которые станут ясны из следующего абзаца, это выражение заключим в круглые скобки. Наконец, второе ‘the’ заменим специальным метасимволом ⌈\1⌋. Получится ⌈\<([A-Za-z]+)•+\1\>⌋.

НЕОБЯЗАТЕЛЬНОЕ ПОДВЫРАЖЕНИЕ

❖ *Ответ на вопрос со с. 46.*

В данном случае «необязательный» означает, что часть тега может встречаться один раз, но ее присутствие не требуется. Для подобных ситуаций существует метасимвол ⌜?⌟. Поскольку размер необязательного элемента превышает один символ, необходимо использовать круглые скобки: ⌜(...)?⌟. После вставки этой конструкции наше выражение принимает вид:

⌜<HR(•+SIZE•*=•*[0-9]+)?•*>⌟

Обратите внимание: элемент ⌜•*⌟ вынесен за круглые скобки. Это сделано для того, чтобы выражение успешно находило теги вида <HR•>. Если бы этот элемент находился в скобках, то завершающие пробелы допускались бы только при указании в теге атрибута SIZE.

Также следует заметить, что конструкция ⌜•+⌟ перед SIZE *включена* в круглые скобки. Если вынести ее за скобки, регулярное выражение будет требовать обязательного пробела после HR, даже при отсутствии атрибута SIZE. В этом случае тег <HR> не совпадет.

В программах с поддержкой обратных ссылок круглые скобки «запоминают» текст, совпавший с находящимся в них подвыражением, а специальный метасимвол ⌜\1⌟ представляет этот текст (каким бы он ни был на тот момент) в оставшейся части регулярного выражения.

Конечно, в выражение можно включить несколько пар круглых скобок и ссылаться на совпавший текст при помощи метасимволов ⌜\1⌟, ⌜\2⌟, ⌜\3⌟ и т. д. Пары скобок нумеруются в соответствии с порядковым номером открывающей скобки слева направо, поэтому в выражении ⌜([a-z])([0-9])\1\2⌟ метасимвол ⌜\1⌟ ссылается на текст, совпавший с ⌜[a-z]⌟, а ⌜\2⌟ ссылается на текст, совпавший с ⌜[0-9]⌟.

В нашем примере 'the•the' подвыражение ⌜[A-Za-z]+⌟ совпадает с первым 'the'. Оно находится в первой паре круглых скобок, поэтому на совпавшее 'the' можно ссылаться при помощи метасимвола ⌜\1⌟ — если конструкция ⌜•+⌟ совпадает, то на месте ⌜\1⌟ должно находиться другое слово 'the'. Если и это условие выполняется, ⌜\>⌟ проверяет, что мы находимся на границе слова (тем самым исключаются случайные совпадения в строках типа 'the•theft'). Успешное совпадение всего выражения означает, что мы нашли повторяющееся слово. Впрочем, это не всегда является ошибкой (например, в английском языке допускаются два слова «that» подряд), но найденные подозрительные строки можно просмотреть и самостоятельно принять решение.

Решив включить в книгу этот пример, я опробовал его на подготовленном тексте (моя версия *egrep* поддерживает ⌜\<...\>⌟ и обратные ссылки). Чтобы команда прино-

сила больше пользы и находила повторения вида 'The•the', я включил в командную строку ключ -i, упоминавшийся на с. 40[1].

Команда выглядела так:

```
% egrep -i '\<([a-z]+) +\1\>' файлы...
```

Как ни стыдно признаваться, поиск обнаружил четырнадцать пар ошибочно 'повторяющихся•повторяющихся' слов! Я исправил ошибки и с тех пор всегда включаю подобную проверку по регулярному выражению в процедуру подготовки материала книги, чтобы предотвратить возможное появление подобных ошибок.

При всей полезности этого регулярного выражения необходимо хорошо понимать его принципиальные ограничения. Поскольку *egrep* просматривает каждую строку по отдельности, вы не сможете обнаружить те ситуации, когда слово в конце строки повторяется в начале следующей строки. Для этого нужны более совершенные средства, и некоторые примеры будут рассмотрены в следующей главе.

Экранирование

Я еще не упоминал об одной важной проблеме: как включить в регулярное выражение символ, который обычно интерпретируется как метасимвол? Например, при попытке поиска хоста ega.att.com в Интернете по регулярному выражению ⌜ega.att.com⌟ в результат будут включены строки типа megawatt•computing. Вспомните, о чем говорилось выше: метасимвол ⌜.⌟ совпадает с произвольным символом.

Метапоследовательность, совпадающая с литеральной точкой, состоит из обычной точки и экранирующего префикса \: ⌜ega\.att\.com⌟. Последовательность ⌜\.⌟ называется *экранированной* (*escaped*) *точкой*. Экранирование может выполняться со всеми стандартными метасимволами, кроме метасимволов символьных классов[2].

Экранированный метасимвол теряет свой особый смысл и становится обычным литералом. При желании последовательность «экранирующий префикс+символ» можно интерпретировать как специальную метапоследовательность, совпадающую с указанным литералом.

---

[1] Учтите, что в популярной GNU-версии *egrep* ключ i реализован с ошибкой, из-за которой его действие не распространяется на обратные ссылки. Иначе говоря, *egrep* успешно найдет «the the», но *не* найдет «The the».

[2] Большинство языков программирования и утилит также позволяет использовать экранирование в символьных классах, однако в основных версиях *egrep* эта возможность не поддерживается, а символ '\' в классах интерпретируется как литерал, входящий в список символов.

Другой пример: для поиска слов в круглых скобках (например, '(very)') можно воспользоваться регулярным выражением ⌜\([a-zA-Z]+\)⌟. Обратный слэш в последовательностях ⌜\(⌟ и ⌜\)⌟ отменяет особую интерпретацию символов () и превращает их в литералы, совпадающие с круглыми скобками в тексте.

Если обратный слэш находится перед другим символом, не являющимся метасимволом, он может иметь различный смысл в зависимости от версии программы. Например, мы уже видели, что в некоторых версиях *egrep* ⌜\<⌟, ⌜\>⌟, ⌜\1⌟ и т. д. интерпретируются как метапоследовательности. В последующих главах будут приведены и другие примеры.

Новые горизонты

Надеюсь, эти примеры и объяснения заложили основу для глубокого понимания регулярных выражений. Но, пожалуйста, учтите, что это только начало и вам предстоит еще многое узнать.

Языковая диверсификация

Я упоминал о некоторых возможностях регулярных выражений, поддерживаемых большинством версий *egrep*. Существуют и другие возможности, которые присутствуют лишь в небольшом числе версий. Они будут рассмотрены в следующих главах.

К сожалению, у языка регулярных выражений, как и у естественных языков, существуют различные диалекты и ответвления. Почти каждая новая программа с поддержкой регулярных выражений изобретает какое-нибудь новое «улучшение». В результате регулярные выражения постоянно развиваются, и за многие годы появились многочисленные диалекты (flavors) регулярных выражений. В последующих главах вас ждут еще много примеров.

Смысл регулярного выражения

В самом общем смысле регулярное выражение либо совпадает внутри некоторого фрагмента текста (при использовании *egrep* — в строке), либо не совпадает. При построении регулярного выражения приходится постоянно следить за тем, чтобы регулярное выражение:

- совпадало там, где нужно;
- не совпадало там, где не нужно.

Кроме того, хотя *egrep* не следит за тем, где именно в строке произошло совпадение, в некоторых случаях этот вопрос может оказаться важным. Предположим, вы работаете с текстом, в котором встречается строка:

```
…zip Is 44272. If you write, send $4.95 to cover postage and…
```

Если вы просто ищете строку по шаблону ⌜`[0-9]+`⌟, вас не интересует, в каком из чисел произошло совпадение. Но если вы собираетесь *что-то сделать* с найденным числом (сохранить в файле, увеличить, заменить и т. д. — примеры подобных операций приводятся в следующей главе), вопрос о том, *какое именно* число было найдено, становится очень существенным.

Дополнительные примеры

При работе с регулярными выражениями, как и с любым языком, *чрезвычайно полезен практический опыт*, поэтому я привожу еще несколько распространенных примеров регулярных выражений.

Половина хлопот при написании регулярного выражения связана с тем, чтобы оно совпадало в нужных местах. Другая половина — с тем, чтобы регулярное выражение *не* совпадало в ненужных местах. На практике важны оба аспекта, но пока основное внимание будет уделяться первому, т. е. получению успешного совпадения. Хотя примеры анализируются не во всей полноте, так более наглядно прослеживаются основные принципы работы с регулярными выражениями.

Имена переменных

Во многих языках программирования существуют идентификаторы (имена переменных и т. п.), которые состоят только из алфавитно-цифровых символов и знаков подчеркивания, но не могут начинаться с цифры. На языке регулярных выражений эта формулировка записывается в виде ⌜`[a-zA-Z_][a-zA-Z_0-9]*`⌟. Первый класс определяет возможные значения первого символа идентификатора, второй (вместе с суффиксом `*`) определяет оставшуюся часть идентификатора. Если длина идентификатора ограничивается, допустим, 32 символами, звездочку можно заменить выражением ⌜`{0,31}`⌟, если вашей программой поддерживается конструкция ⌜{min, max}⌟ (интервальный квантификатор кратко упоминается на с. 46).

Последовательности символов, заключенные в кавычки

Простое выражение, обозначающее строку в кавычках, выглядит так: ⌜`"[^"]*"`⌟.

Двойные кавычки, ограничивающие регулярное выражение, совпадают с открывающими и закрывающими кавычками строки. Между ними может находиться

все, что угодно... кроме других кавычек! Выражение ⌜[^"]⌟ совпадает с любым символом, кроме ", а звездочка говорит о том, что количество таких символов может быть любым.

Более полезное (хотя и более сложное) определение строки в кавычках позволяет включать в строку внутренние символы «, если перед ними стоит обратный слэш — например, "nail•the•2\"x4\"•plank". Мы вернемся к этому примеру в будущих главах, когда будем подробно рассматривать, как же на самом деле происходит поиск совпадений.

Денежные суммы в долларах (с необязательным указанием центов)

Одно из возможных решений: ⌜\$[0-9]+(\.[0-9][0-9])?⌟.

На верхнем уровне это простое регулярное выражение разбивается на три части: ⌜\$⌟, ⌜...+⌟ и ⌜(...)?⌟. Его можно вольно интерпретировать как «литерал — знак доллара, за которым следует последовательность чего-то такого, а в конце еще может находиться что-то этакое». В данном случае «что-то такое» — это цифра (последовательность цифр образует число), а «что-то этакое» — это десятичная точка, за которой следуют две цифры.

Этот пример наивен по нескольким причинам. Например, это регулярное выражение предполагает, что денежная сумма в долларах будет записываться как $1000, а не как $1,000. Выражение допускает наличие части суммы в центах, но в этом нет большого смысла при использовании *egrep*. В *egrep* важно лишь то, есть совпадение или нет. Наличие чего-то необязательного в конце никак не влияет на совпадение в начале.

С другой стороны, если вам потребуется отыскать строки, содержащие *только* денежную сумму и ничего другого, выражение можно «завернуть» в конструкцию ⌜^...$⌟. В этом случае необязательная дробная часть важна, поскольку она может находиться (или не находиться) между основной суммой и концом строки, а ее наличие или отсутствие влияет на определение совпадения.

Кроме того, приведенное выражение не находит суммы вида '$.49'. Возникает искушение заменить + на *, но такое решение не годится. Почему? Этот вопрос останется открытым до возвращения к этому примеру в главе 5 (☞ 253).

Адреса (URL) HTTP/HTML

Адреса URL могут иметь довольно сложную структуру, поэтому построение регулярного выражения для любых возможных URL оказывается столь же сложной задачей. Тем не менее небольшое смягчение требований позволяет описать боль-

шинство нормальных URL относительно простым выражением. Зачем это может понадобиться? Представьте, что вы ищете в архиве электронной почты адрес URL, который вы точно не помните, но надеетесь узнать по внешнему виду.

В общем случае обычные адреса URL в HTML/HTTP имеют следующую структуру:

```
http://хост/путь.html
```

хотя также часто встречается расширение `.htm`.

Правила, определяющие, что может (или не может) использоваться в качестве *хоста* (т. е. имени компьютера, например `www.yahoo.com`), весьма сложны, но для нашего случая можно предположить, что за префиксом '`http://`' указывается именно хост, поэтому мы обойдемся простым выражением вида ⌜`[-a-z0-9_.]+`⌟. Структура *пути* может быть еще более разнообразной, поэтому мы обозначим путь выражением ⌜`[-a-z0-9_:@&?=+,.!/~+%$]*`⌟. Обратите внимание: перечисление символов в этих классах начинается с дефиса, чтобы он включался в список как литерал, а не интерпретировался как часть интервала (☞ 35).

Объединив все сказанное, можно начать поиски с выражения следующего вида:

```
% egrep -i '\<http //[-a-z0-9_.:]+/[-a-z0-9_:@&?=+,.!/~*%$]*\.html?\>' файлы
```

Наш подход к поиску совпадений был весьма либеральным, поэтому такое выражение совпадет со строкой вида '`http://..../foo.html`', которая, несомненно, не является правильным адресом URL. Насколько это существенно? Все зависит от того, что вы пытаетесь сделать. В ситуации с поиском в архиве электронной почты несколько ложных совпадений ни на что не повлияют. Более того, мы могли бы обойтись еще более простым выражением:

```
% egrep -i '\<http://[^ ]*\.html?\>' файлы…
```

По мере углубления в изучение темы вы убедитесь, что правильная оценка данных в значительной мере определяет компромисс между сложностью и точностью. Мы еще вернемся к примеру с поиском URL в следующей главе.

Теги HTML

Трудно представить, чтобы кто-нибудь стал искать строки, содержащие тот или иной тег HTML, при помощи утилиты типа *egrep*. Однако анализ регулярного выражения для поиска тегов HTML приносит несомненную пользу, особенно когда мы познакомимся с более совершенными средствами в следующей главе.

При виде простейших случаев вроде '`<TITLE>`' и '`<HR>`' напрашивается мысль использовать выражение вида ⌜`<.*>`⌟. Такой упрощенный подход встречается довольно

часто. Конечно, он неверен. Переводя выражение ⌜<.*>⌟ на повседневный язык, мы получаем: «символ '<', за которым следует последовательность произвольных символов, завершенная символом '>'». При подобной формулировке становится ясно, что это выражение может совпасть с последовательностью тегов — например, с помеченной частью строки 'this <I>short</I> example'.

На первый взгляд это выглядит странно, но не забывайте — вы читаете первую главу книги, поэтому ваше понимание темы весьма поверхностно. Я привел этот пример лишь для того, чтобы показать — регулярные выражения не так уж сложны, но без полного понимания темы возможны неожиданности. В следующих главах будут изложены тонкости, которые помогут разобраться в причинах и решить эту проблему.

Время в формате «9:17 am» или «12:30 pm»

Поиск времени тоже может осуществляться с разной степенью точности. Например, выражение

⌜[0-9]?[0-9]:[0-9][0-9]•(am|pm)⌟

успешно находит 9:17•am и 12:30•pm, но с неменьшей легкостью обнаруживает бессмысленное время 99:99•pm.

Нетрудно понять, что если час состоит из двух цифр, то первая цифра может быть только единицей. Но конструкция ⌜1?[0-9]⌟ также допускает 19 часов (и 0 часов), поэтому можно рассмотреть два отдельных случая: ⌜1[012]⌟ для часов из двух цифр и ⌜[1-9]⌟ для часов из одной цифры. В результате получается ⌜(1[012]|[1-9])⌟.

С минутами дело обстоит проще. Первая цифра определяется выражением ⌜[0-5]⌟, а для второй цифры можно оставить ⌜[0-9]⌟. Объединяя все компоненты, мы получаем ⌜(1[012]|[1-9]):[0-5][0-9]•(am|pm)⌟.

Попробуйте воспользоваться аналогичными рассуждениями и построить регулярное выражение для поиска времени в 24-часовом формате, с нумерацией часов от 0 до 23. Чтобы задание было посложнее, разрешите использование начального нуля, по крайней мере, до 09:59. ❖ Попробуйте построить собственное решение, затем сверьтесь с моим вариантом на с. 56.

Терминология регулярных выражений

Выражение (regex)

Как вы уже наверняка заметили, использование полной фразы «регулярное выражение» (regular expression) достаточно утомительно, особенно для тех, кому часто приходится писать ее. Поэтому я обычно использую просто слово «regex»

(реджекс). Оно само так и «катится» по языку (и рифмуется с «FedEx», с твердым «g» (дж), как в слове «regular» (реджуле), а не мягким, как в слове «Regina») и допускает самые необычные способы употребления, например «при применении реджекса...», «начинающие реджексеры» и даже «реджексификация». Для обозначения части программы, которая фактически выполняет поиск совпадений, я использую фразу «реджекс-механизм».

Соответствие

Когда я говорю, что регулярное выражение «соответствует» строке, на самом деле я имею в виду, что оно соответствует *части* строки. Формально выражение ⌜`a`⌟ не *соответствует* строке `cat`, но соответствует *части* этой строки. Это обстоятельство обычно не вызывает непонимания, тем не менее об этом стоит упомянуть.

Метасимвол

Концепция метасимвола (или «метапоследовательности» — я использую эти слова как синонимы) зависит от того, где именно в регулярном выражении он используется. Например, символ ⌜`*`⌟ является метасимволом, но только не внутри символьного класса и только если он не экранируется. «Экранируется» — в том смысле, что перед ним стоит обратный слэш. Впрочем, и это не всегда так. Например, звездочка интерпретируется как литерал в выражении ⌜`\*`⌟ но не в ⌜`\\*`⌟ (когда первый символ \ обеспечивает особую интерпретацию второго символа), хотя в обоих случаях перед звездочкой стоит обратный слэш.

В зависимости от диалекта регулярных выражений существуют различные ситуации, когда некоторый символ считается или не считается метасимволом. В главе 3 эта тема рассматривается более подробно.

Диалект

Как я уже говорил, в разных программах регулярные выражения выполняют разные функции, поэтому наборы метасимволов и другие возможности, поддерживаемые программами, также различаются. Вернемся к примеру с границами слов. Некоторые версии *egrep* поддерживают обозначения `\<...\>`, о которых говорилось выше. В других версиях нет отдельных метасимволов для начала и конца слова, а есть один универсальный метасимвол ⌜`\b`⌟ (который нам еще не встречался — он будет представлен в следующей главе). В третьих версиях поддерживаются все перечисленные метасимволы. Наконец, существуют версии, которые не поддерживают ни один из этих метасимволов.

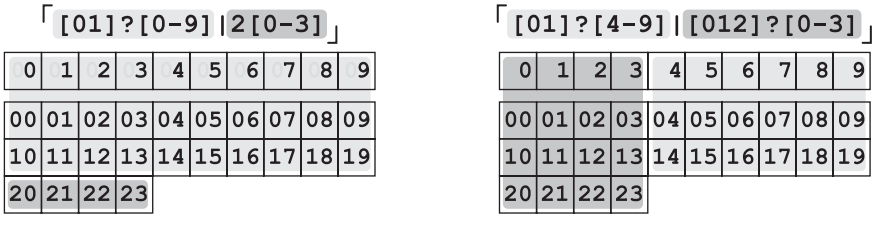
Совокупность этих второстепенных различий в реализации я обозначаю термином «диалект». Однако диалект не сводится к набору поддерживаемых и неподдержи-

ПОИСК ВРЕМЕНИ В 24-ЧАСОВОМ ФОРМАТЕ

❖ *Ответ на вопрос со с. 54.*

Возможны разные решения, но мы воспользуемся уже описанной логикой. На этот раз задача разбивается на три временных интервала: утро (с `00` до `09` часов, возможен начальный ноль), день (с `10` до `19` часов) и вечер (с `20` до `23` часов). Самое прямолинейное решение выглядит так: `⌜0?[0-9]|1[0-9]|2[0-3]⌟`.

Вообще говоря, первые два варианта можно объединить, и тогда запись получится более короткой: `⌜[01]?[0-9]|2[0-3]⌟`. На первый взгляд эквивалентность этих двух записей не очевидна, но на самом деле это так. Возможно, вам поможет приведенный ниже рисунок, на котором затененные группы обозначают числа, соответствующие разным альтернативам.



ваемых метасимволов — за этим понятием кроется нечто большее. Даже если две программы поддерживают `⌜\<...\>⌟`, они могут расходиться во мнениях относительно того, что именно следует считать словом. Если вы хотите на самом деле *извлекать пользу* из программы, это действительно важно.

Не путайте «диалект» с конкретной программой. Подобно тому как два человека могут говорить на одном и том же диалекте, две абсолютно разные программы могут поддерживать одинаковые диалекты регулярных выражений. Кроме того, две одноименные программы (к тому же предназначенные для решения общей задачи) нередко обладают сильно различающимися диалектами. Например, в семействе программ, называемых *egrep*, поддерживается широкий спектр различных диалектов.

В конце 1990-х годов диалект регулярных выражений языка Perl прославился особой мощью и выразительностью, и вскоре другие языки программирования стали предлагать поддержку регулярных выражений в стиле Perl (во многих случаях даже присутствовало явное указание на источник, когда язык объявлялся «Perl-совместимым»). Среди примеров стоит упомянуть PHP, Python, многие пакеты регулярных выражений для Java, Microsoft .NET Framework, Tcl, различные библиотеки C и т. д. Тем не менее все эти диалекты отличались от оригинала по ряду

важных аспектов. Диалект регулярных выражений Perl тоже развивался (причем иногда в него внедрялись удачные новшества, появившиеся в других языках). В итоге общая ситуация становится все более разнообразной и запутанной.

Подвыражение

Термин «подвыражение» (subexpression) на самом деле означает любую часть большего выражения, но обычно он относится к части, заключенной в круглые скобки, или к одной из альтернатив конструкции выбора. Например, в выражении ⌜`^(Subject|Date):•`⌟ часть ⌜`Subject|Date`⌟ обычно именуется подвыражением. Внутри нее альтернативы ⌜`Subject`⌟ и ⌜`Date`⌟ тоже называются подвыражениями. Формально ⌜`S`⌟ также считается подвыражением, так же как и ⌜`u`⌟, и ⌜`b`⌟ и ⌜`j`⌟...

Конструкция типа ⌜`1-6`⌟ не считается подвыражением ⌜`H[1-6]•*`⌟, поскольку она является частью неразрывного «элемента» — символьного класса. С другой стороны, ⌜`H`⌟, ⌜`1-6`⌟ и ⌜`•*`⌟ являются подвыражениями исходного выражения ⌜`H[1-6]•*`⌟.

В отличие от конструкции выбора, квантификаторы (`*`, `+` и `?`) всегда применяются к наименьшему непосредственно предшествующему подвыражению. Вот почему в выражении ⌜`mis+pell`⌟ плюс относится только к ⌜`s`⌟, а не к ⌜`mis`⌟ или ⌜`is`⌟. Конечно, когда квантификатору непосредственно предшествует подвыражение в круглых скобках, все подвыражение (сколь бы сложным оно ни было) воспринимается как единое целое.

Символ

Термин «символ» в информатике имеет много значений. Символ, представленный некоторым байтом, — всего лишь вопрос интерпретации. Значение байта остается неизменным в любом контексте, однако *представляемый* им *символ* зависит от контекста. Например, два байта с десятичными значениями `64` и `53` представляют символы «@» и «5» в кодировке ASCII, но, с другой стороны, в кодировке EBCDIC они соответствуют совершенно другим символам (пробел и какой-то управляющий символ).

А в одной из популярных японских кодировок эти два байта совместно представляют иероглиф 亜. Но в другой японской кодировке этот же иероглиф представляется двумя совершенно другими байтами. Кстати говоря, в популярной кодировке Latin-1 эти два байта представляют два символа «Àµ», а в одном из вариантов Юникода[1] — один корейский иероглиф 쌵. Вывод: интерпретация байтов зависит

[1] Наиболее авторитетным руководством по многобайтовым кодировкам является книга Кена Лунде (Ken Lunde) «CJKV Information Processing». Сокращение CJKV означает «китайский, японский, корейский и вьетнамский» — все языки, в которых используется многобайтовая кодировка.

от кодировки, и чтобы поиск был успешным, ваша интерпретация должна соответствовать интерпретации того инструмента, с которым вы работаете.

До недавнего времени средства обработки текста обычно интерпретировали данные как набор байтов в кодировке ASCII, не обращая внимания на предполагаемую кодировку. Тем не менее все больше современных систем использует Юникод при внутренней обработке данных (Юникод рассматривается в главе 3, ☞ 145). Если подсистема работы с регулярными выражениями в такой системе реализована нормально, пользователю обычно не приходится обращать внимание на подобные проблемы. Однако это «если» весьма существенно, поэтому в главе 3 эта тема рассматривается более подробно.

Пути к совершенствованию

Честно говоря, изучить регулярные выражения не так уж сложно. Но если поговорить со средним пользователем программы или языка с поддержкой регулярных выражений, скорее всего выяснится, что ваш собеседник «немножко разбирается» в них, но не чувствует достаточной уверенности для решения действительно нетривиальных задач или ограничивается только теми программами, которыми он часто пользуется.

Обычно документация по регулярным выражениям ограничивается коротким и неполным описанием одного-двух метасимволов, за которыми следует таблица с перечислением всего остального. В примерах часто используются бессмысленные регулярные выражения типа ⌜`a*((ab)*|b*)`⌟ и потрясающие тексты вроде ‘`a•xxx•ce•xxxxxx•ci•xxx•d`’. Кроме того, в документации полностью игнорируются неочевидные, но важные моменты и часто утверждается, что поддерживаемый диалект полностью совместим с диалектом другой, хорошо известной программы. При этом авторы всегда забывают упомянуть о неизбежных исключениях. В общем, состояние дел с документацией по регулярным выражениям явно нуждается в улучшении.

Я вовсе не утверждаю, что эта глава решит все проблемы или хотя бы даст сколько-нибудь полное представление о регулярных выражениях *egrep*. Скорее она закладывает фундамент, на котором будет построена вся оставшаяся часть книги. Звучит амбициозно, но я надеюсь, что эта книга действительно решит многие проблемы. После первого издания книги я получил много лестных отзывов и очень постарался, чтобы новое издание стало еще лучше — как по глубине изложения, так и по широте представленного материала.

Может быть, из-за традиционных недостатков, присущих документации, я постарался приложить дополнительные усилия и изложить материал действительно понятно. Для полноценного использования регулярных выражений вы должны *действительно* понять их.

Это и хорошо, и плохо.

Хорошо, потому что вы научитесь мыслить регулярными выражениями. Вы узнаете, на какие различия и особенности следует в первую очередь обращать внимание при знакомстве с новой программой, обладающей собственным диалектом. Вы научитесь выражать свои мысли даже на слабом, усеченном диалекте регулярных выражений. Вы поймете, почему одно выражение эффективнее другого, а также сможете оценить компромиссы между сложностью и точностью решения задачи и принять правильное решение. Столкнувшись с особенно сложным выражением, вы будете точно знать, как оно будет обработано программой. Короче говоря, потенциал регулярных выражений в полной мере раскроется перед вами.

Но для этого вам придется основательно потрудиться. Материал книги разбит на три глобальные темы.

- **Общие принципы использования регулярных выражений.** В большинстве программ существуют более совершенные средства работы с регулярными выражениями, чем в утилите *egrep*. Прежде чем подробно рассматривать процесс написания регулярных выражений, используемых на практике, необходимо разобраться с общими принципами их использования. Мы займемся этой темой со следующей главы.

- **Возможности регулярных выражений.** Правильный выбор инструмента для решения конкретной проблемы наполовину решает задачу, поэтому я не хочу ограничиваться использованием одной утилиты по всей книге. Разные программы (а иногда даже разные версии одной программы) обладают разными возможностями и поддерживают разные метасимволы. Прежде чем переходить к подробностям использования, мы подробно изучим обстановку. Этой теме посвящена глава 3.

- **Механизм обработки регулярных выражений.** Чтобы мы могли изучать полезные (но нередко сложные) примеры, необходимо выяснить, как же организуется поиск по регулярным выражениям. Как вы убедитесь, порядок обработки некоторых метасимволов может играть очень важную роль. Более того, обработка регулярных выражений может быть реализована различными способами, поэтому разные программы часто выполняют с одним выражением разные действия. Эта обширная тема рассматривается в главах 4, 5 и 6.

Последний пункт — самый важный и одновременно самый сложный в изложении. Теоретические рассуждения покажутся скучноватыми нетерпеливому читателю, стремящемуся поскорее добраться до самого интересного — до реальных задач. Тем не менее механизм обработки регулярных выражений является ключом к их *подлинному пониманию*.

Возможно, кто-то возразит: чтобы научиться водить машину, необязательно знать, как она устроена. Но аналогия с вождением машины в данном случае неуместна.

Я хочу, чтобы вы научились решать задачи с использованием регулярных выражений, а для этого вам придется самостоятельно строить регулярные выражения. Так что лучше провести аналогию не с вождением автомобиля, а с его самостоятельной сборкой. Чтобы построить свой автомобиль, необходимо знать, как он устроен.

В главе 2 вы получите дополнительный опыт практического вождения. В главе 3 приводится краткая история автомобилестроения и подробно рассматривается строение кузова (диалектов регулярных выражений). В главе 4 вы познакомитесь со строением двигателя. В главе 5 приводятся дополнительные практические примеры. Глава 6 показывает, как настраивать различные двигатели, а в последующих главах рассматриваются конкретные модели. Нам предстоит провести много времени, копаясь под капотом (особенно в главах 4, 5 и 6), поэтому не забудьте надеть рабочий комбинезон и запаситесь тряпками.

Заключение

В табл. 1.3 приведена сводка метасимволов *egrep*, рассмотренных в этой главе.

Кроме того, вы должны четко понимать перечисленные ниже положения.

- Не все версии программы *egrep* одинаковы. Они часто различаются по набору поддерживаемых метасимволов и их интерпретации — за подробностями обращайтесь к документации.
- Круглые скобки применяются для группировки (☞ 38), сохранения совпавшего текста(☞ 39) и ограничения конструкций выбора (☞ 51).
- Символьные классы занимают особое место — в них действуют совершенно иные правила использования метасимволов, отличные от «основного» языка регулярных выражений (☞ 33).
- Конструкции выбора и символьные классы принципиально отличаются друг от друга. Они решают разные задачи, которые лишь в одной специализированной ситуации выглядят похожими (☞ 37).
- Инвертированный символьный класс «позитивен» — он предполагает наличие символа, а не его отсутствие. Поскольку список символов инвертируется, совпадающий символ должен быть одним из тех, которые *не* перечислены в классе (☞ 35).
- Полезный ключ `-i` отменяет учет регистра символов при сравнении (☞ 40).
- Существует три типа экранирования:
  - ⌜\⌟ + метасимвол — метапоследовательность, обозначающая соответствующий литерал (например, ⌜*⌟ обозначает литерал-звездочку).

Таблица 1.3. Сводка метасимволов egrep

Метасимвол	Название	Интерпретация
Элементы, обозначающие отдельный символ		
`.`	*Точка*	Один любой символ
`[…]`	*Символьный класс*	Любой из перечисленных символов
`[^…]`	*Инвертированный символьный класс*	Любой символ, не перечисленный в классе
`\символ`	*Экранирование*	Если перед *метасимволом* ставится экранирующий префикс \, то *символ* интерпретируется как соответствующий литерал
Квантификаторы		
`?`	*Вопросительный знак*	Допускается один экземпляр (ни один не требуется)
`*`	*Звездочка*	Допускается любое количество экземпляров (ни один не требуется)
`+`	*Плюс*	Требуется один экземпляр, допускается любое количество экземпляров
`{min, max}`	*Интервальный квантификатор**	Требуется минимум экземпляров, допускается максимум экземпляров
Позиционные метасимволы		
`^`	*Крышка, циркумфлекс*	Позиция в начале строки
`$`	*Доллар*	Позиция в конце строки
`\<`	*Граница слова**	Позиция в начале слова
`\>`	*Граница слова**	Позиция в конце слова
Прочие метасимволы		
`|`	*Конструкция выбора*	Любое из перечисленных выражений
`(...)`	*Круглые скобки*	Ограничение конструкции выбора, группировка для применения квантификаторов и «сохранение» текста для обратных ссылок
`\1, \2, ...`	*Обратная ссылка**	Текст, ранее совпавший с первой, второй и т. д. парами круглых скобок
* Не поддерживается некоторыми версиями *egrep*.		

- ⌜\⌟ + некоторые метасимволы — метапоследовательность, смысл которой зависит от конкретной реализации (например, ⌜\<⌟ часто означает «начало слова»).

- `[\]` + любой другой символ — просто указанный символ (иначе говоря, символ \ игнорируется).

Следует отметить, что в большинстве версий *egrep* обратный слэш не имеет особой интерпретации внутри символьных классов и потому не может использоваться для нужд экранирования.

- Элементы, к которым применяются метасимволы ? и *, не обязаны действительно совпадать с какой-то частью строки для получения «успешного совпадения». Они совпадают *всегда*, даже если совпадают с «ничем» (☞ 43).

Личные заметки

Задача с повторяющимися словами, описанная в начале главы, выглядит довольно сложной, но возможности регулярных выражений позволили нам практически полностью решить ее при помощи такого ограниченного инструмента, как *egrep*, и притом в первой главе книги. Я хотел привести и более эффектные примеры, но потом решил, что лучше направить свои усилия на укрепление надежного фундамента для последующих глав. Я испугался, что какой-нибудь новичок прочитает эту главу, заполненную всевозможными правилами, исключениями из правил, предупреждениями и т. д., и подумает: «А стоит ли с этим связываться?»

Недавно мои братья обучали своих друзей играть в *шафкопф* (*schaffkopf*) — карточную игру, в которую играют уже несколько поколений моей семьи. Она значительно интереснее, чем кажется на первый взгляд, но начинающим приходится нелегко. После получасовых мучений моя двоюродная сестра Лиз, образец терпения, окончательно запуталась в сложных правилах и сказала: «Может, лучше сыграем в “пьяницу”?» Но в конце концов они засиделись за игрой до поздней ночи. Стоило преодолеть первую «гору» на пути обучения, как простой азарт уже не отпустил их. Мои братья знали, что так и произойдет, но им пришлось затратить время и усилия на то, чтобы Лиз и другие новички оценили новую игру.

Возможно, кому-то из читателей понадобится некоторое время на то, чтобы привыкнуть к регулярным выражениям. До тех пор пока вы не почувствуете настоящего удовольствия от решения ваших собственных задач, регулярные выражения могут показаться несколько абстрактной теорией. Надеюсь, вы удержитесь от желания «сыграть в рамми». Когда вы поймете, какие возможности открывают перед вами регулярные выражения, небольшие усилия по их изучению покажутся вам сущим пустяком.

2 Дополнительные примеры

Помните задачу с повторяющимися словами из первой главы? Я уже говорил, что ее полное решение в языке типа Perl состоит из нескольких строк. Например, оно может выглядеть так:

```
$/ = ".\n";
while (<>) {
  next if !s/\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$3\e[m/ig;
  s/^(?:[^\e]*\n)+//mg;     # Удалить непомеченные строки
  s/^$ARGV: /mg;            # Начинать строку с имени файла
  print;
}
```

Да, перед вами *вся* программа.

Даже если вы знакомы с языком Perl, я не рассчитываю на то, что вы разберетесь в ней (*пока!*). Это, скорее, пример, выходящий за рамки возможностей *egrep* и призванный разжечь ваш интерес к регулярным выражениям. Вся основная работа этой программы выполняется тремя регулярными выражениями:

- ⌜\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)⌟
- ⌜^(?:[^\e]*\n)+⌟
- ⌜^⌟

Хотя пример написан на Perl, эти три регулярных выражения могут использоваться практически без изменений (или с минимальными изменениями) в ряде других языков, включая PHP, Python, Java, Visual Basic .NET, Tcl и т. д.

Конечно, ⌜^⌟ узнать нетрудно, но в остальных выражениях присутствуют элементы, не встречавшиеся в *egrep*. Дело в том, что в Perl и *egrep* используются разные диалекты регулярных выражений. Отличаются некоторые обозначения, и Perl (как и большинство современных программ) обладает гораздо более богатым набором метасимволов. Примеры будут приведены в этой главе.

О примерах

В этой главе мы рассмотрим несколько задач-примеров (проверка пользовательского ввода; работа с заголовками электронной почты; преобразование простого текста в HTML) и воспользуемся ими для общего обзора регулярных выражений. В ходе работы над примерами я буду «мыслить вслух», что поможет читателю составить представление о том, как строятся регулярные выражения. Попутно будут рассмотрены некоторые синтаксические конструкции и средства, не поддерживаемые в *egrep*, к тому же мы будем часто отвлекаться от основной темы и исследовать различные важные концепции.

В конце этой и в последующих главах будут приведены примеры на разных языках, в том числе PHP, Java и VB.NET, но большая часть примеров этой главы написана на Perl. Все эти языки (да и многие другие) по широте возможностей работы с регулярными выражениями значительно превосходят *egrep*, поэтому применение любого из них в практических примерах позволит нам узнать что-нибудь интересное. Я решил начать с Perl в основном из-за того, что среди всех распространенных языков он обладает наиболее интегрированной и доступной поддержкой регулярных выражений. Кроме того, в Perl существует много вспомогательных конструкций, которые выполняют всю «черную работу» и позволяют нам сконцентрироваться на регулярных выражениях.

Вспомните пример с проверкой файлов, описанный на с. 26, где требовалось проверить, что количество вхождений строки ‘`ResetSize'` в каждый файл точно совпадает с количеством вхождений ‘`SetSize'`. Для решения этой задачи я воспользовался Perl, а команда выглядела так:

```
% perl -0ne 'print "$ARGV\n" if s/ResetSize//ig != s/SetSize//ig' *
```

(я не рассчитываю, что вы поймете эту команду, а лишь надеюсь, что на вас произведет впечатление лаконичность решения).

Я люблю Perl, но не собираюсь уделять ему слишком много внимания. Не забывайте: эта глава посвящена *регулярным выражениям*. Приведу слова одного профессора, обращенные к студентам-первокурсникам: «Мы изучаем общие концепции компьютерных технологий, но рассматриваем их на примере Pascal[1]».

Поскольку эта глава не предполагает знания Perl, я в общих чертах представлю этот язык, чтобы вы могли разобраться в приведенных примерах (глава 7, в которой рассматриваются всевозможные нюансы работы Perl, требует некоторых познаний

[1] Pascal — традиционный язык программирования, первоначально разработанный в учебных целях. Спасибо Уильяму Ф. Мэттону (William F. Matton) и его профессору за удачную аналогию.

в языке). Даже если вам приходилось программировать на разных языках, Perl на первый взгляд выглядит довольно странно; это объясняется компактностью его синтаксиса и семантической многозначностью. Ради наглядности я отказался от использования многих возможностей Perl и постарался представить программы в более общем, приближенном к псевдокоду стиле. Хотя приведенные примеры нельзя назвать «плохими», это далеко не лучшие образцы «Пути программирования Perl». Зато вы *увидите* некоторые замечательные применения регулярных выражений.

Краткий курс Perl

Мощный сценарный язык Perl был создан в конце 1980-х годов на основе идей, почерпнутых из ряда других языков программирования. Многие его концепции обработки текста и регулярных выражений позаимствованы из двух специализированных языков, *awk* и *sed*, заметно отличающихся от «традиционных» языков типа C или Pascal.

Perl существует на многих платформах, включая DOS/Windows, Mac-OS, OS/2, VMS и Unix. Он в значительной степени ориентирован на обработку текстов и является самым распространенным инструментом для работы с текстовыми данными в WWW. За информацией о том, где достать версию Perl для вашей системы, обращайтесь на сайт *www.perl.com*.

На момент написания книги существовал Perl версии 5.8, но примеры этой главы работают в версиях начиная с 5.005.

Разберем простой пример:

```
$celsius = 30;
$fahrenheit = ($celsius * 9 / 5) + 32;  # Вычислить температуру
                                        # по шкале Фаренгейта
print "$celsius C is $fahrenheit F.\n"; # Вывести обе температуры
```

Программа выводит следующий результат:

```
30 C is 86 F.
```

Имена простых переменных (таких, как `$fahrenheit` и `$celsius`) всегда начинаются со знака доллара и могут содержать число или строку любой длины (в данном примере используются только числа). Комментарии начинаются со знака `#` и продолжаются до конца строки.

Если вы привыкли к традиционным языкам типа C, C#, Java и VB.NET, вероятно, вас больше всего удивит присутствие в строках Perl имен переменных, заключенных в кавычки. В строке `"$celsius C is $fahrenhe it F.\n"` каждая переменная

заменяется своим значением. В данном случае полученная строка затем выводится (символ \n начинает новую строку на экране).

По своим управляющим структурам Perl напоминает другие распространенные языки:

```
$celsius = 20;
while ($celsius <= 45)
{
  $fahrenheit = ($celsius * 9 / 5) + 32; # Вычислить температуру
                                         # по шкале Фаренгейта
  print "$celsius C is $fahrenheit F.\n";
  $celsius = $celsius + 5;
}
```

Блок кода, находящийся в теле цикла `while`, выполняется многократно, пока условие (в данном случае **`$celsius <= 45`**) остается истинным. Если сохранить этот фрагмент в файле (например, *temps*), его можно запустить из командной строки.

Вот как это выглядит:

```
% perl -w temps
20 C is 68 F.
25 C is 77 F.
30 C is 86 F.
35 C is 95 F.
40 C is 104 F.
45 C is 113 F.
```

Ключ `-w` не является обязательным и вообще не относится к регулярным выражениям. Он приказывает Perl проверить вашу программу и выдать предупреждения по тем аспектам, которые выглядят подозрительно (например, использование неинициализированных переменных и т. д. — заранее объявлять переменные в Perl не требуется). Я использую его лишь потому, что это рекомендуется делать всегда.

А теперь посмотрим, что нам предлагает Perl в области работы с регулярными выражениями.

Поиск по регулярному выражению

В Perl существует несколько способов применения регулярных выражений. Самый простой вариант — поиск совпадения регулярного выражения в тексте, хранящемся в переменной. Следующий фрагмент проверяет содержимое переменной `$reply` и сообщает, состоит ли она из одних цифр:

```
if ($reply =~ m/^[0-9]+$/) {
    print "only digits\n";
} else {
    print "not only digits\n";
}
```

Первая строка выглядит немного странно. Регулярное выражение `^[0-9]+$`, а окружающая конструкция **`m/…/`** сообщает Perl, что с этим выражением нужно сделать. **`m`** означает поиск *совпадения регулярного выражения* в тексте, а символ `/` определяет границы регулярного выражения[1]. Предшествующая конструкция `=~` связывает `m/…/` со строкой, в которой происходит поиск (в данном случае с содержимым переменной `$reply`).

Не путайте **`=~`** с `=` или с `==`. Оператор `==` проверяет, равны ли два числа (как вы вскоре увидите, при проверке равенства двух *строк* применяется оператор **`eq`**). Оператор `=` присваивает значение переменной, например **`$celsius = 20`**. Наконец, оператор **`=~`** связывает команду поиска со строкой, в которой будет происходить поиск (в данном примере использована команда поиска `m/^[0-9]+$/`, а целевая строка хранится в переменной `$reply`). В других языках программирования аналогичная операция поиска выполняется несколько иначе, в чем вы сможете убедиться при изучении примеров в следующей главе.

Оператор **`=~`** можно читать как «совпадает», «соответствует». Таким образом, строку:

```
if ($reply =~ m/^[0-9]+$/)
```

можно прочитать так: «Если регулярное выражение `^[0-9]+$` совпадает в тексте, хранящемся в переменной `$reply`, то...» Результат выполнения команды **`$reply =~ m/^[0-9]+$/`** равен *true*, если выражение `^[0-9]+$` совпадает в строке `$reply`, или *false* в противном случае. На основании этого значения команда `if` выбирает выводимое сообщение. Обратите внимание: результат команды **`$reply =~ m/[0-9]+/`** (тот же, что и раньше, но без ограничивающих символов `^` и `$`) равен *true*, если `$reply` содержит *хотя бы одну* цифру. Конструкция `^…$` проверяет, что `$reply` содержит *только* цифры.

Давайте объединим два предыдущих примера. Мы предложим пользователю ввести значение и затем проверим его при помощи регулярного выражения, чтобы убе-

[1] Во многих случаях использование спецификатора m является необязательным. Данный пример может быть также записан как: `$reply =~ /^[0-9]+$/`. Многие читатели, знакомые с Perl, найдут такую форму записи более естественной. На мой взгляд, спецификатор `m` является дополнительным наглядным уточнением назначения регулярного выражения, поэтому я решил использовать его.

диться в том, что было введено число. Если проверка даст положительный результат, мы вычисляем и выводим эквивалентную температуру по шкале Фаренгейта. В противном случае выводится предупреждающее сообщение:

```
print "Enter a temperature in Celsius:\n";
$celsius = <STDIN>; # Получить одну строку от пользователя
chopr($celsius);    # Удалить из $celsius завершающий символ новой строки

if ($celsius =~ m/^[0-9]+$/) {
    $fahrenheit = ($celsius * 9 / 5) + 32; # Вычислить температуру
                                           # по шкале Фаренгейта
    print "$celsius C = $fahrenheit F\n";
} else {
    print "Expecting a number, so I don't understand \"$celsius\".\n";
}
```

Обратите внимание на префикс \ перед кавычками в завершающей команде `print`. Здесь он помогает отличить внутренние кавычки от кавычек, в которые заключена строка. В большинстве языков программирования некоторые символы в строках приходится экранировать, как и метасимволы в регулярных выражениях. В языке Perl связь между строкой и регулярным выражением не так важна, но в языках типа Java, Python и других она играет исключительно важную роль. В разделе «Многоликие метасимволы» (☞ 74) эта тема рассматривается подробнее. На общем фоне выделяется только язык VB.NET, в котором кавычки в строковых литералах экранируются не префиксом (`\"`), а удваиванием (`""`).

Предположим, программа хранится в файле *c2f*. При запуске она выводит следующий результат:

```
% perl -w c2f
Enter a temperature in Celsius:
22
22 C = 71.599999999999994316 F
```

Ой... Оказывается, простая команда `print` плохо подходит для вывода вещественных чисел.

Чтобы не увязнуть в технических деталях Perl, я не стану вдаваться в подробные объяснения и просто скажу, что для улучшения внешнего вида результата можно воспользоваться командой форматного вывода `printf`:

```
printf "%.2f C is %.2f F\n", $celsius, $fahrenheit;
```

Функция `printf` аналогична функции `printf` языка C или команде `format` языков Pascal, Tcl, *elisp* и Python. Команда изменяет не значения переменных, а лишь их внешний вид при выводе. Теперь результат смотрится гораздо лучше:

```
Enter a temperature in Celsius:
22
22.00 C = 71.60 F
```

Переходим к реальным примерам

Пожалуй, наш пример было бы неплохо усовершенствовать, чтобы он поддерживал отрицательные и дробные значения температуры. С точки зрения математики ничего не изменится — обычно Perl не различает целые и вещественные числа. Тем не менее мы должны модифицировать регулярное выражение, чтобы оно разрешало ввод отрицательных и вещественных величин. Вставка `(?=)` допускает присутствие минуса в начале отрицательного числа. А если сделать еще один шаг и вставить `[=+]?`, будет разрешен и начальный плюс для положительных чисел.

Чтобы регулярное выражение поддерживало наличие необязательной дробной части, в него добавляется `(\.[0-9]*)?`. Комбинация `\.` совпадает с обычным символом «точка», поэтому выражение `\.[0-9]*` совпадает с точкой, за которой следует любое количество необязательных цифр. Поскольку подвыражение `\.[0-9]*` заключается в конструкцию `(…)?`, оно как единое целое становится необязательным (и этим принципиально отличается от выражения `\.?[0-9]*`, ошибочно допускающего совпадение дополнительных цифр даже в том случае, если отсутствует совпадение для `\.`).

Объединяя все сказанное, мы получаем:

```
if ($celsius =~ m/^[-+]?[0-9]+(\.[0-9]*)?$/) {
```

Команда допускает такие числа, как `32`, `-3.723` и `+98.6`. В действительности она не идеальна, поскольку не поддерживает чисел, начинающихся с десятичной точки (например, `.357`). Разумеется, пользователь может добавить начальный ноль (т. е. `0.357`), поэтому вряд ли это можно считать крупным недостатком. Проблема вещественных чисел ставит ряд интересных задач, подробно рассматриваемых в главе 5 (☞ 253).

Побочные эффекты успешных совпадений

Давайте расширим наш пример, чтобы пользователь мог вводить температуру как по Цельсию, так и по Фаренгейту. В зависимости от выбранной шкалы к введенному значению присоединяется суффикс `C` или `F`. Чтобы суффикс прошел через регулярное выражение, после выражения для числа просто добавляется `[CF]`. Однако мы должны изменить саму программу так, чтобы она автоматически определяла шкалу введенной температуры и вычисляла значение по другой шкале.

В первой главе говорилось о том, что некоторые версии *egrep* поддерживают метасимволы ⌜\1⌟, ⌜\2⌟, ⌜\3⌟, которые используются внутри регулярного выражения для ссылки на текст, совпавший с предшествующим подвыражением в круглых скобках (☞ 47). Perl и большинство других современных языков с поддержкой регулярных выражений также поддерживают эти метасимволы, но к ним добавляются переменные, которые могут использоваться *за пределами* регулярного выражения после успешного поиска.

Примеры того, как это делается в других языках, будут приведены в следующей главе (☞ 184), а в Perl используются служебные переменные `$1`, `$2`, `$3` и т. д., которые представляют текст совпадения первого, второго, третьего и т. д. подвыражения в круглых скобках. Хотя на первый взгляд они выглядят немного странно, это *действительно* переменные. Просто имена переменных представляют собой числа. Perl присваивает им значения при каждом успешном совпадении регулярного выражения.

На всякий случай повторю: метасимвол ⌜\1⌟ используется в регулярном выражении для ссылки на текст, совпавший ранее в ходе текущей попытки поиска, а переменная `$1` может использоваться в программе для ссылки на этот текст после успешного совпадения.

Чтобы не загромождать пример и сосредоточиться на новых аспектах, я временно удалю подвыражение для идентификации дробной части, но мы к нему еще вернемся. Итак, чтобы увидеть переменную `$1` в действии, сравните:

```
$celsius =~ m/^[-+]?[0-9]+[CF]$/
$celsius =~ m/^([-+]?[0-9]+)([CF])$/
```

Изменяют ли новые круглые скобки общий смысл выражения? Чтобы ответить на этот вопрос, мы должны знать:

- обеспечивают ли они группировку символов для * или других квантификаторов?
- ограничивают ли они конструкцию ⌜|⌟?

На оба вопроса ответ отрицательный, поэтому совпадение всего выражения остается неизменным. Однако в круглые скобки заключены два подвыражения, совпадающие с «интересными» частями проверяемой строки. Как видно из рис. 2.1, в `$1` сохраняется введенное число, а в `$2` — суффикс (`C` или `F`). Из блок-схемы на рис. 2.2 видно, что это позволяет нам легко выбрать дальнейшие действия после применения регулярного выражения.

Программа преобразования температуры

```
print "Enter a temperature (i.e. 32F, 100C):\n";
$input = <STDIN>; # Получить одну строку от пользователя
chomp($input);    # Удалить из $input завершающий символ новой строки
```

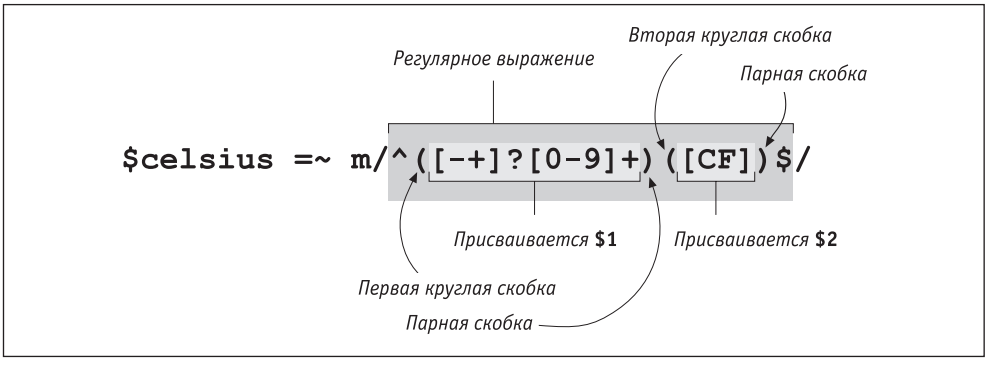


Рис. 2.1. Сохранение текста в круглых скобках

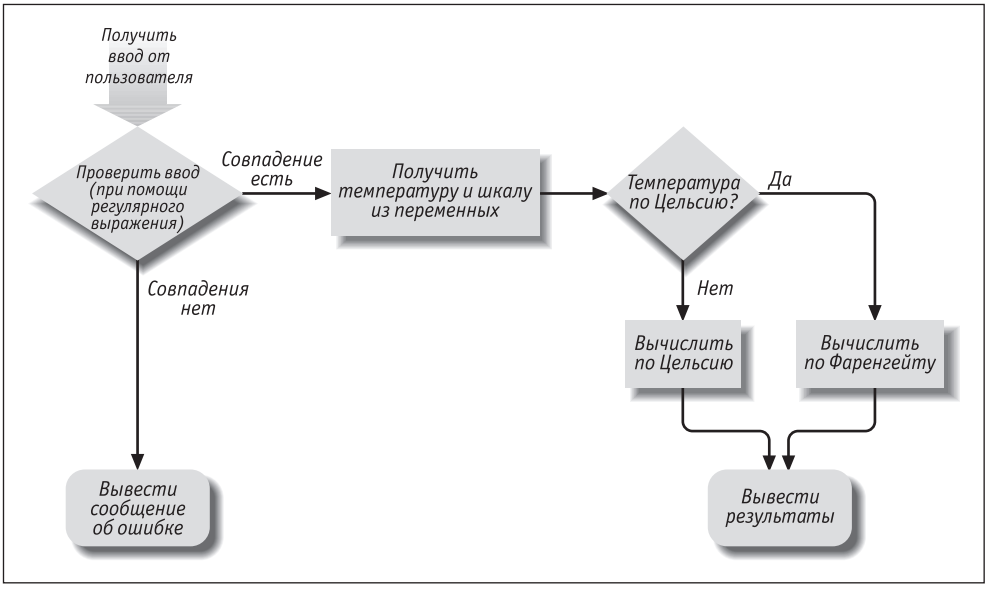


Рис. 2.2. Блок-схема программы преобразования температуры

```
if ($input =~ m/^([-+]?[0-9]+)([CF])$/)
{
    # Обнаружено совпадение. $1 содержит число, $2 - символ "C" или "F"
    $InputNum = $1;  # Сохранить в именованных переменных,
    $type     = $2;  # чтобы программу было легче читать
    if ($type eq "C") {    # 'eq' проверяет равенство двух строк
        # Температура введена по Цельсию, вычислить по Фаренгейту
        $celsius = $InputNum;
        $fahrenheit = ($celsius * 9 / 5) + 32;
    } else {
```

```
        # Видимо, температура введена по Фаренгейту. Вычислить по Цельсию.
        $fahrenheit = $InputNum;
        $celsius = ($fahrenheit - 32) * 5 / 9;
    }
    # Известны обе температуры, переходим к выводу результатов
    printf "%.2f C = %.2f F\n", $celsius, $fahrenheit;
} else {
    # Регулярное выражение не совпало, вывести предупреждение.
    print "Expecting a number followed by \"C\" or \"F\",\n";
    print "so I don't understand \"$input\".\n";
}
```

Если бы эта программа хранилась в файле *convert*, то пример ее использования мог бы выглядеть так:

```
% perl -w convert
Enter a temperature (e.g. 32F, 100C):
39F
3.89 C = 39.00 F
% perl -w convert
Enter a temperature (e.g. 32F, 100C):
39C
39.00 C = 102.20 F
% perl -w convert
Enter a temperature (e.g. 32F, 100C):
oops
Expecting a number followed by "C" or "F",
so I don't understand "oops".
```

Взаимодействие регулярных выражений с логикой программы

В таких нетривиальных языках программирования, как Perl, применение регулярных выражений может переплетаться с логикой самой программы. Давайте внесем в нашу программу три полезных изменения: обеспечим поддержку вещественных чисел, как это было сделано ранее, поддержим ввод суффиксов `f` и `c` в нижнем регистре и разрешим, чтобы число отделялось от буквы пробелами. После внесения всех изменений можно будет вводить данные вида ‘`98.6•f`’.

Вы уже знаете, как обеспечить возможность ввода вещественных чисел — для этого в выражение включается конструкция `(\.[0-9]*)?`:

```
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)([CF])$/)
```

Обратите внимание: вставка происходит *внутри* первой пары круглых скобок. Поскольку первые скобки используются для сохранения числа, в них также должна

быть включена его дробная часть. Однако появление новой пары круглых скобок, пусть даже предназначенных только для применения квантификатора, приводит к побочному эффекту — содержимое этих скобок также сохраняется в переменной. Поскольку открывающая скобка является второй слева в выражении, дробная часть числа сохраняется в `$2`. Сказанное поясняет рис. 2.3.

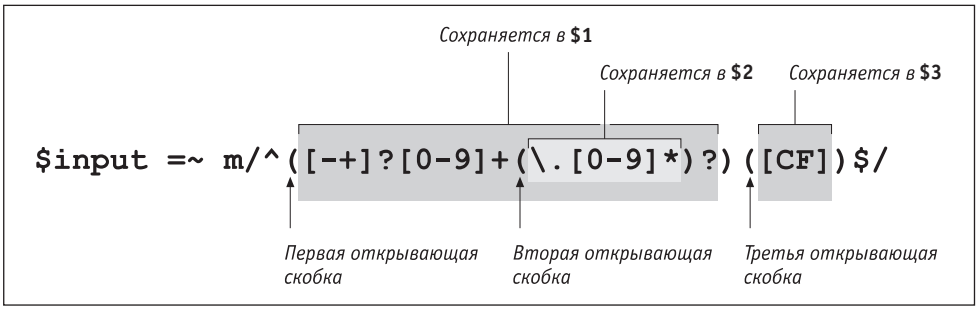


Рис. 2.3. Вложенные круглые скобки

Как видно из рисунка, появление новых круглых скобок в предшествующей части выражения не изменяет смысла выражения ⌜`[CF]`⌟ напрямую, но отражается на нем косвенно. Дело в том, что круглые скобки, в которые заключено это подвыражение, становятся третьей парой, а это означает, что в присваивании `$type` необходимо указать `$3` вместо `$2` (впрочем, во врезке на следующей странице описано альтернативное решение).

Пробелы между числом и суффиксом вызывают меньше проблем. Мы знаем, что обычный пробел в регулярном выражении соответствует ровно одному пробелу в тексте, поэтому используем конструкцию ⌜`•*`⌟, чтобы разрешить любое количество пробелов (ни один из которых не является обязательным):

```
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)•*([CF])$/)
```

Такое решение обеспечивает определенную гибкость, но раз уж мы стремимся создать пример, пригодный для практического использования, давайте включим в регулярное выражение поддержку других разновидностей *пропусков* (*whitespace*). Одной из распространенных разновидностей пропусков являются символы табуляции. Конечно, конструкция ⌜`⇥*`⌟ не позволит использовать пробелы, поэтому мы должны сконструировать символьный класс для обеих разновидностей пропусков: ⌜`[•⇥]*`⌟.

Кстати, чем это выражение принципиально отличается от ⌜`(•*|⇥*)`⌟? ❖ Подумайте над этим вопросом, затем переверните страницу и проверьте свой ответ.

ОТВЕТ НА ВОПРОС

❖ *Ответ на вопрос со с. 73.*

Чем ⌜[•⇥]*⌟ отличается от ⌜[•*|⇥*]⌟?

Выражение ⌜(•*|⇥*)⌟ совпадает либо с ⌜•*⌟, либо с ⌜⇥*⌟. Иначе говоря, в тексте должны стоять либо пробелы (или ничего), либо символы табуляции (или ничего). Однако это выражение не позволяет *комбинировать* пробелы с символами табуляции.

⌜[•⇥]*⌟ означает ⌜[•⇥]⌟, повторенное любое количество раз. В строке «tabspcspc» оно совпадает трижды — сначала с символом табуляции, а затем с пробелами.

Выражение ⌜[•⇥]*⌟ логически эквивалентно ⌜(•|⇥)*⌟, хотя по причинам, объясненным в главе 4, символьный класс обычно работает более эффективно.

В этой книге пробелы и табуляции легко различаются благодаря использованным мною обозначениям • и ⇥. К сожалению, на экране различить их сложнее. Если вы увидите что-нибудь вроде [ ]*, логично будет предположить, что это пробел и символ табуляции, но до проверки полностью уверенным в этом быть нельзя. Для удобства в регулярных выражениях Perl существует метасимвол ⌜\t⌟, который просто совпадает с символом табуляции. Единственным преимуществом этого символа перед литералом является его визуальная наглядность, и я часто использую его в своих выражениях. Таким образом, ⌜[•⇥]*⌟ превращается в ⌜[•\t]*⌟.

Существуют и другие вспомогательные метасимволы: ⌜\n⌟ (новая строка), ⌜\f⌟ (перевод формата) и ⌜\b⌟ (забой). Вообще говоря, ⌜\b⌟ в одних ситуациях означает забой, а в других — границу слова. Так какое же из двух значений соответствует символу? Об этом вы узнаете в следующем разделе.

Многоликие метасимволы

Символ \n уже встречался нам в предыдущих примерах, однако он находился внутри строки, а не в регулярном выражении. *Строки* Perl обладают собственными метасимволами, которые не имеют ничего общего с метасимволами *регулярных выражений*. Программисты-новички очень часто путают их (хотя, например, в языке VB.NET количество строковых метасимволов чрезвычайно мало). Некоторые строковые метасимволы очень похожи на аналогичные метасимволы регулярных выражений. Например, при помощи строкового метасимвола \t можно вставить символ табуляции в строку, а при помощи метасимвола регулярных выражений ⌜\t⌟ в выражение включается элемент, совпадающий с символом табуляции.

НЕСОХРАНЯЮЩИЕ КРУГЛЫЕ СКОБКИ: `(?:...)`

На рис. 2.3 круглые скобки в выражении `(\.[0-9]*)?` использованы с целью группировки, чтобы мы могли применить квантификатор «?» ко всей конструкции `\.[0-9]*` и сделать ее необязательной. Но при этом возникает побочный эффект: текст, совпавший с выражением в круглых скобках, сохраняется в переменной `$2`, которая не используется в нашей программе. Конечно, было бы хорошо использовать при группировке такую разновидность круглых скобок, которая бы не приводила к затратам (и возможным недоразумениям), связанным с сохранением текста в неиспользуемых переменных.

Такая возможность существует в Perl, а с недавнего времени — и в других диалектах регулярных выражений. Вместо конструкции `(...)`, производящей группировку с сохранением, используется специальная разновидность круглых скобок `(?:...)`, которая обеспечивает группировку без сохранения. При такой записи «открывающая круглая скобка» состоит из трех символов `(?:`, что выглядит довольно странно. В данном случае вопросительный знак не имеет отношения к «необязательному» метасимволу `?` (на с. 126 объясняется, почему была выбрана эта странная запись).

Итак, выражение приходит к виду:

```
if ($input =~ m/^([-+]?[0-9]+(?:\.[0-9]*)?)([CF])$/)
```

Теперь при том, что подвыражение `[CF]` заключено в третий набор круглых скобок, совпадающий с ним текст обозначается переменной `$2`, поскольку конструкция `(?:...)` при подсчете не учитывается.

Такое решение обладает двумя преимуществами. Первое — повышение эффективности поиска за счет исключения лишнего сохранения (проблема эффективности подробно рассматривается в главе 6). Второе преимущество — точное указание типа скобок упрощает последующее чтение программы и снимает вопросы относительно того, для чего нужна та или иная пара скобок.

С другой стороны, конструкция `(?:...)` усложняет восприятие выражения с первого взгляда. Не перечеркивает ли это ее преимуществ? Лично я предпочитаю точно указывать тип необходимых скобок, но в данном случае игра не стоит свеч. Например, о повышении эффективности говорить не приходится, поскольку поиск производится всего один раз (в отличие от многократного поиска в цикле).

В этой главе чаще используются обычные (сохраняющие) круглые скобки `(...)` — не потому, что мы используем свойство сохранения, а просто ради наглядности.

Такое сходство удобно, но при этом крайне важно, чтобы вы не путали разные типы метасимволов. В простейших случаях вроде `\t` это кажется несущественным, но,

как будет показано при описании различных языков и программ, умение различать метасимволы, используемые в каждой ситуации, играет очень важную роль.

На самом деле мы уже встречались с примерами конфликта метасимволов. В главе 1 при работе с *egrep* регулярные выражения, как вы помните, обычно заключались в апострофы. Командная строка *egrep* целиком передается в приглашении командного интерпретатора, и интерпретатор распознает некоторые из своих метасимволов. Например, для него пробел является метасимволом, отделяющим команду от аргументов, а аргументы — друг от друга. Во многих командных интерпретаторах апострофы являются метасимволами, которые сообщают интерпретатору о том, что в заключенном между ними тексте не следует распознавать другие метасимволы (в DOS для этой цели используются кавычки).

Апострофы позволяют включать в регулярное выражение пробелы. Без апострофов командный интерпретатор самостоятельно интерпретирует пробелы вместо того, чтобы предоставить *egrep* возможность интерпретировать их *по-своему*. Во многих командных интерпретаторах также используются метасимволы `$`, `*`, `?` и т. д., часто встречающиеся в регулярных выражениях.

Впрочем, весь этот разговор о метасимволах командного интерпретатора и строковых метасимволах Perl не имеет прямого отношения к регулярным выражениям, но непосредственно связан с *применением* регулярных выражений в реальных ситуациях. В этой книге нам встретятся многочисленные и порой довольно сложные ситуации, в которых используется многоуровневое взаимодействие метасимволов.

Так как же насчет ⌜`\b`⌟? В Perl этот метасимвол обычно совпадает с границей слова, но в символьном классе ему соответствует символ «забой». Граница слова в символьном классе не имеет смысла, поэтому Perl имеет полное право присвоить этому метасимволу какой-то другой смысл (равно как и другие диалекты регулярных выражений).

Обобщенный пропуск (метасимвол \s)

При рассказе о пропусках мы рассматривали регулярное выражение ⌜`[•\t]*`⌟. Это неплохо, однако во многих диалектах регулярных выражений используется удобная сокращенная запись: ⌜`\s`⌟. В отличие от метасимвола ⌜`\t`⌟, который представляет литерал-табуляцию, метасимвол ⌜`\s`⌟ обеспечивает сокращенную запись для целого символьного класса, означающего «любой пропуск». К этой категории относятся пробелы, табуляции, символы новой строки и возврата каретки. В нашем примере символы новой строки и возврата каретки не используются, однако вводить выражение ⌜`\s*`⌟ удобнее, чем ⌜`[•\t]*`⌟. Вскоре вы привыкнете к этому метасимволу и будете легко узнавать конструкцию ⌜`\s*`⌟ даже в сложных выражениях.

Наша команда теперь выглядит так:

```
$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/
```

Наконец, суффикс может вводиться не только в верхнем, но и в нижнем регистре. Задача решается простым включением символов нижнего регистра в символьный класс: `[CFcf]`. Впрочем, я хочу продемонстрировать еще один способ:

```
$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i
```

Ключ `i` является *модификатором*. Его присутствие после `m/.../` сообщает Perl о том, что поиск должен осуществляться без учета регистра символов. Символ `i` является частью не регулярного выражения, а синтаксической оболочки `m/.../` и указывает Perl, как именно должно обрабатываться указанное регулярное выражение. Аналогичный пример уже встречался ранее, когда мы рассматривали ключ `-i` программы *egrep* (☞ 40).

Впрочем, постоянно говорить «модификатор `i`» неудобно, поэтому обычно используется обозначение `/i` (хотя при использовании модификатора дополнительный символ `/` не нужен). Запись `/i` дает лишь один пример определения модификаторов в Perl; в следующей главе будут представлены другие способы, а также будет показано, как аналогичные возможности реализуются в других языках. На страницах книги вам встретятся и другие модификаторы, в том числе `/g` (модификатор глобального поиска) и `/x` (модификатор свободной записи выражений), с которыми вы познакомитесь в этой главе.

Попробуем запустить новую версию программы после всех изменений:

```
% perl -w convert
Enter a temperature (e.g. 32F, 100C):
32 f
0.00 C = 32.00 F
% perl -w convert
Enter a temperature (e.g. 32F, 100C):
50 c
10.00 C = 50.00 F
```

Стоп! При втором запуске мы ввели 50° по Цельсию, которые почему-то интерпретируются как 50° по Фаренгейту! Просмотрите текст программы. Вы видите, почему это произошло?

Обратитесь к следующей части программы:

```
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i)
{
    ⋮
    $type = $3;            # Использование $type вместо $3
                           # упрощает чтение программы
    if ($type eq "C") {    # 'eq' проверяет равенство двух строк
      ⋮
    } else {
      ⋮
```

Мы изменили регулярное выражение, чтобы оно допускало суффикс `f` в нижнем регистре, но забыли предусмотреть его обработку в оставшейся части программы. Сейчас, если `$type` не содержит в точности символ `C`, мы предполагаем, что пользователь ввел температуру по Фаренгейту. Поскольку шкала Цельсия также может обозначаться символом `c`, необходимо изменить условие проверки `$type`:

```
if ($type eq "C" or $type eq "c") {
```

Вообще говоря, раз уж книга посвящена регулярным выражениям, логичнее было бы использовать команду:

```
if ($type =~ m/c/i) {
```

В любом случае программа работает так, как нужно. Ниже приведен окончательный вариант ее кода. Приведенные примеры наглядно показывают, что применение регулярных выражений может быть взаимосвязано с остальной программой.

Программа преобразования температуры — окончательная версия

```
print "Enter a temperature (e.g. 32F, 100C):\n";
$input = <STDIN>;  # Получить одну строку от пользователя.
chomp($input);     # Удалить из $input завершающий символ новой строки.

if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i)
{
    # Обнаружено совпадение. $1 содержит число, $3 - символ "C" или "F".
    $InputNum = $1;  # Сохранить в именованных переменных,
    $type     = $3;  # чтобы программу было легче читать.
    if ($type =~ m/c/i) {    # "c" или "C"?
        # Температура введена по Цельсию, вычислить по Фаренгейту
        $celsius = $InputNum;
        $fahrenheit = ($celsius * 9 / 5) + 32;
    } else {
        # Видимо, температура введена по Фаренгейту. Вычислить по Цельсию.
        $fahrenheit = $InputNum;
        $celsius = ($fahrenheit - 32) * 5 / 9;
    }
    # Известны обе температуры, переходим к выводу результатов
    printf "%.2f C = %.2f F\n", $celsius, $fahrenheit;
} else {
    # Регулярное выражение не совпало, вывести предупреждение.
    print "Expecting a number followed by \"C\" or \"F\",\n";
    print "so I don't understand \"$input\".\n";
}
```

Лирическое отступление

Хотя эта глава началась с ускоренного описания Perl, стоит ненадолго отвлечься и особо выделить некоторые аспекты, относящиеся к регулярным выражениям.

1. Регулярные выражения Perl отличаются от регулярных выражений *egrep*; почти каждая программа поддерживает собственный диалект регулярных выражений. Регулярные выражения Perl относятся к тому же типу, но обладают расширенным набором метасимволов. Во многих языках (Java, Python, языки .NET и Tcl) реализованы диалекты, аналогичные диалекту Perl.

2. Для поиска совпадений регулярного выражения в строке используется конструкция **`Svariable =~ m/`***`регулярное_выражение`*`/`. Символ `m` определяет операцию поиска *совпадения* (*match*), а символы `/` ограничивают регулярное выражение, но не являются его частью. Результатом проверки является логическая величина, `true` или `false`.

3. Концепция метасимволов (символов с особой интерпретацией) используется не только при работе с регулярными выражениями. Как говорилось выше при обсуждении командных интерпретаторов и строк, заключенных в кавычки, концепция служебных символов встречается во многих контекстах. Знание различных контекстов (командные интерпретаторы, регулярные выражения, строки и т. д.), их метасимволов и возможностей окажет существенную помощь при изучении Perl, PHP, Java, Tcl, GNU Emacs, awk, Python и других нетривиальных языков. Также следует помнить о том, что символьные классы регулярных выражений обладают собственным мини-языком с отдельным набором метасимволов.

4. К числу самых полезных метасимволов, поддерживаемых в регулярных выражениях Perl, принадлежат следующие (некоторые из приведенных метасимволов еще не упоминались в книге).

`\t`	символ табуляции
`\n`	символ новой строки
`\r`	символ возврата курсора
`\s`	класс, совпадающий с любым «пропускным» символом (пробел, табуляция, новая строка, подача листа и т. д.)
`\S`	все, что не относится к ⌈`\s`⌋
`\w`	⌈`[a-zA-Z0-9_]`⌋ (часто используется в конструкции ⌈`\w+`⌋ для поиска слов)
`\W`	все, что не относится к ⌈`\w`⌋, т. е. ⌈`[^a-zA-Z0-9_]`⌋
`\d`	⌈`[0-9]`⌋, т. е. цифра
`\D`	все, что не относится к ⌈`\d`⌋, т. е. ⌈`[^0-9]`⌋

5. При наличии модификатора `/i` поиск производится без учета регистра символов. Хотя в тексте обычно используется обозначение «`/i`», в действительности после завершающего ограничителя ставится только символ «`i`».

6. Для группировки символов без сохранения может использоваться непривлекательная конструкция `(?:…)`.

7. При обнаружении совпадения в переменные `$1`, `$2`, `$3` заносится текст, совпавший с соответствующими подвыражениями в круглых скобках. Благодаря этим переменным регулярные выражения могут использоваться для извлечения данных из строки (в других языках для получения аналогичной информации применяются другие средства; многочисленные примеры будут приведены в следующей главе).

Подвыражения нумеруются в соответствии с номером по порядку открывающей круглой скобки, начиная с 1. Подвыражения могут быть вложенными — например, `(Washington(•DC)?)`. Обычные (сохраняющие) круглые скобки `(…)` могут быть применены только с целью группировки, но и в этом случае соответствующая специальная переменная заполняется текстом совпадения.

Модификация текста с использованием регулярных выражений

Во всех примерах, рассмотренных выше, мы ограничивались поиском и, в отдельных случаях, «извлечением» информации из строки. Сейчас мы перейдем к *подстановке*, или *поиску с заменой* — возможности, поддерживаемой Perl и многими другими программами.

Как вы уже знаете, конструкция **`$var =~ m/`***`выражение`*`/`** сопоставляет регулярное выражение с содержимым переменной и возвращает `true` или `false` в зависимости от результата. Аналогичная конструкция **`$var =~ s/`***`выражение`*`/`***`замена`*`/`** представляет собой следующий шаг в работе с текстом: если регулярное выражение совпадает в строке `$var`, то фактически совпавший текст заменяется заданной строкой. При этом используется такое же регулярное выражение, как и в `m/…/`, но строка замены (между средним и последним символом `/`) интерпретируется по правилам строк, заключенных в кавычки. Это означает, что она может содержать ссылки на переменные (в частности, очень удобно использовать переменные `$1`, `$2` и т. д., ссылающиеся на компоненты найденного совпадения).

Таким образом, конструкция **`$var =~ s/…/…/`** изменяет значение переменной `$var` (впрочем, если совпадение не обнаружено, замена не производится и переменная сохраняет прежнее значение). Например, если переменная `$var` содержит строку `Jeff•Friedl`, то при выполнении команды

```
$var =~ s/Jeff/Jeffrey/;
```

в переменную $var заносится текст Jeffrey•Friedl. Но если переменная $var изначально содержала строку Jeffrey•Friedl, то после выполнения этой команды она примет вид Jeffreyrey•Friedl. Обычно при подобных заменах используются метасимволы границ слов. Как упоминалось в первой главе, некоторые версии *egrep* поддерживают метасимволы ⌜\<⌟ и ⌜\>⌟ для обозначения *начала* и *конца слова*. В Perl для этой цели существует универсальный метасимвол ⌜\b⌟:

```
$var =~ s/\bJeff\b/Jeffrey/;
```

А теперь каверзный вопрос. В конструкции s/…/…/, как и в конструкции m/…/, могут использоваться модификаторы (например, /i — см. с. 77). Модификаторы указываются после строки замены. К какому результату приведет выполнение следующей команды?

```
$var =~ s/\bJeff\b/Jeff/i;
```

❖ Переверните страницу и проверьте свой ответ.

Пример: письмо на стандартном бланке

Рассмотрим несерьезный пример, демонстрирующий использование переменной в строке замены. Предположим, вы генерируете письма по шаблону, в котором специальной разметкой выделены переменные фрагменты:

```
Dear =FIRST=,
You have been chosen to win a brand new =TRINKET=! Free!
Could you use another =TRINKET= in the =FAMILY= household?
Yes =SUCKER=, I bet you could! Just respond by.....
```

Чтобы заполнить шаблон данными конкретного получателя, вы присваиваете переменным необходимые значения:

```
$given = "Tom";
$family = "Cruise";
$wunderprize = "100% genuine faux diamond";
```

После подготовки шаблон заполняется следующими командами:

```
$letter =~ s/=FIRST=/$given/g;
$letter =~ s/=FAMILY=/$family/g;
$letter =~ s/=SUCKER=/$given $family/g;
$letter =~ s/=TRINKET=/fabulous $wunderprize/g;
```

Каждая команда ищет в тексте простой маркер и, обнаружив его, заменяет текстом, который должен присутствовать в итоговом сообщении. Текст замены также

ОТВЕТ НА ВОПРОС

❖ *Ответ на вопрос со с. 81.*

Что делает команда `$var =~ s/\bJeff\b/Jeff/i`?

Из-за того, как сформулирован этот вопрос, он действительно может оказаться каверзным. Если бы регулярное выражение имело вид `⌜\bJEFF\b⌟`, `⌜\bjeff\b⌟` или даже `⌜\bjEfF\b⌟`, возможно, смысл этой команды стал бы более очевидным. При наличии модификатора `/i` слово «Jeff» ищется без учета регистра символов. После этого оно заменяется словом «Jeff», записанным именно этими символами (модификатор `/i` не влияет на текст замены, хотя существуют и другие модификаторы, о которых речь пойдет в главе 7).

В итоге текст «`jeff`» с произвольным сочетанием регистра символов заменяется текстом «`Jeff`» (именно в таком виде).

представлен строкой Perl и поэтому может содержать ссылки на переменные, как в нашем примере. Например, подчеркнутая часть команды `s/=TRINKET=/fabulous $wunderprize/g` интерпретируется по тем же правилам, что и строка `"fabulous $wunderprize"`. Если вы пишете всего одно сообщение, эти переменные можно опустить и непосредственно ввести нужный текст, но показанный метод позволяет автоматизировать процесс рассылки (например, если данные получателей загружаются из списка).

Модификатор «глобального поиска» `/g` нам еще не встречался. Он говорит о том, что после первой подстановки команда `s/…/…/` должна продолжить поиск совпадений (и произвести дополнительные замены). Модификатор `/g` необходим, если проверяемый текст может содержать несколько экземпляров искомого текста и вы хотите произвести все возможные замены, не ограничиваясь первым найденным совпадением.

Результат вполне предсказуем:

```
Dear Tom,
You have been chosen to win a brand new fabulous 100% genuine faux diamond! Free!
Could you use another fabulous 100% genuine faux diamond in the Cruise household?
Yes Tom Cruise, I bet you could! Just respond by...
```

Пример: обработка биржевых котировок

Рассмотрим другой пример — проблему, с которой я столкнулся во время написания на Perl программы для работы с биржевыми котировками. Я получал коти-

ровки вида «9,0500000037272». Конечно, настоящее значение было равно 9,05, но из-за особенностей внутреннего представления вещественных чисел Perl выводил их в таком виде, если при выводе не использовалось форматирование. В обычной ситуации я бы просто воспользовался функцией `printf` для вывода числа с двумя разрядами в дробной части, как в примере с преобразованием температур, но в данном случае этот вариант не годился. Дело в том, что дробная часть, равная 1/8, должна выводиться в виде «.125», и в этом случае необходимы три цифры вместо двух.

В формальном виде моя задача звучала так: «Всегда выводить первые две цифры дробной части, а третью — лишь в том случае, если она не равна нулю. Все остальные цифры игнорируются». В результате число `12.3750000000392` или уже правильное `12.375` возвращается в виде «12,375», а `37.500` сокращается до «37,50». Именно то, что нам требовалось.

Но как реализовать это требование? Строка, в которой выполняется поиск, хранится в переменной `$price`, поэтому мы воспользуемся командой

```
$price =~ s/(\.\d\d[1-9]?)\d*/$1/
```

(напомню: метасимвол ⌜`\d`⌟, упоминавшийся на с. 79, представляет цифру).

Начальная конструкция ⌜`\.`⌟ нужна для того, чтобы совпадение начиналось с десятичной точки. Затем два метасимвола ⌜`\d\d`⌟ совпадают с двумя цифрами, следующими за точкой. Подвыражение ⌜`[1-9]?`⌟ совпадает с ненулевой цифрой, если она следует за первыми двумя. Все совпавшие до настоящего момента символы образуют часть, которую мы хотим *оставить*, поэтому они заключаются в круглые скобки для сохранения в переменной `$1`. Затем значение `$1` используется в строке замены. Если ничего больше не совпало, строка заменяется сама собой — не слишком интересно. Однако вне круглых скобок `$1` продолжается поиск других символов. Эти символы не включаются в строку замены и поэтому они фактически удаляются из числа. В данном случае «удаляемый» текст состоит из всех дальнейших цифр, т. е. ⌜`\d*`⌟ в конце регулярного выражения.

Запомните этот пример; мы вернемся к нему в главе 4, при изучении механизма поиска совпадений. Из этого примера следует ряд очень интересных выводов.

Автоматизация редактирования

Во время работы над этой главой я столкнулся еще с одним простым, но вполне реальным примером. Я подключился к компьютеру на другом берегу Тихого океана, причем сеть работала на редкость медленно. Реакция на простое нажатие клавиши Enter следовала только через минуту. Я должен был внести несколько

мелких -исправлений в файл, чтобы заработала важная программа. Все, что требовалось, — заменить в файле каждый вызов `sysread` на `read`. Исправлений было немного, но при такой замедленной реакции сама идея запуска полноэкранного редактора выглядела нереально.

Ниже приведена команда, которая автоматически внесла все необходимые изменения:

```
% perl -p -i -e 's/sysread/read/g' имя_файла
```

Команда выполняет программу Perl **`s/sysread/read/g`**. (Да, это действительно целая программа, о чем свидетельствует ключ `-e`, после которого в командной строке следует программа.) Ключ `-p` означает, что подстановка выполняется в каждой строке указанного файла, а ключ `-i` означает, что после подстановки результаты записываются обратно в файл.

Обратите внимание: в команде не указано имя переменной (**`$var =~`** …), поскольку ключ `-p` обеспечивает неявное выполнение программы для каждой строки файла. Кроме того, модификатор `/g` гарантировал замену всех экземпляров в одной строке.

Хотя я применил эту программу к одному файлу, с таким же успехом можно было указать в командной строке имена нескольких файлов; Perl применил бы подстановку к каждой строке каждого файла. В этом случае одна простая команда произвела бы массовое редактирование во множестве файлов. Конкретная реализация специфична для Perl, но мораль данного примера состоит в том, что регулярные выражения в сценарных языках обладают огромными возможностями даже в малых дозах.

Маленькая почтовая утилита

Рассмотрим еще один пример. Допустим, у нас имеется файл с сообщением электронной почты и мы хотим подготовить файл с ответом. В процессе подготовки каждая строка исходного сообщения должна быть процитирована в ответе, чтобы мы могли легко вставить свой текст в каждую часть сообщения. Кроме того, из заголовка исходного сообщения необходимо удалить лишние строки, а также подготовить заголовок ответного сообщения.

Некоторые поля заголовка (дата, тема и прочие) представляют интерес для нас, но остальные поля необходимо удалить. Если сценарий, который мы собираемся написать, называется *mkreply*, а исходное сообщение хранится в файле *king.in*, шаблон ответа строится следующей командой:

```
% perl -w mkreply king.in > king.out
```

(На всякий случай напоминаю: ключ `-w` включает выдачу предупреждений, ☞ 66.)

ПРИМЕР СООБЩЕНИЯ

```
From elvis Thu Feb 29 11:15 2007
Received: from elvis@localhost by tabloid.org (8.11.3) id KA8CMY
Received: from tabloid.org by gateway.net (8.12.5/2) id N8XBK
To: jfriedl@regex.info (Jeffrey Friedl)
From: elvis@tabloid.org (The King)
Date: Thu, Feb 29 2007 11:15
Message-Id: <2007022939939.KA8CMY@tabloid.org>
Subject: Be seein' ya around
Reply-To: elvis@hh.tabloid.org
X-Mailer: Madam Zelda's Psychic Orb [version 3.7 PL92]

Sorry I haven't been around lately. A few years back I checked
into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
The Duke says "hi".
    Elvis
```

Мы хотим, чтобы итоговый файл *king.out* выглядел примерно так:

```
To: elvis@hh.tabloid.org (The King)
From: jfriedl@regex.info (Jeffrey Friedl)
Subject: Re: Be seein' ya around

On Thu, Feb 29 2007 11:15 The King wrote:
|> Sorry I haven't been around lately. A few years back I checked
|> into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
|> The Duke says "hi".
|>        Elvis
```

Проанализируем постановку задачи. Для построения нового заголовка необходимо знать адрес получателя (в данном случае `elvis@hh.tabloid.org`, прочитанный из поля `Reply-To` оригинала), настоящее имя получателя (`The King`), имя и адрес отправителя, а также тему. Кроме того, для вывода вступительной строки перед телом сообщения необходимо знать дату.

Задача делится на три фазы:

1) извлечение данных из заголовка сообщения;

2) вывод заголовка ответа;

3) вывод строк сообщения с префиксом `|>•`.

Пожалуй, я немного опережаю события — чтобы обработать данные, необходимо сначала прочитать их в программе. К счастью, в Perl эта задача легко решается при помощи волшебного оператора `<>`. Когда эта странная конструкция присваивается обычной переменной (`$variable = <>`), в переменную заносится следующая строка

входных данных. Входные данные читаются из файлов, имена которых передаются сценарию Perl в командной строке (файл *king.in* в приведенном примере).

Не путайте оператор `<>` с командой перенаправления данных `> имя_файла` командного интерпретатора или операторами Perl `>=/<=`. Это всего лишь своеобразный аналог функции `getline()` в языке Perl.

После того как все входные данные будут прочитаны, `<>` для удобства возвращает неопределенное значение (интерпретируемое как логическое значение `false`), поэтому при обработке файлов часто применяется следующая конструкция:

```
while ($line = <>) {
    ...работаем с $line...
}
```

Похожая конструкция будет использована и в нашей задаче, но в соответствии с ее постановкой заголовок необходимо обрабатывать отдельно. Заголовок состоит из всех строк, находящихся перед первой пустой строкой; далее следует тело сообщения. Чтобы ограничиться чтением заголовка, можно воспользоваться следующим фрагментом:

```
# Обработка заголовка
while ($line = <>) {
    if ($line =~ m/^\s*$/) {
      last; # Прервать цикл while, продолжить ниже
    }
    ...Обработать строку заголовка...
}
...Обработка прочих строк сообщения...
   ⋮
```

Поиск пустой строки, завершающей заголовок, осуществляется при помощи регулярного выражения `^\s*$`. Выражение ищет начало строки (имеется у всех строк), за которым следует любое количество пропускных символов (хотя на самом деле их быть не должно, если не считать завершающий символ новой строки), после чего логическая строка завершается[1]. Ключевое слово `last` прерывает цикл `while`, останавливая обработку заголовка.

Итак, внутри цикла, после проверки пустой строки, со строкой заголовка можно сделать все, что потребуется. Мы должны выделить из нее необходимые данные — тему и дату сообщения.

---

[1] Я использую термин «логическая строка», потому что в общем случае регулярное выражение может применяться к тексту, содержащему несколько логических строк. Метасимволы `^` и `$` обычно совпадают только в начале и конце всего текста (противоположный пример будет приведен позже). В данном случае различия несущественны, так как вследствие специфики алгоритма мы *знаем*, что переменная `$line` всегда состоит из одной логической строки.

БУДЬТЕ ВНИМАТЕЛЬНЫ С「.*」

Выражение 「.*」 часто обозначает «последовательность любых символов», поскольку точка может совпадать с чем угодно (или в некоторых программах — с чем угодно, кроме символа новой строки). Звездочка же означает, что допускается любое количество символов, но ни один не является обязательным. Такая конструкция весьма полезна.

Однако в ней таятся «подводные камни», с которыми может столкнуться пользователь, не понимающий тонкостей применения этой конструкции в контексте большого выражения. Мы уже столкнулись с одним примером (☞ 54), а другие примеры неоднократно встретятся нам при подробном рассмотрении этой темы в главе 4 (☞ 216).

Для выделения темы используется распространенный прием, который мы будем часто использовать в будущем:

```
if ($line =~ m/^Subject: (.*)/i) {
    $subject = $1;
}
```

В этом фрагменте мы ищем строки, начинающиеся с символов «`Subject:•`». После того как эта часть выражения совпадет, следующее подвыражение 「`.*`」 совпадает со всеми остальными символами в строке. Поскольку подвыражение 「`.*`」 заключено в круглые скобки, тема сообщения заносится в переменную `$1`. В нашем примере содержимое этой переменной просто сохраняется в переменной `$subject`. Конечно, если регулярное выражение не совпадает в строке (а в большинстве строк дело обстоит именно так), условие команды `if` не выполняется, и для этой строки значение переменной `$subject` не присваивается.

Аналогично происходит поиск полей `Date` и `Reply-To`:

```
if ($line =~ m/^Date: (.*)/i) {
    $date = $1;
}
if ($line =~ m/^Reply-To: (.*)/i) {
    $reply_address = $1;
}
```

С полем `From:` придется повозиться подольше. Во-первых, нам нужна строка, которая начинается с «`From:`», а не первая строка, начинающаяся с «`From•`». Речь идет о строке:

```
From: elvis@tabloid.org (The King)
```

В ней содержится адрес, а также имя отправителя, заключенное в круглые скобки. Нас интересует имя.

Чтобы пропустить адрес, мы используем выражение ⌜^From:•(\S+)⌟. Как упоминалось выше, ⌜\S⌟ совпадает со всеми символами, которые не являются пропусками (☞ 79), поэтому ⌜\S+⌟ совпадает до первого пропуска (или до конца целевого текста). В данном случае совпадение определяет адрес отправителя. После успешного совпадения мы хотим выделить текст, заключенный в круглые скобки. Конечно, для этого нужно сначала найти сами скобки, поэтому мы используем конструкции ⌜\(⌟ и ⌜\)⌟ (экранируя скобки для того, чтобы отменить их метасимвольную интерпретацию). Внутри скобок в совпадение должны попасть все символы — кроме других круглых скобок! Задача решается выражением ⌜[^()]*⌟ (вспомните: метасимволы символьных классов отличаются от метасимволов «обычных» регулярных выражений; внутри символьного класса круглые скобки не имеют специальной интерпретации, поэтому экранировать их не нужно).

Итак, объединив все сказанное, мы получаем:

⌜From:•(\S+)•\(([^()]*)\)⌟

Чтобы вы не запутались в многочисленных круглых скобках, на рис. 2.4 структура этого выражения изображена более наглядно.

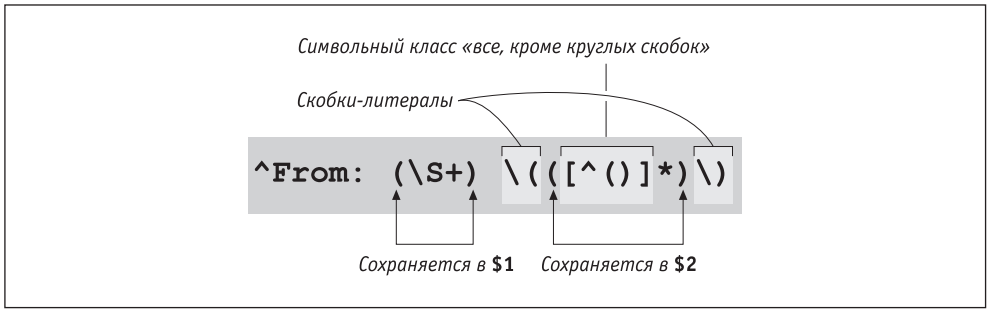


Рис. 2.4. Вложенные круглые скобки; $1 и $2

Когда регулярное выражение, показанное на рис. 2.4, совпадает, имя отправителя сохраняется в переменной `$2`, а возможный адрес — в переменной `$1`:

```
if ($line =~ m/^From: (\S+) \(([^()]*)\)/i) {
    $reply_address = $1;
    $from_name = $2;
}
```

Не все сообщения электронной почты содержат строку заголовка `Reply-To`, поэтому содержимое `$1` используется в качестве предварительного обратного адреса. Если позднее в заголовке найдется поле `Reply-To`, переменной `$reply_address` будет присвоено новое значение. В результате фрагмент обработки заголовка выглядит так:

```
while ($line = <>)
{
    if ($line =~ m/^\s*$/ ) {   # Если строка пустая...
      last; # Немедленно прервать цикл while
    }

    if ($line =~ m/^Subject: (.*)/i) {
        $subject = $1;
    }
    if ($line =~ m/^Date: (.*)/i) {
        $date = $1;
    }
    if ($line =~ m/^Reply-To: (\S+)/i) {
        $reply_address = $1;
    }
    if ($line =~ m/^From: (\S+) \(([^()]*)\)/i) {
        $reply_address = $1;
        $from_name = $2;
    }
}
```

Каждая строка заголовка проверяется по всем регулярным выражениям, и если она совпадает с одним из них, соответствующей переменной присваивается значение. Многие строки заголовка не совпадут ни с одним регулярным выражением и в итоге будут проигнорированы.

После завершения цикла `while` выводится заголовок ответа[1]:

```
print "To: $reply_address ($from_name)\n"
print "From: jfriedl\@regex.info (Jeffrey Friedl)\n";
print "Subject: Re: $subject\n";
print "\n" ; # Пустая строка, отделяющая заголовок
             # от основного текста сообщения
```

Обратите внимание на включение в тему префикса `Re:`, являющегося неформальным признаком ответа. Непосредственно перед выводом основного текста сообщения добавляется команда:

```
print "On $date $from_name wrote:\n";
```

Для всех остальных входных данных (основного текста сообщения) мы просто выводим каждую строку с добавлением префикса '`|>•`':

```
while ($line = <>) {
    print "|> $line";
}
```

[1] В строках, заключенных в кавычки, и регулярных выражениях Perl большинство символов @ должно экранироваться (☞ 112).

Символ новой строки здесь не нужен, поскольку переменная `$line` наследует его из входных данных.

Кстати говоря, команду вывода строки с префиксом цитирования можно записать с использованием регулярного выражения:

```
$line =~ s/^/|> /;
print $line;
```

Подстановка ищет совпадение выражения `^` и, конечно, находит его в начале строки. Впрочем, реальные символы при этом не совпадают, поэтому подстановка заменяет «ничто» в начале строки комбинацией '|>•', т. е. фактически вставляет '|>•' в начало строки. Такое экзотическое применение регулярных выражений в данном случае не оправдано, но в конце главы будет продемонстрирован схожий (и значительно более полезный) пример.

Реальные проблемы, реальные решения

Если уж мы рассматриваем реальный пример, вероятно, следует указать на его реальные недостатки. Во-первых, как упоминалось выше, примеры этой главы демонстрируют общие принципы использования регулярных выражений, а Perl — всего лишь удобное средство. Использованный код Perl не всегда является наиболее эффективным, но он достаточно наглядно показывает, как работать с регулярными выражениями.

Кроме того, в реальном мире такие простые задачи встречаются редко. Строка `From:` может кодироваться в нескольких разных форматах; в нашей программе обрабатывается лишь один частный случай. Если поле заголовка хоть частично расходится с шаблоном, переменной `$from_name` не присваивается значение и она остается неопределенной. Оптимальное решение заключается в модификации регулярного выражения и поддержке разных форматов адреса/имени, но в качестве предварительной меры после проверки исходного сообщения (и перед выводом шаблона) можно вставить следующий фрагмент:

```
if (   not defined($reply_address)
    or not defined($from_name)
    or not defined($subject)
    or not defined($date)
{
    die "couldn't glean the required information";
}
```

Функция Perl `defined` проверяет, имеет ли переменная определенное значение, а функция `die` выдает сообщение об ошибке и завершает программу.

Кроме того, наша программа предполагает, что строка `From:` расположена в заголовке перед строкой `Reply-To:`. Если строка `From:` окажется на втором месте, она сотрет адрес отправителя `$reply_address`, взятый из строки `ReplyTo:`.

Проблемы, проблемы...

Сообщения электронной почты генерируются разными программами, каждая из которых следует собственным представлениям о стандарте, поэтому обработка электронной почты может оказаться весьма непростой задачей. Как выяснилось при попытке запрограммировать некоторые действия на Pascal, без регулярных выражений эта задача становится невероятно сложной — настолько, что мне было проще написать на Pascal пакет для работы с регулярными выражениями в стиле Perl, чем пытаться сделать все непосредственно на Pascal. Я воспринимал силу и гибкость регулярных выражений как нечто привычное, пока не столкнулся с ситуацией, когда они вдруг стали недоступными. Не хотелось бы мне снова оказаться в подобной ситуации!

Разделение разрядов числа запятыми

Разделение групп разрядов в больших числах часто улучшает внешний вид отчетов. Команда

```
print "The US population is $pop\n";
```

выводит строку вида «The US population is 298444215», однако большинству читателей было бы удобнее работать со строкой «298,444,215». Как использовать регулярное выражение в этом случае?

При ручной расстановке запятых мы отсчитываем группы из трех цифр от правого края числа и вставляем запятую, если слева еще остаются цифры. Конечно, было бы удобно имитировать этот процесс при помощи регулярного выражения, но регулярные выражения обычно обрабатываются слева направо. Более формально постановка задачи выглядит так: запятые вставляются во всех позициях, у которых количество цифр справа кратно трем, а слева есть хотя бы одна цифра. Подобные задачи легко решаются при помощи относительно нового средства, называемого *позиционной проверкой* (lookaround).

Конструкции позиционной проверки обладают определенным сходством с метасимволами границ слов (`\b`, `^` и `$`) — они тоже совпадают не с символами, а с *позициями* в тексте. Однако позиционная проверка носит гораздо более общий характер, чем специализированные метасимволы границ слов.

Одна из разновидностей позиционной проверки — *опережающая проверка* — анализирует текст, расположенный справа, и проверяет возможность совпадения подвыражения. Если совпадение возможно, проверка считается успешной. Позитивная опережающая проверка задается специальной последовательностью ⌜`(?=…)`⌟. Например, подвыражение ⌜`(?=\d)`⌟ совпадает в тех позициях, за которыми следует цифра. Также существует *ретроспективная проверка*, при которой текст анализируется в обратном направлении (к левому краю). Ретроспективная проверка задается специальной последовательностью ⌜`(?<=…)`⌟. Например, подвыражение ⌜`(?<=\d)`⌟ совпадает в тех позициях, слева от которых находится цифра (т. е. в позиции после цифры).

Позиционная проверка не «поглощает» текст

Оба вида позиционной проверки (опережающая и ретроспективная) обладают одним важным свойством: хотя механизм регулярных выражений выполняет некоторые действия, проверяя совпадения их подвыражений, текст при этом не «поглощается». Возможно, это звучит не совсем внятно, поэтому я лучше приведу пример. Ниже выделено совпадение регулярного выражения ⌜`Jeffrey`⌟ в строке:

```
... by Jeffrey Friedl.
```

Однако в конструкции опережающей проверки то же самое выражение, ⌜`(?=Jeffrey)`⌟, совпадает только в отмеченной позиции:

```
... by ▲Jeffrey Friedl.
```

Опережающая проверка использует свое подвыражение для поиска, но находит только *позицию*, с которой начинается совпадение, а не фактически совпадающий *текст*. Впрочем, объединение опережающей проверки с выражением, совпадающим с символами текста (например, ⌜`Jeff`⌟), открывает дополнительные возможности. Объединенное выражение ⌜`(?=Jeffrey)Jeff`⌟, изображенное на рис. 2.5, совпадает с текстом «Jeff» только в том случае, если он является частью слова «Jeffrey». Выражение совпадет в строке

```
... by Jeffrey Friedl.
```

как совпало бы отдельное выражение ⌜`Jeff`⌟, но оно *не* совпадет в строке

```
... by Thomas Jefferson
```

Само по себе выражение ⌜`Jeff`⌟ совпало бы и в этой строке, но из-за отсутствия позиции, в которой могло бы совпасть выражение ⌜`(?=Jeffrey)`⌟, совпадение для объединенного выражения не находится. Не огорчайтесь, если вы пока не видите особой практической пользы от опережающей проверки. Сосредоточьте основное

внимание на механике ее работы — вскоре будут приведены реальные примеры, в которых эта польза проявляется более наглядно.

Стоит заметить, что выражение ⌜`(?=Jeffrey)Jeff`⌟ фактически эквивалентно выражению ⌜`Jeff(?=rey)`⌟. Оба выражения совпадают с текстом «Jeff» только в том случае, если он является частью слова «Jeffrey».

Обратите внимание и на важность порядка перечисления объединяемых подвыражений. Выражение ⌜`Jeff(?=Jeffrey)`⌟ не совпадает ни в одной из приведенных строк — оно совпадает только с текстом «Jeff», сразу же после которого следует «Jeffrey».

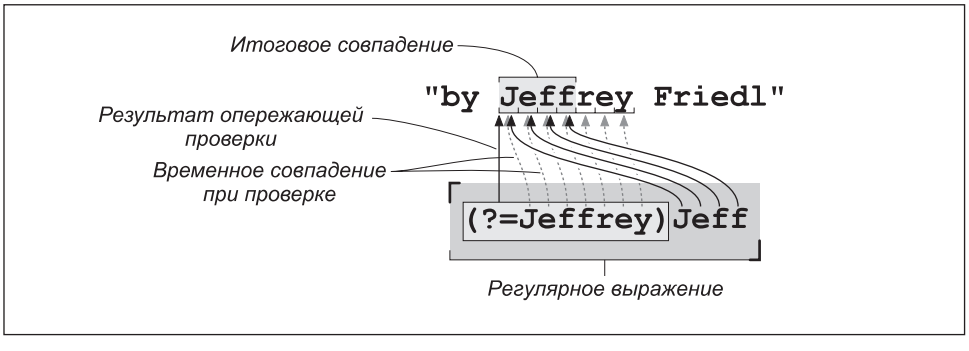


Рис. 2.5. Поиск совпадения для выражения ⌜`(?=Jeffrey)Jeff`⌟

Другое важное обстоятельство, относящееся к конструкциям позиционной проверки, — их странный внешний вид. В этих конструкциях, как и в несохраняющих круглых скобках «`(?:…)`» (о которых говорилось на с. 75), «открывающая скобка» представляет собой особую последовательность символов. Существует несколько разных последовательностей со специальной «открывающей скобкой», но все они начинаются с двух символов «`(?`». Символ, следующий за вопросительным знаком, определяет выполняемую функцию. Мы уже встречались с несохраняющими скобками «`(?:…)`», с конструкциями опережающей «`(?=…)`» и ретроспективной «`(?<=…)`» проверки.

Дополнительные примеры опережающей проверки

Вскоре мы вернемся к задаче разделения разрядов запятыми, а пока рассмотрим еще несколько примеров опережающей проверки. Начнем с включения апострофа в строку «Jeffs» и превращения ее в «Jeff's». Задача легко решается без применения опережающей проверки, простой командой **`s/Jeffs/Jeff's/g`** (модификатор глобальной замены `/g` упоминался на с. 82). Еще лучше привязать замену к границам слова при помощи метасимволов: **`s/\bJeffs\b/Jeff's/g`**.

В принципе, можно использовать и более экзотическую конструкцию типа `s/\b(Jeff)(s)\b/$1'$2/g`, но это выглядит чрезмерным усложнением простой задачи, и мы пока остановимся на выражении `s/\bJeffs\b/Jeff's/g`. Теперь сравните его со следующим выражением:

```
s/\bJeff(?=s\b)/Jeff'/g
```

Регулярное выражение почти не изменилось, просто завершающее подвыражение ⌜`s\b`⌟ теперь используется для опережающей проверки. На рис. 2.6 показано, как происходит поиск совпадения для этого регулярного выражения. В соответствии с изменениями в шаблоне поиска из строки замены исключен символ '`s`'.

После того как совпадение для ⌜`Jeff`⌟ будет найдено, выполняется опережающая проверка. Она успешна лишь в том случае, если конструкция ⌜`s\b`⌟ совпадает в данной позиции (т. е. за '`Jeff`' следует символ '`s`' и граница слова). Но так как ⌜`s\b`⌟ входит в подвыражение, участвующее в опережающей проверке, обнаруженный символ '`s`' не считается частью окончательного совпадения. Вспомните: ⌜`Jeff`⌟ совпадает с текстом, а подвыражение опережающей проверки всего лишь «выбирает» позицию в строке. Следовательно, опережающая проверка в данном случае служит только одной цели — отсеять совпадения первой части регулярного выражения при несовпадении подвыражения опережающей проверки. Или — другое объяснение — опережающая проверка позволяет проверить наличие всей строки ⌜`Jeffs`⌟, но включить в совпадение только ⌜`Jeff`⌟.

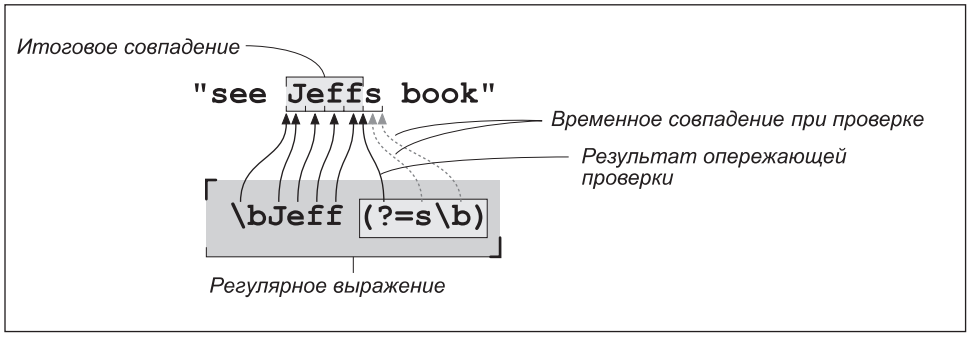


Рис. 2.6. Поиск совпадения для выражения ⌜`\bJeff(?=s\b)`⌟

Почему в совпадение включается меньше символов, чем было фактически проверено? Обычно потому, что этот текст проверяется повторно какой-то из следующих частей регулярного выражения. Соответствующий пример с разделением разрядов в числах встретится через несколько страниц. В текущем примере причина другая: мы проверяем наличие всего текста ⌜`Jeffs`⌟, потому что апостроф должен включаться именно в этот текст, но фактическое совпадение ограничивается строкой

‘Jeff’, чтобы уменьшить длину строки замены. Поскольку символ ‘s’ не входит в совпадение, его не нужно заменять. Вот почему этот символ был исключен из строки замены.

Итак, несмотря на различия в регулярных выражениях и строках замены, результат остается одним и тем же. Возможно, вся эта акробатика с регулярными выражениями отдает теорией, но я преследую определенную цель. Давайте сделаем следующий шаг.

При переходе от первого примера ко второму завершающий символ ⌜s⌟ был перемещен из «основного» регулярного выражения в условие опережающей проверки. А что если сделать нечто подобное с начальным текстом ⌜Jeff⌟ и перенести его в условие *ретроспективной* проверки? Получится выражение ⌜(?<=\bJeff)(?=s\b)⌟ (рис. 2.7), которое читается так: «Найти позицию, перед которой находится текст ‘Jeff’, а после которой находится текст ‘s’.». Данная формулировка точно описывает позицию, в которой вставляется апостроф. Итак, используя это выражение в команде подстановки, мы получаем:

```
s/(?<=\bJeff)(?=s\b)/'/g
```

Результат получается весьма любопытным. Регулярное выражение вообще не совпадает ни с каким текстом — оно совпадает с позицией, в которой мы хотим вставить апостроф. Найденное «ничто» в этой позиции заменяется апострофом. Нечто похожее встречалось буквально несколько страниц назад, когда мы использовали выражение s/^/|>•/ для включения в строку префикса ‘|>•’.

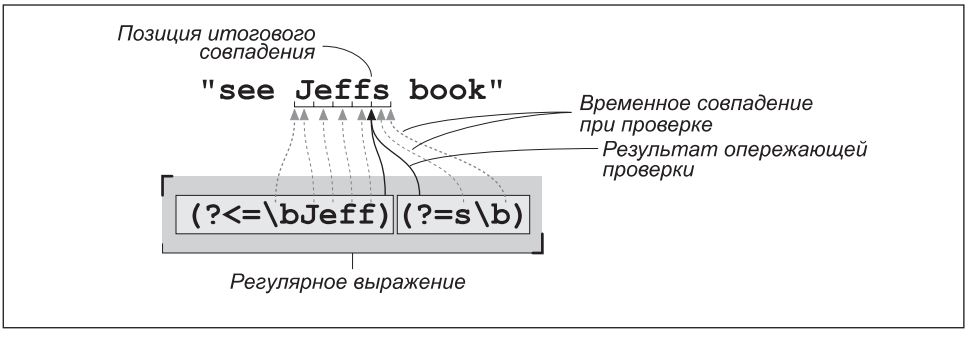


Рис. 2.7. Поиск совпадения для выражения ⌜(?<=\bJeff)(?=s\b)⌟

Изменится ли смысл выражения, если изменить порядок следования двух позиционных проверок конструкций? Другими словами, к какому результату приведет команда **s/(?=s\b)(?<=\bJeff)/'/g**? ❖ Подумайте над этим вопросом, затем переверните страницу и проверьте свой ответ.

ОТВЕТ НА ВОПРОС

❖ *Ответ на вопрос со с. 95.*

Что получится при изменении порядка следования проверок?

В данном примере порядок ⌜`(?=s\b)`⌟ и ⌜`(?<=\bJeff)`⌟ роли не играет. Независимо от того, как происходит проверка (сначала справа, потом слева или наоборот), комбинация двух проверок совпадет только в том случае, если обе проверки в некоторой позиции дают положительный результат. Например, в строке '`Thoma␣s•Jeff␣erson`' оба выражения, ⌜`(?=s\b)`⌟ и ⌜`(?<=\bJeff)`⌟, смогут отыскать совпадения (в двух отмеченных позициях), но поскольку эти позиции не совпадают, комбинация двух проверок даст отрицательный результат.

На данном этапе вполне оправданно употреблять выражение «комбинация двух проверок», поскольку смысл ее достаточно понятен. Однако в других случаях понимание того, как механизм регулярных выражений отыскивает совпадение, может быть не столь очевидным. В главе 4 подробно рассматривается, как этот механизм работает и какое влияние в действительности это оказывает на смысл наших регулярных выражений.

Подведем итог. В табл. 2.1 перечислены рассмотренные нами способы замены строки `Jeffs` строкой `Jeff's`.

Таблица 2.1. Решения задачи «Jeffs»

Решение	Комментарии
`s/\bJeffs\b/Jeff's/g`	Самое простое, прямолинейное и эффективное решение; я бы остановился на нем, если бы не желание продемонстрировать другие интересные подходы к решению задачи. Регулярное выражение просто совпадает с текстом «`Jeffs`» без всяких дополнительных проверок
`s/\b(Jeff)(s)\b/$1'$2/g`	Лишние сложности. Выражение также совпадает с текстом «`Jeffs`»
`s/\bJeff(?=s\b)/Jeff'/g`	Символ «`s`» не включается в совпадение, но в данном случае этот пример не имеет практической ценности и приводится лишь для демонстрации позиционной проверки
`s/(?<=\bJeff)(?=s\b)/'/g`	Регулярное выражение вообще не совпадает с символами текста. Оно использует опережающую и ретроспективную проверку для выделения *позиции*, в которую нужно вставить апостроф. Чрезвычайно полезно для демонстрации возможностей позиционной проверки

Решение	Комментарии
`s/(?=s\b)(?<=\bJeff)/'/g`	В точности совпадает с предыдущим выражением, но две проверки меняются местами. Поскольку подвыражения не совпадают с символами текста, порядок их применения не влияет на наличие или отсутствие совпадения

Прежде чем вернуться к примеру с разделением разрядов запятыми, позвольте задать еще один вопрос. Предположим, мы хотим искать текст «Jeffs» без учета регистра символов, но сохранить исходный регистр после вставки апострофа. Какие из перечисленных выражений позволяют добиться желаемого результата простым добавлением модификатора `/i`? Подсказка: все, кроме двух. ❖ Подумайте над вопросом, затем переверните страницу и проверьте свой ответ.

Возвращаемся к задаче вставки запятых

Наверное, вы уже поняли, что пример «Jeffs» имеет нечто общее с задачей разделения разрядов: в обоих случаях мы хотим вставить нечто в *позицию*, описываемую регулярным выражением.

Формальная постановка задачи уже приводилась выше: «вставить запятые во всех позициях, у которых количество цифр справа кратно трем, а слева есть хотя бы одна цифра». Второе требование относительно легко выполняется при помощи ретроспективной проверки. Одной цифры слева достаточно для выполнения этого требования, а этот критерий задается выражением ⌜`(?<=\d)`⌟.

Переходим к условию «количество цифр справа кратно трем». Группа из трех цифр определяется выражением ⌜`\d\d\d`⌟. Заключим ее в конструкцию ⌜`(...)+`⌟, чтобы совпадение могло состоять из нескольких групп, и завершим метасимволом ⌜`$`⌟, чтобы гарантировать отсутствие символов после совпадения. Само по себе выражение ⌜`(\d\d\d)+$`⌟ совпадает с группами из трех цифр, следующими до конца строки, но в конструкции опережающей проверки ⌜`(?=...)`⌟ оно совпадает с *позицией*, справа от которой до конца строки следуют группы из трех цифр — например, в отмеченных позициях текста '␣123␣456␣789'. Однако перед первой цифрой запятая не ставится, поэтому совпадение дополнительно ограничивается ретроспективной проверкой ⌜`(?<=\d)`⌟.

Фрагмент

```
$pop =~ s/(?<=\d)(?=(\d\d\d)+$)/,/g;
print "The US population is $pop\n";
```

выводит строку «The US population is 298,444,215», как и предполагалось. На первый взгляд кажется странным, что конструкция ⌜`\d\d\d`⌟ заключена в сохраняющие

ОТВЕТ НА ВОПРОС

❖ *Ответ на вопрос со с. 97.*

Какие из выражений позволяют добиться сохранения регистра символов в задаче «Jeffs» простым добавлением модификатора `/i`?

Чтобы сохранить регистр символов неизменным, необходимо замещать только «поглощаемые» символы (т. е. не просто вставлять текст '`Jeff's`') или вообще не «поглощать» никакие символы. Первый вариант реализует второе решение из табл. 2.1, оно сохраняет найденный текст в переменных `$1` и `$2` и затем восстанавливает его. Последние два решения из указанной таблицы реализуют второй подход, «не поглощая ни одного символа». Так как эти решения не поглощают текст, в них отсутствует потребность его восстановления.

Первое и третье решения просто вставляют жестко определенную строку замены. Они не позволяют сохранить регистр символов при использовании модификатора `/i`. В результате строка `JEFFS` будет замещена строками `Jeff's` и `Jeff'S` соответственно.

круглые скобки. В данном случае скобки предназначены только для группировки трех цифр, а сохранение текста в переменной `$1` программа не использует.

Вместо простых скобок также можно использовать `(?:…)` — несохраняющие скобки, о которых упоминалось во врезке на с. 75. В результате регулярное выражение принимает вид `(?<=\d)(?=(?:\d\d\d)+$)`. Такой вариант более точен — если позднее кто-нибудь будет читать этот фрагмент, ему не придется размышлять над тем, где используется значение переменной `$1`, ассоциированной с круглыми скобками. Он также чуть более эффективен, поскольку подсистеме обработки регулярных выражений не нужно сохранять совпавший текст. С другой стороны, даже с обычными круглыми скобками `(…)` выражение никак не назовешь понятным, а конструкция `(?:…)` его только усложняет. На этот раз я выбрал более простой вариант. Это стандартный компромисс, постоянно возникающий при работе с регулярными выражениями. Лично я обычно использую `(?:…)` там, где это уместно, но отдаю предпочтение наглядности там, где пытаюсь что-то объяснить или продемонстрировать (как в большинстве примеров книги).

Границы слов и негативные проверки

Предположим, мы хотим усовершенствовать это выражение для разделения разрядов в числах, находящихся внутри строк. Пример:

```
$text = "The population of 298444215 is growing";
...
$text =~ s/(?<=\d)(?=(\d\d\d)+$)/./g;
print "$text\n";
```

В этом варианте выражение не работает, поскольку метасимвол ⌜$⌟ требует, чтобы группы из трех цифр следовали до конца строки. Его нельзя просто убрать, потому что тогда запятые вставляются во всех позициях, у которых слева имеется цифра, а справа — хотя бы одна группа из трех цифр; получится строка вида «...of 2,9,8,4,4,4,215 is...».

А что если заменить ⌜$⌟ метасимволом границы слова ⌜\b⌟? Хотя мы работаем с числовыми данными, концепция «слов» языка Perl нам поможет. Как упоминалось в комментариях к метасимволу ⌜\w⌟ (☞ 79), в Perl и в большинстве других программ считается, что слова состоят из алфавитно-цифровых символов и символов подчеркивания. Следовательно, любая позиция, слева от которой находится один из перечисленных символов (например, последняя цифра), а справа — другой символ (например, конец строки или пробел после числа), считается границей слова.

Конструкция «позиция, у которой слева то, а справа это» выглядит знакомо, не правда ли? Мы уже встречались с ней в примере «Jeffs». Единственное различие состоит в том, что на этот раз выражение справа должно не совпадать с заданными символами. Оказывается, то, что до настоящего момента называлось «опережающей проверкой» и «ретроспективной проверкой», правильнее было бы называть *позитивной опережающей проверкой* и *позитивной ретроспективной проверкой*, потому что заданное условие должно выполняться. Как видно из табл. 2.2, у этих проверок также существуют антиподы — негативная опережающая проверка и негативная ретроспективная проверка. Как подсказывает само название, проверка считается успешной *при отсутствии* совпадения для заданного подвыражения.

Таблица 2.2. Четыре разновидности позиционных проверок

Тип	Регулярное выражение	Успешна, если подвыражение...
Позитивная ретроспективная проверка	(?<=...)	*Может* совпасть *слева*
Негативная ретроспективная проверка	(?<!...)	*Не может* совпасть *слева*
Позитивная опережающая проверка	(?=...)	*Может* совпасть *справа*
Негативная опережающая проверка	(?!...)	*Не может* совпасть *справа*

Итак, если граница слова находится в позиции, слева от которой находится символ ⌜\w⌟, а справа — символ, не совпадающий с ⌜\w⌟, то позиция начала слова определяется выражением ⌜(?<!\w)(?=\w)⌟, а позиция конца слова — аналогичным выражением ⌜(?<=\w)(?!\w)⌟. Объединив эти два выражения, мы могли бы использовать ⌜(?<!\w)(?=\w)|(?<=\w)(?!\w)⌟ для замены ⌜\b⌟. На практике было бы неразумно прибегать к подобным заменам в языках с непосредственной поддержкой метасимвола ⌜\b⌟, гораздо более наглядного и эффективного, но каждый из вариантов может принести пользу в определенной ситуации (☞ 180).

Впрочем, для нашего примера достаточно выражения ⌜(?!\d)⌟, означающего отсутствие цифр справа. Подставляя его вместо ⌜\b⌟ или ⌜$⌟, мы получаем:

```
$text =~ s/(?<=\d)(?=(\d\d\d)+(?!\d))/,/g;
```

Обновленный вариант нормально работает с текстом вида «...tone of 12345Hz», и это хорошо. К сожалению, он также сработает в строке «...the 1970s...». Более того, все рассмотренные выражения найдут год в строке «...in 1970...», что совсем нежелательно. Вы должны хорошо знать данные, к которым применяется регулярное выражение, и решить, когда это выражение уместно применить (а если в данных присутствуют годы, такое регулярное выражение явно неуместно).

При определении границ по отсутствию совпадений для указанных символов мы использовали негативную опережающую проверку, ⌜(?!\w)⌟ и ⌜(?!\d)⌟. Возможно, вы помните метасимвол ⌜\D⌟ («символ, не являющийся цифрой» — с. 79) и полагаете, что вместо ⌜(?!\d)⌟ можно было использовать его. Такое решение было бы ошибочным. В определении «символ, не являющийся цифрой» обязательно присутствует символ, просто он не является цифрой. Если после цифры в тексте нет *ничего*, ⌜\D⌟ не совпадет (аналогичная ситуация упоминается во врезке на с. 38).

Разделение разрядов без ретроспективной проверки

Ретроспективная проверка поддерживается (да и используется) значительно реже, чем опережающая проверка. Опережающая проверка появилась в мире регулярных выражений за несколько лет до появления ретроспективной, и хотя в Perl поддерживаются оба варианта, многие другие языки ограничиваются только опережающей проверкой. По этой причине было бы полезно узнать, как решить проблему с разделением разрядов без ретроспективной проверки. Пример:

```
$text =~ s/(\d)(?=(\d\d\d)+(?!\d))/$1,/g;
```

От предыдущего примера этот отличается тем, что конструкция позитивной ретроспективной проверки, в которую был заключен начальный символ ⌜\d⌟, заменена сохраняющими круглыми скобками, а в строку замены перед запятой включена соответствующая переменная $1.

А если опережающая проверка тоже не поддерживается? Мы можем заменить ⌜(?!\d)⌟ на ⌜\b⌟, но будет ли методика, использованная для исключения ретроспективной проверки, работать с опережающей проверкой? Другими словами, подойдет ли следующая команда:

```
$text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g;
```

❖ Подумайте над этим вопросом, а затем проверьте свой ответ на с. 102.

Преобразование текста в HTML

Давайте напишем простую программу для преобразования простого текста в HTML. Разработка универсальной утилиты, которая бы могла использоваться в любой ситуации, — довольно сложная задача, поэтому в этом разделе мы ограничимся простейшим примером, в первую очередь предназначенным для учебных целей.

Во всех примерах, встречавшихся до настоящего момента, регулярные выражения применялись к переменным, содержащим ровно одну логическую строку текста. В новом примере будет проще (и интереснее) считать, что весь преобразуемый текст содержится в одной большой строке. На Perl подобные строки легко создаются следующей конструкцией:

```
undef $/;   # Перейти в режим поглощения файла
$text = <>; # Полностью загрузить первый файл, указанный в командной строке
```

Допустим, файл примера состоит из трех коротких строк:

```
This is a sample line.
It has three lines.
That's all
```

В этом случае переменная `$text` будет содержать следующий текст:

```
This is a sample file.NLIt has three lines.NLThat's allNL
```

Хотя в некоторых системах текст в переменной может иметь следующий вид:

```
This is a sample file.CRNLIt has three lines.CRNLThat's allCRNL
```

В большинстве систем логические строки завершаются символом новой строки, но в некоторых системах (прежде всего в Windows) используется комбинация «возврат каретки+новая строка». Мы должны позаботиться о том, чтобы наша простая программа работала в обоих случаях.

Специальные символы

Прежде всего необходимо защитить символы `&`, `<` и `>` в исходном тексте, заменив их соответствующими кодами HTML: `&`, `<` и `>`. Эти символы в HTML имеют особый смысл, поэтому без кодировки возникнут проблемы с выводом информации. Я называю это простое преобразование «подготовкой текста для HTML»:

```
$text =~ s/&/&amp;/g;  # Защитить основные символы HTML
$text =~ s/</&lt;/g;   #   ... &, < и >
$text =~ s/>/&gt;/g;   # их безопасными сущностями HTML
```

ОТВЕТ НА ВОПРОС

❖ *Ответ на вопрос со с. 100.*

Подойдет ли команда **`$text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g`** для разделения разрядов числа?

Нет, команда будет работать не так, как нужно, — она будет оставлять результаты вида «298,444215». Дело в том, что цифры, совпавшие с подвыражением `「(\d\d\d)+」`, в этом варианте являются частью последнего совпадения, поэтому они становятся недоступными для следующей итерации регулярного выражения, обусловленной модификатором `/g`.

Очередная итерация продолжает анализ текста с той позиции, где завершилась предыдущая итерация. Нам бы хотелось, чтобы это была позиция вставки очередной запятой, чтобы мы могли продолжить поиск позиций числа, в которых можно вставить дополнительные запятые. Но в данном случае следующая итерация начнется с позиции, расположенной за последней цифрой в числе. Опережающая проверка использовалась как раз для того, чтобы проверенные символы не включались в совпавший текст.

На самом деле это выражение все-таки может использоваться для решения задачи. Если выражение многократно применяется средствами языка (например, в цикле `while`), модифицированный текст каждый раз обрабатывается заново. В этом случае при каждой итерации добавляется новая запятая (к каждому числу в строке из-за наличия модификатора `/g`). Пример:

```
while ( $text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g ) {
    # Внутри цикла ничего не происходит
    # мы просто применяем регулярное выражение,
    # пока для него находятся совпадения.
}
```

При замене используется модификатор `/g`, обеспечивающий преобразование всех символов в целевом тексте (без него замена ограничилась бы первым вхождением каждого символа в строку). Преобразование должно начинаться с **`&`**, поскольку этот символ присутствует в строке замены во всех трех командах.

Разделение абзацев

После замены специальных символов абзацы помечаются тегом разделения абзацев HTML `<p>`. Будем считать, что конец абзаца обозначается пустой строкой. Существует несколько способов идентификации пустой строки. Сначала может возникнуть идея использовать конструкцию вида

```
$text =~ s/^$/<p>/g;
```

Фактически условие поиска означает: «Найти позицию начала строки, сразу же после которой следует позиция конца строки». Действительно, как было показано в ответе на с. 34, подобное решение годится для программ типа *egrep*, которые всегда работают с текстом, состоящим из одной логической строки. Это решение подойдет и для Perl в контексте предыдущего примера с электронной почтой, в котором было точно известно, что каждый фрагмент текста содержит ровно одну логическую строку.

Но, как упоминалось в сноске на с. 86, метасимволы ⌜^⌟ и ⌜$⌟ обычно относятся не к *логическим строкам*, а к абсолютным позициям начала и конца рассматриваемого *текста*[1]. Следовательно, если целевая строка содержит несколько логических строк, придется поискать другое решение.

К счастью, в большинстве языков с поддержкой регулярных выражений существует простое решение — *расширенный режим привязки*, в котором метасимволы ⌜^⌟ и ⌜$⌟ применяются именно к *логическим строкам*, что и требуется в нашем примере. В языке Perl этот режим активизируется модификатором `/m`:

```
$text =~ s/^$/<p>/mg;
```

Обратите внимание на слияние модификаторов `/m` и `/g` (при указании нескольких модификаторов вы можете объединять их в произвольном порядке). Использование аналогичных модификаторов в других языках будет рассмотрено в следующей главе.

Таким образом, если команда применяется к исходному тексту «`...chapter.`␤␤`Thus`», мы получим желаемый результат «`...chapter.`␤`<p>`␤`Thus...`».

Однако такое решение не работает в том случае, если «пустая» строка содержит пробелы или другие пропускные символы. Выражение ⌜^•*$⌟ учитывает возможные пробелы, а команда ⌜^[•\t\r]*$⌟ — пробелы, символы табуляции и символы возврата каретки, вставляемые в некоторых системах перед символом новой строки. Эти выражения принципиально отличаются от ⌜^$⌟ тем, что они совпадают с *символами* текста, тогда как ⌜^$⌟ совпадает только с *позицией*. Но в данном примере все эти пробелы, символы табуляции и возврата каретки нам не нужны, поэтому их можно включить в совпадение (а затем заменить абзацным тегом).

Если вы помните о существовании метасимвола ⌜\s⌟ (☞ 76), возникает желание использовать выражение ⌜^\s*$⌟, как было сделано в примере с электронной почтой на с. 85. Если заменить ⌜[•\t\r]⌟ на ⌜\s⌟, возможность совпадения ⌜\s⌟ с символом новой строки изменяет общий смысл выражения: вместо «найти *строки*, не со-

[1] На самом деле ⌜$⌟ интерпретируется несколько сложнее, чем простой «конец строки», хотя в контексте нашего примера это несущественно. Дополнительная информация приведена в описании на с. 175.

держащие ничего, кроме пропусков» оно означает «найти *группы строк*, не содержащих ничего, кроме пропусков». Если в тексте идут несколько пустых строк подряд, выражение `⌜^\s*$⌟` совпадет сразу со всеми. Впрочем, для нас это даже хорошо — замена оставит один тег `<p>` вместо нескольких подряд, как получилось бы с исходным выражением.

Итак, если переменная `$text` содержит строку

```
with.¶¶¶⇥• ¶Therefore
```

и мы применяем к ней команду

```
$text =~ s/^[ \t\r]*$/<p>/mg;
```

результат будет выглядеть так:

```
with.¶<p>¶<p>¶<p>¶Therefore
```

Но если использовать команду

```
$text =~ s/^\s*$/<p>/mg;
```

мы получим более желательный результат:

```
with.¶<p>¶Therefore
```

По этой причине в окончательной версии программы будет использовано выражение `⌜^\s*$⌟`.

Преобразование адресов в ссылки

Следующий этап преобразования текста в HTML заключается в идентификации адресов электронной почты и их преобразовании в ссылки «mailto». Таким образом, фрагмент «jfriedl@oreilly.com» превращается в `<a•href="mailto:jfriedl@oreilly.com>jfriedl@oreilly.com</a>`.

Поиск и проверку почтового адреса было бы вполне естественно проводить при помощи регулярного выражения. Официальная спецификация адреса достаточно сложна; вместо ее точной реализации мы воспользуемся упрощенным синтаксисом, который будет работать с большинством обычных адресов. В базовом варианте адрес электронной почты имеет формат *имя_пользователя@имя_хоста*. Прежде чем рассматривать регулярные выражения для идентификации обеих составляющих, мы сначала обсудим контекст, в котором они будут использоваться:

```
$text =~ s/\b(имя_пользователя\@имя_хоста)\b/<a href="mailto:$1">$1<\/a>/g;
```

Обратите внимание на два подчеркнутых символа \ — в регулярном выражении и в начале тега . Эти символы выполняют разные функции. Рассмотрение \@ откладывается на будущее (☞ 112), а пока достаточно сказать, что по правилам Perl литералы @ в регулярных выражениях должны экранироваться.

На этой стадии проще объяснить роль символа '\' перед символом '/' в строке замены. Вы уже знаете, что базовая форма поиска и замены в Perl имеет формат s/*выражение*/*замена*/*модификаторы*, а компоненты команды разделяются символом /. Если символ / входит в какой-либо из компонентов, он должен экранироваться, чтобы Perl интерпретировал его не как разделитель, а как часть регулярного выражения или строки замены. Следовательно, если в строке замены присутствует текст , он должен быть представлен в виде <\/a>, что и сделано в приведенном примере.

Такое решение работает, но выглядит громоздко, поэтому Perl позволяет использовать в команде нестандартные ограничители — например, s!*выражение*!*замена*!*модификаторы* или s{*выражение*}{*замена*}*модификаторы*. В этих случаях символ / в строке замены не конфликтует с ограничителями, поэтому экранировать его не нужно. Во втором примере используются удобные парные ограничители, поэтому в дальнейшем я буду использовать эту форму.

Но вернемся к приведенной выше команде. Обратите внимание — вся адресная часть заключена между метасимволами ⌈\b…\b⌋. Это сделано для того, чтобы предотвратить внутренние совпадения вида '*jfriedl@oreilly.com*piler'. Подобные ошибки маловероятны, но защититься от них привязкой к границам слов совсем несложно, поэтому я буду использовать этот способ. Также обратите внимание на то, что весь адрес заключен в круглые скобки. Это сделано для того, чтобы сохранить совпавший адрес и включить его в строку замены: '<a•href="mailto:$1">$1'.

Поиск имен пользователя и хоста

Перейдем к непосредственному поиску адреса электронной почты. Для этого необходимо построить регулярные выражения, совпадающие с компонентами *имя_пользователя* и *имя_хоста*. Имена хостов (такие, как regex.info или www.oreilly.com) состоят из групп букв, разделенных запятыми и заканчивающихся стандартными суффиксами 'com', 'edu', 'info', 'uk' и т. д. Простейшее выражение для адреса электронной почты имеет вид ⌈\w+\@\w+(\.\w+)+⌋, при этом имя пользователя и каждая составляющая имени хоста представляется подвыражением ⌈\w+⌋. Впрочем, на практике этого может оказаться недостаточно. В именах пользователей иногда встречаются точки и дефисы (хотя имена пользователей, начинающиеся с этих символов, встречаются крайне редко), поэтому вместо ⌈\w+⌋ лучше использовать ⌈\w[-.\w]*⌋. Выражение требует, чтобы имя начиналось с символа ⌈\w⌋, но внутри имени также могут встречаться точки и дефисы. Обратите внимание: дефис

поставлен на первом месте в символьном классе, чтобы он интерпретировался как литерал, а не как часть интервальной конструкции типа `a-z`. Символьный класс `.-\w` почти всегда дает ошибочный результат и состоит из произвольного набора букв, цифр и специальных символов в зависимости от программы и кодировки, используемой компьютером.

С именем хоста дело обстоит несколько сложнее. Точки в данном случае являются «жесткими» разделителями, а это означает, что между ними должны находиться разделяемые символы. Именно по этой причине даже в упрощенной версии, приведенной выше, имя хоста представляется выражением ⌜`\w+(\.\w+)+`⌟, а не ⌜`[\w.]+`⌟, которое бы ошибочно совпадало в строке '`..x..`'. Но даже первое выражение совпадает в строке '`Artichokes 4@1.00`', поэтому мы должны быть еще точнее.

Одно из решений заключается в явном перечислении всех возможных значений последнего компонента — например, ⌜**`\w+(\.\w+)*\.(com|edu|info)`**⌟. Полный список альтернатив в действительности имеет вид **`com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z]`**, но мы будем использовать укороченный вариант, чтобы не загромождать пример. Совпадение начинается с ⌜`\w+`⌟, затем следуют дополнительные необязательные компоненты ⌜`\.\w+`⌟, а в конце находится один из стандартных завершителей, входящих в список.

Вообще говоря, метасимвол ⌜`\w`⌟ не совсем подходит для данного случая. Он совпадает с цифрами и буквами ASCII, что нас устраивает, но в некоторых системах он также может допускать буквы, не входящие в набор ASCII (à, ç, Ξ, Æ и т. д.), а в некоторых диалектах — и символы подчеркивания. Все эти дополнительные символы запрещены в именах хостов. Следовательно, вместо ⌜`\w`⌟ правильнее было бы использовать ⌜`[a-zA-Z0-9]`⌟ или ⌜`[a-z0-9]`⌟ с модификатором `/i` (поиск совпадения без учета регистра). Имена хостов также могут содержать дефисы, поэтому символьный класс принимает вид ⌜`[-a-z0-9]`⌟ (еще раз напомню, что дефис должен стоять на первом месте). Итак, имя хоста представляется выражением ⌜`[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)`⌟.

В данном примере, как и во всех примерах регулярных выражений, необходимо учитывать контекст использования. Например, само по себе выражение ⌜`[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)`⌟ может совпасть в строке '`run C:\\startup.command at startup`', однако в контексте нашей программы можно быть уверенным в том, что оно совпадет там, где нужно, и не совпадет в других местах. В сущности, его можно было бы прямо сейчас включить в команду

```
$text =~ s{\b(имя_пользователя\@имя_хоста)\b}{<a href="mailto:$1">$1</a>}gi;
```

(эта команда приводилась выше, но здесь в нее включены фигурные скобки и модификатор `/i`), но тогда команда никак не поместится в одной строке. Конечно, Perl абсолютно безразличен к эстетике программы, но у меня на этот счет другое

мнение! Позвольте представить модификатор /x, который позволяет записать регулярное выражение в виде:

```
$text =~ s{
    \b
    # Сохранить адрес в $1...
    (
       имя_пользователя
       \@
       имя_хоста
    )
    \b
}{<a href="mailto:$1">$1</a>}gix;
```

Совсем другое дело! Модификатор /x завершает этот фрагмент (вместе с модификаторами /g и /i) и обеспечивает две простые, но чрезвычайно полезные возможности. Во-первых, «посторонние» пропуски игнорируются, что позволяет представить регулярное выражение в формате, удобном для чтения. Во-вторых, в регулярное выражение могут включаться комментарии, начинающиеся с префикса #.

Выражаясь точнее, модификатор /x превращает пропуски в игнорируемые метасимволы, а символ # — в метасимвол, который означает: «игнорируйте меня и весь текст до следующего символа новой строки» (☞ 151). Они не воспринимаются как метасимволы в символьных классах (отсюда следует, что символьные классы *не могут* свободно форматироваться даже с модификатором /x), причем пропуски и # можно экранировать, как любой другой метасимвол, чтобы они интерпретировались буквально. Конечно, для представления пропусков можно использовать метасимвол ⌜\s⌟, как в команде **m/<a \s+ href=…>/x**.

Вы должны четко понять, что действие модификатора /x распространяется только на регулярное выражение, а не на строку замены. Даже несмотря на то, что мы выбрали синтаксическую форму s{…}{…}, в которой модификаторы следуют после завершающего символа '}' (например, }x), в тексте модификатор x все равно будет именоваться «модификатором /x».

Все вместе

Итак, теперь мы можем объединить регулярные выражения для имени пользователя и имени хоста с синтаксическими конструкциями, описанными выше. Получается следующая программа:

```
undef $/:   # Перейти в режим «поглощения файла»
$text = <>; # Полностью загрузить первый файл, указанный в командной строке

$text =~ s/&/&amp;/g;  # Защитить основные символы HTML
$text =~ s/</&lt;/g;   #   ... &, < и >
```

```
$text =~ s/>/&gt;/g;   # их безопасными сущностями HTML

$text =~ s/^\s*$/<p>/mg;  # Разделить абзацы

# Преобразование адресов электронной почты в ссылки
$text =~ s{
    \b
    # Сохранить адрес в $1...
    (
       \w[-.\w]*                                  # Имя пользователя
       \@
       [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)  # Имя хоста
    )
    \b
}{<a href="mailto:$1">$1</a>}gix;

print $text; # Вывод преобразованного текста
```

Все регулярные выражения работают с текстом, объединенным в одну длинную строку, но обратите внимание: модификатор `/m` необходим только в команде замены для разделения абзацев, поскольку в ее выражении присутствуют метасимволы `^` и `$`. Впрочем, в других командах этот модификатор не принесет вреда (разве что вызовет некоторое недоумение у того, кто будет разбираться в программе).

Преобразование HTTP URL в ссылки

Остается выделить в строке обычные HTTP URL и преобразовать их в эквивалентные ссылки. При этом текст вида «*http://www.yahoo.com*» преобразуется в ссылку `<a•href="http://www.yahoo.com/">http://www.yahoo.com</a>`.

Простейший HTTP URL имеет формат «`http://`*хост*`/`*путь*», причем `/`*путь* не является обязательным. Следовательно, общая конструкция выглядит так:

```
$text =~ s{
    \b
    # Сохранить URL в $1...
    (
       http://хост
       (
            / путь
       )?
    )
}{<a href="$1">$1</a>}gix;
```

Для идентификации хоста можно воспользоваться тем же подвыражением, которое было использовано для идентификации адреса электронной почты. Путь может содержать различные символы; в предыдущей главе использовалось выражение

`[-a-z0-9_:@&?=+,.!/~*'%$]*` (☞ 52), включающее большинство символов ASCII, кроме пропусков, управляющих и некоторых других символов (`<>( ) { }` и т. д.).

Но перед тем как использовать это выражение в Perl, мы должны выполнить еще одну операцию — экранирование символов `@` и `$`. Подробные объяснения снова откладываются на будущее (☞ 112). Подставляя выражения для хоста и пути, получаем:

```
$text =~ s{
    \b
    # Сохранить URL в $1...
    (
       http://[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info) \b  # Хост
       (
            / [-a-z0-9_:@&?=+,.!/~*'%$]*      # Необязательный путь
       )?
    )
}{<a href="$1">$1</a>}gix;
```

Обратите внимание на отсутствие `\b` после пути — URL может заканчиваться знаком препинания:

http://www.oreilly.com/catalog/regex3/

Наличие `\b` привело бы к тому, что такие адреса считались бы недопустимыми.

Однако на практике, вероятно, стоит установить искусственные ограничения на возможные завершения URL. Рассмотрим следующий текст:

```
Read "odd" news at http://dailynews.yahoo.com/h/od, and
maybe some tech stuff at http://www.slashdot.com!
```

Текущая версия регулярного выражения совпадает с помеченным текстом, хотя совершенно очевидно, что конечные знаки препинания не должны входить в URL. При поиске совпадения для URL в английском тексте было бы логично не включать в совпадение завершающие символы `.,?!` (данное требование не входит в стандарт, однако это эвристическое правило работает в абсолютном большинстве случаев). Задача решается простым включением негативной ретроспективной проверки «не является ни одним из символов `.,?!`» (т. е. `?<![.,?!])` в конце подвыражения пути). Итак, после обнаружения потенциального совпадения для URL ретроспективная проверка «оглядывается назад» и убеждается в том, что последний символ не принадлежит к числу запрещенных. Если условие не выполняется, механизм регулярных выражений должен заново вычислить совпадение для URL, чтобы условие соблюдалось. Таким образом, знак-нарушитель исключается из совпадения, чтобы не нарушалось условие ретроспективной проверки (другой способ решения этой проблемы приведен в главе 5, ☞ 267).

После вставки нового фрагмента мы получаем следующую программу:

```
undef $/;   # Перейти в режим поглощения файла
$text = <>; # Полностью загрузить первый файл, указанный в командной строке

$text =~ s/&/&amp;/g;     # Заменить основные символы HTML...
$text =~ s/</&lt;/g;      # &, < и >
$text =~ s/>/&gt;/g;      # их безопасными сущностями HTML

$text =~ s/^\s*$/<p>/mg;  # Разделить абзацы

# Преобразовать адреса электронной почты в ссылки...
$text =~ s{
    \b
    # Сохранить адрес в $1...
    (
       \w[-.\w]*                                # Имя пользователя
       \@
       [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)  # Имя хоста
    )
    \b
}{<a href="mailto:$1">$1</a>}gix;

# Преобразовать HTTP URL в ссылки...
$text =~ s{
    \b
    # Сохранить URL в $1...
    (
       http://[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info) \b  # Хост
       (
            / [-a-z0-9_:@&?=+,.!/~*'%$]*    # Необязательный путь
              (?>![.,?!])       # Не может заканчиваться символами [.,?!]
       )?
    )
}{<a href="$1">$1</a>}gix;

print $text; # Вывод преобразованного текста
```

Построение библиотеки регулярных выражений

Обратите внимание: два имени хоста определяются одним и тем же выражением. Следовательно, при обновлении одного выражения мы должны обязательно синхронизировать его со вторым выражением. Вместо того чтобы создавать в программе источник для потенциальных ошибок, проще объявить переменную `$HostnameRegex`, как это сделано в следующем измененном фрагменте:

```
$HostnameRegex = qr/[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)/i;

# Преобразовать адреса электронной почты в ссылки...
$text =~ s{
```

```
    \b
    # Сохранить адрес в $1...
    (
       \w[-.\w]*                                  # имя пользователя
       \@
       $HostnameRegex                             # имя хоста
    )
    \b
}{<a href="mailto:$1">$1</a>}gix;

# Преобразовать HTTP URL в ссылки...
$text =~ s{
    \b
    # Сохранить URL в $1...
    (
       http://$HostnameRegex \b             # имя хоста
       (
            / [-a-z0-9_:@&?=+,.!/~*'%$]*    # необязательный путь
              (?<![.,?!])       # не может заканчиваться символами [.,?!]
       )?
    )
}{<a href="$1">$1</a>}gix;
```

В первой строке используется оператор Perl `qr`. Как и операторы `m` и `s`, он получает регулярное выражение (т. е. `qr/…/` по аналогии с `m/…/` и `s/…/…/`), но вместо того, чтобы применять его к некоторому тексту, он преобразует переданное выражение в *объект регулярного выражения*, который может быть сохранен в переменной. Позднее этот объект может использоваться вместо регулярного выражения или даже внутри другого выражения (как в нашем примере, где объект, присвоенный переменной `$HostnameRegex`, используется в двух операциях замены). Подобный подход чрезвычайно полезен, поскольку программа становится более наглядной. Но на этом его преимущества не кончаются: регулярное выражение для имени хоста определяется всего в одном месте, а затем может многократно использоваться в программе. Дополнительные примеры «библиотечных регулярных выражений» такого рода встречаются в главе 6 (☞ 350), а подробное пояснение этой темы приведено в главе 7 (☞ 382).

В других языках предусмотрены свои средства создания объектов регулярных выражений; некоторые объекты в общих чертах рассматриваются в следующей главе, а пакеты Java и платформа .NET подробно описаны в главах 8 и 9.

Почему иногда приходится экранировать символы $ и @

Вероятно, вы заметили, что символ '`$`' используется как в качестве метасимвола конца строки, так и при интерполяции (т. е. подстановке значения) переменной. Обычно смысл '`$`' интерпретируется однозначно, однако в символьных классах

ситуация усложняется. В символьном классе символ '$' никак не может обозначать конец строки, поэтому в этой ситуации Perl считает его признаком интерполяции переменной, если только символ не экранируется. Экранированный символ '$' просто считается обычным членом символьного класса. Именно это нам и нужно в данном примере, поэтому знак '$' во второй части выражения для поиска URL экранируется.

Сказанное относится и к символу @. Символ @ используется в Perl в качестве префикса имен массивов, которые могут интерполироваться в строковых литералах. Если мы хотим, чтобы литерал @ был частью регулярного выражения, символ необходимо экранировать; в противном случае он будет считаться признаком интерполяции массива.

В одних языках (в частности, в Java, VB.NET, C, C#, Emacs и awk) интерполяция переменных не поддерживается. Другие языки (Perl, PHP, Python, Ruby и Tcl) интерполяцию поддерживают, но каждый делает это по-своему. Данная тема развивается в следующей главе (☞ 140).

Задача с повторяющимися словами

Надеюсь, задача с повторяющимися словами из главы 1 пробудила в вас интерес к регулярным выражениям. В самом начале главы я подразнил вас загадочным набором символов, которые я назвал решением задачи:

```
$/ = ".\n";
while (<>) {
    next if !s/\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$3\e[m/ig;
    s/^(?:[^\e]*\n)+//mg;    # Удалить непомеченные строки.
    s/^/$ARGV: /mg;          # Начинать строку с имени файла.
    print;
}
```

Теперь, когда вы хотя бы немного разбираетесь в Perl, нетрудно узнать знакомые элементы: `<>`, три `s/…/…/` и `print`. И все же непонятного остается больше! Если в этой главе вы впервые встретились с Perl (и до этой книги никогда не работали с регулярными выражениями), вероятно, вы бы предпочли заняться чем-нибудь попроще.

Но как мне кажется, это регулярное выражение не такое уж сложное. Прежде чем подробно рассматривать программу, стоит вспомнить постановку задачи, описанную в начале главы 1, и посмотреть на результат тестового запуска:

```
% perl -w FindDbl ch01.txt
ch01.txt: check for doubled words (such as this this), a common problem with
ch01.txt: * Find doubled words despite capitalization differences, such as with
```

```
ch01.txt: despite capitalization differences, such as with 'The
ch01.txt: the...', as well as allow differing amounts of whitespace
ch01.txt: /\<(1,000,000|million|thousand thousand)/. But alternation can't
ch01.txt: of this chapter. If you knew the the specific doubled word
```

Перейдем к рассмотрению программы и начнем с языка Perl. Затем будет кратко рассмотрено решение на языке Java и продемонстрирован другой подход к работе с регулярными выражениями. В следующем листинге используется подстановка в форме `s{`*выражение*`}{`*замена*`}`*модификаторы* с модификатором `/x`, упрощающим чтение программы (а вместо '`next if !`' используется более понятная запись `next unless`). В остальном эта версия идентична приведенной выше.

Поиск повторяющихся слов

```
$/ = ".\n";   # ❶ Особый режим чтения; фрагменты завершаются
              #     комбинацией символов "точка - новая строка"
while (<>)    # ❷
{
    next unless s{ # ❸ Далее следует регулярное выражение
        ### Поиск совпадения слова
        \b              # Начало слова...
        ([a-z]+)        # Найти слово, заполнить переменную $1 (и \1)

        ### Далее следует произвольное количество пробелов и тегов <...>
        (               # Сохранить промежуточные символы в $2
           (?:          # (Несохраняющие скобки для группировки альтернатив)
                \s      # Пропуски (в том числе символы новой строки)
                |       # или
                <[^>]+> # конструкции вида <TAG>
            )+          # Обязателен хотя бы один
                        # из перечисленных элементов.
        )

        ### Проверить повторное совпадение первого слова
        (\1\b)       # \b исключает внутренние совпадения в других словах.
                  # Копия сохраняется в $3.
        # (конец регулярного выражения)
    }
    # Далее следует строка замены и модификаторы /i, /g, /x
    {\e[7m$1\e[m$2\e[7m$3\e[m}igx;  # ❹
    s/^(?:[^\e]*\n)+//mg;    # ❺ Удалить непомеченные строки
    s/^/$ARGV: /mg;          # ❻ Начинать строку с имени файла
    print;
}
```

В этой программе есть несколько идей, с которыми мы еще не встречались. Позвольте мне вкратце описать эти моменты, а также логику, лежащую в их основе.

За дополнительными подробностями я рекомендую обратиться к страницам справочного руководства Perl (или главу 7). В описании, следующем ниже, слово «волшебный» означает: «потому что это является одной из особенностей языка Perl, с которым вы еще не знакомы».

❶ Поскольку решение задачи с повторяющимися словами должно работать даже в том случае, когда повторяющиеся слова находятся в разных строках файла, мы не можем использовать обычный режим построчной обработки, как в примере с электронной почтой. Присваивание специальной переменной `$/` (да, это действительно переменная!) переводит последующий оператор `<>` в волшебный режим, при котором он возвращает не отдельные строки, а фрагменты, приблизительно совпадающие с абзацами. Возвращаемое значение представляет собой одну последовательность символов, которая может состоять из нескольких логических строк.

❷ Вы обратили внимание, что значение, возвращаемое оператором `<>`, ничему не присваивается? В условиях цикла `while` оператор `<>` волшебным образом присваивает строку специальной переменной по умолчанию[1]. Эта переменная содержит строку, с которой по умолчанию работает `s/…/…/` и которая выводится командой `print`. Использование переменной по умолчанию делает программу более компактной, но и менее понятной для новичков, поэтому я рекомендую пользоваться явными операндами до тех пор, пока вы не почувствуете себя более уверенно.

❸ Команда `next unless` перед командой подстановки заставляет Perl отменить обработку текущей строки (и перейти к следующей), если подстановка ничего не дает. Нет смысла продолжать обработку строки, в которой не были найдены повторяющиеся слова.

❹ Строка замены в действительности имеет вид `"$1$2$3"` с несколькими промежуточными Escape-последовательностями ANSI, обеспечивающими цветовое выделение двух повторяющихся слов (но не тех символов, которые их разделяют). Последовательность `\e[7m` начинает выделение, а `\e[m` — завершает его (`\e` в строках и регулярных выражениях Perl является сокращенным обозначением символа, с которого начинаются Escape-последовательности ANSI).

Присмотревшись к круглым скобкам в регулярном выражении, вы поймете, что `"$1$2$3"` соответствует совпавшей части. Поэтому, если не считать добавления Escape-последовательностей, вся команда замены фактически является (относительно медленной) пустой операцией.

[1] Речь идет о переменной `$_` (да, она действительно так называется!), используемой в качестве операнда по умолчанию во многих функциях и операторах.

Мы знаем, что `$1` и `$3` содержат одинаковые слова (собственно, для этого и была написана программа!), поэтому в замене можно было использовать одну из этих переменных. Тем не менее слова могут различаться регистром символов, поэтому я включил в строку замены обе переменные.

❺ Фрагмент текста может состоять из нескольких логических строк, но после пометки всех повторяющихся слов мы удаляем из выходных данных те логические строки, в которых отсутствует признак Escape-последовательности `\e`. В результате остаются лишь те строки, в которых присутствуют повторяющиеся слова. Благодаря наличию модификатора расширенного режима привязки `/m` регулярное выражение `⌜^([^\e]*\n)+⌟` находит логические строки, не содержащие `\e`, а пустая строка замены приводит к их удалению. В результате во входных данных остаются лишь логические строки, содержащие символ `\e`, т. е. строки с повторяющимися словами[1].

❻ Волшебная переменная `$ARGV` выводит имя входного файла. При наличии модификаторов `/m` и `/g` эта подстановка включает имя входного файла в начало каждой логической строки, оставшейся в выходных данных после удаления.

Наконец, команда `print` выводит оставшуюся часть строки вместе с Escape-последовательностями ANSI. Цикл `while` повторяет эти действия для всех строк (а точнее, фрагментов текста, примерно соответствующих абзацам), прочитанных из входных данных.

Операторы, функции и объекты

Я уже говорил о том, что Perl в этой главе используется лишь как средство, демонстрирующее основные принципы работы с регулярными выражениями. Несомненно, это очень полезное средство, но необходимо еще раз подчеркнуть, что задача может быть решена с применением регулярных выражений во многих других языках.

И все же демонстрация несколько упрощается тем, что на фоне других языков высокого уровня поддержка регулярных выражений в Perl является полноправной составляющей, интегрированной непосредственно в язык. Иначе говоря, в языке существуют базовые операторы, которые работают с регулярными выражениями точно так же, как операторы `+`, и работают с числами, что приводит к сокращению объема «синтаксического багажа», необходимого для работы с регулярными выражениями.

Во многих языках такая возможность отсутствует. По причинам, изложенным в главе 3 (☞ 129), во многих современных языках вместо этого предусмотрены

[1] Такое решение предполагает, что входной файл не содержит служебных символов ANSI. В противном случае программа может включить в выходные данные ошибочные строки.

специальные функции и объекты для применения и выполнения других операций с регулярными выражениями. Например, в языке может существовать функция, которой при вызове передается строка, интерпретируемая как регулярное выражение, и текст, в котором производится поиск; функция возвращает `true` или `false` в зависимости от того, совпадает ли регулярное выражение в тексте. Впрочем, чаще эти две операции (интерпретация строки как регулярного выражения и применение выражения к тексту) разбиваются на две и более отдельные функции, как показано в следующем листинге программы на языке Java. В программе используется пакет `java.util.regex` из стандартной поставки Java 1.4.

В начале листинга встречаются три регулярных выражения, использованных в примере с Perl, передаваемые в строковом виде функциям `Pattern.compile`. Непосредственное сравнение показывает, что в Java-версиях присутствуют лишние символы \, но это всего лишь побочный эффект синтаксических требований Java, в соответствии с которыми регулярные выражения должны определяться в строковом виде. Символы \, используемые в регулярном выражении, экранируются для того, чтобы анализатор строк Java не пытался интерпретировать их по-своему (☞ 73).

Вероятно, вы также заметили, что регулярные выражения определяются не в основной части программы, а в начале листинга, в секции инициализации. Функция `Pattern.compile` всего лишь анализирует строку регулярного выражения и строит внутреннюю «откомпилированную» версию, которая присваивается переменной типа `Pattern` (`regex1` и т. д.). Затем в основной части программы откомпилированная версия применяется к тексту в синтаксисе `regex1.matcher(text)`, а результат используется для выполнения замены. Подробности откладываются до следующей главы, сейчас речь идет о другом: при изучении любого языка с поддержкой регулярных выражений необходимо различать два аспекта — диалект регулярных выражений и синтаксис работы с регулярными выражениями в самом языке.

Поиск повторяющихся слов на языке Java

```
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class TwoWord
{
  public static void main(String [] args)
  {
    Pattern regex1 = Pattern.compile(
      "\\b([a-z]+)((?:\\s|\\<[^>]+\\>)+)(\\1\\b)",
      Pattern.CASE_INSENSITIVE);
    String replace1 = "\033[7m$1\033[m$2\033[7m$3\033[m";
    Pattern regex2 = Pattern.compile("^(?:[^\\e]*\\n)+", Pattern.MULTILINE);
```

```
    Pattern regex3 = Pattern.compile("^([^\\n]+)", Pattern.MULTILINE);

    // Для каждого аргумента командной строки....
    for (int i = 0; i < args.length; i++)
    {
      try {
        BufferedReader in = new BufferedReader(new FileReader(args[i]));
        String text;

        // Для каждого абзаца в файле.....
        while ((text = getPara(in)) != null)
        {
          // Применить три замены
          text = regex1.matcher(text).replaceAll(replace1);
          text = regex2.matcher(text).replaceAll("");
          text = regex3.matcher(text).replaceAll(args[i] + ": $1");

          // Вывести результаты
          System.out.print(text);
      }
     } catch (IOException e) {
       System.err.println("can't read ["+args[i]+"]: " + e.getMessage());
      }
    }
  }

  // Подпрограмма читает следующий абзац и возвращает его в виде строки
  static String getPara(BufferedReader in) throws java.io.IOException
  {
    StringBuffer buf = new StringBuffer();
    String line;

    while ((line = in.readLine()) != null &&
           (buf.length() == 0 || line.length() != 0))
    {
        buf.append(line + "\n");
    }
    return  buf.length() == 0 ? null : buf.toString();
  }
}
```

3 Регулярные выражения: возможности и диалекты

Итак, вы получили общее представление о регулярных выражениях и нескольких программах, в которых они поддерживаются. Может возникнуть обманчивое впечатление, что теперь вы готовы к тому, чтобы успешно пользоваться ими в любой другой программе. Но даже простое сравнение нескольких версий *egrep* в первой главе и языков Perl и Java во второй главе наглядно показывает, что внешний вид, а также особенности применения регулярных выражений сильно изменяются от программы к программе.

При рассмотрении регулярных выражений в контексте любого языка или программы учитываются три основных фактора.

- Состав и интерпретация поддерживаемых метасимволов. Часто называется «диалектом» регулярного выражения.
- Особенности взаимодействия регулярных выражений с языком или программой, в частности синтаксис операций с регулярными выражениями, поддерживаемые операции и требования к тексту, с которым они работают.
- Специфика применения регулярных выражений к тексту. Метод реализации механизма регулярных выражений, выбранный разработчиком языка или программы, заметно влияет на результаты применения любого регулярного выражения.

Регулярные выражения и автомобили

Перечисленные факторы напоминают те соображения, которыми люди обычно руководствуются при покупке автомобиля. Как при изучении диалекта регулярных выражений вы сначала обращаете внимание на метасимволы, так и у машины сначала бросаются в глаза внешние стороны — форма кузова, отделка и прочие приятные мелочи вроде проигрывателя компакт-дисков и кожаных сидений. Подобные вещи обычно щедро расписываются на глянцевых страницах рекламных брошюр; список метасимволов вроде приведенного на с. 61 можно считать их эквивалентом

в мире регулярных выражений. Бесспорно, это очень важная информация, но это не все.

Важным аспектом является взаимодействие регулярных выражений с управляющей программой. Одни составляющие интерфейса выполняют «косметические» функции (как, например, непосредственный синтаксис операций с регулярными выражениями в программе), другие более существенны (например, состав поддерживаемых операций и удобство работы с ними). В нашей аналогии с автомобилями можно провести параллель с «интерфейсом» машины с пассажиром. В нем есть как второстепенные аспекты (например, с какой стороны производится заправка или наличие выдвигающихся стекол), так и весьма важные — например, тип коробки передач. Третий фактор определяет удобство и функциональность: поместится ли машина в ваш гараж? А можно ли в ней перевезти матрас? Лыжи? Пятерых взрослых людей? И насколько удобно этим пятерым будет выходить из машины (конечно, с четырьмя дверьми им будет удобнее, чем с двумя)? Многие из этих аспектов также упоминаются в рекламных брошюрах, но про них обычно пишут самым мелким шрифтом на последней странице.

Остается лишь разобраться с двигателем и тем, насколько хорошо он справляется со своей задачей. На этом аналогия кончается, потому что большинство водителей, по крайней мере, минимально представляет возможности двигателя своей машины: если он работает на бензине, вряд ли кто-нибудь станет заливать в него дизельное топливо. Но в мире регулярных выражений даже самые существенные подробности работы конкретной реализации, а также их влияние на построение и использование регулярных выражений обычно не включаются в документацию. Тем не менее именно эти подробности настолько важны для практической работы с регулярными выражениями, что им в книге посвящена целая глава.

Для чего написана эта глава

Как нетрудно предположить по названию, в этой главе представлен обзор основных возможностей и диалектов регулярных выражений. В ней рассматриваются стандартные метасимволы и некоторые аспекты взаимодействия регулярных выражений с теми программами, которые обеспечивают их работу. Вероятно, вы узнали первые два пункта списка, приведенного в начале. Третий пункт — принципы работы механизма регулярных выражений и что это для нас значит в практическом аспекте — будет рассмотрен в следующих главах.

Прежде всего скажу, что эта глава не является справочником по применению регулярных выражений. Я не пытаюсь описывать особенности использования регулярных выражений в разных утилитах и языках программирования, встречающихся в примерах этой главы. В этой главе приводится общий обзор регулярных выражений и тех программных средств, в которых они реализованы. Если вы живете

в пещере и работаете только с одной программой, возможно, вы сможете счастливо прожить, не задумываясь о том, что другие программы (или другие версии той же программы) работают иначе. Но так не бывает, поэтому знание «родословной» вашей любимой программы обеспечит вас интересными и ценными сведениями.

История регулярных выражений

Сначала я хочу поведать краткую историю эволюции некоторых диалектов регулярных выражений и соответствующих программ. Берите горячую чашку (или запотевшую кружку) своего любимого напитка и устраивайтесь поудобнее. Вы познакомитесь с историей (порой довольно странной) становления регулярных выражений. Этот обзор добавит новых красок в общую картину и поможет вам лучше осознать, почему ситуация с регулярными выражениями выглядит именно так, а не иначе. Для особо любознательных включены сноски с дополнительной информацией, но вообще эту главу можно рассматривать как легкое, развлекательное чтиво.

Происхождение регулярных выражений

Семена регулярных выражений были посажены в начале 1940-х годов. Двое нейрофизиологов, Уоррен Маккаллох (Warren McCulloch) и Уолтер Питтс (Walter Pitts), занимались моделированием работы нервной системы на нейронном уровне[1]. Регулярные выражения воплотились в реальность через несколько лет, когда математик Стивен Клин (Stephen Kleene) дал формальное описание этих моделей при помощи алгебры, которую он назвал *регулярными множествами* (*regular sets*). Он разработал для регулярных множеств простую математическую запись, которую и назвал регулярными выражениями.

В 1950–1960-х регулярные выражения стали предметом серьезного изучения математиками-теоретиками. Роберт Констейбл (Robert Constable) написал хорошую статью[2] для специалистов-математиков.

---

[1] Статья *A logical calculus of the ideas immanent in nervous activity* была впервые опубликована в бюллетене *Bulletin of Math. Biophysics* (номер 5, 1943 г.) и позднее перепечатана в *Embodiments of Mind* (MIT Press, 1965 г.). Статья начинается с интересного обзора поведения нейронов (оказывается, скорость распространения внутринейронных импульсов меняется от 1 до 150 метров в секунду!), после чего погружается в бездну формул, в которых я так и не разобрался.

[2] Robert L. Constable, *The Role of Finite Automata in the Development of Modern Computing Theory*, в The Kleene Symposium, Eds. Barwise, Keisler, and Kunen (North-Holland Publishing Company, 1980), 61–83.

Хотя существуют свидетельства о более ранних работах, первой публикацией, посвященной применению регулярных выражений в области компьютерных технологий, которую мне удалось обнаружить, была статья Кена Томпсона *Regular Expression Search Algorithm* 1968 года[1]. В этой статье Томпсон описывает компилятор регулярных выражений, генерирующий объектный код IBM 7094. Это подтолкнуло его к работе над *qed* — редактором, который был положен в основу известного редактора UNIX *ed*.

Регулярные выражения *ed* уступали по своим возможностям выражениям *qed*, но именно они впервые получили широкое распространение за пределами теоретических кругов. Одна из команд *ed* выводила строки редактируемого файла, в которых находилось совпадение для заданного регулярного выражения. Эта команда, `g/Regular Expression/p`, читалась как «Global Regular Expression Print» («глобальный вывод по регулярному выражению»). Функция оказалась настолько полезной, что была преобразована в отдельную утилиту. Так появилась программа *grep*, по образцу которой позднее была создана ее расширенная версия — *egrep*.

Метасимволы grep

Регулярные выражения, поддерживаемые ранними программами, заметно уступали по своим возможностям выражениям *egrep*. Метасимвол * поддерживался, но + и ? не поддерживались (причем отсутствие последнего было особенно сильным недостатком). Для группировки метасимволов в *grep* использовалась конструкция `\(…\)`, а *не*экранированные круглые скобки использовались для представления литералов[2]. Программа *grep* поддерживала привязку к позициям строки, но в ограниченном варианте. Если символ ^ находился в начале регулярного выражения, он представлял собой метасимвол, совпадающий с началом строки. В противном случае он вообще не считался метасимволом и просто обозначал соответствующий литерал. Аналогично, символ $ считался метасимволом только в конце регулярного выражения. В результате терялась возможность использования выражений вида `⌜end$|^start⌟`. Впрочем, это несущественно, поскольку конструкция выбора все равно не поддерживалась!

Взаимодействие метасимволов также отличалось некоторыми особенностями. Например, один из главных недостатков *grep* заключался в том, что квантификатор * не мог применяться к выражениям в круглых скобках, а применялся только к литералам, символьным классам или метасимволу «точка». Следовательно, в *grep*

[1] *Communications of the ACM*, Vol.11, No. 6, June 1968.

[2] Историческая справка: *ed* (и *grep*) использовали в качестве ограничителей экранированные скобки вместо простых, потому что Кен Томпсон считал, что регулярные выражения будут в основном использоваться для работы с программным кодом C и поиск скобок-литералов будет происходить чаще, чем применение обратных ссылок.

скобки предназначались только для сохранения совпавшего текста, но не для общей группировки. Более того, в некоторых ранних версиях *grep* не допускалось вложение круглых скобок.

Эволюция grep

Хотя *grep* существует во многих системах и в наши дни, я в основном говорил об этой программе в прошедшем времени, поскольку речь шла о диалекте регулярных выражений в старых версиях 30-летней давности. Однако технология не стоит на месте, и со временем старые программы дополняются новыми возможностями. Программа *grep* не является исключением.

AT&T Bell Labs дополнили *grep* новыми возможностями — например, интервальным квантификатором `\{`*min*, *max*`\}`, позаимствованным из программы *lex*. Также была исправлена ошибка с ключом `-y`, который должен был обеспечивать поиск без учета регистра, но работал ненадежно. Одновременно в Беркли были добавлены метасимволы начала и конца слова, а ключ `-y` был переименован в `-i`. К сожалению, * и другие квантификаторы все еще не могли применяться к выражениям в круглых скобках.

Эволюция egrep

К этому времени Альфред Ахо (Alfred Aho), также работавший в AT&T Bell Labs, написал программу *egrep*, которая поддерживала большую часть набора метасимволов, описанного в главе 1. Еще важнее тот факт, что программа была реализована совершенно иным (и в общем случае более эффективным) способом. В *egrep* не только появились новые квантификаторы `+` и `?`, но и они наряду с другими квантификаторами стали применяться к выражениям в круглых скобках, что значительно расширило возможности регулярных выражений *egrep*.

Также была добавлена конструкция выбора, а якорные метасимволы получили «равноправие», т. е. могли использоваться практически в любом месте регулярного выражения. Конечно, у *egrep* были свои проблемы — иногда программа находила совпадение, но не включала его в результат, а также не поддерживала некоторые распространенные возможности. И все же пользы она приносила на порядок больше.

Появление других видов

В это время появились и начали развиваться другие программы (такие, как *awk*, *sed* и *lex*). Разработчик, которому нравилась какая-то возможность одной программы, часто пытался реализовать ее в другой программе. Результат иногда получался, мягко говоря, неэстетичным. Например, если вам вдруг захотелось включить в *grep*

поддержку квантификатора «плюс», для этой цели нельзя было использовать символ '+', поскольку в *grep* он традиционно не являлся метасимволом, и неожиданное превращение удивило бы пользователей. Поскольку комбинация '\+' в обычных условиях встречается редко, именно ее связали с метасимволом «один или больше».

Иногда реализация новых возможностей сопровождалась появлением новых ошибок. Бывало и так, что добавленная возможность позднее исключалась. Многие неочевидные аспекты, связанные с диалектом конкретной программы, документировались неполно или вообще не документировались, поэтому новые программы либо изобретали собственный стиль, либо пытались имитировать то, что «где-то работало».

Умножьте это обстоятельство на прошедшее время и количество программ, и в результате получится сущая неразбериха (особенно когда автор программы пытается заниматься несколькими делами сразу).

POSIX — попытка стандартизации

Стандарт POSIX (Portable Operating System Interface — переносимый интерфейс операционной системы) был предложен в 1986 году для обеспечения переносимости программ между операционными системами. Некоторые части этого стандарта связаны с регулярными выражениями и традиционными средствами, в которых они используются, поэтому данная тема представляет для нас интерес. Впрочем, ни один из диалектов, описанных в книге, не обеспечивает полного соответствия спецификации POSIX. В попытке упорядочить этот хаос POSIX делит распространенные диалекты на две категории: *BRE* (basic regular expressions, т. е. «базовые регулярные выражения») и *ERE* (extended regular expressions, т. е. «расширенные регулярные выражения»). POSIX-совместимые программы поддерживают одну из этих категорий. Метасимволы двух категорий POSIX перечислены в табл. 3.1.

Таблица 3.1. Категории диалектов регулярных выражений в стандарте POSIX

Метасимволы	BRE	ERE
Точка, ^, $, [...], [^...]	✓	✓
Произвольное число	*	*
Квантификаторы + и ?		+ ?
Интервальный квантификатор	\{min, max\}	\{min, max\}
Группировка	\(...\)	(...)
Применение квантификаторов к скобкам	✓	✓
Обратные ссылки	от \1 до \9	
Конструкция выбора		✓

Одной из важных особенностей стандарта POSIX является понятие *локального контекста* (*locale*) — совокупности параметров, описывающих языковые и национальные правила: формат даты, времени и денежной величины, интерпретация символов активной кодировки и т. д. Локальные контексты упрощают адаптацию программ для других языков. Они не относятся к специфике регулярных выражений, однако могут влиять на их применение. Например, при работе в локальном контексте с кодировкой Latin-1 (ISO-8859-1), à и À (коды 224 и 160 соответственно) считаются «буквами». При любом применении регулярных выражений, при котором игнорируется регистр символов, эти два символа будут считаться идентичными.

Другой пример — метасимвол `\w`, обычно обозначающий «символ слова» (во многих диалектах это понятие эквивалентно `[a-zA-Z0-9_]`). POSIX не требует, но допускает поддержку этого метасимвола. При поддержке `\w` в поиск включаются все буквы и цифры, определенные в локальном контексте, а не только те, которые определены в кодировке ASCII.

Однако следует учесть, что эта сторона локальных контекстов в значительной мере теряет свою значимость при использовании программ с поддержкой Юникода. Юникод рассматривается на с. 146.

Пакет Генри Спенсера

В 1986 году Генри Спенсер (Henry Spencer) выпустил первый пакет для работы с регулярными выражениями, написанный на языке C. Любой желающий мог бесплатно включить этот пакет (первый на тот момент) в свою программу. Все программы, использовавшие этот пакет (а такие программы были, и немало), поддерживали один и тот же согласованный диалект регулярных выражений — если только автор не вносил в него сознательные изменения.

Эволюция Perl

Примерно в то же время Ларри Уолл (Larry Wall) начал работу над программой, которая позднее превратилась в язык Perl. Ларри уже был автором программы *patch*, заметно упростившей разработку распределенных приложений, но Perl было суждено произвести настоящую революцию.

Первая версия Perl была выпущена в декабре 1987 года. Язык пользовался огромным успехом; в нем были объединены многие полезные возможности других языков, причем основной упор был сделан на повседневную, практическую *полезность*.

Одной из самых заметных особенностей Perl стал набор операторов для работы с регулярными выражениями в лучших традициях специализированных программ *sed* и *awk* — в сценарном языке общего назначения подобное встречалось впервые.

Код механизма регулярных выражений Ларри позаимствовал из *rn* — программы просмотра электронных новостей, написанной самим Ларри. В свою очередь, программа *rn* создавалась на базе механизма регулярных выражений Emacs Джеймса Гослинга (James Gosling)[1]. По стандартам того времени диалект Perl считался достаточно мощным, но его даже отдаленно нельзя сравнить с современными аналогами. Главные недостатки состояли в том, что он поддерживал не более 9 пар круглых скобок, не более 9 альтернатив в конструкции выбора `|`, и что хуже всего — не поддерживалось применение `|` в круглых скобках. Кроме того, не поддерживался поиск без учета регистра символов, не допускалось включение `\w` в символьные классы (а метасимволы `\s` и `\d` вообще не поддерживались). Также отсутствовал интервальный квантификатор `{min, max}`.

Версия 2 вышла в июне 1988 года. Ларри полностью переработал весь код работы с регулярными выражениями и воспользовался улучшенной версией пакета Генри Спенсера, упоминавшегося в предыдущем разделе. Регулярные выражения по-прежнему не могли содержать более 9 пар круглых скобок, но теперь в них могла использоваться конструкция `|`. Добавилась поддержка `\s` и `\d`, а в `\w` был включен символ подчеркивания, поскольку это соответствовало синтаксису имен переменных в Perl. Более того, эти метасимволы могли включаться в символьные классы (их антиподы, `\D`, `\W` и `\S`, тоже поддерживались, но включать их в символьные классы *запрещалось*, и вообще они иногда работали неправильно). Также появилась поддержка очень важного модификатора `/i`, позволявшего выполнять поиск без учета регистра.

Версия 3 появилась больше года спустя, в октябре 1989 года. В ней добавился модификатор `/e`, значительно расширявший возможности оператора замены, и были исправлены некоторые ошибки с обратными ссылками. Появились интервальные квантификаторы, хотя, к сожалению, они были недостаточно надежны. Что еще хуже, в версии 3 механизм регулярных выражений не всегда правильно работал с 8-разрядными данными, а это приводило к непредсказуемым последствиям, если входные данные выходили за пределы ASCII.

Версия 4 вышла через полгода, в марте 1991 года, и в течение двух следующих лет она постепенно совершенствовалась до своего последнего обновления в феврале 1993 года. К этому времени ошибки были исправлены, а ограничения сняты (допускалось использование `\D` и других подобных метасимволов в символьных классах, а количество круглых скобок в выражениях практически не ограничивалось). Также велась работа над оптимизацией механизма регулярных выражений, но настоящий переворот произошел лишь в 1994 году.

---

[1] Позднее Джеймс Гослинг займется разработкой собственного языка Java. Как ни странно, в этом языке отсутствует встроенная поддержка регулярных выражений. Тем не менее в Java 1.4 входит замечательный пакет для работы с регулярными выражениями, подробно описанный в главе 8.

Версия 5 была официально выпущена в октябре 1994 года. Язык Perl был основательно переработан и стал гораздо более приятным во всех отношениях. В области регулярных выражений были добавлены внутренние оптимизации и несколько новых метасимволов, в том числе метасимвол `\G`, повышающий эффективность итеративного поиска (☞ 177), несохраняющие круглые скобки (☞ 45), минимальные квантификаторы (☞ 190), опережающая проверка (☞ 92) и модификатор `/x`[1] (☞ 107).

Помимо непосредственной практической пользы эти изменения наглядно показали, что регулярные выражения сами по себе могут стать мощным языком программирования, причем возможности для усовершенствования в этой области еще не исчерпаны.

Новые возможности (несохраняющие круглые скобки и опережающую проверку) необходимо было как-то выразить на уровне синтаксиса. Парные скобки — `(…)`, `[…]`, `<…>` и `{…}` — не могли использоваться для этой цели, поэтому Ларри выбрал конструкции '`(?`', с которыми мы работаем сегодня. Эта неприглядная комбинация была выбрана потому, что ранее в регулярных выражениях Perl она считалась недопустимой, поэтому за ней можно было смело закрепить новый смысл. Ларри также предвидел возможность новых дополнений; ограничив состав символов, которые могут следовать после комбинации '`(?`', он зарезервировал место для будущих усовершенствований.

Дальнейшие версии Perl работали более устойчиво, содержали меньше ошибок, больше внутренних оптимизаций и новых возможностей. Хочется верить, что в этом есть некоторая заслуга первого издания настоящей книги, поскольку я занимался анализом и тестированием подсистемы регулярных выражений. Результаты, которые я отправлял Ларри, а также в группу Perl Porters, помогали отыскать новые области для усовершенствования.

За последние годы в регулярных выражениях появились такие новшества, как ограниченная ретроспективная проверка (☞ 92), атомарная группировка (☞ 187) и поддержка Юникода. Поддержка условных конструкций (☞ 188) подняла регулярные выражения на новый уровень и позволила принимать решения типа «если-то-иначе» на уровне регулярного выражения. А если этого недостаточно, в языке появились новые конструкции, позволяющие смешивать программный код с регулярным выражением (☞ 410). В книге рассматривается Perl версии 5.8.8.

---

[1] Кстати говоря, Ларри добавил модификатор `/x` после того, как получил от меня сообщение с описанием длинного и сложного регулярного выражения. Я «приукрасил» это выражение для наглядности. Ларри решил, что подобные вещи должны выполняться в Perl на программном уровне, и включил модификатор `/x`.

Частичная консолидация диалектов

Новые возможности Perl 5 как нельзя лучше совпали по времени с пришествием World Wide Web. Язык Perl создавался для работы с текстом, а построение веб-страниц сводится именно к обработке текста, так что вполне понятно, почему Perl быстро стал *основным* языком веб-разработок. Популярность Perl заметно увеличилась, и это в немалой степени объясняется широтой возможностей его регулярных выражений.

Разработчики других языков не остались равнодушными к мощи регулярных выражений Perl. Постепенно появлялось все больше новых пакетов для работы с регулярными выражениями, «Perl-совместимых» в той или иной степени. Среди них стоит упомянуть пакеты для Tcl, Python, семейства языков Microsoft .NET, Ruby, PHP, C/C++ и несколько пакетов для Java.

В 1997 году появилась еще одна форма консолидации (что по времени совпало с выходом первого издания этой книги), когда Филип Хейзел (Philip Hazel) разработал библиотеку PCRE (Perl Compatible Regular Expressions — библиотека регулярных выражений, совместимых с Perl) — тщательно продуманный механизм регулярных выражений, который очень близко повторяет синтаксис и семантику регулярных выражений Perl. Другие разработчики смогли интегрировать PCRE в свои программные продукты и языки и тем самым обеспечить своих пользователей богатством и выразительностью регулярных выражений. В настоящее время PCRE используется в широко известных программных продуктах, таких как PHP, Apache Version 2, Exim, Postfix и Nmap[1].

Версии программ, упоминаемых в книге

В табл. 3.2 перечислены номера версий некоторых программ и библиотек, рассматриваемых в книге. Более старые версии обычно содержат меньше возможностей и больше ошибок (впрочем, в новых версиях могут появиться свои ошибки).

Таблица 3.2. Версии некоторых программ, упоминаемых в книге

GUN awk 3.1	`java.util.regex` (Java 1.5.0, A.K.A Java 5.0)	Procmail 3.22
GNU egrep/grep 2.5.1	.NET Framework 2.0	Python 2.3.5
GNU Emacs 21.3.1	PCRE 6.6	Ruby 1.8.4
flex 2.5.31	Perl 5.8.8	GNU sed 4.0.7
MySQL 5.1	PHP (семейство функций preg) 5.1.4 / 4.4.3	Tcl 8.4

[1] Библиотека PCRE доступна для скачивания по адресу *ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/*

На первый взгляд

Достаточно взглянуть лишь на некоторые аспекты распространенных программ, чтобы понять, как сильно они отличаются друг от друга. В табл. 3.3 приведена очень поверхностная сводка диалектов некоторых программ.

Такие таблицы часто приводятся в книгах для наглядной демонстрации различий между разными инструментами. Но эта таблица является, в лучшем случае, верхушкой айсберга — у всех перечисленных возможностей существуют десятки важных аспектов, которые часто упускаются из виду при поверхностном знакомстве.

Прежде всего, сами программы изменяются со временем. Например, Tcl когда-то не поддерживал обратных ссылок и привязки к границам слов, а теперь поддерживает. Сначала границы слов обозначались некрасивыми конструкциями `[:<:]` и `[:>:]`, они поддерживаются до сих пор, но сейчас вместо них рекомендуется использовать более современные метасимволы `\m`, `\M` и `\y` (начало границы слова, конец границы слова или и то и другое).

Такие программы, как *grep* и *egrep*, не имеют единого источника — фактически они создаются всеми желающими и поддерживают любой диалект по усмотрению автора. Человеческая природа такова, что каждый наделяет свою программу какими-то отличительными особенностями (например, GNU-версии многих распространенных программ часто обладают большими возможностями и работают надежнее других версий).

Таблица 3.3. Поверхностный обзор диалектов некоторых распространенных программ

Возможность	Современные версии grep	Современные версии egrep	GNU Emacs	Tcl	Perl	.NET	Пакет для Java от Sun
`*, ^, $, […]`	✓	✓	✓	✓	✓	✓	✓
`? + |`	`\? \+ |`	`? + |`	`? + |`	`? + |`	`? + |`	`? + |`	`? + |`
Группировка	`\(...\)`	`(...)`	`\(...\)`	`(...)`	`(...)`	`(...)`	`(...)`
`(?:...)`					✓	✓	✓
Границы слов		`\< \>`	`\< \> \b \B`	`\m \M \y`	`\b \B`	`\b \B`	`\b \B`
`\w, \W`		✓	✓	✓	✓	✓	✓
Обратные ссылки	✓		✓	✓	✓	✓	✓
✓ — поддерживается							

Впрочем, наряду с очевидными различиями существуют и другие, менее заметные. При беглом взгляде на таблицу может создаться впечатление, что регулярные вы-

ражения Perl, .NET и Java абсолютно одинаковы, что не соответствует действительности. Ниже перечислены лишь некоторые вопросы, которые приходят в голову при виде табл. 3.3.

- Могут ли * и другие квантификаторы применяться к выражениям, заключенным в круглые скобки?
- Может ли точка совпадать с символом новой строки? А инвертированные символьные классы? И как насчет нуль-символа (`NUL`)?
- К какой строке привязываются якорные метасимволы — целевой или логической (т. е. распознают ли они символы новой строки, находящиеся внутри целевой строки)? Являются ли они полноправными метасимволами или допускаются только в определенных частях регулярного выражения?
- Распознаются ли экранированные символы в символьных классах? Какие еще символы разрешаются или запрещаются в символьных классах?
- Разрешается ли вложение круглых скобок? Если разрешается, то на сколько уровней (и вообще, сколько круглых скобок может присутствовать в выражении)?
- Если в выражениях разрешены обратные ссылки, то правильно ли они совпадают при поиске без учета регистра? Насколько «разумно» работают обратные ссылки в нетривиальных ситуациях?
- Допускается ли экранирование восьмеричных кодов символов в формате `\123`? Если допускается, то как разрешаются конфликты с обратными ссылками? А как насчет экранирования шестнадцатеричных кодов? И кто *в действительности* поддерживает восьмеричные и шестнадцатеричные коды — механизм регулярных выражений или какая-то другая часть программы?
- Совпадает ли `\w` только с алфавитно-цифровыми символами или допускаются еще какие-то другие символы? (Среди программ с поддержкой `\w`, упоминаемых в табл. 3.3, встречается несколько разных интерпретаций.) В какой степени `\w` согласуется с традиционными метасимволами границ слов в отношении того, что является или не является «символом слова»? Учитывает ли он локальный контекст или поддерживает Юникод?

Использование в качестве начального руководства даже такой краткой сводки, как приведена в табл. 3.3, заставляет учитывать множество факторов. Если вы будете знать, что под гладкой поверхностью кроется немало подводных камней, вам будет легче обойти их.

Как упоминалось в начале главы, иногда все сводится к поверхностным различиям в синтаксисе, но некоторые аспекты более глубоки. Например, если вы знаете, что

выражение, которое в *egrep* выглядит как ⌜`(Jul|July)`⌟, в GNU Emacs должно записываться в виде ⌜`\(Jul\|July\)`⌟, можно подумать, что дальше все идет одинаково. Это не всегда так — различия в семантике поиска совпадений (или по крайней мере в том, как она выглядит извне) являются очень важным фактором, который нередко упускается из виду, но иногда объясняет различия в совпадениях двух внешне идентичных примеров: один из них всегда совпадает с текстом '`Jul`' даже в случае применения к строке '`July`'. Именно семантика объясняет, почему выражения с обратным порядком альтернатив, ⌜`(July|Jul)`⌟ и ⌜`\(July|Jul\)`⌟, совпадают с одним и тем же текстом. Напомню, что этой теме посвящена вся следующая глава.

Конечно, то, что программа может *сделать* с регулярным выражением, нередко играет более важную роль, чем поддерживаемый ею диалект регулярных выражений. Даже если бы регулярные выражения Perl уступали *egrep* по своим возможностям, благодаря гибким средствам их использования Perl все равно приносил бы больше практической пользы. В этой главе мы рассмотрим ряд отдельных возможностей, а подробное описание языков будет отложено до следующей главы.

Основные операции с регулярными выражениями

Вторым фактором, упомянутым в начале главы, является синтаксическая «обертка», которая сообщает приложению: «вот регулярное выражение, а вот то, что с ним нужно сделать». Программа *egrep* является вырожденным примером, поскольку регулярное выражение передается в виде аргумента в командной строке. Дополнительные синтаксические «украшения» (например, апострофы, использованные в главе 1) связаны с требованиями командного интерпретатора, а не *egrep*. В более сложных системах (например, подсистемах регулярных выражений в языках программирования) необходимы более сложные синтаксические конструкции, которые сообщают системе, что именно следует считать регулярным выражением и как оно должно использоваться.

Итак, на следующем шаге мы должны выяснить, что же можно сделать с результатами совпадения. В качестве простейшего примера и на этот раз можно рассматривать программу *egrep*, которая ограничивается одной операцией (выводом строк, содержащих совпадения), но как видно из предыдущей главы, настоящие возможности раскрываются лишь при выполнении нетривиальных операций. Примером таких возможностей можно считать уже рассмотренный поиск *совпадения* (проверка наличия совпадения в строке и возможно — извлечение информации из строки) и *поиск с заменой* (модификация строки по результатам поиска). Существует несколько разновидностей как самих операций, так и способов их выполнения в конкретных языках.

В общем случае в языке программирования реализуется один из трех подходов к обработке регулярных выражений: интегрированный, процедурный и объектно-ориентированный. В первом случае регулярные выражения встраиваются непосредственно в язык, как это сделано в Perl. В двух других случаях регулярные выражения не входят в низкоуровневый синтаксис языка. Обычным функциям передаются обычные строки, которые затем интерпретируются как регулярные выражения. В зависимости от функции затем выполняются соответствующие операции с регулярными выражениями. Те или иные производные этого подхода используются в большинстве языков программирования (не считая Perl), в том числе в Java, языках .NET, Tcl, Python, PHP, Emacs lisp и Ruby.

Интегрированный интерфейс

Нам уже встречались примеры интегрированных средств работы с регулярными выражениями в Perl, как в примере на с. 86:

```
if ($line =~ m/^Subject: (.*)/i) {
    $subject = $1;
}
```

В этом фрагменте имена переменных выделены курсивом; синтаксические конструкции регулярных выражений выделены жирным шрифтом, а регулярное выражение подчеркнуто. Perl применяет регулярное выражение `^Subject:•(.*)` к тексту, хранящемуся в переменной `$line`, и в случае обнаружения совпадения выполняет следующий блок программы. В данном примере переменная `$1` представляет текст, совпавший с подвыражением в круглых скобках. Этот текст позднее присваивается переменной `$subject`.

Другим примером интегрированной обработки является включение регулярных выражений в конфигурационные файлы, как в почтовой программе UNIX *procmail*. В конфигурационном файле регулярные выражения используются для маршрутизации сообщений между секциями, в которых производится их непосредственная обработка. Здесь ситуация еще проще, чем в Perl, поскольку операнды (сообщения) задаются автоматически.

Однако «за кулисами» при этом происходят события более сложные, чем кажется на первый взгляд. Интегрированный подход упрощает ситуацию с точки зрения программиста за счет маскировки механики подготовки регулярных выражений, настройки поиска, применения регулярного выражения и получения результатов. В результате обычные случаи реализуются очень легко, но, как вы вскоре убедитесь, в некоторых ситуациях подобная интеграция приводит к снижению эффективности и появлению громоздких выражений.

Прежде чем переходить к подробностям, давайте посмотрим, что же именно скрывается от программиста при интегрированной обработке. Для этого стоит поближе познакомиться с другими методами.

Процедурный и объектно-ориентированный интерфейс

Два других типа интерфейса для работы с регулярными выражениями — процедурный и объектно-ориентированный — имеют много общего. В обоих случаях функциональность регулярного выражения обеспечивается не встроенными операторами, а обычными функциями (процедурный подход) или конструкторами и методами (объектно-ориентированный подход). Вместо специализированных выражений-операндов используются обычные строковые аргументы, которые интерпретируются функциями, конструкторами и методами как регулярные выражения.

В следующем разделе приведены примеры на языках Java, VB.NET, PHP и Python.

Работа с регулярными выражениями в Java

Ниже приведен пример с выделением подстроки «Subject» на языке Java с использованием пакета `java.util.regex` (язык Java подробно описан в главе 8).

```
import java.util.regex.*; // Получение доступа к классам
                          // регулярных выражений
 ⋮
❶ Pattern r = Pattern.compile("^Subject: (.*)", Pattern.CASE_INSENSITIVE);
❷ Matcher m = r.matcher(line);
❸ if (m.find()) {
❹    subject = m.group(1);
}
```

Как и в предыдущем примере, имена переменных выделены курсивом; синтаксические конструкции регулярных выражений выделены жирным шрифтом, а само регулярное выражение подчеркнуто. Точнее, подчеркнут обычный строковый литерал, который интерпретируется как регулярное выражение.

В этом фрагменте продемонстрирован объектно-ориентированный подход к работе с регулярными выражениями, обеспечиваемый двумя классами пакета `java.util.regex` от Sun: `Pattern` и `Matcher`. Выполняются следующие действия:

❶ Анализируется регулярное выражение и компилируется во внутреннюю форму, обеспечивающую совпадение без учета регистра символов. Результат представляет собой объект `Pattern`.

❷ Объект регулярного выражения ассоциируется с текстом, к которому он применяется. Результатом является объект `Matcher`.

❸ Регулярное выражение применяется к тексту. Метод `find` проверяет, существует ли совпадение в ранее ассоциированном тексте, и возвращает результат проверки.

❹ При наличии совпадения текст, совпавший с первой парой сохраняющих круглых скобок, присваивается переменной.

Подобные действия выполняются (явно или косвенно) любой программой, использующей регулярные выражения. Perl скрывает подробности от программиста, а в реализации Java они обычно лежат на поверхности.

Пример процедурного подхода

Пакет регулярных выражений для Java, разработанный в Sun, также поддерживает несколько вспомогательных функций, маскирующих большую часть выполняемых операций. Программисту не приходится сначала создавать объект регулярного выражения, а затем вызывать методы этого объекта; эти статические функции сами создают временный объект и удаляют его после завершения операции.

В следующем примере используется функция `Pattern.matches()`:

```
if (! Pattern.matches("\\s*", line))
{
    //  ... Строка не пустая ...
}
```

Регулярное выражение неявно заключается между символами ⌜^...$⌟ и возвращает логическую величину — признак возможного совпадения во входной строке. Многие пакеты предоставляют как процедурный, так и объектно-ориентированный интерфейс, как это сделано в реализации от Sun. Различия между двумя подходами часто связаны с удобством применения (процедурный интерфейс удобнее для простых задач, но дает более громоздкую запись в сложных ситуациях), функциональностью (процедурный интерфейс в общем случае уступает по своим возможностям своим объектно-ориентированным аналогам) и эффективностью (в любой конкретной ситуации один из вариантов оказывается более эффективным, чем другой, — эта тема подробно рассматривается в главе 6).

Существует несколько пакетов регулярных выражений для языка Java, но компании Sun удалось обеспечить более высокую степень интеграции своего пакета с языком. Например, пакет был интегрирован со строковым классом, поэтому предыдущий пример также можно записать в следующем виде:

```
if (! line.matches("\\s*", ))
{
    //  ... Строка не пустая ...
}
```

По своей эффективности подобное решение несколько уступает правильному применению объектно-ориентированного интерфейса, поэтому его не стоит использовать в циклах, критичных по времени, однако в «житейских» случаях оно вполне удобно.

Работа с регулярными выражениями в VB и других языках .NET

Все механизмы регулярных выражений решают одни и те же базовые задачи, однако они по-разному предоставляют доступ к своей функциональности и результатам программисту, причем различия существуют даже в реализациях, относящихся к одному типу. Ниже приведен пример с поиском подстроки «Subject» в VB.NET (платформа .NET подробно рассматривается в главе 9):

```
Imports System.Text.RegularExpressions   ' Получение доступа к классам
                                         ' регулярных выражений
   ⋮
Dim R as Regex = New Regex("^Subject: (.*)", RegexOptions.IgnoreCase)
Dim M as Match = R.Match(line)
If M.Success
    subject = M.Groups(1).Value
End If
```

В целом пример похож на аналогичную программу на языке Java, если не считать того, что .NET объединяет шаги ❷ и ❸ и использует дополнительную ссылку `Value` на шаге ❹. Чем объясняются эти различия? Не стоит полагать, что один вариант явно лучше или хуже другого, — решения выбирались разработчиками, твердо уверенными, что использованный подход на тот момент являлся оптимальным (об этом — чуть ниже).

.NET также поддерживает несколько функций, обеспечивающих процедурный интерфейс. Следующий фрагмент обнаруживает пустые строки:

```
If Not Regex.IsMatch(Line, "^\s*$") Then
    //  ... Строка не пустая ...
End If
```

В отличие от функции `Pattern.matches` из реализации Sun, неявно заключающей регулярное выражение между метасимволами `^...$`, разработчики Microsoft решили придать своей функции более общий характер. Она представляет собой простую «обертку» для базовых объектов, но зато программисту не приходится вводить лишние символы и плодить лишние переменные (при минимальном снижении эффективности).

Работа с регулярными выражениями в PHP

Ниже приводится решение примера ⌜Subject⌟ на языке PHP с использованием семейства функций preg, предназначенных для работы с регулярными выражениями. (Описание PHP вы найдете в главе 10.)

```
if (preg_match('/^Subject: (.+)/i', $line, $matches))
     $Subject = $matches[1];
```

Работа с регулярными выражениями в Python

Рассмотрим последний пример — реализацию примера ⌜Subject⌟ на языке Python:

```
import re;
    ⋮
R = re.compile("^Subject: (.*)", re.IGNORECASE);
M = R.search(line)
if M:
    subject = M.group(1)
```

Представленное решение также имеет много общего с приведенными выше.

Чем объясняются различия в реализациях?

Почему в одном языке задача решается одним способом, а в другом то же самое делается совершенно иначе? Отчасти это может объясняться спецификой самого языка, но в основном зависит от прихотей и уровня мастерства программистов, занимающихся разработкой пакетов. Существует несколько независимых пакетов для работы с регулярными выражениями в языке Java; все они были написаны авторами, которым понадобились возможности, не поддерживаемые исходной реализацией от Sun. Каждый пакет обладает своими достоинствами и недостатками, но обратите внимание на любопытное обстоятельство: у каждого пакета есть свои особенности использования, по которым он заметно отличается от других пакетов и от исходной реализации Sun.

Другим ярким примером существующих отличий может служить язык PHP, который включает в себя *три* не связанных между собой механизма регулярных выражений, каждый из которых предлагает свой собственный набор функций. В разные моменты времени разработчики PHP, не удовлетворенные имеющейся функциональностью, вносили дополнения в ядро PHP и соответствующие наборы функций. (Вообще, семейство функций «preg» можно рассматривать как надмножество для всех остальных наборов, как своего рода обложку книги.)

Поиск с заменой

Пример с поиском подстроки «Subject» слишком тривиален, и на нем трудно продемонстрировать различия между разными реализациями. В этом разделе будет рассмотрен более сложный пример, в котором эти различия выделяются нагляднее.

В предыдущей главе (☞ 107) был приведен пример использования средств поиска и замены языка Perl для преобразования адресов в ссылки `mailto`:

```
$text =~ s{
    \b
    # Сохранить адрес в $1...
    (
      \w[-.\w]*                          # имя пользователя
      @
      [-\w]+(\.[-\w]+)*\.(com|edu|info)  # имя хоста
    )
    \b
}{<a href="mailto:$1">$1</a>}gix;
```

Оператор поиска и замены в языке Perl работает непосредственно с самой строкой, в том смысле, что переменная, которая участвует в операции поиска с заменой, по окончании операции оказывается модифицированной. В большинстве других языков сначала создается *копия* оригинального текста, после чего в копии производится поиск с заменой. Это бывает очень удобно, когда необходимо сохранить оригинальную строку неизменной, однако если этого не требуется, возникает необходимость записывать результат операции в ту же самую переменную. Давайте посмотрим, как та же операция выполняется в других языках.

Поиск с заменой на языке Java

Ниже приведен пример поиска с заменой, реализованный с использованием пакета `java.util.regex` компании Sun:

```
import java.util.regex.*; // Получение доступа к классам
                          // регулярных выражений
     ⋮
Pattern r = Pattern.compile(
    "\\b                                                  \n"+
    "# Сохранить адрес в $1...                            \n"+
    "(                                                    \n"+
    "  \\w[-.\\w]*                     # Имя пользователя \n"+
    "  @                                                  \n"+
    "  [-\\w]+(\\.[-\\w]+)*\\.(com|edu|info)   # Имя хоста\n"+
    ")                                                    \n"+
    "\\b                                                  \n",
```

```
    Pattern.CASE_INSENSITIVE|Pattern.COMMENTS);

Matcher m = r.matcher(text);
text = m.replaceAll("<a href=\"mailto:$1\">$1</a>");
```

В этом листинге стоит обратить внимание на ряд обстоятельств. Наверное, самое важное из них — то, что каждый символ '\' в регулярном выражении представляется символами '\\' в строковом литерале. Таким образом, '\\w' в строке означает `\w` в регулярном выражении. В целях отладки было бы полезно воспользоваться командой **`System. out.println(r.pattern())`**; и вывести регулярное выражение в том виде, в котором оно передается функции. Одна из причин, по которой я включил символы новой строки в регулярное выражение, заключается в том, что оно лучше смотрится при выводе. Есть и другая причина — каждый знак '#' начинает комментарий, который продолжается до ближайшего символа новой строки. Следовательно, по крайней мере, пара символов новой строки понадобится хотя бы для того, чтобы ограничить комментарии.

Для обозначения специальных условий в Perl используются модификаторы `/g`, `/i` и `/x` (*глобальная замена*, *поиск без учета регистра*, *свободное форматирование* — ☞ 183). В пакете `java.util.regex` для этой цели используются разные функции (`replaceAll` вместо `replace`) или флаги, передаваемые при вызове (например, `Pattern.CASE_INSENSITIVE` и `Pattern.COMMENTS`).

Поиск с заменой на языке VB.NET

Общий подход в VB.NET выглядит аналогично:

```
Dim R As Regex = New Regex _
 ("\b                                                  " & _
  "(?# Сохранить адрес в $1... )                       " & _
  "(                                                   " & _
  "  \w[-.\w]*               (?# имя пользователя)     " & _
  "  @                                                 " & _
  "  [-\w]+(\.[-\w]+)*\.(com|edu|info) (?# имя хоста) " & _
  ")                                                   " & _
  "\b                                                  ",  _
  RegexOptions.IgnoreCase Or RegexOptions.IgnorePatternWhitespace)

text = R.Replace(text, "<a href=""mailto:${1}"">${1}</a>")
```

Из-за недостаточной гибкости строковых литералов VB.NET (они не могут продолжаться в другой строке, в них не могут входить символы новой строки) работать с длинными регулярными выражениями в VB.NET не так удобно, как в других языках. С другой стороны, поскольку «\» в VB.NET не является строковым метасимволом, выражения не перегружаются лишними символами. Учтите, что кавычки

являются метасимволами в строковых литералах VB.NET и экранируются удвоением (иначе говоря, если потребуется включить в содержимое строки кавычку, в литерал включаются две кавычки подряд).

Поиск с заменой на языке PHP

Ниже приводится вариант реализации поиска с заменой в языке PHP:

```
$text = preg_replace('{
                        \b
                        # Сохранить адрес в $1...
                        (
                            \w[-.\w]*        # имя пользователя
                            @
                            [-\w]+(\.[-\w]+)*\.(com|edu|info) # имя хоста
                        )
                        \b
                      }ix',
                      '<a href="mailto:$1">$1</a>', # замена строки $text);
```

Как и в двух предыдущих примерах (Java и VB.NET), результат операции поиска с заменой должен записываться обратно в переменную `$text`, а в остальном этот пример весьма сходен с реализацией в языке Perl.

Поиск и замена в других языках

Ниже приводится краткий обзор других языков и программ.

Awk

В языке awk используется интегрированный подход — конструкция */регулярное_выражение/* ищет совпадения в текущей входной строке, а конструкция «**`var ~ …`**» выполняет поиск в других данных. Именно awk повлиял на синтаксис операций с регулярными выражениями языка Perl (впрочем, при выборе оператора подстановки Perl за образец была взята программа sed). В ранних версиях awk операция подстановки не поддерживалась, но в современных версиях появился оператор `sub(…)`:

```
sub(/mizpel/, "misspell")
```

Эта команда применяет регулярное выражение ⌜`mizpel`⌟ к текущей строке, заменяя первый найденный экземпляр строкой `misspell`. Сравните с командой Perl **`s/mizpel/misspell/`**.

Для замены всех экземпляров в строке вместо модификатора `/g` или его аналога в awk используется другая функция: **`gsub(/mizpel/, "misspell")`**.

Tcl

В Tcl используется процедурный синтаксис, который на первый взгляд выглядит довольно странно, если вы не знакомы с правилами оформления строк в Tcl. Приведенный выше пример на Tcl может выглядеть так:

```
regsub mizpel $var misspell newvar
```

Команда проверяет содержимое переменной `var`, заменяет первый экземпляр ⌜`mizpel`⌟ строкой `misspell` и присваивает полученный текст переменной `newvar` (которая в данном случае *не* снабжается префиксом `$`). По правилам Tcl на первом месте передается регулярное выражение, на втором — целевая строка, на третьем — строка замены и на четвертом — имя целевой переменной. Tcl также позволяет передавать при вызове `regsub` дополнительные параметры. Например, ключ `all` обеспечивает глобальную замену всех найденных экземпляров (не только первого):

```
regsub -all mizpel $var misspell newvar
```

Ключ `-nocase` заставляет механизм регулярных выражений игнорировать регистр символов (по аналогии с флагом *egrep* `-i` или модификатором Perl `/i`).

GNU Emacs

В мощном текстовом редакторе GNU Emacs (в дальнейшем просто Emacs) поддерживается встроенный язык программирования *elisp* (Emacs lisp) и многочисленные функции для работы с регулярными выражениями. Одна из важнейших функций, `re-search-forward`, получает в качестве аргумента обычную строку и интерпретирует ее как регулярное выражение, после чего ищет текст от текущей позиции до первого совпадения или отменяет поиск, если совпадение отсутствует.

Именно эта функция вызывается при выполнении команды поиска по регулярному выражению в редакторе.

Как видно из табл. 3.3 (☞ 128), для диалекта регулярных выражений Emacs характерно наличие многочисленных символов \. Например, регулярное выражение ⌜`\<([a-z]+\)\([\n•\t]\|<[^>]+>\)+\1\>`⌟ находит в тексте повторяющиеся слова (см. главу 1). Непосредственно использовать это выражение нельзя, поскольку механизм регулярных выражений Emacs не понимает символов `\n` и `\t`. С другой стороны, эти символы поддерживаются для строк Emacs, заключенных в кавычки, причем замена их кодами соответствующих символов производится еще до того, как они станут видны механизму регулярных выражений. В этом состоит заметное преимущество

передачи регулярных выражений в строковом формате. Впрочем, есть и недостатки. Из-за частого использования символа \ в диалекте *elisp* регулярные выражения порой выглядят так, словно кто-то рассыпал упаковку зубочисток. Ниже приведена небольшая функция для поиска следующего повторяющегося слова:

```
(defun FindNextDbl ()
"move to next doubled word, ignoring <...> tags"   (interactive)
(re-search-forward "\\<\\([a-z]+\\)\\([\n \t]\\|<[^>]+>\\)+\\1\\>")
)
```

Если объединить эту функцию с командой **(define-key global-map "\Cx\C-d" 'FindNextDbl)**, вы сможете использовать последовательность «Control-x Control-d» для быстрого поиска повторяющихся слов.

Итоги

Как видите, разные программы обладают разными функциональными возможностями и средствами для их реализации. Если вы только начинаете работать на этих языках, вероятно, у вас будет немало затруднений. Не бойтесь! Чтобы освоить какой-то конкретный инструмент, нужно просто понять логику его работы.

Строки, кодировки и режимы

Прежде чем переходить к рассмотрению распространенных метасимволов, мы должны рассмотреть ряд общих тем: интерпретация строк как регулярных выражений, кодировки и режимы поиска совпадений.

Теоретически эти концепции чрезвычайно просты, причем некоторые из них остаются простыми даже на практике. Тем не менее в ряде случаев общая картина искажается мелкими деталями, нюансами и расхождениями между реализациями, из-за которых бывает трудно разобраться, как именно они работают. В нескольких ближайших разделах рассматриваются как простые, так и нетривиальные проблемы, с которыми вы столкнетесь на практике.

Строки как регулярные выражения

На первый взгляд все просто: в большинстве языков, за исключением Perl, *awk* и *sed*, регулярные выражения передаются в виде обычных строк, которые часто определяются в виде литералов вида `"^From:(.*)"`. И все же у многих программистов (особенно на первых порах) часто возникают затруднения с использованием метасимволов строковых литералов языка при построении регулярных выражений.

Строковые литералы любого языка обладают собственным набором метасимволов. Более того, в ряде языков существует несколько разновидностей строковых литералов, поэтому единых правил не существует, однако общие принципы остаются одними и теми же. В строковых литералах многих языков распознаются экранированные последовательности вида `\t`, `\\` и `\x2A`, интерпретируемые в процессе построения содержимого строки. Самая распространенная проблема, связанная с употреблением регулярных выражений, заключается в том, что каждый символ `\` в регулярном выражении представляется двумя символами `\\` в соответствующем строковом литерале. Например, для получения регулярного выражения `⌜\n⌟` необходима запись вида `"\\n"`.

Если забыть про удвоение символа `\` в строковом литерале и использовать литерал вида `"\n"`, во многих языках вы получите символ ⌜NL⌟, который по чистой случайности делает то же самое, что и `⌜\n⌟`. Или, вернее, почти то же самое — если регулярное выражение задается в режиме свободного форматирования `/x`, ⌜NL⌟ обращается в ничто, тогда как `⌜\n⌟` остается регулярным выражением, совпадающим с символом новой строки. Если забыть об этом, возможны неприятности. В табл. 3.4 приведен ряд примеров с метасимволами `\t` и `\x2A` (`2A` — ASCII-код символа '*'). Вторая пара примеров в таблице демонстрирует неожиданные результаты в том случае, если вы забыли о метасимволах строковых литералов.

В разных языках используются разные строковые литералы, но некоторые из них отличаются от других тем, что символ `\` не является метасимволом. Например, в строковых литералах VB.NET существует всего один метасимвол — кавычка. Ниже рассматриваются особенности строковых литералов нескольких распространенных языков. Впрочем, независимо от правил построения строковых литералов при работе с ними вы должны руководствоваться главным вопросом: что именно получит механизм регулярных выражений после того, как строка будет обработана по правилам языка?

Таблица 3.4. Примеры строковых литералов

Строковый литерал	**"[\t\x2A]"**	**"[\\t\\x2A]"**	**"\t\x2A"**	**"\\t\\x2A"**
Содержимое строки	`[`⇥`*]`	`[\t\x2A]`	⇥`*`	`[\t\x2A]`
Как регулярное выражение	`⌜[`⇥`*]⌟`	`[\t\x2A]`	⇥`*`	`[\t\x2A]`
Совпадает	Символ табуляции или *	Символ табуляции или *	Любое количество символов табуляции	Символ табуляции, за которым следует *
В режиме `/x`	Символ табуляции или *	Символ табуляции или *	*Ошибка*	Символ табуляции, за которым следует *

Строки Java

Начало этой главы дает представление о строковых литералах Java: они заключаются в кавычки, а символ \ является метасимволом. Поддерживаются многие стандартные комбинации — '\t' (символ табуляции), '\n' (символ новой строки), '\\' (символ \) и т. д. Использование символа \ в комбинации, не поддерживаемой строковыми литералами Java, приводит к ошибке.

Строки VB.NET

Строковые литералы VB.NET также заключаются в кавычки, но в остальном они значительно отличаются от литералов Java. В строках VB.NET распознается только одна метапоследовательность: пара кавычек в строковом литерале транслируется в одну кавычку в строке. Например, строка `"he said ""hi""\."` преобразуется в выражение ⌜`he said "hi"\.`⌟.

Строки C#

Хотя во внутренней работе всех языков Microsoft .NET Framework используется общий механизм регулярных выражений, в каждом языке установлены свои правила для строк, используемых при создании выражений-аргументов. Простые строковые литералы Visual Basic уже были описаны выше. В другом языке .NET, C#, существует два типа строковых литералов.

C# поддерживает стандартные строки в кавычках, аналогичные тем, что упоминались в начале главы, за единственным исключением — для включения кавычки в строку вместо `\"` используется последовательность `""`. Наряду с ними в C# также поддерживаются буквально интерпретируемые строки вида `@"..."`, в которых последовательности символом \ не распознаются, а поддерживается одна специальная последовательность: пара кавычек преобразуется в целевой строке в одну кавычку. Это означает, что выражение ⌜`\t\x2A`⌟ может быть создано как в виде строки `"\\t\\x2A"`, так и в виде `@"\t\x2A"`. Из-за простоты и удобства при построении регулярных выражений обычно используются буквально интерпретируемые строки `@"..."`.

Строки PHP

В PHP также существует два типа строк, однако они существенно отличаются от любого из типов C#. В строках, заключенных в кавычки, могут использоваться стандартные последовательности типа '\n', но при этом также поддерживается интерполяция переменных, знакомая нам по Perl (☞ 111), и специальная последовательность `{...}` для вставки в строку результатов выполнения кода, заключенного между фигурными скобками.

Расширенные возможности этой разновидности строк PHP означают, что в регулярные выражения часто приходится вставлять лишние символы \, однако другая их особенность помогает избавиться от этой необходимости. В строковых литералах Java и C# наличие неопознанных комбинаций с \ приводит к ошибке, но в строках PHP такие комбинации просто включаются в содержимое строки. Комбинация `\t` опознается в строках, заключенных в кавычки, поэтому метасимвол ⌜`\t`⌟ представляется строкой `"\\t"`. С другой стороны, строка `"\w"` преобразуется в ⌜`\w`⌟, потому что `\w` не принадлежит к числу распознаваемых комбинаций. Иногда эта дополнительная возможность оказывается удобной, но она усложняет работу со строками, заключенными в кавычки, поэтому в PHP также поддерживается другая, более простая разновидность — строки, заключенные в апострофы.

Строки PHP, заключенные в апострофы, также помогают избавить строки от лишних символов на манер строк VB.NET или строк `@"..."` языка C#, но делают это несколько иначе. В строке, заключенной в апострофы, комбинация `\'` вставляет один апостроф в содержимое строки, а комбинация `\\` в конце строки позволяет закончить строку символом \. Все остальные символы (в том числе и \) не считаются специальными и копируются в целевую строку без изменений. Таким образом, строка `'\t\x2A'` создает регулярное выражение ⌜`\t\x2A`⌟. Благодаря своей простоте строки PHP, заключенные в апострофы, лучше всего подходят для работы с регулярными выражениями PHP.

Строки PHP, заключенные в апострофы, подробнее рассматриваются в главе 10 (☞ 552).

Строки Python

Язык Python поддерживает несколько разновидностей строковых литералов. Для создания строк в качестве ограничителей можно использовать как апострофы, так и кавычки, но в отличие от PHP между ними нет никаких различий. В Python также существуют строки с «тройными ограничителями» в форме `'''...'''` или `"""..."""`, которые в отличие от других строк могут содержать неэкранированные символы новой строки. Для всех четырех типов ограничителей распознаются стандартные последовательности вида `\n`; при этом следует учитывать, что (как и в PHP) неопознанные комбинации включаются в строку (в отличие от Java и C#, где такие неопознанные последовательности вызывают ошибку).

Как и в PHP и C#, в Python поддерживается «упрощенная» запись строк, которая обозначается префиксом `r`, стоящим перед открывающей кавычкой любого из четырех типов кавычек, описанных выше (по аналогии с синтаксисом C# `@"..."`). Например, строка `r"\t\x2A"` порождает выражение ⌜`\t\x2A`⌟. Но в отличие от других языков в упрощенных строках Python *все* символы \ сохраняются в строке, в том числе и те, которые экранируют кавычки (чтобы кавычки могли включаться

в строку): так, строка `r"he said \"hi\"\."` порождает регулярное выражение ⌜`he said \"hi\"\.`⌟. Это не создает проблем при использовании строк в регулярных выражениях, поскольку диалект регулярных выражений Python интерпретирует ⌜`\"`⌟ как ⌜`"`⌟, но при желании вы можете воспользоваться другой формой кавычек: `r'he said "hi"\.´`

Строки Tcl

Tcl отличается от других языков тем, что в нем вообще нет строковых литералов как таковых. Командная строка делится на «слова», которые интерпретируются командами Tcl как строки, имена переменных, регулярные выражения или все, что угодно, в соответствии со спецификой команды. В процессе разбора строки на слова происходит распознавание и преобразование стандартных последовательностей типа `\n`, а символы `\` в неизвестных комбинациях просто удаляются. Слова могут заключаться в кавычки, но при отсутствии внутренних пропусков это не обязательно.

В Tcl также поддерживается упрощенная форма, аналогичная упрощенным строкам Python, но вместо `r'…'` используются фигурные скобки: `{…}`. Внутри скобок все символы, кроме комбинации «\+новая строка», интерпретируются буквально, поэтому для получения выражения ⌜`\t\x2A`⌟ можно использовать запись `{\t\x2A}`.

Внутри фигурных скобок могут находиться дополнительные пары фигурных скобок (при условии их правильного вложения). Невложенные фигурные скобки должны экранироваться бэкслэшем, хотя бэкслэш *остается* в значении строки.

Литералы регулярных выражений в Perl

Во всех примерах на языке Perl, встречавшихся в книге, регулярные выражения задавались в виде литералов. Оказывается, они также могут задаваться в виде строк. Например, команда

```
$str =~ m/(\w+)/;
```

также может быть записана в виде

```
$regex = '(\w+)';
$str =~ $regex;
```

или

```
$regex = "(\\w+)";
$str =~ $regex;
```

(хотя литералы могут быть значительно эффективнее ☞ 308, 436).

Если регулярное выражение задается в виде литерала, Perl обеспечивает ряд дополнительных возможностей, не предусмотренных самим регулярным выражением, в том числе:

- интерполяция переменных (подстановка значения переменной в качестве части регулярного выражения);
- поддержка литерального режима с использованием синтаксиса `\Q...\E` (☞ 154);
- необязательная поддержка конструкции `\N{`*имя*`}`, позволяющей задавать символы по именам Юникода. Например, для поиска текста '¡Hola!' может использоваться выражение `\N{INVERTED EXCLAMATION MARK}Hola!`.

В Perl литералы регулярных выражений разбираются как особая разновидность строк. Следует упомянуть, что перечисленные возможности также характерны для строк Perl, заключенных в кавычки. Вы должны хорошо понимать, что эти возможности поддерживаются *не* механизмом регулярных выражений. Поскольку абсолютное большинство регулярных выражений, используемых в программах Perl, задается при помощи литералов, некоторые программисты считают конструкцию `\Q...\E` частью языка регулярных выражений Perl. Но если вам когда-нибудь придется работать с регулярными выражениями, прочитанными из конфигурационного файла (переданными в командной строке и т. д.), необходимо знать, каким аспектом языка обеспечивается та или иная возможность.

Дополнительная информация приведена в главе 7, начиная со с. 362.

Проблемы кодировки символов

Кодировка символов представляет собой набор правил, определяющих интерпретацию значений байтов. Байт с десятичным значением 110 в кодировке ASCII интерпретируется как символ 'n', а в кодировке EBCDIC — как символ '>'. Почему? Потому что так кто-то решил — в самих кодах и в символах нет ничего такого, что заставило бы предпочесть одну кодировку другой. Байты одинаковы, меняется только их интерпретация.

В кодировке ASCII определяются символы только для половины значений, которые могут храниться в одном байте. В кодировке ISO-8859-1 (обычно называемой Latin-1) пустые места заполняются буквами с диакритическими знаками и специальными символами, что упрощает ее использование в различных языках. Например, в кодировке Latin-1 байт с десятичным кодом 234 интерпретируется как буква ê и не считается неопределенным, как в ASCII.

Возникает важный вопрос: если мы считаем, что некоторый набор байтов *должен интерпретироваться* по правилам конкретной кодировки, всегда ли мнение программы будет совпадать с нашим? Допустим, имеются четыре байта с кодами 234,

116, 101 и 115, которые должны интерпретироваться в кодировке Latin-1 (французское слово «êtes»), и мы хотим использовать для поиска регулярное выражение ⌜^\w+$⌟ или ⌜^\b⌟. Это возможно, только если в представлении программы метасимволы \w и \b интерпретируют эти байты как символы Latin-1; в противном случае совпадение, вероятно, не будет найдено.

Поддержка кодировок при работе с регулярными выражениями

В мире существует много кодировок. Если вы работаете с одной конкретной кодировкой, задайте себе несколько важных вопросов.

- Понимает ли программа эту кодировку?
- Откуда она узнает, что данные должны интерпретироваться именно в этой кодировке?
- Каким уровнем возможностей работы с регулярными выражениями обладает данная кодировка?

Уровень работы с регулярными выражениями определяется рядом важных факторов, в том числе:

- Распознаются ли символы, закодированные несколькими байтами, как таковые? С чем совпадают выражения ⌜.⌟ и ⌜^x⌟ — с одним *символом* или с одним *байтом*?
- Правильно ли метасимволы \w, \d, \s, \b и подобные им воспринимают весь набор символов данной кодировки? Например, мы знаем, что ê — это буква, но известно ли об этом метасимволам \w и \b?
- Пытается ли программа расширить интерпретацию символьных классов? Совпадает ли ê с символьным классом [a-z]?
- Правильно ли работает поиск без учета регистра со всеми символами? Например, считаются ли символы ê и Ê эквивалентными?

Некоторые вопросы не так просты, как может показаться на первый взгляд. Например, метасимвол \b из пакета java.util.regex правильно интерпретирует все «словесные» символы Юникода, а метасимвол \w из того же пакета понимает только базовые символы ASCII. Далее в этой главе встретятся и другие примеры.

Юникод

Существует масса заблуждений по поводу того, что же означает термин «Юникод». На базовом уровне Юникод представляет собой *набор символов*, или *концептуаль-*

ную кодировку, т. е. логическое отображение между числами и символами. Например, корейский символ 삵 ассоциируется с числом 49 333. При записи символов Юникода это число, *называемое кодовым пунктом*, обычно отображается в шестнадцатеричной форме с префиксом U+. В шестнадцатеричной системе счисления десятичному числу 49 333 соответствует число `C0B5`, поэтому этот символ представляется записью U+`C0B5`. Для многих символов в концептуальное представление Юникода включаются дополнительные атрибуты типа «3 — это цифра» или «É — прописная буква, которой в нижнем регистре соответствует буква é».

Однако на базовом уровне ничего не говорится о том, как же эти числа кодируются в компьютерных данных. Существует несколько вариантов такой кодировки, включая кодировку UCS-2 (все символы кодируются двумя байтами), UCS-4 (все символы кодируются четырьмя байтами), UTF-16 (большинство символов кодируется двумя, но некоторые символы — четырьмя байтами) и UTF-8 (символы кодируются переменным числом байтов, от одного до шести). Вопрос о том, какая из этих кодировок используется во внутренних операциях той или иной программы, обычно не представляет интереса для пользователя программы. Как правило, пользователю приходится думать о том, как преобразовать внешние данные (например, прочитанные из файла) из конкретной кодировки (ASCII, Latin-1, UTF-8 и прочих) в формат, используемый программой. Программы с поддержкой Юникода обычно предоставляют различные функции кодирования/декодирования, которые и осуществляют преобразование.

В программах, работающих с Юникодом, регулярные выражения часто поддерживают метапоследовательность `\u`*num*, которая может использоваться для поиска конкретного символа Юникода (☞ 159). Код обычно задается в виде числа, состоящего из четырех шестнадцатеричных цифр, поэтому обозначение `\uC0B5` совпадает с 삵. Важно понимать, что запись `\uC0B5` всего лишь ассоциируется с символом Юникода U+`C0B5` и ничего не говорит о том, сколькими байтами этот символ представляется (это зависит от конкретной кодировки, используемой во внутренних операциях программы для представления пунктов Юникода). Если во внутренней работе программы используется кодировка UTF-8, может оказаться, что символ представляется тремя байтами. Вас как пользователя программы с поддержкой Юникода это беспокоить не должно. (Хотя иногда в этом возникает необходимость, например, при работе с семейством функций preg языка PHP и модификатором `u`; ☞ 556.)

Тем не менее имеется ряд смежных вопросов, которые необходимо учитывать...

Символы и комбинации

Понятие «символ» с точки зрения программиста не всегда совпадает с понятием «символ» с точки зрения Юникода или программы, поддерживающей Юникод.

Например, большинство из нас скажет, что буква à является отдельным символом, но в Юникоде код этого символа состоит из двух кодовых пунктов: U+0061 (a) в комбинации с диакритическим знаком U+0300 (`). В Юникоде существуют различные комбинационные символы, которые должны следовать за базовым символом (и объединяться с ним). Это несколько усложняет задачу механизма регулярных выражений — например, должен ли метасимвол ⌜.⌟ совпадать с частью кода или со всей комбинацией U+0061 + U+0300?

На практике многие программы считают понятия «символ» и «пункт» синонимами, вследствие чего метасимвол ⌜.⌟ совпадает с каждым пунктом в отдельности, будь то базовый символ или любой из комбинационных символов. Таким образом, à (U+0061 + U+0300) совпадает с выражением ⌜^..$⌟ и не совпадает с ⌜^.$⌟.

В Perl и PCRE (и семействе функций preg языка PHP) поддерживается удобная метапоследовательность \X, которая ведет себя так, как обычно ожидают от ⌜.⌟ — она совпадает с базовым *символом*, за которым следует произвольное количество комбинационных символов. За дополнительной информацией обращайтесь к с. 163.

Помните о комбинационных символах при непосредственном вводе символов регулярного выражения в редакторе с поддержкой Юникода. Если регулярное выражение завершается акцентированным символом (например, Å), представленным в виде 'A'+'°', скорее всего, оно не совпадет в строке, содержащей однопунктовую версию A (однопунктовые версии рассматриваются в следующем подразделе). Кроме того, с точки зрения самого механизма регулярных выражений этот символ воспринимается как два отдельных символа, поэтому запись ⌜[…Å…]⌟ добавляет в символьный класс сразу два символа, как при явном перечислении ⌜[…A°…]⌟.

Если за таким символом следует квантификатор, он применяется только к диакритическому знаку, как в выражении ⌜A°+⌟.

Наличие разных представлений

Теоретически Юникод устанавливает однозначное соответствие между символами и их числовыми представлениями, но во многих ситуациях один символ может иметь несколько представлений. В предыдущем разделе я уже упоминал о том, что символ à представляется пунктом U+0061, за которым следует U+0300. Тем не менее этот символ *также* кодируется одним пунктом U+00E0. Для чего нужна двойная кодировка? Чтобы упростить преобразование символов между Юникодом и Latin-1. Если у вас имеется текст в кодировке Latin-1, который преобразуется в Юникод, вероятно, буква à будет преобразована в U+00E0, но нельзя исключать и возможность ее преобразования в комбинацию U+0061 + U+0300. Довольно часто вы ничего не можете сделать, чтобы повлиять на выбор того или иного варианта представления символов, однако пакет `java.util.regex` поддерживает специальный ключ поиска `CANON_EQ`, который обеспечивает одинаковое совпадение для

«канонически эквивалентных» символов даже в том случае, если их представления в Юникоде различны (☞ 460).

Существует и другая проблема того же рода: разные символы могут выглядеть одинаково, что порой вызывает недоразумения при подготовке проверяемого текста. Например, латинскую букву I (U+0049) часто путают с греческой буквой «йота» (U+0399). А при добавлении *диалитики* получается буква Ї, которая может кодироваться четырьмя разными способами (U+00CF; U+03AA; U+0049 U+0308; U+0399 U+0308). Таким образом, вам придется вручную ввести все четыре возможности при конструировании выражения, совпадающего с Ї, — и это далеко не единственный пример.

Также часто встречаются одиночные символы, которые на первый взгляд выглядят как несколько символов. Например, в Unicode определяется символ «SQUARE HZ» (U+3390), который имеет вид Hz и выглядит очень похожим на два обычных символа Hz (U+0048 U+007A).

Хотя в настоящее время специальные символы вроде Hz используются очень редко, их постепенное распространение повысит сложность программ обработки текста, поэтому все программисты, работающие с Юникодом, должны учитывать эти потенциальные проблемы. Например, наряду с уже перечисленными вопросами может возникнуть необходимость в поддержке как обычных (U+0020), так и неразрывных (U+00A0) пробелов — или еще десятка разных пробелов, определенных в Юникоде.

Юникод 3.1+ и пункты, находящиеся после U+FFFF

В спецификацию Юникода версии 3.1, вышедшей в середине 2001 года, были включены символы с кодовыми пунктами, выходящими за границу U+FFFF (в предыдущих версиях Юникода была сформулирована возможность размещения символов в этом интервале, но ни один символ фактически не определялся). Например, в новой спецификации появился специальный символ для музыкального знака C Clef (U+1D121). Программы, рассчитанные на работу с пунктами только от U+FFFF и ниже, не распознают этот символ. У большинства программ в конструкции \u*код* может задаваться только шестнадцатеричное число из четырех цифр.

Программы, способные воспринимать символы с этими новыми пунктами взамен или в дополнение к нотации \u*код*, обычно предоставляют конструкцию \x{*код*}, где *код* может содержать любое количество цифр. Таким образом, для поиска символа C Clef можно воспользоваться конструкцией \x{1D121}.

Завершители строк

В Юникоде определяется несколько символов (а также одна последовательность из двух символов), которые считаются *завершителями строк* (*line terminators*). Завершители строк перечислены в табл. 3.5.

Таблица 3.5. Завершители строк в Юникоде

Символ	Код	Описание
LF	U+000A	Перевод строки (ASCII)
VT	U+000B	Вертикальная табуляция (ASCII)
FF	U+000C	Перевод формата (ASCII)
CR	U+000D	Возврат курсора (ASCII)
CR/LF	U+000D U+000A	Возврат курсора/перевод строки (ASCII-последовательность)
NEL	U+0085	Следующая строка (Юникод)
LS	U+2028	Разделитель строк (Юникод)
PS	U+2029	Разделитель абзацев (Юникод)

При наличии полноценной поддержки со стороны программы завершители строк влияют на результаты чтения строк из файла (в сценарных языках — включая файл, из которого читается программа). В регулярных выражениях они могут влиять как на то, какие символы совпадают с ⌜`.`⌟ (☞ 151), так и на возможность совпадения метасимволов ⌜`^`⌟, ⌜`$`⌟ и ⌜`\Z`⌟ (☞ 152).

Режимы обработки регулярных выражений и поиска совпадений

В большинстве механизмов регулярных выражений поддерживается несколько режимов интерпретации или применения выражения. Мы уже встречались с примерами двух типов — модификаторами Perl `/x` (режим, допускающий пропуски и комментарии в регулярных выражениях ☞ 106) и `/i` (поиск без учета регистра ☞ 77).

Режимы могут применяться глобально ко всему регулярному выражению или, как во многих современных диалектах, к отдельным подвыражениям. Глобальное применение реализуется при помощи различных модификаторов и ключей, как, например, модификатор Perl `/i`, модификатор шаблона `i` в PHP или флаг `Pattern.CASE_INSENSITIVE` пакета `java.util.regex` (☞ 98). Частичное применение, если оно поддерживается, обеспечивается конструкциями вида ⌜`(?i)`⌟ (включение поиска без учета регистра) и ⌜`(?-i)`⌟ (отключение). В некоторых диалектах также поддерживаются конструкции ⌜`(?i:…)`⌟ и ⌜`(?-i:…)`⌟, которые включают и отключают поиск без учета регистра для указанных подвыражений.

Активизация этих режимов в регулярных выражениях рассматривается позднее в этой главе (☞ 182). В этом разделе всего лишь рассматриваются некоторые режимы, существующие в большинстве систем.

Режим поиска без учета регистра символов

Режим, при котором игнорируется регистр символов при поиске (`[b]` совпадает с 'b' и 'B'), поддерживается практически во всех диалектах и программах. Он зависит от должной поддержки на уровне кодировки, поэтому вы должны помнить обо всех предупреждениях, упоминавшихся выше.

Как ни странно, поддержка этого режима традиционно содержала множество ошибок. Большинство этих ошибок было исправлено, но некоторые остаются до сих пор. Например, до сих пор режим поиска без учета регистра не распространяется на восьмеричные и шестнадцатеричные коды символов в Ruby.

В Юникоде поиск без учета регистра (который в Юникоде называется «поиском неточного совпадения») сопряжен с некоторыми дополнительными обстоятельствами. Прежде всего, не во всех алфавитах предусмотрено деление символом по регистрам, а в некоторых предусмотрен особый *титульный регистр*, используемый только в начале слов. В отдельных случаях не существует однозначного соответствия между верхним и нижним регистром. Общеизвестный пример: греческая буква «сигма» (`Σ`) имеет два представления в нижнем регистре, `ς` и `σ`; теоретически при поиске без учета регистра все три символа должны считаться эквивалентными (из всех протестированных систем это было так только в Perl и в Java-пакете `java.util.regex`).

Но это еще не все: один символ в другом регистре иногда представляется несколькими символами. Как известно, символ ß в верхнем регистре представляется комбинацией из двух символов «SS», но только Perl корректно обрабатывает его.

Существуют и другие проблемы, обусловленные применением Юникода. Скажем, у символа ǰ (`U+01F0`) нет односимвольной версии в верхнем регистре — он представляется только комбинацией `U+006A` и `U+030C` (☞ 147). Однако при этом оба варианта, прописной и строчный, должны совпадать при поиске без учета регистра. Существуют сходные ситуации, где совпадение должно реализовываться для трех представлений одного и того же символа. К счастью, большая часть таких символов используется довольно редко.

Свободное форматирование и комментарии

В режиме свободного форматирования пропуски за пределами символьных классов в основном игнорируются. Учитываются пропуски внутри символьных классов (кроме `java.util.regex`), а между `#` и концом строки могут находиться комментарии. Выше приводились примеры для Perl (☞ 141), Java (☞ 136) и VB.NET (☞ 137).

Было бы неправильно утверждать, что *все* пропуски за пределами символьных классов игнорируются (это так только для пакета `java.util.regex`). Правильнее

было бы сказать, что они преобразуются в метасимвол, который «ничего не делает». Различия важны в конструкциях вида ⌜\12•3⌟ — эта конструкция в режиме свободного форматирования воспринимается как последовательность ⌜\12⌟, за которой следует ⌜3⌟, а не ⌜\123⌟, как можно было бы предположить.

Разумеется, вопрос о том, что считается или не считается «пропуском», зависит от действующей кодировки и полноты ее поддержки. Большинство программ опознает только пробелы из кодировки ASCII.

Совпадение точки со всеми символами («однострочный режим»)

Обычно метасимвол ⌜.⌟ не совпадает с символом новой строки. Исходные инструменты UNIX работали в построчном режиме, поэтому до появления *sed* и *lex* сама проблема совпадения с символом новой строки была неактуальной. К моменту появления этих программ запись ⌜.*⌟ стала распространенной идиомой «совпадения до конца строки», поэтому новые инструменты запрещали выход за границу строки, чтобы программисты могли пользоваться знакомыми конструкциями[1]. В связи с этим в программных инструментах, допускающих возможность работы с многострочным текстом (таких, как текстовые редакторы), обычно запрещалось совпадение ⌜.⌟ с символом конца строки.

В современных языках программирования режимы, в которых точка совпадает или не совпадает с символом новой строки, могут быть одинаково удобными. Выбор режима для конкретной ситуации зависит от самой ситуации. Многие программы теперь позволяют установить нужный режим на уровне регулярного выражения.

Из этого распространенного правила существует ряд исключений. Системы с поддержкой Юникода (например, пакет для работы с регулярными выражениями в Java от Sun) позволяют изменить состав допустимых совпадений для точки и исключить из него любой из односимвольных завершителей строк Юникода (☞ 150). В Tcl в общем случае точка совпадает со всеми символами, но при этом существуют особые режимы «поддержки новой строки» и «частичной поддержки новой строки», в которых точка и инвертированные символьные классы не совпадают с символом новой строки.

Неудачное название

При первом появлении модификатора `/s` в Perl этот режим был назван «однострочным» (single-line). Этот неудачный термин существует до настоящего времени и порождает массу недоразумений, поскольку он не имеет ничего общего

[1] Как мне объяснил Кен Томпсон, автор *ed*, это предотвращало «чрезмерное разрастание» совпадений ⌜.*⌟.

с метасимволами ⌜^⌟ и ⌜$⌟, на которые распространяется «многострочный» режим, обсуждаемый ниже. «Однострочный» режим означает всего лишь то, что точка может совпасть с любым символом.

Расширенный режим привязки к границам строк («многострочный» режим)

Расширенный режим привязки к границам строк влияет на совпадение якорных метасимволов ⌜^⌟ и ⌜$⌟. Метасимвол ⌜^⌟ обычно не совпадает с внутренними символами новой строки, а только с началом того текста, к которому применяется регулярное выражение. Тем не менее в расширенном режиме он также может совпасть с внутренним символом новой строки, поэтому при наличии таких символов ⌜^⌟ фактически разбивает целевой текст на логические строки. Пример уже встречался нам в предыдущей главе (☞ 103) при разработке программы Perl, преобразующей текст в код HTML. Весь текстовый документ содержался в одной строковой переменной, что позволило использовать команду **`s/^$/<p>/mg`** для преобразования текста «tags.␤␤It's...» в «tags.␤ **`<p>`** ␤It's...». Команда заменяет пустую «строку» абзацным тегом.

С метасимволом ⌜$⌟ ситуация во многом аналогичная, хотя возможность совпадения ⌜$⌟ определяется более сложными правилами (☞ 175). Впрочем, в контексте этого раздела расширенный режим просто относит позиции перед внутренними символами новой строки к числу тех, в которых может совпасть метасимвол ⌜$⌟.

В программах, поддерживающих этот режим, часто существуют метасимволы ⌜\A⌟ и ⌜\Z⌟, аналогичные ⌜^⌟ и ⌜$⌟ за одним исключением: многострочный режим на них *не* распространяется. Это означает, что ⌜\A⌟ и ⌜\Z⌟ никогда не совпадают с внутренними символами новой строки. Некоторые реализации также допускают совпадение ⌜$⌟ и ⌜\Z⌟ перед символом новой строки, завершающим текст. В таких реализациях часто поддерживается метасимвол ⌜\Z⌟, который игнорирует все символы новой строки и совпадает *только* в самом конце текста. Подробности приведены на с. 175.

Как и в случае с ⌜.⌟, из общего правила существуют исключения. Текстовые редакторы типа GNU Emacs обычно допускают совпадения якорных метасимволов с внутренними символами новой строки, поскольку в редакторе это вполне логично. С другой стороны, в *lex* метасимвол ⌜$⌟ совпадает только перед символами новой строки (хотя ⌜^⌟ сохраняет традиционное значение).

Системы с поддержкой Юникода (такие, как пакет `java.util.regex`) могут допускать совпадение якорных метасимволов с любыми завершителями строк (☞ 149). Якорные метасимволы Ruby обычно *совпадают* со всеми внутренними символами новой строки, а метасимвол Python ⌜\Z⌟ ведет себя как ⌜\z⌟, а не как обычный ⌜$⌟.

По традиции этот режим называется многострочным режимом. Хотя он не имеет отношения к «однострочному» режиму, из-за сходства названий возникает ошибочное впечатление, что эти режимы как-то связаны. Один режим изменяет возможности совпадения метасимвола ⌜.⌟, а другой влияет на совпадения ⌜^⌟ и ⌜$⌟. Другая проблема заключается в том, что эти два режима интерпретируют символы новой строки с разных точек зрения. В первом режиме метасимвол ⌜.⌟ переходит от «специальной» интерпретации символа новой строки к «обычной», а во втором происходит обратное — «обычная» интерпретация символов новой строки метасимволами ⌜^⌟ и ⌜$⌟ заменяется «специальной»[1].

Литеральный режим

В литеральном режиме не распознается большинство метасимволов (или все метасимволы) регулярных выражений. Например, регулярное выражение ⌜[a-z]*⌟ совпадает со строкой '[a-z]*'. Литеральный поиск эквивалентен простому поиску строки («найти эту строку» вместо «найти совпадение для этого регулярного выражения»), а программы с поддержкой регулярных выражений обычно предоставляют отдельную поддержку простого строкового поиска. Больший интерес представляет случай, когда литеральный режим применяется не ко всему регулярному выражению, а к его части. Например, в регулярных выражениях PCRE (т. е. в PHP) и Perl поддерживается специальная последовательность \Q…\E, внутри которой игнорируются все метасимволы (разумеется, кроме \E).

Стандартные метасимволы и возможности

Остальная часть этой главы — следующие 30 страниц — представляет собой обзор наиболее распространенных метасимволов и концепций современных регулярных выражений. Конечно, здесь не анализируются все существующие метасимволы, и ни одна программа не поддерживает всего, что здесь перечислено.

В определенном смысле это сводка того, что вы видели в двух начальных главах, но в свете более широкого и сложного мировоззрения, представленного в начале этой главы. Если вы впервые беретесь за этот раздел, можете на скорую руку просмотреть его и перейти к следующим главам. Вы сможете возвращаться к нему по мере необходимости.

---

[1] В Tcl точка обычно совпадает с любым символом, поэтому в каком-то смысле он «прямолинейнее» других языков. В регулярных выражениях Tcl символы новой строки не интерпретируются каким-либо особым образом (ни метасимволом ⌜.⌟, ни якорными метасимволами), но за счет использования режимов они могут стать особыми. Однако в других системах используется иной подход, поэтому у пользователей этих систем часто возникают трудности с Tcl.

Одни программы обрастают новыми полезными возможностями, другие по своей прихоти и в соответствии с внутренними требованиями изменяют стандартные правила. Эта глава в основном посвящена общим аспектам работы с регулярными выражениями, хотя иногда я буду давать комментарии по конкретным утилитам. В этом разделе я всего лишь пытаюсь описать некоторые распространенные метасимволы и область их применения, а также некоторые проблемы, о которых следует помнить. Во время чтения рекомендую держать под рукой руководство по той программе, с которой вы часто работаете.

Конструкции, рассматриваемые в этой части

Представления символов

с. 157	Сокращенные обозначения символов: **\n**, **\t**, **\a**, **\b**, **\e**, **\f**, **\r**, **\v**...
с. 159	Восьмеричные коды: *код*
с. 159	Шестнадцатеричные коды/Юникод: **\x***код*, **\x{***код***}**, **\u***код*, **\U***код*...
с. 161	Управляющие символы: **\c***символ*
Символьные классы и аналогичные конструкции	
с. 161	Обычные классы: **[a-z]** и **[^a-z]**
с. 162	Почти любой символ: точка
с. 163	Точно один байт: \C
с. 163	Комбинационные последовательности Юникода: \X
с. 164	Сокращенные обозначения классов: \w, \d, \s, \W, \D, \S
с. 164	Блоки, категории и свойства Юникода: \p{*свойство*},\P{*свойство*}
с. 170	Операции с классами: **[[a-z]&&[^aeiou]]**
с. 172	«Символьные классы» POSIX: **[[:alpha:]]**
с. 173	«Объединяющие последовательности» POSIX: **[[.span-11.]]**
с. 174	«Символьные эквиваленты» POSIX: **[[=n=]]**
с. 174	Синтаксические классы Emacs
Якорные метасимволы и другие проверки с нулевой длиной совпадения	
с. 175	Начало строки/логической строки: **^**, **\A**
с. 175	Конец строки/логической строки: **$**, **\Z**, **\z**
с. 177	Начало совпадения (или конец предыдущего совпадения): **\G**
с. 180	Границы слов: **\b**, **\B**, **\<**, **\>** ...
с. 181	Опережающая проверка **(?=...)**, **(?!...)**; ретроспективная проверка **(?<=...)**, **(?<!...)**

Комментарии и модификаторы режимов	
с. 182	Модификаторы режимов **(?***модификатор***)** — например, **(?i)** или **(?-i)**
с. 183	Интервальное изменение режима **(?***модификатор***:...)** — например, **(?i:...)**
с. 183	Комментарии: **(?#...)** и **#...**
с. 183	Литеральный текст: **\Q...\E**
Группировка, сохранение, условные и управляющие конструкции	
с. 184	Сохраняющие круглые скобки: **(...)**, **\1**, **\2**, ...
с. 185	Группирующие круглые скобки: **(?:...)**
с. 186	Именованное сохранение: **(?<***Имя***>...)**
с. 187	Атомарная группировка: **(?>...)**
с. 187	Конструкция выбора: ...|...|...
с. 188	Условная конструкция: **(?** *if then* **|** *else***)**
с. 189	Максимальные квантификаторы: *****, **+**, **?**, **{**min**,** max**}**
с. 190	Минимальные квантификаторы: ***?**, **+?**, **??**, **{**min**,** max**}?**
с. 190	Захватывающие квантификаторы: ***+**, **++**, **?+**, **{**min**,** max**}+**

Представления символов

Метасимволы этой группы обеспечивают наглядные средства для поиска символов, которые трудно представить иным образом.

Сокращенные обозначения символов

Во многих программах существуют метасимволы для представления машинно-зависимых управляющих символов, которые трудно вводить с клавиатуры или выводить на экран.

`\a` **Сигнал** (при «выводе» раздается звуковой сигнал). Обычно соответствует ASCII-символу `<BEL>`, код `007` (в восьмеричной системе).

`\b` **Забой**. Обычно соответствует ASCII-символу `<BS>`, код `010` (в восьмеричной системе). Обратите внимание: во многих диалектах ⌜`\b`⌟ интерпретируется как символ забоя только внутри символьных классов, а за их пределами — как граница слова (☞ 180).

`\e` **Символ Escape**. Обычно соответствует ASCII-символу `<ESC>`, код `033` (в восьмеричной системе).

Таблица 3.6. Сокращенные обозначения, поддерживаемые некоторыми программами

	\b граница слова	\b забой	\a сигнал	\e ASCII-символ Escape	\f подача листа	\n новая строка	\r возврат курсора	\t табуляция	\v вертикальная табуляция
Программа	**Сокращенные обозначения символов**								
Python	✓	✓$_C$	✓		✓	✓	✓	✓	✓
Tcl	как \y	✓	✓	✓	✓	✓	✓	✓	✓
Perl	✓	✓$_C$	✓	✓	✓	✓	✓	✓	
Java	✓$_X$	×$_X$	✓	✓	✓$_{SR}$	✓$_{SR}$	✓$_{SR}$	✓$_{SR}$	✓
GNU awk		✓	✓		✓	✓	✓	✓	✓
GNU sed	✓					✓			
GNU Emacs	✓	×$_S$	×$_S$	×$_S$	×$_S$	×$_S$	×$_S$	×$_S$	×$_S$
.NET	✓	✓$_C$	✓	✓	✓	✓	✓	✓	✓
PHP (функции preg)	✓	✓$_C$	✓	✓	✓	✓	✓	✓	
MySQL									
GNU grep/egrep	✓								
flex		✓	✓		✓	✓	✓	✓	✓
Ruby	✓	✓$_C$	✓	✓	✓	✓	✓	✓	✓

Информация о версиях программ приводится на с. 127.

✓ — поддерживается; ✓$_C$ — поддерживается только в классах;

✓$_{SR}$ — поддерживается (также поддерживается строковыми литералами);

✓$_X$ — поддерживается (но в строковых литералах последовательность интерпретируется иначе);

×$_X$ — не поддерживается (но в строковых литералах последовательность интерпретируется иначе);

×$_S$ — не поддерживается (но поддерживается строковыми литералами).

Для каждого приложения выбран тип строк, наиболее удобный при работе с регулярными выражениями (с. 140).

`\f` **Перевод формата**. Обычно соответствует ASCII-символу `<FF>`, код `014` (в восьмеричной системе).

`\n` **Новая строка**. На большинстве платформ (включая Unix и DOS/ Windows) обычно соответствует ASCII-символу `<LF>`, код `012` (в восьмеричной системе). В системе MacOS обычно соответствует ASCII-символу `<CR>`, код `015` (в вось-

меричной системе). В Java и языках .NET всегда соответствует ASCII-символу <LF> независимо от платформы.

\r **Возврат каретки**. Обычно соответствует ASCII-символу <CR>. В системе MacOS обычно соответствует ASCII-символу <LF>. В Java и языках .NET всегда соответствует ASCII-символу <CR> независимо от платформы.

\t **Обычная (горизонтальная) табуляция**. Обычно соответствует ASCII-символу <HT>, код 011 (в восьмеричной системе).

\v **Вертикальная табуляция**. Обычно соответствует ASCII-символу <VT>, код 013 (в восьмеричной системе).

В табл. 3.6 перечислены некоторые стандартные программы и поддерживаемые ими сокращенные обозначения управляющих символов. Как упоминалось выше, некоторые языки также поддерживают аналогичные обозначения для метасимволов строковых литералов. Обязательно ознакомьтесь с этим разделом (☞ 141), в котором описаны некоторые типичные проблемы, обусловленные наличием двух классов метасимволов.

Зависимость от операционной системы

Как видно из приведенного на предыдущей странице списка, символы \n и \r во многих программах[1] зависят от операционной системы, поэтому при их использовании необходимо тщательно обдумать свой выбор. Если вы хотите, например, чтобы символ обозначал «новую строку» во всех системах, где будет работать ваш сценарий, воспользуйтесь обозначением \n. Если вам нужен символ с конкретным значением байта (например, при программировании для стандартных протоколов типа HTTP), воспользуйтесь записью \012 или другим значением, соответствующим вашему стандарту (\012 — восьмеричный код символа). Если нужно найти символы завершения строк в DOS, используйте последовательность ⌜\015\012⌟. Конструкция ⌜\015?\012⌟ подходит для поиска символов завершения строк в DOS и Unix (обратите внимание: она совпадает именно с символами, чтобы получить совпадение *в позиции* начала или конца строки, используйте якорные метасимволы ☞ 175).

---

[1] Если программа написана на C или C++, то результат преобразования \-последовательностей регулярного выражения в \-последовательности C зависит от компилятора, поскольку стандартная библиотека C оставляет выбор кодов символов на усмотрение разработчика компилятора. На практике поддержка новой строки в компиляторах для любой платформы стандартизируется, поэтому можно смело считать, что выбор символов зависит от операционной системы. Более того, в разных системах различаются только символы \n и \r, поэтому остальные символы можно считать стандартными.

Восьмеричные коды: \число

Реализации, поддерживающие восьмеричную запись (т. е. запись чисел в системе счисления с основанием 8), в общем случае позволяют задавать байты и символы по их кодам, состоящим из двух или трех цифр. Например, ⌜`\015\012`⌟ соответствует последовательности ASCII-символов CR/LF. Восьмеричные коды позволяют легко вставлять в выражения символы, которые трудно вставить другим способом. Например, в Perl можно использовать для ASCII-символа Escape обозначение ⌜`\e`⌟, но в awk такая возможность отсутствует. Поскольку в awk поддерживаются восьмеричные коды, символ Escape можно вставить непосредственно в виде ASCII-кода: ⌜`\033`⌟.

В табл. 3.7 приведены сведения о поддержке восьмеричных кодов в некоторых программах.

В некоторых реализациях предусмотрен особый случай — совпадение ⌜`\0`⌟ с нуль-байтом. В других реализациях поддерживаются все восьмеричные коды, состоящие из одной цифры (как правило, только в том случае, если при этом не поддерживаются обратные ссылки вида ⌜`\1`⌟). При возникновении конфликтов неоднозначная последовательность обычно интерпретируется как обратная ссылка, а не как восьмеричный код. Некоторые реализации (как, например, `java.util.regex`) допускают восьмеричные коды из четырех цифр; это сделано для соблюдения правила, по которому любой восьмеричный код должен начинаться с `0`.

Вероятно, вас интересует, как интерпретируются недопустимые последовательности вида `\565` (8-разрядные восьмеричные коды лежат в интервале от `\000` до `\377`). Похоже, половина реализаций оставляет их в виде значения, выходящего за пределы байта (которое может совпасть с расширенным символом при поддержке Юникода), а другая половина урезает их до одного байта. В общем случае рекомендуется ограничиваться восьмеричными кодами из интервала до `\377`.

Шестнадцатеричные коды и Юникод: \хкод, \х{код}, \икод, \Икод

По аналогии с восьмеричными кодами во многих программах существует возможность ввода кодов в шестнадцатеричной системе счисления (с основанием 16) при помощи префиксов `\x`, `\u` и `\U`. Например, последовательность ⌜`\x0D\x0A`⌟ соответствует последовательности ASCII-символов CR/LF. В табл. 3.7 приведена информация о поддержке шестнадцатеричных кодов в некоторых популярных программах.

Наряду с выбором синтаксиса необходимо учитывать, сколько цифр может содержать код, а также возможно ли его заключение в скобки (или это является обязательным требованием). Эта информация тоже приводится в табл. 3.7.

Таблица 3.7. Поддержка восьмеричных и шестнадцатеричных кодов в регулярных выражениях некоторых программ

	Обратные ссылки	Восьмеричные коды	Шестнадцатеричные коды
Python	✓	`\0, \07, \377`	`\xFF`
Tcl	✓	`\0, \77, \777`	`\x... \uFFF; \UFFFFFFFF`
Perl	✓	`\0, \77, \777`	`\xF; \xFF; \x{...}`
Java	✓	`\07, \077, \0377`	`\xFF; \uFFFF`
GNU awk		`\7, \77, \377`	`\x...`
GNU sed	✓		
GNU Emacs	✓		
.NET	✓	`\0, \77, \377`	`\xFF; \uFFFF`
PHP (функции preg)	✓	`\0, \77, \377`	`\xF, \xFF, \x{...}`
MySQL			
GNU egrep	✓		
GNU grep	✓		
flex		`\7, \77, \377`	`\xF, \xFF`
Ruby	✓	`\7, \77, \377`	`\xF, \xFF`

`\0` — `\0` совпадает с нуль-байтом, но другие восьмеричные коды из одной цифры не поддерживаются;

`\7`, `\77` — поддерживаются восьмеричные коды из одной и двух цифр;

`\07` — поддерживаются восьмеричные коды из двух цифр, начинающиеся с 0;

`\077` — поддерживаются восьмеричные коды из трех цифр, начинающиеся с 0;

`\377` — поддерживаются восьмеричные коды из трех цифр до `\377`;

`\0377` — поддерживаются восьмеричные коды из четырех цифр до `\0377`;

`\777` — поддерживаются восьмеричные коды из трех цифр до `\777`;

`\x...` — `\x` допускает любое количество цифр;

`\x{...}` — `\x{...}` допускает любое количество цифр;

`\xF`, `\xFF` — `\x` поддерживает шестнадцатеричные коды из одной и двух цифр;

`\uFFFF` — `\u` поддерживает шестнадцатеричные последовательности из четырех цифр;

`\UFFFF` — `\U` поддерживает шестнадцатеричные последовательности из четырех цифр;

`\UFFFFFFFF` — `\U` поддерживает шестнадцатеричные последовательности из восьми цифр.

Информация о версиях программ приводится на с. 127.

Управляющие символы: \ссимвол

Во многих диалектах поддерживается последовательность ⌜\c*символ*⌟, предназначенная для идентификации управляющих символов с кодами, меньшими 32 (некоторые диалекты допускают более широкий интервал). Например, ⌜\cH⌟ совпадает с символом Control+H, представляющим в ASCII клавишу Backspace, а ⌜\cJ⌟ совпадает с символом перевода строки ASCII (этот символ часто представляется метасимволом ⌜\n⌟, но в зависимости от платформы также используется метасимвол ⌜\r⌟ (☞ 158).

Системы, поддерживающие эту конструкцию, различаются в технических деталях. Вы всегда можете с полной уверенностью использовать прописные английские буквы, как в приведенных выше примерах. В большинстве реализаций также допускается использование строчных английских букв, хотя они, например, не поддерживаются пакетом регулярных выражений для Java от Sun. Точная интерпретация неалфавитных символов практически полностью зависит от диалекта, поэтому я рекомендую использовать в конструкции \c только буквы верхнего регистра.

Отмечу, что данная возможность также поддерживается в GNU Emacs, но при этом используется громоздкая метапоследовательность ⌜?\^*символ*⌟ (например, ⌜?^H⌟ означает забой).

Символьные классы и их аналоги

В современных диалектах предусмотрено несколько способов определения набора символов, разрешенных в некоторой позиции регулярного выражения, однако простые символьные классы поддерживаются всеми программами.

Обычные классы: [a-z] и [^a-z]

Базовая концепция символьного класса уже рассматривалась выше, но позвольте мне снова подчеркнуть, что правила интерпретации метасимволов изменяются в зависимости от того, принадлежат они символьному классу или нет. Например, ⌜*⌟ никогда не является метасимволом в символьном классе, а ⌜-⌟ в общем случае интерпретируется как метасимвол. Интерпретация некоторых метапоследовательностей (таких, как ⌜\b⌟) в символьном классе иногда отличается от их интерпретации за пределами класса (☞ 156).

В большинстве систем в общем случае порядок перечисления символов в классе несущественен, а использование интервалов вместо списка не влияет на скорость обработки (т. е. [0-9] ничем не отличается от [9081726354]). Тем не менее не-

которые реализации (например, пакет регулярных выражений для Java от Sun) не обеспечивают полной оптимизации классов, поэтому в общем случае рекомендуется по возможности использовать интервалы, которые обычно работают быстрее.

Символьный класс всегда определяет *позитивное условие*. Другими словами, чтобы совпадение было успешным, символ в тексте должен совпасть с одним из перечисленных символов. Для инвертированных символьных классов совпавший символ должен быть одним из символов, *не* входящих в класс. Инвертированный символьный класс удобно рассматривать как «символьный класс с инвертированным списком» (обязательно ознакомьтесь с предупреждением об инвертированных символьных классах и точке в следующем разделе). Раньше конструкция вида `[^LMNOP]` была эквивалентной `[\x00-KQ-\xFF]`. В восьмиразрядных системах эта эквивалентность сохраняется и до сих пор, но в таких системах, как Юникод, в которых коды символов превышают 255 (`\xFF`), инвертированные классы вида `[^LMNOP]` включают многие тысячи символов кодировки — все, за исключением `L`, `M`, `N`, `O` и `P`.

При использовании интервалов необходимо хорошо знать кодировку, в которой определяется интервал. Например, интервал `[a-Z]` с большой вероятностью задан ошибочно и уж заведомо не ограничивается алфавитными символами. Одним из вариантов задания набора алфавитных символов является конструкция `[a-zA-Z]` (по крайней мере, для кодировки ASCII). Для Юникода см. описание конструкции `\p{L}` в подразделе «Свойства Юникода» на с. 164. При работе с двоичными данными вполне оправданно использование интервалов вида `\x80-\xFF`.

Точка — (почти) любой символ

В одних программах точка является сокращенным обозначением символьного класса, совпадающего с любым возможным символом, а в других — с любым символом, *кроме символа новой строки*. Это тонкое различие играет важную роль в программах, допускающих наличие нескольких логических строк в целевом тексте. В частности, приходится учитывать следующие аспекты.

- В некоторых системах с поддержкой Юникода (например, в пакете регулярных выражений для Java от Sun) точка в общем случае не совпадает с завершителями строк Юникода (☞ 149).
- Совпадение для точки может изменяться в зависимости от режима поиска (☞ 151).
- В стандарте POSIX сказано, что точка не совпадает с нуль-символом (т. е. символом, код которого равен нулю), но большинство сценарных языков допускает включение нуль-символов в текст (и точка совпадает с этими символами).

Точка и инвертированные символьные классы

При работе с программами, допускающими поиск в многострочном тексте, следует помнить, что точка обычно не совпадает с символом новой строки, но инвертированные классы типа ⌜[^"]⌟ обычно с этим символом совпадают. Таким образом, переход от ⌜".*"⌟ к ⌜"[^"]*"⌟ может преподнести сюрприз. Состав допустимых совпадений для точки можно изменить при помощи режима поиска — см. раздел «Совпадение точки со всеми символами («однострочный режим»)» на с. 152.

Точно один байт

В Perl и PCRE (т. е. в PHP) поддерживается метасимвол \C, который соответствует точно одному *байту*, даже если этот байт — часть последовательности, представляющей код одного *символа* (причем все остальные метасимволы продолжают работать именно с символами). Неправильное использование этого метасимвола может приводить к внутренним ошибкам, поэтому следует использовать его только в случае полного контроля над всеми последствиями. Я не могу представить себе ситуацию, когда этот метасимвол действительно мог бы пригодиться, поэтому далее упоминать о нем не буду.

Комбинационные последовательности символов Юникода: \X

Perl и PHP поддерживают метасимвол \X как сокращение шаблона ⌜\P{M}\p{M}*⌟, который представляет собой расширенный вариант ⌜.⌟. Он соответствует *базовому символу* (любому, не соответствующему \p{M}), за которым может следовать произвольное число *комбинационных символов* (все, что соответствует \p{M}).

Как уже говорилось ранее (☞ 149), Юникод определяет начертание отдельных символов с диакритическими знаками, например à (базовый символ ‘a’ U+0061 комбинируется с диакритическим знаком ‘`’ U+0300), как комбинацию из базового и комбинационного символов. Для получения требуемого результата может потребоваться использовать несколько комбинационных символов. Например, если потребуется создать символ ‘ç’, сделать это можно, вставив символ «c», за которым следуют седиль ‘¸’ и значок краткости ‘˘’ (U+0063 U+0327 U+0306).

При попытке отыскать совпадения со словами «francais» и «français» использование шаблона ⌜fran.ais⌟ или ⌜fran[cç]ais⌟ может не обеспечивать результат, поскольку в обоих шаблонах предполагается, что символу ‘ç’ соответствует один кодовый пункт Юникода U+00C7, и не учитывается, что этому символу может соответствовать символ ‘c’ со следующей за ним седилью (U+0063 U+0327). Для обеспечения точного совпадения можно было бы использовать шаблон ⌜fran(c¸?|ç)ais⌟, но в данном случае неплохой заменой для ⌜fran.ais⌟ мог бы служить шаблон ⌜fran\Xais⌟.

Кроме того, что \X соответствует любым комбинационным символам, существует еще два отличия \X от точки. Первое отличие заключается в том, что \X всегда соответствует символу новой строки и другим завершителям строк Юникода (☞ 149), тогда как поведение точки зависит от выбранного режима обработки регулярных выражений (☞ 152). Второе отличие состоит в том, что в однострочном режиме точка соответствует любому символу, тогда как ⌜\X⌟ не может соответствовать начальному комбинационному символу (не идущему вслед за базовым).

Сокращенные обозначения классов: \w, \d, \s, \W, \D, \S

В большинстве программ существуют следующие удобные сокращения для конструкций, которые обычно оформляются в виде класса.

\d	**Цифра**. Обычно эквивалентно ⌜[0-9]⌟ или в некоторых программах с поддержкой Юникода — любым цифрам Юникода
\D	**Не-цифра**. Обычно эквивалентно ⌜[^\d]⌟
\w	**Символ, входящий в слово**. Часто эквивалентно ⌜[a-zA-Z0-9_]⌟, хотя одни программы исключают символ подчеркивания, а другие дополняют его всеми алфавитно-цифровыми символами в соответствии с локальным контекстом (☞ 123). При поддержке Юникода \w обычно соответствует любому алфавитно-цифровому символу (важное исключение: в пакете регулярных выражений java.util.regex и в PCRE, расширении для PHP, метасимвол ⌜\w⌟ интерпретируется в точности как ⌜[a-zA-Z0-9_]⌟)
\W	**Символ, не входящий в слово**. Обычно эквивалентно ⌜[^\w]⌟
\s	**Пропуск**. В системах, ограничивающихся поддержкой ASCII, часто эквивалентно ⌜[\f\n\r\t\v]⌟. В системах с поддержкой Юникода также часто включает управляющий символ «следующей строки» U+0085, а иногда и свойство ⌜\p{Z}⌟ (см. ниже)
\S	**Не-пропуск** (обычно эквивалентно ⌜[^s]⌟)

Как говорилось на с. 123, локальный контекст POSIX может влиять на интерпретацию некоторых обозначений (особенно \w). В программах с поддержкой Юникода у метасимвола \w обычно имеется больше потенциальных совпадений — например, \p{L} с символом подчеркивания.

Свойства Юникода, алфавиты и блоки: \p{свойство}, \P{свойство}

На концептуальном уровне Юникод представляет собой отображение множества символов на множество кодов (☞ 146), но стандарт Юникода не сводится к простому перечислению пар. Он также определяет атрибуты символов (например,

«этот символ является строчной буквой», «этот символ пишется справа налево», «этот символ является диакритическим знаком, который должен объединяться с другим символом» и т. д.).

Уровень поддержки этих атрибутов зависит от конкретной программы, но многие программы с поддержкой Юникода позволяют находить хотя бы некоторые из них при помощи конструкций ⌜`\p{`*атрибут*`}`⌟ (символ, обладающий указанным атрибутом) и ⌜`\P{`*атрибут*`}`⌟ (символ, не обладающий атрибутом). Для примера рассмотрим конструкцию ⌜`\p{L}`⌟ (‘`L`’ — атрибут «буква» в отличие от цифр, знаков препинания и т. п.), которая представляет пример *общего свойства* (также называемого *категорией*). Вскоре вы познакомитесь с другими атрибутами, для проверки которых используются конструкции ⌜`\p{…}`⌟ и ⌜`\P{…}`⌟, но общие свойства встречаются чаще всего.

Общие свойства перечислены в табл. 3.8. Каждый символ (а точнее, каждый кодовый пункт, включая и те, для которых символы не определены) может быть идентифицирован по крайней мере по одному общему свойству. Имена общих свойств задаются одиночными символами (‘`L`’ — буква, ‘`S`’ — специальный знак и т. д.), но в некоторых системах существуют более информативные синонимы (‘`Letter`’, ‘`Symbol`’ и т. д.). В частности, они поддерживаются в Perl.

Таблица 3.8. Базовые свойства Юникода

Класс	Синоним и описание
`\p{L}`	`\p{Letter}` — символы, считающиеся буквами
`\p{M}`	`\p{Mark}` — различные символы, существующие не самостоятельно, а лишь в сочетании с другими базовыми символами (диакритические знаки, рамки и т. д.)
`\p{Z}`	`\p{Separator}` — символы, выполняющие функции разделителей, но не имеющие собственного визуального представления (разнообразные пробелы и т. д.)
`\p{S}`	`\p{Symbol}` — различные декоративные элементы и знаки
`\p{N}`	`\p{Number}` — цифры
`\p{P}`	`\p{Punctuation}` — знаки препинания
`\p{C}`	`\p{Other}` — прочие символы (редко используется при работе с обычным текстом)

В некоторых системах на однобуквенные имена свойств можно ссылаться без использования фигурных скобок (например, `\pL` вместо `\p{L}`). В отдельных системах требуется (или просто допускается) использовать префикс «`In`» или «`Is`» перед

буквой (например, `\p{IsL}`). При описании конкретных атрибутов будут приведены примеры ситуаций, когда префикс `In`/`Is` является обязательным[1].

Как показано в табл. 3.9, каждое однобуквенное общее свойство Юникода может делиться на несколько подсвойств, обозначаемых двумя буквами (табл. 3.9), например свойство «буква» может подразделяться на: «строчная буква», «прописная буква», «первая буква слова», «буква-модификатор» и «остальные буквы». Каждому из подсвойств соответствует точно один кодовый пункт.

Кроме того, в некоторых реализациях поддерживается специальное композитное подсвойство `\p{L&}`, которое обеспечивает сокращенную запись для всех букв, имеющих регистр, эквивалентное ⌜`\p{Lu}\p{Ll}\p{Lt}`⌟.

В табл. 3.9 включены полные имена синонимов (например, «`Lowercase_Letter`» вместо «`Ll`»), поддерживаемых некоторыми реализациями. Стандарт рекомендует поддерживать разные формы синонимов (например, '`LowercaseLetter`', '`LOWERCASE_LETTER`', '`Lowercase•Letter`', '`Lowercase-letter`' и т. д.), но по соображениям единства стиля желательно всегда использовать формы, приведенные в табл. 3.9.

Таблица 3.9. Базовые подсвойства Юникода

Свойство	Синоним и описание
`\p{Ll}`	`\p{Lowercase_Letter}` — строчные буквы
`\p{Lu}`	`\p{Uppercase_Letter}` — прописные буквы
`\p{Lt}`	`\p{Titlecase_Letter}` — буквы, находящиеся в начале слова (например, символ Dž в отличие от строчного символа dž и прописного символа DŽ)
`\p{L&}`	Сокращенная запись для всех символов, обладающих свойствами `\p{Lt}`, `\p{Ll}` и `\p{Lu}`
`\p{Lm}`	`\p{Modifier_Letter}` — небольшое подмножество специальных символов, по внешнему виду схожих с буквами
`\p{Lo}`	`\p{Other_Letter}` — буквы, не имеющие регистра и не являющиеся модификаторами, в том числе буквы иврита, бенгали, японского и тибетского языков и т. д.
`\p{Mn}`	`\p{Non_Spacing_Mark}` — «символы», изменяющие другие символы (акценты, умляуты, признаки тональности и т. д.)

[1] Как вы вскоре узнаете, в истории с префиксами `Is/In` много путаницы и недоразумений. В предыдущих версиях Юникода рекомендовалось одно, в ранних реализациях часто встречалось совсем другое. В процессе разработки Perl 5.8 я общался с группой программистов и постарался по возможности прояснить ситуацию в Perl. Теперь в Perl действует простое правило: «Вы не обязаны использовать префиксы '`Is`' и '`In`', за исключением работы с блоками Юникода (☞ с. 168); в этом случае должен использоваться префикс '`In`'.

Свойство	Синоним и описание
`\p{Mc}`	`\p{Spacing_Combining_Mark}` — модификаторы, занимающие собственное место в тексте (в основном обозначения гласных в тех языках, в которых они поддерживаются, включая бенгальский, гуджарати, тамильский, телугу, каннада, малаялам, синхала, мьянмар и кхмерский)
`\p{Me}`	`\p{Enclosing_Mark}` — небольшой набор знаков, внутри которых могут находиться другие символы (круги, квадраты, ромбы и буквицы)
`\p{Zs}`	`\p{Space_Separator}` — различные типы пробелов (обычный пробел, неразрывный пробел и пробелы фиксированной ширины)
`\p{Zl}`	`\p{Line_Separator}` — символ LINE SEPARATOR (U+2028)
`\p{Zp}`	`\p{Paragraph_Separator}` — символ PARAGRAPH SEPARATOR (U+2029)
`\p{Sm}`	`\p{Math_Symbol}` — +, /, ÷, ≷, ...
`\p{Sc}`	`\p{Currency_Symbol}` — $, ₡, ¥, €...
`\p{Sk}`	`\p{Modifier_Symbol}` — в основном разновидности комбинационных символов, но также включает ряд самостоятельных символов
`\p{So}`	`\p{Other_Symbol}` — различные декоративные символы, псевдографика, азбука Брайля и т. д.
`\p{Nd}`	`\p{Decimal_Digit_Number}` — цифры от 0 до 9 в различных алфавитах (кроме китайского, японского и корейского)
`\p{Nl}`	`\p{Letter_Number}` — в основном римские цифры
`\p{No}`	`\p{Other_Number}` — числа в верхних и нижних индексах, дроби; символы, представляющие числа, которые не являются цифрами (кроме китайского, японского и корейского алфавитов)
`\p{Pd}`	`\p{Dash_Punctuation}` — всевозможные дефисы и отрезки
`\p{Ps}`	`\p{Open_Punctuation}` — такие символы, как (⩔, ⟪, ...
`\p{Pe}`	`\p{Close_Punctuation}` — такие символы, как), ⩖, ⟫ ...
`\p{Pi}`	`\p{Initial_Punctuation}` — такие символы, как «, “, <, ...
`\p{Pf}`	`\p{Final_Punctuation}` — такие символы, как », ”, >, ...
`\p{Pc}`	`\p{Connector_Punctuation}` — знаки препинания, имеющие особый лингвистический смысл (например, символ подчеркивания)
`\p{Po}`	`\p{Other_Punctuation}` — остальные знаки препинания: !, &, ·, :, ⋮, ... и т. д.
`\p{Cc}`	`\p{Control}` — управляющие символы ASCII и Latin-1 (TAB, LF, CR, ...)
`\p{Cf}`	`\p{Format}` — неотображаемые символы, являющиеся признаками форматирования *(zero width joiner, activate Arabic form shaping, ...)*
`\p{Co}`	`\p{Private_Use}` — кодовые пункты, предназначенные для закрытого использования (логотипы компаний и т. д.)
`\p{Cn}`	`\p{Unassigned}` — кодовые пункты, которым не присвоены символы

Алфавиты

В некоторых системах поддерживается поиск совпадения с указанием имени *алфавита* (системы письменности), для чего используется синтаксис `\p{…}`. Например, конструкция `\p{Hebrew}` (если она поддерживается) совпадает с символами, входящими в набор символов иврита. При поиске по алфавиту не учитываются совпадения общих символов, которые также могут использоваться в других системах письменности (например, пробелы и знаки препинания).

Одни алфавиты применяются только в одном языке (`Gujarati`, `Thai`, `Cherokee`…), другие используются в нескольких языках (`Latin`, `Cyrillic`). В некоторых языках применяется несколько алфавитов, например, в японском языке используются символы из алфавитов `Hiragana`, `Katakana`, `Han` («Китайские иероглифы») и `Latin`. За полным списком алфавитов обращайтесь к документации по вашей системе.

Алфавит содержит не все символы некоторой письменности, а лишь те символы, которые используются исключительно (или преимущественно) в ней. Стандартные символы (например, пробелы и знаки препинания) ни к какому алфавиту не относятся. Считается, что они входят в универсальный псевдоалфавит `IsCommon`, совпадение с которым определяется конструкцией `\p{IsCommon}`. Второй псевдоалфавит, `Inherited`, состоит из различных комбинационных символов, наследующих алфавит от базового символа, за которым они следуют.

Блоки

Отдаленно похожи на алфавиты, но уступают им по возможностям. *Блок* определяет интервал кодовых пунктов в кодировке Юникод. Например, блок `Tibetan` относится к 256 кодовым пунктам с кодами от `U+0F00` до `U+0FFF`. В Perl и `java.util.regex` символы этого блока идентифицируются выражением `\p{InTibetan}`, а на платформе .NET — выражением `\p{IsTibetan}` (подробности будут приведены ниже).

В Юникоде определено много блоков: для большей части систем письменности (`Hebrew`, `Tamil`, `Basic_Latin`, `Hangul_Jamo`, `Cyrillic`, `Katakana`…) и для специальных категорий символов (`Currency`, `Arrows`, `Box_Drawing`, `Dingbats`…).

Блок `Tibetan` является исключительно удачным примером блока, поскольку все символы, определенные в нем, относятся к тибетскому языку, а за пределами блока не существует ни одного символа, специфического для этого языка. Тем не менее блоки уступают алфавитам по ряду характеристик:

- Блок может содержать кодовые пункты, которым не были назначены символы. Например, в блоке `Tibetan` свободны около 25 % кодовых пунктов.
- Не все символы, логически связанные с блоком, фактически входят в него. Например, блок `Currency` не содержит универсальный символ денежной единицы

‘¤’ и такие важные символы, как $, ¢, £, € и ¥ (к счастью, вместо него можно воспользоваться свойством \p{Sc}).

- В блоках часто имеются посторонние символы. Например, знак ¥ (символ йены) входит в блок Latin_1_Supplement.
- Символы одного *алфавита* иногда разделяются по нескольким блокам. Например, символы греческого языка присутствуют в блоках Greek и Greek_Extended.

Свойства блоков поддерживаются чаще, чем свойства алфавитов. Эти два понятия часто путают из-за сходства имен (например, в Юникоде поддерживается как алфавит Tibetan, так и блок Tibetan).

Более того, как видно из табл. 3.10, синтаксис имен еще не стандартизирован. В Perl и java.util.regex принадлежность к блоку Tibetan определяется выражением ⌜\p{InTibetan}⌟, а в .NET Framework используется конструкция ⌜\p{IsTibetan}⌟ (ситуация дополнительно усложняется тем, что в Perl поддерживается альтернативное представление для *алфавита* Tibetan).

Другие свойства и атрибуты

Далеко не все, о чем говорилось выше, поддерживается всеми программами. В табл. 3.10 приведена дополнительная информация о том, какие возможности присутствуют в наиболее популярных языках.

Таблица 3.10. Возможности свойств/алфавитов/блоков

Возможность	Perl	Java	.NET	PHP/ PCRE
✓ Базовые свойства типа \p{L}	✓	✓	✓	✓
✓ Короткие обозначения базовых свойств типа \pL	✓	✓		✓
Длинные обозначения типа \p{IsL}	✓	✓		
✓ Базовые свойства с полными именами типа \p{Letter}	✓			
✓ Регистровые символы \p{L&}	✓			✓
✓ Обозначения алфавитов типа \p{Greek}	✓			✓
Длинные обозначения алфавитов типа \p{IsGreek}	✓			
✓ Блоки типа \p{Cyrillic}	при отсутствии алфавита	✓		
✓ Длинные обозначения блоков типа \p{InCyrillic}	✓	✓		

Таблица 3.10 (*окончание*)

Возможность	Perl	Java	.NET	PHP/ PCRE
Длинные обозначения блоков типа `\p{IsCyrillic}`			✓	
✓ Инверсия `\P{…}`	✓	✓	✓	✓
Инверсия `\p{^…}`	✓			✓
✓ `\p{Any}`	✓	`\p{all}`		✓
✓ `\p{Assigned}`	✓	`\P{Cn}`	`\P{Cn}`	`\p{Cn}`
✓ `\p{Unassigned}`	✓	`\p{Cn}`	`\p{Cn}`	`\p{Cn}`

В левом столбце помечены возможности, которые рекомендуется поддерживать в новых реализациях (информация о версиях приводится на с. 127).

В Юникоде также определяются другие атрибуты, для работы с которыми используется конструкция ⌜`\p{…}`⌟. К их числу относятся направление вывода (слева направо, справа налево и т. п.), гласные звуки, ассоциированные с символами, и т. д. Некоторые реализации даже позволяют определять собственные свойства во время работы программы. Дополнительная информация приводится в документации по программе.

Простое вычитание классов: [[a-z]-[aeiou]]

Платформа .NET предоставляет возможность простого «вычитания» классов, что позволяет исключать из одного класса символы, входящие в другой класс. Например, классу ⌜`[[a-z]-[aeiou]]`⌟ будут соответствовать символы, соответствующие классу ⌜`[a-z]`⌟, за исключением символов, соответствующих классу ⌜`[aeiou]`⌟, т. е. согласные буквы латинского алфавита.

Еще один пример — классу ⌜`[\p{P}-[\p{Ps}\p{Pe}]]`⌟ будут соответствовать символы, соответствующие классу `\p{P}`, за исключением символов, соответствующих классу ⌜`[\p{Ps}\p{Pe}]`⌟, т. е. все знаки препинания, за исключением открывающих и закрывающих знаков препинания, таких как 》 и (.

Операции с классами: [[a-z]&&[^aeiou]]

Пакет регулярных выражений для Java от Sun поддерживает операции множеств с символьными классами (объединение, вычитание, пересечение). Синтаксис их несколько отличается от синтаксиса простого вычитания, описанного в предыдущем разделе (в частности, в языке Java синтаксис операции вычитания выглядит несколько странно, например множество всех согласных букв латинского алфавита определяется формулой `[[a-z]&&[^aeiou]]`). Прежде чем рассматривать операцию

вычитания в подробностях, познакомимся с двумя основными операциями над символьными классами — объединения (OR) и пересечения (AND).

Операция объединения включает в классы новые символы. Запись выглядит как определение класса внутри класса: выражение `[abcxyz]` также можно записать в виде `[[abc][xyz]]`, `[abc[xyz]]` или `[[abc]xyz]`. Операция объединения создает новое множество из всех элементов, присутствующих хотя бы в одном из множеств-аргументов. На концептуальном уровне она аналогична оператору «поразрядной дизъюнкции», который во многих языках представляется оператором '|' или 'or'. В символьных классах объединения в основном используются для удобства записи, хотя в некоторых ситуациях полезной оказывается возможность включения *инвертированных* классов.

Операция пересечения является концептуальным аналогом оператора «поразрядной конъюнкции». Итоговое множество состоит из элементов, присутствующих в обоих множествах-аргументах. Например, выражение `[\p{InThai}&&\P{Cn}]` совпадает с кодовыми пунктами блока `Thai`, с которыми *ассоциируются* символы. Для этого вычисляется пересечение (т. е. совокупность символов, входящих в каждое из множеств) между классами `\p{InThai}` и `\P{Cn}`. Как говорилось выше, `\P{…}` (с прописной P) совпадает со всем, что *не* обладает заданным атрибутом. Таким образом, `\P{Cn}` совпадает со всеми кодовыми пунктами, которые *не обладают* атрибутом отсутствия ассоциированных символов, а проще говоря, со всеми пунктами, у которых имеются *ассоциированные* символы (если бы разработчики Sun поддержали атрибут Assigned вместо `\P{Cn}` в данном примере, можно было бы использовать запись `\p{Assigned}`).

Будьте внимательны и не путайте пересечение с объединением. Интуитивность этих терминов зависит от точки зрения. Например, конструкция `[[this][that]]` обычно формулируется в виде «допускаются символы, совпадающие с `[this]` *или* `[that]`», но ее с таким же успехом можно прочитать как «список допустимых символов состоит из `[this]` *и* `[that]`». Две точки зрения на одно и то же явление.

Смысл операции пересечения более очевиден, например, выражение `[\p{InThai}&&\P{Cn}]` обычно читается как: «символы, соответствующие одновременно классу `\p{InThai}` *и* `\P{Cn}`», хотя его можно прочитать и как: «список символов, представляющий собой пересечение `\p{InThai}` и `\P{Cn}`».

Эти две различных точки зрения могут вносить путаницу: вместо использования союзов И и ИЛИ иногда бывает удобнее использовать термин ПЕРЕСЕЧЕНИЕ и союз И.

Вычитание

Возвращаясь к примеру `[\p{InThai}&&\P{Cn}]`, будет полезно заметить, что конструкция `\P{Cn}` эквивалентна `[^\p(Cn}]`, поэтому все выражение можно переписать в несколько усложненном виде: `[\p{InThai}&& [^\p{Cn}]]`. Более того,

категория «пункты с ассоциированными символами в блоке `Thai`» эквивалентна категории «все пункты блока `Thai` *за исключением* пунктов, *не имеющих ассоциированных* символов». Двойное отрицание слегка усложняет формулировку, но вы можете легко убедиться, что выражение `[\p{InThai}&&[^\p{Cn}]]` означает «`\p{InThai}` *минус* `\p{Cn}`».

Итак, мы возвращаемся к примеру ⌜`[[a-z]&&[^aeiou]]`⌟, с которого начинается этот раздел. В нем продемонстрирована операция *вычитания классов*. Суть состоит в том, что выражение ⌜`[`*this*`&&[^`*that*`]]`⌟ означает: «[*this*] минус [*that*]». Лично у меня от двойного отрицания `&&` и `[^…]` голова идет кругом, поэтому я предпочитаю просто запомнить шаблон ⌜`[…&&[^…]]`⌟.

Имитация операций с множествами с использованием опережающей проверки

Если ваша программа не поддерживает операции с множествами, но поддерживает опережающую проверку (☞ 181), желаемого результата можно добиться другим способом. Выражение ⌜`[\p{InThai}&&[^\p{Cn}]]`⌟ можно записать в виде ⌜`(?!\p{Cn})\p{InThai}`⌟[1]. Хотя такое решение уступает по эффективности нормально реализованным операциям с множествами, опережающая проверка обладает достаточно гибкими возможностями. Этот пример можно записать в четырех разных вариантах (в .NET `InThai` заменяется на `IsThai`, ☞ 171):

```
(?!\p{Cn})\p{InThai}
(?=\P{Cn})\p{InThai}
\p{InThai}(?<!\p{Cn})
\p{InThai}(?<=\P{Cn})
```

Групповые выражения в стандарте POSIX: [[:alpha:]]

То, что мы обычно называли *символьным классом*, в стандарте POSIX было решено назвать *групповым выражением* (*bracket expression*). В POSIX термин «символьный класс» относится к специальной конструкции, используемой *внутри* группового выражения[2], которую можно рассматривать как своего рода предшественника свойств символов в Юникоде.

---

[1] Вообще говоря, в Perl этот конкретный пример можно записать просто в виде ⌜`\p{Thai}`⌟, поскольку в этом языке `\p{Thai}` является *алфавитом*, который заведомо не содержит пунктов без ассоциированных символов. Между алфавитом и блоком `Thai` также существуют и более тонкие различия. Например, в данном случае в алфавите отсутствуют некоторые специальные символы, входящие в блок. Полезно иметь под рукой документацию с описанием того, что именно входит в тот или иной блок или алфавит. Все необходимые подробности вы можете найти на сайте *http://unicode.org*.

[2] Обычно в этой книге термины «символьный класс» и «групповое выражение POSIX» используются как синонимы, описывающие конструкцию в целом, а термин «символьный класс POSIX» относится к особой псевдоинтервальной конструкции, описанной в этом разделе.

Символьный класс POSIX представляет собой одну из нескольких специальных метапоследовательностей, используемых внутри групповых выражений в стандарте POSIX. Примером является конструкция `[:lower:]`, соответствующая любой букве нижнего регистра в текущем локальном контексте (☞ 124). Для обычного английского текста конструкция `[:lower:]` означает интервал `a-z`. Поскольку вся метапоследовательность действительна только внутри группового выражения, класс, сравнимый с ⌜`[a-z]`⌟, имеет вид ⌜`[[:lower:]]`⌟. Да, это выглядит уродливо, но предоставляет дополнительную возможность включения других символов — ö, ñ и т. д. (если в локальном контексте они действительно являются символами «нижнего регистра»).

Точный список символьных классов POSIX зависит от локального контекста, но следующие конструкции поддерживаются в большинстве систем.

`[:alnum:]`	алфавитные символы и цифровые символы
`[:alpha:]`	алфавитные символы
`[:blank:]`	пробел и табуляция
`[:cntrl:]`	управляющие символы
`[:digit:]`	цифры
`[:graph:]`	отображаемые символы (не пробелы, не управляющие символы и т. д.)
`[:lower:]`	алфавитные символы нижнего регистра
`[:print:]`	аналог `[:graph:]`, но включает пробел
`[:punct:]`	знаки препинания
`[:space:]`	все пропуски (`[:blank:]`, символ новой строки, возврат курсора и т. д.)
`[:upper:]`	алфавитные символы верхнего регистра
`[:xdigit:]`	цифры, допустимые в шестнадцатеричных числах (т. е. 0-9a-fA-F)

Системы, в которых поддерживаются свойства Юникода (☞ 165), могут по своему усмотрению расширять поддержку этих свойств до конструкций POSIX. Обычно предпочтение отдается свойствам Юникода, поскольку они обладают большими возможностями, нежели их аналоги в POSIX.

«Объединяющие последовательности» в групповых выражениях POSIX: [[.span-ll.]]

В локальном контексте могут определяться *объединяющие последовательности*, описывающие интерпретацию некоторых символов или совокупностей символов при сортировке и других операциях. Например, в испанском языке символы `ll` (как, например, в слове *tortilla*), традиционно сортируются, как если бы это был одиночный символ между `l` и `m`, а символ ß в немецком языке находится между `s` и `t`, но сортиру-

ется как два символа ss. Эти правила выражаются объединяющими последовательностями, которым, например, могут быть присвоены имена span-ll и eszet.

В механизме регулярных выражений, соответствующем стандарту POSIX, объединяющие последовательности, отображающие несколько физических символов на один логический символ (как в span-ll), рассматриваются как один символ. Это означает, что символьный класс типа ⌜[^abc]⌟ совпадет с последовательностью 'll'.

Для включения элементов объединяющих последовательностей в групповые выражения используется обозначение **[.….]**: выражение ⌜torti[[.spanll.]]a⌟ совпадает с tortilla. Объединяющая последовательность позволяет осуществлять сравнение символов, которые представляют собой комбинации других символов. Кроме того, становятся возможными ситуации, при которых групповое выражение совпадает с последовательностью из нескольких физических символов!

«Символьные эквиваленты» в групповых выражениях POSIX: [[=n=]]

В некоторых локальных контекстах определяются *символьные эквиваленты* (*character equivalents*), указывающие, что какие-то из символов должны считаться идентичными при выполнении сортировки и других аналогичных операций. Например, локальный контекст может определить класс-эквивалент 'n', содержащий символы 'n' и 'ñ', или класс 'a', содержащий символы 'a', 'à' и 'á'. Используя запись, аналогичную приведенной выше конструкции **[:…:]**, и заменив двоеточия знаками равенства, можно ссылаться на классы-эквиваленты в групповых выражениях; например, ⌜**[[=n=][=a=]]**⌟ совпадает с любым из перечисленных символов.

Если символьный эквивалент с однобуквенным именем используется, но не определяется в локальном контексте, он по умолчанию совпадает с объединяющей последовательностью с тем же именем. Локальные контексты обычно содержат объединяющие последовательности для всех обычных символов ([.a.], [.b.], [.c.] и т. д.), поэтому при отсутствии специальных эквивалентов конструкция ⌜**[[=n=][=a=]]**⌟ по умолчанию считается идентичной ⌜**[na]**⌟.

Синтаксические классы Emacs

В GNU Emacs и его семействе не поддерживаются традиционные метасимволы ⌜\w⌟, ⌜\s⌟ и т. д.; вместо них используется специальный синтаксис ссылок на «синтаксические классы»:

\s*символ*	совпадает с символами, принадлежащими синтаксическому классу Emacs, определяемому заданным *символом*
\S*символ*	совпадает с символами, не принадлежащими синтаксическому классу Emacs

Например, ⌜\sw⌟ означает «символ, входящий в слово», а ⌜\s-⌟ — «символ-пропуск». В большинстве других систем они записываются в виде ⌜\w⌟ и ⌜\s⌟.

Особенность синтаксических классов Emacs заключается в том, что точный состав входящих в них символов может изменяться во время работы программы. Например, концепция символов, входящих в слова, может изменяться в зависимости от типа редактируемого файла.

Якорные метасимволы и другие проверки с нулевой длиной совпадения

Якорные метасимволы и другие «проверки с нулевой длиной совпадения» совпадают не с реальными символами, а с позициями в тексте.

Начало физической или логической строки: ^, \A

Метасимвол ⌜^⌟ совпадает в начале текста, в котором производится поиск, или в расширенном режиме привязки к границам строк (☞ 153) — в позиции после любого символа новой строки. В некоторых системах в расширенном режиме ⌜^⌟ также может совпадать с завершителями строк Юникода (☞ 149).

Метасимвол ⌜\A⌟ (если он поддерживается) совпадает только в начале текста независимо от режима поиска.

Конец физической или логической строки: $, \Z, \z

Как видно из табл. 3.11, концепция «конца строки» несколько сложнее парной ей концепции начала строки. В разных реализациях ⌜$⌟ интерпретируется по-разному. В самом распространенном варианте он совпадает с концом целевого текста перед *завершающим* символом новой строки. Это сделано для того, чтобы выражения вида ⌜s$⌟ («строка, заканчивающаяся символом s») совпадали с комбинациями символов '...sNL' в тексте, когда строки завершаются буквой s и символом новой строки.

В других распространенных интерпретациях ⌜$⌟ совпадает только в конце всего фрагмента или перед любым символом новой строки. В некоторых системах с поддержкой Юникода символы новой строки в этих правилах заменяются завершителями строк Юникода (☞ 149). (Например, язык Java реализует самую сложную семантику ⌜$⌟ в отношении завершителей Юникода, ☞ 462.)

Выбор режима (☞ 153) может повлиять на интерпретацию ⌜$⌟, в результате он будет соответствовать любым промежуточным символам новой строки в блоке текста (а также любым завершителям Юникода).

Таблица 3.11. Якорные метасимволы в некоторых сценарных языках

	Java	Perl	PHP	Python	Ruby	Tcl	.NET
Обычно...							
^ совпадает в начале строки	✓	✓	✓	✓	✓	✓	✓
^ совпадает после любого символа новой строки					✓2		
$ совпадает в конце текста	✓	✓	✓	✓	✓	✓	✓
$ совпадает перед символом новой строки, завершающим текст	✓1	✓	✓	✓	✓		✓
$ совпадает перед *любым* символом новой строки					✓2		
Поддерживает расширенный режим привязки к границам строк (☞ 153)	✓	✓	✓	✓	✓		✓
В расширенном режиме привязки к границам строк...							
^ совпадает в начале строки	✓	✓	✓	✓	—	✓	✓
^ совпадает после любого символа новой строки	✓1	✓	✓	✓	—	✓	✓
$ совпадает в конце текста	✓	✓	✓	✓	—	✓	✓
$ совпадает перед *любым* символом новой строки	✓1	✓	✓	✓	—	✓	✓
\A всегда совпадает как обычный ^	✓	✓	✓	✓	°4	✓	✓
\Z всегда совпадает как обычный $	✓1	✓	✓	°3	°5	✓	✓
\z всегда совпадает только в конце строки	✓	✓	✓	—	—	✓	✓

Индексы обозначают:

1 — пакет регулярных выражений для Java от Sun поддерживает *завершители строк* Юникода (☞ 150);

2 — в Ruby метасимволы $ и ^ совпадают на внутренних символах новой строки, а метасимволы \A и \Z — нет;

3 — в Python метасимвол \Z совпадает только в конце строки;

4 — в Ruby метасимвол \A, в отличие от ^, совпадает только в начале строки;

5 — в Ruby метасимвол \Z, в отличие от $, совпадает в конце строки или перед символом новой строки, завершающим строку.

(Информация о версиях приведена на с. 127.)

Метасимвол ⌜\Z⌟ (если он поддерживается) обычно соответствует метасимволу ⌜$⌟ в стандартном режиме, что часто означает совпадение в конце текста или перед символом новой строки, завершающим текст. Парный метасимвол ⌜\z⌟ совпадает только в конце строки без учета возможных символов новой строки. Некоторые исключения перечислены в табл. 3.11.

Начало совпадения (или конец предыдущего совпадения): \G

Метасимвол ⌜\G⌟ впервые появился в Perl и предназначался для проведения глобального поиска с модификатором /g (☞ 82). Он совпадает с позицией, в которой завершилось предыдущее совпадение. При первой итерации \G совпадает только в начале строки, как и метасимвол ⌜\A⌟.

Если попытка поиска завершилась неудачей, позиция ⌜\G⌟ возвращается в начало строки. Таким образом, при многократном применении регулярного выражения (как при использовании команды ⌜s/…/…/g⌟ или других средств глобального поиска) неудача при очередном поиске приводит к сбросу позиции ⌜\G⌟ для применения следующего выражения.

В Perl метасимвол ⌜\G⌟ обладает тремя уникальными особенностями, которые мне кажутся весьма интересными и полезными.

- Позиция, связанная с ⌜\G⌟, является атрибутом *целевой строки*, а не регулярного выражения. Это позволяет последовательно применить к строке несколько регулярных выражений, причем метасимвол ⌜\G⌟ в каждом из них продолжит поиск с той позиции, на которой он был прерван для предыдущего выражения.
- Операторы регулярных выражений Perl поддерживают модификатор /c (☞ 395), при помощи которого можно указать, что в случае неудачного поиска позиция ⌜\G⌟ *не* сбрасывается, а остается в последней позиции. В сочетании с первым пунктом это помогает организовать проверку по нескольким регулярным выражениям с одной позиции целевого текста, которая смещается только при обнаружении совпадения.
- Текущую позицию метасимвола ⌜\G⌟ можно получить и изменить с использованием средств, не имеющих отношения к регулярным выражениям (функция pos ☞ 394). Например, ее можно задать так, чтобы поиск начинался с заданной позиции. Кроме того, поддержка этой возможности в языке позволяет имитировать функциональность предыдущего пункта, если она не поддерживается напрямую.

Примеры практического использования этих возможностей приводятся во врезке на следующей странице. Несмотря на наличие этих удобных возможностей, у метасимвола ⌜\G⌟ в Perl имеется недостаток — он работает надежно лишь в том случае, если находится в самом начале регулярного выражения. К счастью, именно там он используется наиболее естественным образом.

НЕТРИВИАЛЬНОЕ ИСПОЛЬЗОВАНИЕ \G В PERL

Ниже приведен фрагмент, который выполняет простейшую проверку кода HTML в переменной `$html` и убеждается в том, что переменная содержит конструкции из очень ограниченного подмножества команд HTML (допускаются простые теги `<IMG>` и `<A>`, а также простые подстановки типа `>`). Я использовал нечто похожее на Yahoo! для проверки того, что представленный пользователем код HTML удовлетворяет некоторым требованиям.

Работа программы в значительной степени зависит от поведения оператора Perl `m/…/gc`, который применяет регулярное выражение к целевой строке и продолжает поиск от конца последнего успешного совпадения, но *не* сбрасывает эту позицию в случае неудачи (☞ 396).

Благодаря этой особенности выражения, используемые в следующем примере, «коллективно» обрабатывают строку. Теоретически происходящее напоминает одну большую конструкцию выбора с перечислением всех выражений в качестве альтернатив, но приведенное решение позволяет выполнить фрагмент кода для каждого совпадения, а также включать и исключать выражения во время выполнения программы.

```
my $need_close_anchor = 0; # True, если ранее был обнаружен тег <A>,
                           # для которого еще не найден парный тег </A>.

while (not $html =~ m/\G\z/gc) # Пока не будут обработаны все символы
{                              # до конца строки
  if ($html =~ m/\G(\w+)/gc) {
      ... В $1 теперь хранится слово или число – например,
      можно проверить, не содержит ли текст нецензурные выражения ...
  } elsif ($html =~ m/\G[^<>&\w]+/gc) {
          # Прочие конструкции, не входящие в синтаксис HTML, игнорируются.
  } elsif ($html =~ m/\G<img\s+([^>]+)>/gci) {
      ... Обнаружен тег IMG – проверить его правильность ...
      ⋮
  } elsif (not $need_close_anchor and $html =~ m/\G<A\s+([^>]+)>/gci){
      ... Найден тег ссылки – проверить его ...
      ⋮
      $need_close_anchor = 1; # Теперь нужен закрывающий тег </A>
  } elsif ($need_close_anchor and $html =~ m{\G</A>}gci){
      $need_close_anchor = 0; # Получили необходимое, сбросить флаг
  } elsif ($html =~ m/\G&(#\d+|\w+);/gc){
      # Разрешить подстановки вида &gt; и &#123
  } else {
      # Совпадения не найдены - вероятно, произошла ошибка.
      # Извлечь дюжину символов из кода HTML и использовать их,
      # чтобы сообщение об ошибке стало более содержательным.
      my $location = pos($html); # Запомнить, где начинается
                                 # непредвиденный код HTML.
```

```
        my ($badstuff) = $html =~ m/\G(.{1,12})/s;
        die "Unexpected HTML at position $location: $badstuff\n";
    }
}

# Убедиться, что в коде HTML не осталось незакрытого тега <A>
if ($need_close_anchor) {
    die "Missing final </A>"
}
```

Конец предыдущего или начало текущего совпадения?

Одно из различий между реализациями определяется тем, с чем же в действительности совпадает ⌜\G⌟ — с началом текущего или с концом предыдущего совпадения? В подавляющем большинстве случаев это одно и то же, но в некоторых редких случаях эти две позиции могут различаться. Вполне реальная ситуация описана на с. 277, но суть проблемы проще понять на искусственном примере: применении выражения ⌜x?⌟ к строке 'abcde'. Регулярное выражение может успешно совпасть в позиции '▴abcde', но текст в совпадении не участвует. При глобальном поиске с заменой регулярное выражение применяется повторно, причем если не принять особых мер, — с той позиции, в которой закончилось предыдущее совпадение. Чтобы предотвратить зацикливание, при обнаружении подобной позиции механизм регулярных выражений принудительно переходит к следующему символу ⌜\G⌟ (☞ 198). В этом легко убедиться, применив команду s/x?/!/g к строке 'abcde', — вы получите строку '!a!b!c!d!e!'.

Один из побочных эффектов подобного перевода позиции заключается в том, что «конец предыдущего совпадения» после этого отличается от «начала текущего совпадения». В таких случаях возникает вопрос: с какой из двух позиций совпадает ⌜\G⌟? В Perl применение команды s\Gx?/!/g к строке 'abcde' порождает строку '!abcde', следовательно, в Perl ⌜\G⌟ на самом деле совпадает только с концом предыдущего совпадения. При искусственном смещении позиции ⌜\G⌟ гарантированно терпит неудачу.

С другой стороны, применение той же команды в ряде других программ приводит к результату '!a!b!c!d!e', из чего следует, что \G совпадает в начале каждого текущего совпадения и решение об этом принимается *после* искусственного смещения позиции.

В решении этого вопроса не всегда можно доверять документации, прилагаемой к программе. Я обнаружил, что в документации Microsoft .NET и пакета java.util.regex были приведены неверные сведения, о чем я сообщил разработчикам (после чего данное упущение было исправлено). Тестирование показало, что в PHP

и Ruby ⌜\G⌟ соответствует началу текущего совпадения, а в Perl, `java.util.regex` и языках .NET — концу предыдущего совпадения.

Границы слов: \b, \B, \<, \>, ...

Эти метасимволы совпадают не с символом, а с определенной позицией в строке. Существует два разных подхода. В одном позиция *начала* и *конца слова* обозначается разными метасимволами (обычно \< и \>), а в другом определяется единая метапоследовательность (обычно \b). В обоих случаях также обычно определяется метапоследовательность для любой позиции, *не являющейся границей слова* (обычно \B). Некоторые примеры приведены в табл. 3.12. Программы, которые не различают якорные метасимволы для начала и конца слова, но поддерживают опережающую проверку, могут имитировать их при помощи опережающей проверки. В таблице приведены соответствующие варианты возможного синтаксиса.

Таблица 3.12. Метасимволы границ слов в некоторых программах

Программа	Начало слова ... Конец слова	Граница слова	Не граница слова
GNU awk	\<…\>	\y	\B
GNU *egrep*	\<…\>	\b	\B
GNU Emacs	\<…\>	\b	\B
Java	(?<!\pL)(?=\pL)…(?<=\pL)(?!\pL)	\b☒	\B☒
MySQL	[[:<:]]…[[:>:]]	[[:<:]]|[[:>:]]	
.NET	(?<!\w)(?=\w)…(?<=\w)(?!\w)	\b	\B
Perl	(?<!\w)(?=\w)…(?<=\w)(?!\w)	\b	\B
PHP	(?<!\pL)(?=\pL)…(?<=\pL)(?!\pL)	\b☒	\B☒
Python	(?<!\w)(?=\w)…(?<=\w)(?!\w)	\b	\B
Ruby		\b☒	\B☒
GNU sed	\<\>	\b	\B
Tcl	\m\M	\y	\Y

Элементы, отмеченные ☒, работают только с символами ASCII (или с локальными 8-битовыми кодировками), даже если данный диалект имеет поддержку Юникода. (Информация о версиях приведена на с. 127.)

Граница слова обычно определяется как позиция, с одной стороны от которой находится «символ слова», а с другой — символ, не относящийся к этой категории. У каждой программы имеются свои представления о том, что следует считать «символом слова» с точки зрения границ слов. Было бы логично, если бы границы слова определялась по метасимволам \w, но это не всегда так. Например, в пакете

регулярных выражений `java.util.regex` и PHP `\w` относится только к символам кодировки ASCII вместо Юникода, поддерживаемого в Java, поэтому в таблице используется опережающая проверка со свойством Юникода `\pL` (которое является сокращенной формой `「\p{L}」`; ☞ 165).

В любом случае проверка границы слова всегда сводится к проверке соседних символов. Ни один механизм регулярных выражений не принимает решений на основе лексического анализа — строка «NE14AD8» везде считается словом, а «M.I.T.» к словам не относится.

Опережающая проверка (?=...), (?!...); ретроспективная проверка (?<=...), (?<!...)

Конструкции опережающей и ретроспективной проверки были рассмотрены на достаточно обширном примере в разделе «Разделение разрядов числа запятыми» предыдущей главы (☞ 91). Однако при этом совершенно не упоминался один важный вопрос — какие выражения могут использоваться в обеих разновидностях ретроспективной проверки? В большинстве реализаций устанавливается ограничение на длину текста при ретроспективной проверке (но для опережающей проверки длина текста не ограничивается!).

Самые жесткие правила действуют в Perl и Python, где ретроспективная проверка ограничивается строками фиксированной длины. Например, конструкции `(?<!\w)` и `(?<!this|that)` допустимы, а `(?<!books?)` и `(?<!^\w+:)` запрещены, поскольку они могут совпадать с текстом переменной длины. В таких случаях, как с `(?<!books?)`, задача решается выражением `「(?<!book)(?<!books)」`, но понять его с первого взгляда весьма непросто.

На следующем уровне поддержки в ретроспективной проверке допускаются альтернативы разной длины, поэтому выражение `(?<!books?)` может быть записано в виде `(?<!book|books)`. PCRE (и все функции preg в PHP) это позволяют.

На следующем уровне допускаются регулярные выражения, совпадающие с текстом переменной, но конечной длины. Например, выражение `(?<!books?)` разрешено, а выражение `(?<!^\w+:)` запрещено, поскольку совпадение `\w+` может иметь непредсказуемую длину. Этот уровень поддерживается пакетом регулярных выражений для Java от Sun.

Справедливости ради стоит заметить, что первые три уровня в действительности эквивалентны, потому что все они могут быть выражены, хотя и несколько громоздко, средствами минимального уровня (с фиксированной длиной). Остальные уровни — всего лишь «синтаксическая обертка», позволяющая сделать то же самое более удобным способом. Впрочем, существует и четвертый уровень, при котором подвыражение в ретроспективной проверке может совпадать с произвольным объ-

емом текста, поэтому выражение `(?<!^\w+:)` является разрешенным. Этот уровень, поддерживаемый языками платформы .NET от Microsoft, принципиально превосходит остальные уровни, однако его неразумное применение может серьезно замедлить работу программы (столкнувшись с ретроспективой, которая может совпадать с текстом произвольной длины, механизм вынужден проверять условие от начала строки, что требует огромного объема лишней работы при проверке в конце длинного текста).

Комментарии и модификаторы режимов

Во многих диалектах режимы поиска и форматирования, о которых говорилось выше (☞ 150), могут переключаться самим регулярным выражением во время обработки (так сказать, на лету). Ниже перечислены конструкции, которые при этом используются.

Модификаторы режима: (?модификатор), например (?i) или (?-i)

Многие современные диалекты позволяют менять режимы поиска иформатирования (☞ 150) непосредственно на уровне регулярного выражения. Самый распространенный пример — конструкции ⌜`(?i)`⌟ (включение поиска без учета регистра) и ⌜`(?i)`⌟ (отключение). Например, в выражении ⌜`<B>(?i)very(?-i)</B>`⌟ совпадение для ⌜`very`⌟ ищется без учета регистра символов, но в совпадении для имен тегов регистр учитывается. Например, выражение может совпасть с '`<B>VERY</B>`' и '`<B>Very</B>`', но не с '`<b>Very</b>`'.

Приведенный пример работает в большинстве систем с поддержкой ⌜`(?i)`⌟, включая Perl, `java.util.regex`, Ruby[1] и языки .NET. Он не будет работать ни в Python, ни в Tcl, которые не поддерживают ⌜`(?-i)`⌟.

В большинстве реализаций действие конструкции ⌜`(?i)`⌟, заключенной в круглые скобки, ограничивается этими скобками (т. е. за пределами скобок режим автоматически отключается). Это позволяет просто убрать ⌜`(?-i)`⌟ из выражения и поместить ⌜`(?i)`⌟ сразу после открывающей скобки: ⌜`<B>(?:(?i)very)</B>`⌟.

Возможности модификации режимов не ограничиваются поиском без учета регистра '`i`'. В большинстве систем поддерживаются модификаторы, перечисленные в табл. 3.13. Отдельные системы вводят дополнительные символы для расширения

[1] Этот пример работает в Ruby, однако реализация (?i) содержит ошибку, вследствие которой не работает с альтернативами, разделенными ⌜`|`⌟, набранными в нижнем регистре (однако работает, если альтернативные варианты указаны в верхнем регистре).

круга решаемых задач. В частности, существует множество дополнительных модификаторов в PHP (☞ 555), а также в Tcl (см. документацию к языку).

Таблица 3.13. Стандартные модификаторы режимов

Обозначение	Режим
i	Поиск без учета регистра символов (☞ 151)
x	Свободное форматирование и комментарии (☞ 151)
s	Режим совпадения точки со всеми символами (☞ 152)
m	Расширенный режим привязки к границам строк (☞ 153)

Интервальное изменение режима: (?модификатор:...), например (?i:...)

В системах, поддерживающих интервальные модификаторы, пример из предыдущего раздела можно дополнительно упростить. Интервальные модификаторы имеют синтаксис ⌜`(?i:…)`⌟ и активизируют режим только для части выражения, заключенной в круглые скобки. В этом случае пример ⌜`<B>(?:(?i)very)</B>`⌟ упрощается до ⌜`<B>(?i:very)</B>`⌟.

Там, где этот синтаксис поддерживается, он обычно может использоваться со всеми буквенными обозначениями модификаторов. В Tcl и Python поддерживается форма ⌜`(?i)`⌟, но отсутствует интервальный модификатор ⌜`(?i:…)`⌟.

Комментарии: (?#...) и #...

Некоторые диалекты поддерживают комментарии в форме ⌜`(?#…)`⌟. На практике этот синтаксис используется редко, обычно программисты предпочитают режим свободного форматирования (☞ 151). Тем не менее этот тип комментариев удобен в языках, синтаксис которых затрудняет включение символов новой строки в строковые литералы, например VB.NET (☞ 137, 515).

Литеральный текст: \Q...\E

Специальная последовательность `\Q…\E` впервые появилась в Perl. Все метасимволы, следующие после `\Q`, интерпретируются как литералы (разумеется, кроме `\E`; если последний отсутствует, специальная интерпретация метасимволов подавляется до конца регулярного выражения). Таким образом, последовательности символов, обычно воспринимаемые как метасимволы, интерпретируются как обычный текст. Данная возможность чаще всего используется при подстановке содержимого переменных при построении регулярных выражений.

Допустим, при поиске в сети критерий, полученный от пользователя, был сохранен в переменной `$query`, и вы хотите провести поиск по этому критерию командой `m/$query/i`. Если `$query` содержит текст вроде '`C:\WINDOWS\`', это приведет к неожиданным результатам — произойдет ошибка времени выполнения, поскольку в условии поиска присутствует конструкция, недопустимая в регулярном выражении (завершающий одиночный символ `\`).

В Perl проблема решается командой `m/\Q$query\E/i`; '`C:\WINDOWS\`' фактически превращается в ⌜`C:\\WINDOWS\\`⌟, а в результате поиска будет найден текст '`C:\WINDOWS\`', как и ожидает пользователь.

Данная возможность приносит меньше пользы в системах с процедурным и объектно-ориентированным интерфейсом (☞ 132), поскольку в них регулярные выражения определяются в виде обычных строк. При построении строки можно без особого труда вызвать функцию, которая «защищает» содержимое переменной и делает его безопасным для использования в регулярном выражении. Например, в VB для этой цели вызывается метод `Regex.Escape` (☞ 538), в PHP используется функция `preg_quote` (☞ 587), в Java — метод `quote` (☞ 493).

В настоящее время из всех известных мне механизмов регулярных выражений полная поддержка ⌜`\Q...\E`⌟ реализована только в пакете `java.util.regex` от Sun и в PCRE (т. е. в семействе функций *preg* языка PHP). Но ведь я только что упомянул, что конструкция ⌜`\Q...\E`⌟ впервые появилась в Perl (а также привел пример на Perl) — так почему Perl не включен в это утверждение? Дело в том, что Perl поддерживает ⌜`\Q...\E`⌟ в *литералах* (регулярных выражениях, задаваемых непосредственно в программе), но не в содержимом переменных, которые могут в них интерполироваться. За подробностями обращайтесь к главе 7 (☞ 364).

Пакет `java.util.regex` поддерживает использование конструкции ⌜`\Q...\E`⌟ внутри символьных классов. Реализация ее в версиях Java до 1.6.0 содержит ошибку, поэтому данная конструкция не должна использоваться в этих версиях Java.

Группировка, сохранение, условные и управляющие конструкции

Сохраняющие круглые скобки: (...) и \1, \2, ...

Стандартные круглые скобки обычно выполняют две функции: группировку и сохранение. Они почти всегда включаются в выражения в виде ⌜`(...)`⌟, но в ряде диалектов используется запись ⌜`\(...\)`⌟. К их числу относятся диалекты GNU Emacs, *sed*, *vi* и *grep*.

Сохраняющие скобки идентифицируются по порядковому номеру открывающей скобки от левого края выражения, как показано на с. 70, 75 и 88. Если в диалек-

те поддерживаются *обратные ссылки*, то на текст, совпавший с подвыражением в круглых скобках, можно ссылаться в том же регулярном выражении при помощи метасимволов ⌜\1⌟, ⌜\2⌟ и т. д.

Чаще всего круглые скобки применяются для извлечения данных из строки. Текст, совпавший с подвыражением в круглых скобках (для краткости — «текст, совпавший с круглыми скобками»), остается доступным после применения выражения, при этом в разных программах используется разный синтаксис (например, в Perl — переменные `$1`, `$2` и т. д.). Многие программисты ошибочно пытаются использовать ⌜\1⌟ *за пределами* регулярного выражения; это возможно только в *sed* и *vi*. В табл. 3.14 показано, какой синтаксис используется для работы с совпадающим текстом в ряде программ — как для всего регулярного выражения, так и для определенной пары круглых скобок.

Таблица 3.14. Доступ к тексту совпадений в некоторых программах

Программа	Все совпадение	Первая пара круглых скобок
GNU *egrep*	—	—
GNU Emacs	`(match-string 0)` (`\&` внутри строки замены)	`(match-string 1)` (`\1` внутри строки замены)
GNU awk	`substr($`*текст*`, RSTART, RLENGTH)`	—
MySQL	—	—
Perl (☞ 70)	`$&`	`$1`
PHP (☞ 560)	`$`*matches*`[0]`	*$matches*`[1]`
Python (☞ 135)	*MatchObj*`.group(0)`	*MatchObj*`.group(1)`
Ruby	`$&`	`$1`
GNU sed	`\&` (только при замене)	`\1` (только при замене)
Java (☞ 132)	*MatcherObj*`.group( )`	*MatcherObj*`.group(1)`
Tcl	присваиваются пользовательским переменным командой `regexp`	
VB.NET (☞ 134)	*MatcherObj*`.Groups(0)`	*MatcherObj*`.Groups(1)`
C#	*MatcherObj*`.Groups[0]`	*MatcherObj*`.Groups[1]`
vi	`&`	`\1`
(Информация о версиях приведена на ☞ 128.)		

Группирующие круглые скобки: (?:...)

Группирующие круглые скобки ⌜`(?:…)`⌟ не сохраняют совпавший текст, а лишь группируют компоненты регулярных выражений в конструкции выбора и при применении квантификаторов. Группирующие круглые скобки не учитываются при нуме-

рации переменных $1, $2 и т. д. Например, после успешного совпадения выражения ⌜(1|one)(?:and|or)(2|two)⌟ переменная $1 содержит '1' или 'one', а $2 содержит '2' или 'two'. Группирующие круглые скобки также называются *несохраняющими*.

Несохраняющие круглые скобки полезны по целому ряду причин. Они упрощают применение сложных регулярных выражений, позволяя читателю не думать о том, будет ли совпавший текст использоваться в других местах конструкциями типа $1. Кроме того, они повышают эффективность — механизму регулярных выражений не нужно сохранять совпавший текст, поэтому он быстрее работает и расходует меньше памяти (вопросы эффективности подробно рассматриваются в главе 6).

Несохраняющие круглые скобки также используются при построении регулярных выражений из отдельных компонентов. Вспомните пример на с. 110, где в переменной $HostnameRegex хранилось регулярное выражение для поиска имени хоста. Теперь представьте, что для извлечения пропусков из имени хоста используется следующая команда Perl: m/(\s*)$HostnameRegex(\s*)/. Можно предположить, что после этого начальные пропуски будут храниться в переменной $1, а конечные пропуски — в переменной $2, но это не так: конечные пропуски окажутся в переменной $4, поскольку в определении $HostnameRegex используются две пары *сохраняющих* круглых скобок:

```
$HostnameRegex = qr/[-a-z0-9]+(\.[-az09]+)*\.(com|edu|info)/i;
```

Если бы скобки в этом определении были несохраняющими, использование переменной $HostnameRegex не привело бы к этому сюрпризу. Другое решение проблемы, хотя и недоступное в Perl, основано на использовании именованных сохранений, описанных в следующем подразделе.

Именованное сохранение: (?<Имя>...)

Python и языки .NET позволяют сохранить текст, совпавший с круглыми скобками, под заданным именем. В Python используется синтаксис ⌜(?P<имя>...)⌟, а в языках .NET — синтаксис (?<имя>...) (лично мне этот вариант нравится больше). Пример для .NET:

```
⌜\b(?<Area>\d\d\d)-(?<Exch>\d\d\d)-(?<Num>\d\d\d\d)\b⌟
```

и для Python/PHP:

```
⌜\b(?P<Area>\d\d\d\)-(?P<Exch>\d\d\d)-(?P<Num>\d\d\d\d)\b⌟
```

Приведенное выражение ассоциирует компоненты телефонного номера с именами *Area*, *Exch* и *Num*, после чего на совпавший текст можно ссылаться по имени. Так, в VB.NET и большинстве других языков .NET используется синтаксис *RegexObj*.Groups("Area"), в C# — *RegexObj*.Groups["Area"], в Python —

RegexObj.groups("Area"), а в PHP — $*matches*["Area"]. Работа с данными по именам делает программу более понятной.

Внутри регулярного выражения ссылки на совпавший текст имеют вид ⌜\k<Area>⌟ в .NET и ⌜(?P=Area)⌟ в Python и PHP.

В Python и .NET (но не в PHP) допускается многократное использование имен в выражениях. Например, если код междугородной связи в телефонном номере записывается в виде '(###)' или '###-', для его поиска можно воспользоваться следующим выражением (использован синтаксис .NET): ⌜…(?:\((?<Area>\d\d\d)\)|(?<Area>\d\d\d))…⌟. Какая бы из альтернатив ни совпала, код из трех цифр будет ассоциирован с именем *Area*.

Атомарная группировка: (?>…)

Конструкция атомарной группировки ⌜(?>…)⌟ легко объясняется после знакомства с основными принципами работы механизмов регулярных выражений (☞ 223). Сейчас я скажу только, что текст, совпадающий с подвыражением в круглых скобках, фиксируется (становится атомарным, т. е. неизменяемым) для оставшейся части совпадения, если позднее не окажется, что все совпадение для атомарных круглых скобок не должно быть отброшено или вычислено заново. Неделимую, «атомарную» природу совпадения проще всего пояснить конкретным примером.

Регулярное выражение ⌜¡.*!⌟ совпадает с текстом '¡Hola!', но эта строка не совпадет, если выражение .* заключено в конструкцию атомарной группировки: ⌜¡(?>.*)!⌟. В обоих случаях ⌜.*⌟ сначала распространяется на текст максимальной длины ('¡Hola!'), но в первом случае восклицательный знак заставляет ⌜.*⌟ уступить часть найденных символов (последний символ '!'), чтобы обеспечить общее совпадение. Во втором случае ⌜.*⌟ находится внутри атомарной конструкции, которая никогда и ничего не уступает; на долю завершающего знака '!' ничего не остается, поэтому общее совпадение невозможно.

Атомарная группировка находит ряд важных применений, хотя из данного примера трудно сделать подобный вывод. В частности, атомарная группировка повышает эффективность поиска (☞ 225) и позволяет с высокой степенью точности управлять тем, что может или не может считаться совпадением (☞ 413).

Выбор: …|…|…

Конструкция выбора проверяет наличие совпадений для нескольких подвыражений с заданной позиции. Каждое подвыражение называется *альтернативой*. Символ ⌜|⌟ можно называть по-разному, но чаще всего его называют ИЛИ или *вертикальная черта*. В некоторых диалектах вместо символа ⌜|⌟ используется метапоследовательность ⌜\|⌟.

Практически во всех диалектах конструкция выбора обладает очень низким приоритетом. Это означает, что выражение ⌜`this and|or that`⌟ эквивалентно ⌜`(this and)|(or that)`⌟, а не ⌜`this (and|or) that`⌟, хотя визуально последовательность `and|or` воспринимается как единое целое.

В большинстве диалектов допускаются пустые альтернативы вида ⌜`(this|that|␣)`⌟. Пустое подвыражение означает «совпадение всегда», так что пример логически сравним с `(this|that)`⌟?.[1]

Стандарт POSIX запрещает пустые альтернативы, они не поддерживаются в *lex* и большинстве версий awk. На мой взгляд, такая запись полезна хотя бы из соображений удобства и наглядности. Как однажды объяснил Ларри Уолл, она «выполняет те же функции, как ноль в системе счисления».

Условная конструкция: (?if then | else)

Условная конструкция позволяет реализовать стандартную логику *if/then/else* на уровне регулярного выражения. Часть *if* содержит особую разновидность условного выражения, рассматриваемую ниже, а *then* и *else* представляют собой обычные подвыражения. Если условие *if* истинно, применяется выражение *then*, а если ложно — выражение *else*. (Часть *else* может отсутствовать, в этом случае символ '|' также необязателен.)

Круг допустимых значений *if* зависит от диалекта, но в большинстве реализаций поддерживаются как минимум специальные ссылки на сохраняющие подвыражения и конструкции позиционной проверки.

Использование специальных ссылок в сохраняющих круглых скобках для проверки

Когда часть *if* представляет собой число в круглых скобках, условие считается истинным, если пара сохраняющих круглых скобок с заданным номером уже участвовала в совпадении к настоящему моменту. В следующем примере ищется совпадение для тега HTML <IMG>, отдельного или заключенного между тегами `<A>…</A>`. Выражение приведено в режиме свободного форматирования с комментариями, а условная конструкция (которая в данном случае не содержит части *else*) выделена полужирным шрифтом:

```
( <A\s+[^>]+> \s* )?   # Совпадение для открывающего тега <A>, если он есть
<IMG\s+[^>]+>          # Совпадение для тега <IMG>
```

[1] Строго говоря, выражение ⌜`(this|that|)`⌟ логически эквивалентно ⌜`((?:this|that)?)`⌟. Незначительная разница с ⌜`(this|that)?`⌟ состоит в том, что «совпадение с ничем» не находится *внутри* скобок. Различие на первый взгляд кажется несущественным, но в некоторых языках есть разница между совпадением с ничем и неучастием в совпадении.

```
(?(1)\s*</A>)          # Совпадение для закрывающего тега <A>, если ранее
                       # было найдено совпадение для <A>
```

Условие `(1)` в ⌜`(?(1)…)`⌟ проверяет, участвовала ли первая пара сохраняющих круглых скобок в совпадении. «Участие в совпадении» принципиально отличается от «фактического совпадения с текстом», как наглядно показывает следующий пример.

Рассмотрим два выражения для представления слова, заключенного в необязательные угловые скобки '<…>'. Решение ⌜`(<)?\w+(?(1)>)`⌟ работает, а решение ⌜`(<?)\w+(?(1)>)`⌟ — нет. Между ними существует единственное различие — позиция первого вопросительного знака. В первом (правильном) выражении вопросительный знак относится к сохраняющим скобкам, поэтому сами скобки (и все, что находится внутри них) не обязательны. Во втором (ошибочном) решении сохраняющие скобки обязательны — не обязательно лишь совпадение < *внутри* них, поэтому они «участвуют в совпадении» независимо от того, совпал символ '<' или нет. Это означает, что условие *if* в выражении ⌜`(?(1)…)`⌟ всегда истинно.

При поддержке именованных сохранений (☞ 186) вместо номера в скобках обычно может указываться ассоциированное имя.

Использование позиционной проверки

В условиях *if* также могут использоваться полные конструкции позиционной проверки, такие как ⌜`(?=…)`⌟ и ⌜`(?<=…)`⌟. Если для проверяемого выражения находится совпадение, условие считается истинным и тогда применяется часть *then*. В противном случае применяется часть *else*. Сказанное можно пояснить несколько запутанным примером: ⌜`(?(?<=NUM:)\d+|\w+)`⌟. Приведенное выражение пытается найти совпадение для ⌜`\d+`⌟ в позициях после ⌜`NUM:`⌟, а в других позициях ищется совпадение для ⌜`\w+`⌟. Условие позиционной проверки подчеркнуто.

Другие способы проверки условий

В Perl условная конструкция обладает интересной особенностью: в качестве условия может быть задан произвольный фрагмент кода Perl. Возвращаемое значение этого фрагмента определяет, какая часть условной конструкции должна применяться (*then* или *else*). Дополнительная информация приводится в главе 7 на с. 410.

Максимальные квантификаторы: *, +, ?, {min, max}

Квантификаторы (`*`, `+`, `?` и интервальная конструкция — метасимволы, определяющие *количество* экземпляров повторяющегося элемента) рассматривались в одной из предшествующих глав. Следует помнить, что в некоторых программах вместо ⌜`+`⌟ и ⌜`?`⌟ используются ⌜`\+`⌟ и ⌜`\?`⌟. Кроме того, в отдельных программах квантифика-

торы не могут применяться к обратным ссылкам или выражениям, заключенным в скобки.

Интервалы: {min, max} или \{min, max\}

Интервальный квантификатор «ведет счет» найденных экземпляров совпадения. Он определяет наименьшее количество *обязательных* и наибольшее количество *допустимых* экземпляров. Если указывается только одно число (⌜`[a-z]{3}`⌟ или ⌜`[a-z]\{3\}`⌟ в зависимости от диалекта) и этот синтаксис поддерживается программой, совпадает в точности заданное количество экземпляров. Этот пример эквивалентен ⌜`[a-z][a-z][a-z]`⌟, хотя в разных программах эффективность их применения различается (☞ 319).

Предупреждение: не стоит полагать, что конструкция вида ⌜`X{0,0}`⌟ означает «здесь не должно быть X». Выражение ⌜`X{0,0}`⌟ бессмысленно, поскольку оно означает: «ни один экземпляр ⌜`X`⌟ *не обязателен*, так что можно даже не пытаться их искать». Это равносильно тому, что конструкция ⌜`X{0,0}`⌟ вообще отсутствует — если даже элемент `X` и есть, он может совпасть с одной из следующих частей выражения, поэтому исходный смысл этой конструкции полностью утрачивается.[1] Для проверки условия типа «здесь не должно быть» следует использовать негативную опережающую проверку.

Минимальные квантификаторы: *?, +?, ??, {min, max}?

В некоторых программах поддерживаются нескладные конструкции `*?`, `+?`, `??` и {*min*,*max*}`?`. Они представляют собой *минимальные* версии квантификаторов. Обычные квантификаторы руководствуются критерием максимального совпадения и пытаются найти совпадение как можно большей длины. Минимальные версии ведут себя совершенно противоположным образом и ищут совпадение наименьшей длины, удовлетворяющее заданному критерию. Различия между двумя разновидностями квантификаторов имеют ряд принципиальных аспектов, подробно рассматриваемых в следующей главе (☞ 210).

Захватывающие квантификаторы: *+, ++, ?+, {min, max}+

Захватывающие (*possessive*) квантификаторы в настоящее время поддерживаются только пакетом `java.util.regex` и PCRE (т. е. PHP), но можно предположить, что

[1] Теоретически все сказанное о {0,0} верно. На практике дело обстоит еще хуже — последствия почти непредсказуемы! В некоторых программах (включая GNU *awk*, GNU *grep* и старые версии Perl) конструкция `{0,0}` эквивалентна *, а во многих других (включая большинство известных мне версий *sed* и некоторые версии *grep*) она эквивалентна `?`. Полное безумие!

они получат более широкое распространение в будущем. *Захватывающие квантификаторы* ведут себя как обычные максимальные квантификаторы, но после того, как им будет назначено совпадение, они ни при каких условиях не «возвращают» захваченные символы. Как и в случае с атомарной группировкой, захватывающие квантификаторы гораздо проще понять при хорошем знании общих принципов поиска совпадения (эта тема рассматривается в следующей главе).

В определенном смысле захватывающие квантификаторы относятся к «синтаксическим удобствам», поскольку их можно имитировать при помощи атомарной группировки. Конструкция вида `.++` в точности эквивалентна `(?>.+)`, хотя грамотные реализации оптимизируют захватывающие квантификаторы лучше, чем атомарную группировку (☞ 318).

Путеводитель по серьезным главам

После знакомства с метасимволами, диалектами, синтаксической оберткой и т. д. настало время перейти к техническим подробностям третьего фактора, упоминаемого в начале главы, — непосредственному процессу поиска совпадения для регулярного выражения в тексте. Как будет показано в главе 4 «Механика обработки регулярных выражений», существуют разные варианты реализации механизма, обеспечивающего поиск совпадений, и от выбора того или иного варианта зависит, будет найдено совпадение или нет, *какой* текст в строке совпадет и *сколько времени* займет поиск. Мы рассмотрим все подробности. Заодно этот материал поможет вам более уверенно строить сложные выражения. Глава 5 «Практические приемы построения регулярных выражений» поможет укрепить новые знания на базе нетривиальных примеров.

После этого будет естественным переход к теме главы 6 «Построение эффективных регулярных выражений». Разобравшись в основах работы механизма, мы рассмотрим приемы, позволяющие извлечь максимум пользы из этих знаний. В главе 6 описаны недочеты некоторых распространенных диалектов регулярных выражений (нередко приводящие к неприятным сюрпризам) и рассказано, как обратить их себе на пользу.

Главы 4, 5 и 6 являются истинным стержнем этой книги. Первые три главы всего лишь подводили нас к главной теме, а описание конкретных программ в последующих главах основано на этом материале. Центральные главы никак не назовешь «легким чтивом», но я постарался держаться подальше от алгебры, математики и всего того, что для большинства из нас выглядит китайской грамотой. Вероятно, как обычно бывает при получении большого объема информации, для усвоения новых знаний потребуется некоторое время.

4 Механика обработки регулярных выражений

Предыдущая глава начиналась с аналогии между автомобилями и регулярными выражениями. В ней обсуждались возможности, диалекты и другие аспекты регулярных выражений из разряда публикуемых в рекламных проспектах. В данной главе, в продолжение этой аналогии, мы перейдем к самому важному — механизму обработки регулярных выражений и что происходит, когда он выполняет свою работу.

Зачем нам задумываться о том, как работает поиск? Как мы увидим дальше, самый распространенный из существующих типов механизмов регулярных выражений (тот, который используется в Perl, Tcl, Python, в языках .NET, Ruby, PHP, во всех известных мне пакетах Java и т. д.) работает таким образом, что *сама структура* регулярного выражения может повлиять *на то, совпадет ли* оно в некоторой строке, *где именно* совпадет и *насколько быстро* будет найдено совпадение или принято решение о неудаче. Если эти вопросы важны для вас, значит, для вас и написана эта глава.

Запустить двигатели!

Давайте посмотрим, сколько я еще смогу выжать из своей аналогии с двигателями. Как известно, двигатель нужен для того, чтобы вы могли без особых хлопот попасть из точки А в точку Б. Двигатель работает, а вы расслабляетесь и наслаждаетесь музыкой. Главная задача двигателя — вертеть колеса, а как он это делает, вас не волнует. Но так ли это?

Два вида двигателей

Представьте, что в вашем распоряжении находится электромобиль. Электромобили появились относительно давно, но еще не получили такого распространения, как автомобили с бензиновым двигателем. Но если вы все же пользуетесь электромо-

билем, по крайней мере нужно помнить, что в него не стоит заливать бензин. С бензиновым двигателем дело обстоит сложнее. Электрический двигатель более или менее работает сам по себе, а бензиновому двигателю может потребоваться уход. Небольшая регулировка зазора между электродами свечи, воздушного фильтра или переход на другой сорт бензина — каждый из этих факторов способен существенно улучшить работу двигателя. Но стоит вам ошибиться, и двигатель начнет работать хуже или попросту заглохнет.

Каждый двигатель работает по-своему, но конечный результат один и тот же: колеса крутятся. Если вы захотите куда-нибудь добраться, вам придется еще и крутить руль, но это уже совсем другая история.

Новые стандарты

Теперь давайте усложним ситуацию и добавим в нее новую переменную: калифорнийский стандарт, определяющий допустимый состав выхлопных газов.[1] Одни двигатели соответствуют жестким стандартам, принятым в Калифорнии, другие не соответствуют им, причем это не разные двигатели, а всего лишь разновидности уже существующих моделей. Стандарт регулирует результат работы двигателя — состав выхлопных газов, но ничего не говорит о том, как заставить двигатель работать чище. Итак, два класса двигателей делятся на четыре типа: электрические (соответствующие и несоответствующие) и бензиновые (соответствующие и несоответствующие).

Готов поспорить, что электрический двигатель пройдет проверку без всяких изменений, так как в этом случае стандарт всего лишь «одобряет» уже имеющийся результат. С другой стороны, бензиновый двигатель может потребовать значительной доработки и настройки. Владельцы таких двигателей должны обратить особое внимание на топливо — стоит воспользоваться неправильным сортом бензина, и у вас начнутся большие неприятности.

Влияние стандартов

Экологические стандарты — дело хорошее, но они требуют от водителя большой внимательности и предусмотрительности (по крайней мере, для бензиновых двигателей). Впрочем, этот стандарт не влияет на большинство автомобилистов, поскольку в других штатах действуют собственные законы и калифорнийские стандарты в них не соблюдаются.

---

[1] В Калифорнии установлены довольно жесткие стандарты, регулирующие допустимую степень загрязнения окружающей среды автомобилем. Из-за этого многие автомобили продаются в Америке в двух моделях: «для Калифорнии» и «не для Калифорнии».

Итак, четыре типа двигателей можно разделить на три группы (две группы для бензиновых двигателей, одна для электрических). Вы знаете, чем они отличаются и что в конечном счете все двигатели должны крутить колеса. Правда, вы можете задаться вопросом, какое отношение все сказанное имеет к регулярным выражениям. Гораздо большее, чем вы можете себе представить.

Типы механизмов регулярных выражений

Существует два принципиально разных типа механизмов регулярных выражений: ДКА («электрический») и НКА («бензиновый»). Смысл этих сокращений будет объяснен ниже (☞ 207), а пока просто считайте их техническими терминами (как «электрический» и «бензиновый»).

Оба типа механизмов существуют довольно давно, но тип НКА, как и его бензиновый аналог, встречается чаще. К числу программ, использующих механизм НКА, принадлежат языки .NET, PHP, Ruby, Perl, Python, GNU Emacs, *ed*, sed, *vi*, большинство версий *grep* и даже некоторые версии *egrep* и awk. С другой стороны, механизм ДКА реализован практически во всех версиях *egrep* и awk, а также в *lex* и *flex*. В некоторых системах используется гибридный подход, при котором для каждой ситуации выбирается соответствующий тип двигателя (а иногда переключение происходит даже между разными частями одного выражения для достижения оптимальной комбинации широты возможностей и быстродействия). В табл. 4.1 перечислены некоторые распространенные программы с указанием механизма регулярных выражений, используемого большинством версий. Если ваша любимая программа отсутствует в списке, возможно, информация о ней приводится в разделе «Определение типа механизма».

Таблица 4.1. Некоторые программы и их механизмы регулярных выражений

Тип механизма	Программы
ДКА	awk (большинство версий), *egrep* (большинство версий), *flex*, *lex*, MySQL, Procmail
Традиционный НКА	GNU Emacs, Java, *grep* (большинство версий), *less*, *more*, языки .NET, библиотека PCRE, Perl, PHP (все три семейства функций для работы с регулярными выражениями), Python, Ruby, sed (большинство версий), *vi*
POSIX НКА	*mawk*, утилиты от Mortice Kern Systems, GNU Emacs (по запросу)
Гибридный НКА/ДКА	GNU awk, GNU *grep/egrep*, Tcl

Как было показано в главе 3, двадцать лет эволюции регулярных выражений привели к излишнему разнообразию, постепенно переросшему в хаос. Поэтому

появился стандарт POSIX, который должен был прояснить ситуацию и четко определить, какие метасимволы должны поддерживаться механизмом регулярных выражений и каких *результатов при этом следует ожидать*. Если опустить второстепенные подробности, тип ДКА (наши электрические двигатели) достаточно хорошо соответствовал стандарту, но результаты, традиционно обеспечиваемые НКА, несколько отличались от требований нового стандарта, поэтому потребовалось внести некоторые изменения. В результате механизмы регулярных выражений стали делиться на три типа.

- ДКА (соответствующие стандарту POSIX или нет — практически одно и то же).
- Традиционные НКА (наиболее типичные представители: Perl, .NET, PHP, Java, Python, ...).
- НКА стандарта POSIX.

В данном случае термин «POSIX» относится к *семантике* поиска — предполагаемой последовательности обработки регулярного выражения в соответствии со стандартом POSIX (мы вернемся к этой теме позже в настоящей главе). Не путайте его с употреблением термина «POSIX» в случаях, связанных с *возможностями* регулярных выражений, определяемыми в этом же стандарте (☞ 173). Многие программы ограничиваются поддержкой конкретных возможностей, не поддерживая полной семантики поиска.

Старые программы, обладавшие минимальными возможностями, такие как *egrep*, awk и *lex*, обычно реализовывались на базе «электрического» механизма ДКА, поэтому новый стандарт просто закрепил существующее положение вещей. Тем не менее *существовали* «бензиновые» версии этих программ, и чтобы обеспечить соответствие стандарту POSIX, в них приходилось вносить некоторые изменения. Механизмы, удовлетворяющие «калифорнийскому стандарту» (POSIX НКА), были хороши тем, что выдаваемые ими результаты были стандартными, однако от внесенных изменений они стали более капризными. Если раньше можно было жить с небольшими отклонениями в зазоре электродов, сейчас это было абсолютно нетерпимо. От бензина, который раньше был «достаточно хорош», мотор теперь чихал и останавливался. Но если вы умели ухаживать за своей крошкой, двигатель работал гладко. И притом чисто.

С позиций избыточности

А сейчас я попрошу вас вернуться на пару страниц назад и перечитать историю о двигателях. Каждое из приведенных там высказываний относится и к регулярным выражениям. При втором прочтении возникают некоторые вопросы. В частности, что означает «электрический двигатель более или менее работает сам по себе»?

Какие характеристики влияют на работу «бензиновых» механизмов НКА? Как настроить НКА? Какие особенности присущи POSIX НКА? И что означает «заглохший двигатель» в мире регулярных выражений?

Определение типа механизма

Поскольку тип механизма, используемого программой, в значительной мере определяет поддерживаемые возможности, а также особенности их работы, во многих случаях мы относим механизм к какому-то типу, прогнав его на нескольких тестовых выражениях (а если вы не сможете найти ни одного различия, тогда так ли важны эти различия?). На данном этапе я не ожидаю, что вы разберетесь в том, почему результаты приведенных тестов показывают именно то, что я им приписываю. Я предлагаю положиться на мой выбор, и, если ваша любимая программа отсутствует в табл. 4.1, рекомендую вам провести самостоятельное тестирование и определить тип механизма перед тем, как продолжать чтение следующих глав.

Традиционный механизм НКА или нет?

Из всех типов механизмов регулярных выражений чаще всего используется традиционный механизм НКА, который легко отличить от других. Поддерживаются ли в механизме минимальные квантификаторы (☞ 141)? Если поддерживаются, перед вами почти наверняка традиционный механизм НКА. Как вы вскоре увидите, минимальные квантификаторы в ДКА невозможны, а в POSIX НКА они не имеют смысла. Тем не менее для верности примените регулярное выражение `⌜nfa|nfa•not⌟` к строке '`nfa•not`'; если совпадет только '`nfa`', значит, это традиционный механизм НКА. Если совпадает вся строка '`nfa•not`', это либо POSIX НКА, либо ДКА.

ДКА или POSIX НКА?

Отличить POSIX НКА от ДКА в общем случае бывает несложно — в ДКА не поддерживаются сохраняющие круглые скобки и обратные ссылки. Однако в некоторых гибридных системах используется комбинация двух типов механизмов, и при отсутствии сохраняющих круглых скобок в выражении используется механизм ДКА.

Следующий простой тест поможет получить дополнительную информацию. Примените выражение `X(.+)+X` к строке вида '`=XX======================`', как в следующей команде:

```
echo =XX======================================== | egrep 'X(.+)+X'
```

Если выполнение команды занимает очень много времени, перед вами механизм НКА (а если он не может быть традиционным механизмом НКА по результатам предыдущего теста, значит, это POSIX НКА). Если команда выполняется быстро, это либо ДКА, либо НКА с какой-то хитроумной оптимизацией. Если на экране появилось предупреждение о переполнении стека или прерывании затянувшейся операции, работает НКА.

Основы поиска совпадений

Прежде чем выяснять, чем различаются разные типы двигателей, мы сначала рассмотрим их общие свойства. У всех двигателей имеется нечто общее (хотя бы с чисто практической точки зрения), поэтому эти примеры относятся к двигателям любого типа.

О примерах

В этой главе рассматривается обобщенный полнофункциональный механизм регулярных выражений, поэтому какие-то из описанных возможностей могут не поддерживаться некоторыми программами. Скажем, в моем примере указатель уровня находится слева от масляного фильтра, а у вас он может быть расположен за крышкой распределителя. Вы должны понять основные концепции, чтобы успешно управлять работой своей любимой программы с поддержкой регулярных выражений (а также тех, которые заинтересуют вас в будущем).

В большинстве примеров я продолжаю использовать синтаксис Perl, хотя иногда буду переходить на другой синтаксис, чтобы вы не забывали — *запись* второстепенна, а рассматриваемые вопросы выходят за границы одной программы или диалекта. Если вы встретите незнакомую конструкцию, обращайтесь к главе 3 (☞ 155).

В этой главе подробно описаны практические алгоритмы выполнения поиска. Конечно, было бы удобно свести все к нескольким простым правилам, которые можно просто запомнить, не затрудняя себя доскональным пониманием происходящего. К сожалению, в данном случае сделать это не удастся. Во всей главе я могу выделить лишь два универсальных правила:

1. Предпочтение отдается более раннему совпадению.
2. Стандартные квантификаторы (⌜`*`⌟, ⌜`+`⌟, ⌜`?`⌟ и ⌜`{m,n}`⌟) работают максимально.

Мы рассмотрим эти правила, некоторые следствия из них и многое другое. Начнем с более подробного рассмотрения первого правила.

Правило 1: более раннее совпадение выигрывает

Правило гласит, что более раннему совпадению предпочтение отдается всегда, даже в том случае, если позднее в строке будет обнаружено другое возможное совпадение. Правило ничего не говорит о длине совпадения (вскоре мы обсудим этот вопрос подробнее). Речь идет лишь о том, что из всех возможных совпадений в строке выбирается то, начало которого располагается ближе к началу строки.

Это правило работает следующим образом: сначала механизм пытается найти совпадение от самого начала строки, в которой осуществляется поиск (с позиции перед первым символом). Слово «пытается» означает, что с указанной позиции ищутся все возможные комбинации всего (иногда довольно сложного) регулярного выражения. Если после перебора всех возможностей совпадение так и не будет найдено, вторая попытка делается с позиции, предшествующей второму символу. Эта процедура повторяется для каждой позиции в строке. Результат «совпадение не найдено» возвращается лишь в том случае, если совпадение не находится после перебора всех позиций до конца строки (после завершающего символа).

Следовательно, при поиске совпадения ⌜ORA⌟ в строке FLORAL первая попытка завершается неудачей (поскольку ⌜ORA⌟ не совпадает с FLO). Совпадение, начинающееся со второго символа (LOR), тоже не подходит. Но сопоставление, начинающееся с третьего символа, оказывается удачным, поэтому механизм поиска прекращает перебор и сообщает о найденном совпадении: FLORAL.

Если вы не знаете этого правила, результат иногда оказывается неожиданным. Например, при поиске ⌜cat⌟ в строке

```
The dragging belly indicates your cat is too fat
```

совпадение находится в слове indicates, а не в слове cat, расположенном правее. Слово cat *могло бы* совпасть, но cat в слове indicates начинается раньше, и поэтому совпадает именно оно. В таких приложениях, как *egrep*, учитывается *лишь наличие* совпадения, а не его *точная позиция*, поэтому в них это различие несущественно, но в других операциях (например, при поиске с заменой) оно имеет первостепенное значение.

Простой вопрос: где в строке 'The dragging belly indicates your cat is too fat' совпадет выражение ⌜fat|cat|belly|your⌟? ❖ Проверьте свой ответ на с. 200.

Смещение текущей позиции поиска

Следующий принцип работы механизма регулярных выражений можно рассматривать как аналог автомобильной трансмиссии, соединяющей двигатель с мостом.

Вся настоящая работа (поворот коленчатого вала) осуществляется двигателем, а трансмиссия передает эту работу на колеса.

Если механизм не находит совпадение в начале строки, то «трансмиссия» смещает текущую позицию поиска, и новая попытка осуществляется в следующей позиции строки, затем в следующей и т. д. Во всяком случае, обычно. Если же регулярное выражение начинается с якорного метасимвола начала строки, «трансмиссия» может догадаться, что дальнейшие сдвиги ни к чему не приведут, поскольку успешное совпадение может начинаться только от начала строки. Эта и другие разновидности внутренней оптимизации рассматриваются в главе 6.

Компоненты и части двигателя

Двигатель состоит из множества деталей разных типов и размеров. Чтобы понять, как он работает, необходимо разобраться с тем, как работают его составные части. В регулярных выражениях этим частям соответствуют синтаксические единицы — литералы, квантификаторы (* и прочие), символьные классы, выражения в круглых скобках и т. д. — см. главу 3 (☞ 156). Комбинация этих частей (и их интерпретация механизмом) и делает регулярное выражение тем, чем оно является, поэтому способы объединения и взаимодействие этих частей представляют для нас первоочередной интерес. Начнем с рассмотрения некоторых из этих частей.

Литералы (например, `a`, `\*`, `!`, `枝` и т. д.)

Для литеральных, не-метасимвольных конструкций типа `z` или `!` поиск совпадения сводится к простому выяснению вопроса — совпадает ли символ-литерал с текущим символом текста? Если регулярное выражение содержит только литеральный текст — например, `usa`, — оно интерпретируется как последовательность `u`, затем `s`, после чего `a`. Ситуация немного усложняется при поиске без учета регистра, где `b` может совпадать с `b` и `B`, но и в этом случае все относительно просто (хотя имеются дополнительные аспекты, связанные с применением Юникода ☞ 150).

Символьные классы, точка, свойства Юникода и т. п.

Поиск совпадений для точки, символьных классов, свойств Юникода и т. д. (☞ 151) обычно несложен: независимо от размера символьный класс все равно совпадает только с одним символом[1].

[1] В действительности, как было показано в предыдущей главе (☞ 174), объединяющие последовательности POSIX могут совпадать с несколькими символами, но это особый случай. Кроме того, некоторые символы Юникода могут совпадать с несколькими символами при поиске без учета регистра (☞ 150), хотя в большинстве реализаций такая возможность не поддерживается.

ОТВЕТ

❖ *Ответ на вопрос со с. 198.*

Помните: при каждой попытке осуществляется *полный* поиск регулярного выражения, поэтому ⌜`fat|cat|belly|your`⌟ совпадет с '`The dragging belly indicates your cat is too fat`' раньше, чем с `fat`, хотя ⌜`fat`⌟ в списке альтернатив находится в более ранней позиции.

Конечно, регулярное выражение позволяет найти совпадения и для `fat` с остальными альтернативами, но поскольку они не являются *самым ранним* (т. е. начинающимся в ближайшей к левому краю позиции) совпадением, выбор делается не в их пользу. Как я уже говорил, механизм пытается применить все выражение в текущей позиции, прежде чем переходить к следующей позиции для повторения попытки. В данном примере это означает, что в текущей позиции будут рассмотрены все альтернативы — ⌜`fat`⌟, ⌜`cat`⌟, ⌜`belly`⌟ и ⌜`your`⌟, и только после этого произойдет переход к следующей позиции.

Точка представляет собой сокращенную запись для большого символьного класса, состоящего почти из всех символов (☞ 152), поэтому и здесь поиск обходится без проблем. Так же обстоит дело и с другими сокращенными обозначениями — такими, как ⌜`\w`⌟, ⌜`\W`⌟ и ⌜`\d`⌟.

Сохраняющие круглые скобки

Круглые скобки, используемые только для сохранения совпавшего текста, не влияют на сам процесс поиска.

Якорные метасимволы (например, ⌜`^`⌟, ⌜`\Z`⌟, ⌜`(?<=\d)`⌟, ...)

Якорные метасимволы делятся на две категории: простые (`^`, `$`, `\G`, `\b`, ... ☞ 176) и сложные (опережающая и ретроспективная проверка ☞ 181). Простые якорные метасимволы действительно просты — они либо проверяют свойство определенной позиции в целевой строке (`^`, `\Z`, ...), либо сравнивают два соседних символа целевого текста (`\<`, `\b`, ...). Напротив, конструкции опережающей проверки могут содержать произвольные подвыражения и поэтому могут быть сколь угодно сложными.

Отсутствие обратных ссылок, сохраняющих круглых скобок и минимальных квантификаторов

На первых порах основное внимание будет уделяться сходным чертам двух механизмов, но для начала я опишу одно интересное отличие. Сохраняющие круглые скобки (и связанные с ними обратные ссылки и аналоги `$1`) напоминают бензиновую присадку. Они влияют на работу бензинового двигателя (НКА), но

несущественны для электрических (ДКА) двигателей — прежде всего потому, что бензин в нем вообще не используется. То же самое относится и к минимальным квантификаторам. Сам принцип работы механизма ДКА полностью исключает эти концепции, поэтому в этом механизме они попросту невозможны.[1] Становится понятно, почему программы с механизмом ДКА не обладают этими возможностями. В частности, awk, *lex* и *egrep* не поддерживают обратных ссылок или каких-либо аналогов `$1`.

Однако следует заметить, что GNU-версия *egrep* **поддерживает** обратные ссылки. Для этого под одним капотом устанавливаются два разных двигателя! Сначала механизм ДКА находит возможное совпадение, после чего механизм НКА (реализующий полноценный диалект с обратными ссылками) подтверждает совпадение. Позднее в этой главе будет показано, почему ДКА не позволяет применять обратные ссылки и сохранять текст и зачем вообще нужен такой механизм (у него есть свои большие преимущества — например, очень быстрый поиск совпадений).

Правило 2: квантификаторы работают максимально

Приведенный выше процесс поиска совпадений выглядит весьма прямолинейно. И к тому же кажется неинтересным — трудно *сделать* что-нибудь серьезное без применения более мощных метасимволов (`*`, `+`), конструкции выбора и т. д. Но чтобы понять новые возможности, необходимо больше знать о них.

Во-первых, вы должны запомнить, что стандартные квантификаторы (`?`, `*`, `+` и `{min, max}`) работают *максимально*. Когда квантификатор применяется к подвыражению (например, ⌜`a`⌟ в ⌜`a?`⌟, ⌜(*выражение*)⌟ в ⌜(*выражение*)`*`⌟ или ⌜`[0-9]`⌟ в ⌜`[0-9]+`⌟), существует некоторое минимальное количество повторений, необходимых для успешного совпадения, и максимальное количество повторений, больше которого механизм искать даже не пытается. Об этом говорилось в одной из предыдущих глав — но здесь вступает в силу второе правило регулярных выражений. Если некоторый элемент может совпадать переменное количество раз, механизм всегда пытается найти максимальное количество повторений. (Некоторые диалекты предоставляют другие типы квантификаторов, но в этом разделе рассматриваются только стандартные максимальные квантификаторы.)

Другими словами, *при необходимости* механизм может «согласиться» и на меньшее количество допустимых совпадений, но он всегда пытается обнаружить как можно больше совпадений вплоть до разрешенного максимума. Механизм регулярных выражений соглашается на значения меньше максимума только в одном случае —

[1] Конечно, это вовсе не означает, что нельзя изобрести какое-нибудь комбинированное решение, использующее средства обоих механизмов. Этот вопрос рассматривается во врезке на с. 239.

если слишком большое количество повторений не позволяет найти совпадение для какой-либо последующей части регулярного выражения. Рассмотрим простой пример: выражение ⌜\b\w+s\b⌟, предназначенное для поиска слов, заканчивающихся на 's' (скажем, 'regexes'). Конструкция ⌜\w+⌟ была бы рада совпасть с целым словом, но в этом случае литералу ⌜s⌟ совпадать будет уже не с чем. Чтобы добиться общего совпадения, ⌜\w+⌟ соглашается на совпадение с подстрокой 'regexes': и это позволяет совпасть ⌜s\b⌟ и всему выражению.

Если выясняется, что оставшаяся часть регулярного выражения совпадает лишь в том случае, когда максимальная конструкция не совпадает ни с чем (если нулевое количество повторений разрешено, как для *, ? и интервального квантификатора {0, max})... что ж, это вполне нормально. Но такая ситуация возможна лишь в том случае, когда этого требует одно из дальнейших подвыражений. Максимальные квантификаторы всегда захватывают (или по крайней мере стремятся захватить) больше повторений, чем им требуется по минимуму, поэтому их иногда называют «жадными».

Из этого правила вытекает много полезных (но также и создающих сложности) следствий. В частности, оно объясняет, почему ⌜[0-9]+⌟ совпадает со всем числом в строке March•1998. Найденное совпадение '1' выполняет минимальное требование для квантификатора +. Но из-за того, что квантификатор всегда старается обеспечить максимальное количество повторений, поиск продолжается, и в совпадение включаются символы '998'. Поиск прерывается при обнаружении конца строки — ⌜[0-9]⌟ не может совпасть с «пустым местом», и обработка + на этом завершается.

Пример на заданную тему

Конечно, описанный метод используется не только при поиске чисел. Допустим, у нас имеется строка из заголовка сообщения электронной почты и мы хотим проверить, является ли она строкой темы. Как было показано в одной из предшествующих глав (☞ 87), задача решается при помощи выражения ⌜^Subject:⌟. Но если воспользоваться выражением ⌜^Subject:•(.*)⌟, то программа сохранит тему сообщения в служебной переменной ($1 в Perl).[1]

Прежде чем выяснять, почему ⌜.*⌟ совпадает с текстом темы, необходимо четко понимать следующее: если часть ⌜^Subject:•⌟ совпадает, то и все регулярное выражение заведомо совпадет. Это объясняется тем, что после ⌜^Subject:•⌟ нет ни одного элемента, который бы мог отменить потенциальное совпадение — .* совпа-

[1] Этот пример демонстрирует принцип максимализма с использованием круглых скобок и поэтому подходит только для механизма НКА (поскольку в ДКА такая возможность отсутствует). Тем не менее принцип максимализма действует во всех механизмах, в том числе и в ДКА.

дает *всегда*, поскольку даже худший случай «нет совпадений» для квантификатора * все равно считается успешным.

Тогда зачем включать ⌜.*⌟ в регулярное выражение? Квантификатор ⌜*⌟ постарается найти как можно больше повторений метасимвола «точка», поэтому мы используем его для «заполнения» переменной $1. Круглые скобки не влияют на логику поиска совпадения — в данном случае они используются для сохранения текста, совпавшего с ⌜.*⌟.

После того как ⌜.*⌟ доберется до конца строки, следующего совпадения для точки не находится, поэтому обработка * завершается и начинается поиск совпадения для следующего элемента регулярного выражения (хотя для ⌜.*⌟ совпадений больше нет, возможно, найдется совпадение для следующего элемента). Однако в нашем случае следующего элемента нет, мы достигаем конца регулярного выражения и знаем, что совпадение было найдено успешно.

Чрезмерная жадность

Вернемся к принципу максимализма, согласно которому квантификатор забирает все, что только может забрать. Подумайте, как изменится совпадение и результаты поиска, если добавить второе подвыражение ⌜.*⌟ — ⌜^Subject:␣(.*).*⌟. Ответ: ничего не изменится. Первое подвыражение ⌜.*⌟ (в круглых скобках) оказывается настолько жадным, что совпадает со всей темой сообщения, а на долю второго подвыражения ⌜.*⌟ ничего не остается. Отсутствие совпадающего текста в этом случае не вызывает проблем, поскольку для * наличие фактического совпадения не является обязательным. Если бы второе подвыражение ⌜.*⌟ было также заключено в скобки, то переменная $2 всегда была бы пустой.

Означает ли это, что после ⌜.*⌟ в регулярном выражении не может быть ни одного элемента, для которого требуется реальное совпадение? Конечно, нет. Как было показано в примере ⌜\w+s⌟, там могут находиться любые элементы, которые *заставят* предыдущий «жадный» квантификатор вернуть часть захваченного, необходимую для совпадения всего выражения.

Рассмотрим вполне реальное выражение ⌜^.*([0-9][0-9])⌟. Оно находит две *последние* цифры в строке, где бы они ни располагались, и сохраняет их в переменной $1. Это происходит следующим образом: сначала ⌜.*⌟ сопоставляется со всей строкой. Следующее подвыражение ⌜([0-9][0-9])⌟ является *обязательным*, и при отсутствии совпадения для него «говорит»: «Эй, ⌜.*⌟, ты взял лишнее! Верни мне что-нибудь, чтобы и для меня нашлось совпадение». «Жадные» компоненты всегда сначала стараются захватить побольше, но потом всегда отдают излишки, если при этом достигается общее совпадение. Впрочем, как истинные жадины, они упрямы и добровольно ничего не отдают. Конечно, никогда не уступаются обязательные части — например, первое совпадение квантификатора +.

Учитывая сказанное, применим ⌜^.*([0-9][0-9])⌟ к строке 'about•24•characters•long'. После того как ⌜.*⌟ совпадет со всей строкой, необходимость совпадения для первого класса ⌜[0-9]⌟ заставит ⌜.*⌟ уступить символ 'g' (последний из совпавших). Однако при этом совпадение для ⌜[0-9]⌟ по-прежнему не находится, поэтому ⌜.*⌟ приходится идти на новые уступки — на этот раз это 'n' в слове 'long'. Цикл повторяется еще 15 раз, пока ⌜.*⌟ не доберется до цифры '4'.

К сожалению, совпадение появляется лишь для первого класса ⌜[0-9]⌟. Второму классу, как и прежде, совпадать не с чем. Поэтому ⌜.*⌟ приходится уступать дальше, чтобы обеспечить общее совпадение всего выражения. На этот раз ⌜.*⌟ уступает цифру '2', с которой может совпасть первый класс ⌜[0-9]⌟. Цифра '4' освобождается и совпадает со вторым классом, а все выражение — с 'about•24•char…'. Переменной $1 присваивается строка '24'.

В порядке очереди

Попробуем привести это регулярное выражение к виду ⌜^.*([0-9]+)⌟. Идея состоит в том, чтобы найти не только две последние цифры, а все число независимо от длины. Какой текст совпадет, если применить это выражение к строке 'Copyright 2003.'? ❖ Проверьте свой ответ на с. 206.

Переходим к подробностям

Пожалуй, я должен кое-что пояснить. Фразы типа «⌜.*⌟ *уступает*...» или «⌜[0-9]⌟ *заставляет*...» на самом деле неточны. Я воспользовался этими оборотами, потому что они достаточно наглядны и при этом приводят к правильному результату. Однако события, которые в действительности происходят за кулисами, зависят от базового типа механизма, ДКА или НКА. Пришло время узнать, что же означают эти термины.

Механизмы регулярных выражений

В двух базовых типах механизмов регулярных выражений отражаются два принципиально разных подхода к сравнению регулярного выражения со строкой. Говорят, что механизм НКА «управляется регулярным выражением», а механизм ДКА «управляется текстом».

НКА: механизм, управляемый регулярным выражением

Рассмотрим один из алгоритмов, который может использоваться механизмом для поиска совпадения выражения ⌜to(nite|knight|night)⌟ в тексте '…tonight…'. Ме-

ханизм просматривает регулярное выражение по одному компоненту, начиная с ⌜t⌟, и проверяет, совпадает ли компонент с «текущим текстом». В случае совпадения проверяется следующий компонент. Процедура повторяется до тех пор, пока не будет найдено совпадение для всех компонентов регулярного выражения; в этом случае мы считаем, что найдено общее совпадение.

В примере ⌜to(nite|knight|night)⌟ первым компонентом является литерал ⌜t⌟. Проверка завершается неудачей до тех пор, пока в целевом тексте не будет обнаружен символ 't'. Когда это произойдет, ⌜o⌟ сравнивается со следующим символом, и в случае совпадения управление будет передано следующему компоненту. В данном случае «следующим компонентом» является конструкция выбора ⌜(nite|knight|night)⌟, которая означает «либо ⌜nite⌟, либо ⌜knight⌟, либо ⌜night⌟». Столкнувшись с тремя альтернативами, механизм просто по очереди перебирает их. Мы, существа с хитроумными нейронными сетями в голове, сразу видим, что для строки `tonight` к совпадению приводит третья альтернатива. Но несмотря на свое интеллектуальное происхождение (☞ 120), механизм, управляемый регулярным выражением, придет к этому выводу лишь после перебора всех возможных вариантов.

Проверка первой альтернативы, ⌜nite⌟, подчиняется тому же принципу последовательного сравнения компонентов: «Сначала проверить ⌜n⌟, потом ⌜i⌟, затем ⌜t⌟ и, наконец, ⌜e⌟». Если проверка завершается неудачей (как в нашем примере), механизм переходит к другой альтернативе и так далее до тех пор, пока не будет найдено совпадение или не будут исчерпаны все варианты (тогда механизм сообщает о неудаче). Управление передается внутри регулярного выражения от компонента к компоненту, поэтому я говорю, что такой механизм «управляется регулярным выражением».

Преимущества механизма НКА

Фактически каждое подвыражение в регулярном выражении проверяется независимо от других. Если не считать обратные ссылки, подвыражения никак не связаны между собой — просто они собраны в одно большое выражение. Общие действия механизма в процессе поиска полностью определяются структурой подвыражений и управляющих конструкций регулярного выражения (выбор, круглые скобки и квантификаторы).

Поскольку механизм НКА управляется регулярным выражением, автор выражения (водитель в нашей аналогии) может в значительной мере определять ход дальнейших событий. В главах 5 и 6 показано, как при помощи управления механизмом добиться того, чтобы работа была выполнена правильно и эффективно. Пока эти обещания выглядят несколько расплывчато, но к тому времени все прояснится.

ОТВЕТ

❖ *Ответ на вопрос со с. 204.*

Что сохраняется в круглых скобках при применении выражения ⌜^.*([0-9]+)⌟ к тексту 'Copyright 2003.'?

Идея состоит в том, чтобы найти все число независимо от длины, однако такое решение не работает. Подвыражению ⌜.*⌟ приходится идти на частичные уступки, поскольку для совпадения всего выражения необходимо хотя бы одно совпадение для ⌜[0-9]+⌟. В нашем примере это означает возврат завершающей точки и '3', после чего появляется совпадение для класса ⌜[0-9]⌟. К классу применен квантификатор ⌜+⌟, для которого один экземпляр соответствует минимальным требованиям; дальше в строке идет точка, поэтому второго совпадения нет.

Но в отличие от предыдущего случая никаких обязательных элементов дальше нет, поэтому ⌜.*⌟ не приходится уступать 0 и другие цифры. В принципе подвыражение ⌜[0-9]+⌟ с радостью согласилось бы на такой подарок, но нет: кто первым приходит, того первым и обслуживают. Жадные квантификаторы по-настоящему скупы: если что-то попало им в лапы, они отдадут это «что-то» только по принуждению, а не по доброте душевной. В итоге переменная $1 содержит только символ '3'.

Если такой подход кажется вам нелогичным, подумайте: ⌜[0-9]+⌟ всего на одно совпадение отличается от ⌜[0-9]*⌟, а это выражение из одной категории с ⌜.*⌟. После замены + на * выражение ⌜^.*([0-9]+)⌟ превращается в выражение вида ⌜^.*(.*)⌟, которое подозрительно напоминает ⌜^Subject:•(.*).*⌟ со с. 202, а в этом выражении второе подвыражение ⌜.*⌟ заведомо ни с чем не совпадало.

ДКА: механизм, управляемый текстом

Механизму НКА противопоставляется механизм ДКА, который сканирует строку и следит за всеми «потенциальными совпадениями». В описанном выше примере с tonight механизм узнает о начале потенциального совпадения, как только в строке встречается символ t:

в строке	в регулярном выражении
после …t▲onight…	потенциальные совпадения ⌜to▲(nite|knight|night)⌟

Каждый следующий сканируемый символ обновляет список потенциальных совпадений. Через несколько символов мы приходим к следующей ситуации:

в строке	в регулярном выражении
после …toni▲ght…	потенциальные совпадения ⌜to(ni▲te|knight|night)⌟

с двумя потенциальными совпадениями (одна альтернатива, `knight`, к этому моменту уже отвергается). Проверка следующего символа, `g`, исключает и первую альтернативу. После сканирования `h` и `t` механизм понимает, что он нашел полное совпадение, и успешно завершает свою работу.

Механизм ДКА «управляется текстом», поскольку его работа зависит от каждого просканированного символа строки. В приведенном выше примере частичное совпадение может быть началом любого количества разных (но потенциально возможных) совпадений. Отвергаемые совпадения исключаются из дальнейшего рассмотрения по мере сканирования последующих символов. Возможны также ситуации, при которых найденное «частичное совпадение» является полным совпадением. Например, при обработке круглых скобок в выражении `⌜to(…)?⌟` подвыражение в скобках — необязательное, но из-за максимальности квантификатора механизм всегда пытается найти наибольшее совпадение. Все это время полное совпадение (‘`to`’) уже считается подтвержденным и держится в резерве на тот случай, если не удастся найти более длинное совпадение.

Если очередной символ текста аннулирует все потенциальные совпадения, приходится либо возвращаться к полному совпадению, находящемуся в резерве, либо (при его отсутствии) объявлять, что для текущей начальной позиции совпадений нет.

Сравнение двух механизмов

Если сравнивать эти два механизма на основании только того, о чем я уже рассказал, нетрудно прийти к выводу, что управляемый текстом механизм ДКА должен работать быстрее. Механизм НКА, управляемый регулярным выражением, может терять время при поиске всех заданных подвыражений в одном тексте (как в случае с тремя альтернативами в примере).

И это будет верно. При работе механизма НКА один символ целевого текста может сравниваться во многих разных компонентах регулярного выражения (или даже многократно в одном компоненте). Даже если подвыражение совпадает, возможно, его придется применять снова (а потом еще и еще), поскольку общее совпадение определяется в совокупности с остальными компонентами регулярного выражения. Локальное подвыражение может совпадать или не совпадать, но судить о наличии общего совпадения можно, лишь добравшись до конца регулярного выражения. С другой стороны, механизм ДКА является *детерминированным* — каждый символ в строке проверяется не более одного раза. Если символ совпал, вы еще не знаете, войдет ли он в итоговое совпадение (символ может быть частью потенциального совпадения, которое позднее будет отвергнуто), но, поскольку механизм отслеживает все потенциальные совпадения одновременно, символ достаточно проверить только один раз. Точка.

Две базовые технологии, на основе которых строятся механизмы регулярных выражений, носят устрашающие названия: *недетерминированный конечный автомат* (НКА) и *детерминированный конечный автомат* (ДКА). Надеюсь, вы поняли, почему я предпочитаю использовать простые сокращения «НКА» и «ДКА». На страницах этой книги полные названия больше не встречаются[1].

Последствия для пользователей

Поскольку механизм НКА управляется регулярным выражением, при его использовании необходимо очень хорошо разбираться в том, как происходит поиск совпадений. Как я уже говорил, автор может в значительной степени управлять процессом поиска за счет правильного написания выражения. Вероятно, в примере с `tonight` для повышения эффективности поиска можно было бы записать регулярное выражение в одном из следующих вариантов:

- ⌜`to(ni(ght|te)|knight)`⌟
- ⌜`tonite|toknight|tonigh`⌟
- ⌜`to(k?night|nite`⌟

Эти регулярные выражения эквивалентны с точки зрения совпадающего текста, но при этом они по-разному управляют работой механизма. Пока вы еще недостаточно хорошо разбираетесь в регулярных выражениях и не можете судить, какие варианты работают эффективнее других, но вскоре мы займемся этим вопросом.

В ДКА все наоборот: поскольку механизм отслеживает все совпадения одновременно, любые различия в представлении несущественны, если они в конечном счете определяют один и тот же совпадающий текст. Можно придумать сотню эквивалентных выражений, но поскольку ДКА следит за всеми потенциальными совпадениями (почти волшебным образом — но об этом позднее), конкретный вид представления несущественен. Для «чистого» ДКА даже такие разные выражения, как ⌜`abc`⌟ и ⌜`[aa-a](b|b{1}|b)c`⌟, попросту неразличимы.

При описании механизма ДКА я бы выделил три обстоятельства.

- Совпадения в ДКА ищутся очень быстро.
- Процесс поиска совпадений в ДКА предсказуем и логичен.
- Описывать поиск в ДКА скучно.

---

[1] Вероятно, мне бы следовало объяснить азы теории конечных автоматов... если бы я ее знал! Как указано выше, слово *«детерминированный»* в определении достаточно существенно, однако теоретические обоснования не столь важны, если понимать их практические последствия. К концу этой главы вы будете в них разбираться.

Вскоре мы вернемся к этой теме и рассмотрим ее подробнее. Механизм НКА управляется регулярным выражением, поэтому писать о нем интересно — НКА открывает настоящую свободу для творчества. Правильное построение выражений приносит большую пользу, хотя ошибка может принести еще больший вред. Механизм, как и бензиновый двигатель, тоже может расчихаться и полностью заглохнуть. Но чтобы досконально разобраться во всех тонкостях НКА, необходимо рассмотреть одну из важнейших концепций этого механизма — *возврат* (*backtracking*).

Возврат

Основной принцип работы механизма НКА заключается в следующем: он последовательно рассматривает все подвыражения или компоненты, и когда приходится выбирать между двумя равноправными вариантами — выбирает один вариант и запоминает другой, чтобы позднее вернуться к нему в случае необходимости.

Ситуации, когда механизму НКА приходится выбирать дальнейший ход действий, связаны с использованием квантификаторов (нужно ли пытаться найти следующее совпадение) и конструкций выбора (какую альтернативу искать сейчас, а какую оставить на будущее).

Если выбранный вариант и все выражение успешно совпадают, процесс поиска завершается. Если какая-либо из оставшихся частей регулярного выражения приводит к неудаче, механизм регулярных выражений *возвращается* к развилке, где было принято решение, и продолжает поиск с другим вариантом. В конечном счете механизм перепробует все возможные сочетания компонентов (или по крайней мере столько сочетаний, сколько потребуется для успешного совпадения).

Крошечная аналогия

Представьте себе, что вы ищете выход из лабиринта. На каждой развилке вы оставляете горку хлебных крошек. Если выбранный путь приводит к тупику, вы поворачиваете назад и на ближайшей развилке сворачиваете на дорогу, по которой еще не ходили. Если и эта дорога закончится тупиком, вы возвращаетесь к следующей горке хлебных крошек. В конце концов вы либо найдете дорогу, которая приведет вас к цели, либо переберете все возможные пути.

Существуют разные ситуации, при которых механизму регулярных выражений приходится выбирать один из двух (или более) вариантов. Конструкция выбора является лишь одним из примеров. Другой пример — встретив конструкцию `⌜...x?...⌟`, механизм решает, искать `⌜x?⌟` в тексте или нет. Для `⌜...x+...⌟` такого вопроса не возникает — квантификатор `+` требует хотя бы одного обязательного совпадения, и это требование обсуждению не подлежит. Но если первое совпадение для `⌜x⌟` будет най-

дено, обязательное требование снимается и механизм должен решить, *следует ли* ему искать другие повторения ⌈x⌋. Если будет найдено второе совпадение, механизм решает, нужно ли искать дальше... и дальше... и т. д. Каждый раз, когда механизм принимает решение, он оставляет воображаемую «кучку крошек», которая напоминает о том, что в этой точке остается другой возможный вариант (совпадение или его отсутствие — тот вариант, который не был выбран ранее).

Небольшой пример

Рассмотрим поиск знакомого выражения ⌈to(nite|knight|night)⌋ в строке 'hot•tonic•tonight!' (фраза, конечно, глупая, но зато хороший пример). Первый компонент ⌈t⌋ сравнивается с началом строки. Он не совпадает с h, поэтому в этой позиции все регулярное выражение заведомо не совпадет. Механизм перемещается ко второму символу регулярного выражения, где совпадение тоже не обнаруживается. Затем наступает очередь третьего символа. На этот раз ⌈t⌋ совпадает, но следующий символ ⌈o⌋ отличается от '•' в тексте, поэтому эта попытка тоже завершается неудачей.

Попытка, начинающаяся с ...▲tonic..., представляет больший интерес. После совпадения to появляются три возможных варианта, соответствующие трем разным альтернативам. Механизм регулярных выражений выбирает одну альтернативу и запоминает остальные («оставляет хлебные крошки») на случай, если первая попытка окажется неудачной. Предположим, механизм начал с альтернативы ⌈nite⌋. Это выражение делится на «⌈n⌋+ ⌈i⌋+ ⌈t⌋... », и процесс поиска добирается до символа ...toni▲c..., где происходит несовпадение. В отличие от предыдущих неудач эта не означает перехода к следующей позиции, поскольку остались другие возможные варианты. Механизм выбирает один из них (скажем, ⌈knight⌋), но и здесь происходит немедленная неудача. Остается последний вариант, ⌈night⌋. Совпадение не происходит и на этот раз. Поскольку это был последний неопробованный вариант, неудача означает неудачу всей попытки, начинающейся с ...▲tonic..., поэтому механизм переходит к следующей позиции.

Ситуация снова становится интересной, когда механизм начинает поиск совпадения с позиции ...▲tonight! На этот раз альтернатива ⌈night⌋ совпадает до самого конца (что означает общее совпадение, поэтому теперь механизм может сообщить об успешном завершении работы).

Два важных замечания

Концепция возврата выглядит довольно просто, но для ее применения на практике необходимо учитывать некоторые технические детали. Во-первых, какой вариант из нескольких возможных должен проверяться в первую очередь? Во-вторых, ка-

кой из сохраненных вариантов должен использоваться механизмом при возврате? Ответ на первый вопрос представляет собой важный принцип:

> В тех ситуациях, где механизм выбирает между попытками найти совпадение или отказаться от его поиска (например, при использовании квантификаторов `?`, `*` и других), механизм всегда сначала *пытается* найти совпадение для *максимальных* квантификаторов и пропустить совпадение для *минимальных* квантификаторов.

Это простое правило имеет далекоидущие последствия. Во-первых, оно частично объясняет принцип максимализма в регулярных выражениях. Для полноты картины необходимо знать, какой из сохраненных вариантов должен выбираться при возврате. Простой ответ выглядит так:

> При локальной неудаче происходит возврат к последнему из сохраненных вариантов. Перебор вариантов осуществляется по правилу LIFO (последним пришел — первым обслужен).

Это правило легко понять по нашей аналогии с крошками — оказавшись в тупике, вы просто идете в обратную сторону до тех пор, пока не наткнетесь на горку хлебных крошек. На вашем пути первой встретится та горка, которая была оставлена последней. При описании LIFO также часто используется классическая аналогия со стопкой тарелок: первой со стопки снимается та тарелка, которая была поставлена в нее последней.

Сохраненные состояния

В терминологии регулярных выражений НКА «хлебные крошки» на развилках называются *сохраненными состояниями* (*saved states*). Сохраненное состояние определяет точку, с которой при необходимости можно начать очередную проверку. В нем сохраняется как текущая позиция в регулярном выражении, так и позиция в строке, с которой начинается проверка. Поскольку сохраненные состояния являются основой работы механизма НКА, позвольте мне пояснить сказанное на простых, но поучительных примерах. Если вы и так понимаете, как работают сохраненные состояния, то следующий раздел можно пропустить.

Поиск без возврата

Рассмотрим простой пример — поиск выражения ⌜`ab?c`⌟ в строке `abc`. После совпадения с ⌜`a`⌟ *текущее состояние* выглядит следующим образом:

в строке ‘a▲bc’	в выражении ⌜a▲b?c⌟

Но теперь, перед поиском ⌈b?⌋, механизм должен принять решение — пытаться найти ⌈b⌋ или нет? Из-за своей жадности квантификатор ? начинает искать совпадение. Но для того, чтобы в случае неудачи он мог вернуться к этой точке, в пустой на этот момент список сохраненных состояний добавляется следующая запись:

в строке 'a▲bc'	в выражении ⌈ab?▲c⌋

Это означает, что механизм позднее продолжит поиск с компонента, следующего в регулярном выражении *после* ⌈b?⌋, и сопоставит его с текстом, находящимся до b (т. е. в текущей позиции). В сущности, это означает, что литерал ⌈b⌋ пропускается, что допускает квантификатор ?.

Успешно разместив «горку хлебных крошек», механизм переходит к проверке ⌈b⌋. В нашем примере для него находится совпадение, поэтому новое текущее состояние выглядит так:

в строке 'ab▲c'	в выражении ⌈ab?▲c⌋

Последний компонент, ⌈c⌋, тоже совпадает. Следовательно, механизм нашел общее совпадение для всего регулярного выражения. Единственное сохраненное состояние становится ненужным, и механизм попросту забывает о нем.

Поиск после возврата

Если бы поиск производился в строке 'ac', все происходило бы точно так же до попытки найти совпадение для ⌈b⌋. Конечно, на этот раз совпадение не обнаруживается. Это означает, что путь, пройденный при попытке найти совпадение для ⌈...?⌋, оказался тупиковым. Но поскольку у нас имеется сохраненное состояние, к которому можно вернуться, «локальная неудача» вовсе не означает общей неудачи. Механизм возвращается к последнему сохраненному состоянию и превращает его в новое текущее состояние. В нашем примере восстанавливается состояние

в строке 'a▲c'	в выражении ⌈ab?▲c⌋

сохраненное перед поиском ⌈b⌋. На этот раз ⌈c⌋ и c совпадают, что обеспечивает общее совпадение.

Несовпадение

Рассмотрим поиск того же регулярного выражения в строке 'abX'. Перед попыткой поиска ⌈b⌋ квантификатор ? сохраняет состояние:

в строке 'a▲bX'	в выражении ⌈ab?▲c⌋

Совпадение для ⌜b⌟ находится, однако этот путь позднее оказывается тупиковым, поскольку ⌜c⌟ не совпадает с X. Неудача приводит к восстановлению сохраненного состояния. Механизм сравнивает ⌜c⌟ с символом b, «выпавшим» из совпадения в результате возврата. Разумеется, и эта проверка завершается неудачей. При наличии других сохраненных состояний произошел бы следующий возврат, но таких состояний нет, поэтому попытка найти совпадение в текущей начальной позиции завершается неудачей.

Работа закончена? Нет. Механизм перемещается к следующей позиции в строке и снова пытается применить регулярное выражение. Происходит своего рода «псевдовозврат». Исходное состояние для повторного поиска выглядит так:

в строке ‘a▲bX’	в выражении ⌜▲ab?c⌟

Весь поиск повторяется с новой позиции, но, как и прежде, все пути ведут к неудаче. После провала еще двух попыток (с позиций abX механизм сообщает о том, что совпадение для всего выражения найти не удалось).

Минимальный поиск

Вернемся к исходному примеру, но на этот раз — с минимальным квантификатором. Допустим, мы ищем совпадение для регулярного выражения ⌜ab??c⌟ в строке ‘abc’. После обнаружения совпадения для ⌜a⌟ текущее состояние поиска выглядит так:

в строке ‘a▲bc’	в выражении ⌜a▲b??c⌟

Перед применением ⌜b??⌟ механизм регулярных выражений должен принять решение: попытаться найти совпадение для ⌜b⌟ или пропустить его? Поскольку мы имеем дело с минимальным квантификатором ??, сначала выбирается первый вариант, но информация об этом решении сохраняется. Позднее механизм сможет вернуться и попытаться найти совпадение в том случае, если дальнейший поиск завершится неудачей. В пустой список сохраненных состояний включается новый элемент:

в строке ‘a▲bc’	в выражении ⌜a▲bc⌟

Это означает, что позднее механизм сможет продолжить поиск совпадения для ⌜b⌟ с позиции, непосредственно предшествующей символу b (мы знаем, что совпадение будет найдено, но механизму регулярных выражений об этом еще неизвестно; он даже не знает, дойдет ли дело до этой попытки). Сохранив состояние, механизм продолжает обработку выражения после принятого решения о том, чтобы не искать совпадение для минимального квантификатора:

в строке ‘a▲bc’	в выражении ⌜ab??▲c⌟

Подвыражение ⌜c⌟ не совпадает с ‘b’, поэтому механизм возвращается к сохраненному состоянию:

в строке ‘a▲bc’	в выражении ⌜a▲bc⌟

На этот раз совпадение благополучно находится, а следующее подвыражение ⌜c⌟ совпадает с ‘c’. В результате обнаруживается такое же совпадение, как с максимальным квантификатором ⌜ab?c⌟, но поиск ведется по другому пути.

Возврат и максимализм

При работе с программами, использующими средства возврата механизма НКА, необходимо хорошо понимать, как происходит возврат. Это позволяет создавать выражения, которые решают нужную задачу и притом с максимальной эффективностью. Вы уже видели, как действует принцип максимализма для квантификатора ⌜?⌟ и принцип минимализма для квантификатора ⌜??⌟. Давайте посмотрим, как эти принципы проявляются для квантификаторов * и +.

Квантификаторы *, + и возврат

Если рассматривать выражение ⌜x*⌟ как более или менее эквивалентное ⌜x?x?x?x?x?x?…⌟ (а точнее, ⌜(x(x(x(x…?)?)?)?)?⌟)[1], этот квантификатор не так уж сильно отличается от того, который мы уже рассматривали. Прежде чем проверять очередной символ, механизм сохраняет состояние на случай неудачи при проверке (или последующих проверках), чтобы поиск совпадений можно было продолжить после *. Это происходит снова и снова, пока попытка сопоставления * не завершится неудачей.

Например, при поиске ⌜[0-9]+⌟ в строке ‘a•1234•num’ класс ⌜[0-9]⌟ не совпадает с пробелом после 4. К этому моменту появляется четыре сохраненных состояния, соответствующих позициям, в которых возможен возврат для квантификатора +:

```
a 1▲234 num
a 12▲34 num
a 123▲4 num
a 1234▲ num
```

В сохраненных состояниях отражается тот факт, что наличие совпадения для ⌜[0-9]⌟ в каждой из перечисленных позиций было необязательным. Когда ⌜[0-9]⌟ не совпадает с пробелом, механизм возвращается к самому новому из сохранен-

[1] Просто для сравнения вспомните о том, что ДКА не зависит от формы представления выражения; в ДКА все три примера действительно идентичны.

ных состояний (последнему из перечисленных) и продолжает работу с позиции ‘a•1234▲•num’ в тексте и ⌜[0-9]+▲⌟ в регулярном выражении. Однако текущая позиция уже находится в конце регулярного выражения. Заметив это, механизм понимает, что он нашел общее совпадение.

Обратите внимание: строка ‘a•▲1234•num’ отсутствует среди перечисленных, поскольку первое совпадение для квантификатора + является обязательным. Как вы думаете, будет ли она присутствовать в списке для регулярного выражения ⌜[0-9]*⌟? (*Внимание — каверзный вопрос!*) ❖ Переверните страницу, чтобы проверить ответ.

Возвращаемся к более сложному примеру

После всего сказанного давайте вернемся к примеру ⌜^.*([0-9][0-9])⌟ со с. 202. На этот раз для объяснения того, почему поиск производится именно так, а не иначе, мы не будем ссылаться на «жадность». Знание механики поиска НКА позволяет точно описать все происходящее.

В качестве примера будет использована строка ‘CA•95472,•USA’. После успешного совпадения ⌜.*⌟ до конца строки в списке сохраненных состояний оказывается 13 записей. Сохраненные состояния накапливаются из-за того, что точка с квантификатором * совпала с 13 элементами, которые (при необходимости) могут оказаться необязательными. Это позволяет продолжить подбор вариантов совпадения с позиции выражения ⌜^.*▲([0-9][0-9])⌟ и каждой позиции в строке, для которой сохранялось состояние.

Механизм достигает конца строки и передает управление первому классу ⌜[0-9]⌟. Конечно, попытка найти совпадение проваливается. Нет проблем: у нас в запасе есть сохраненное состояние (даже целая дюжина состояний). Происходит возврат, и механизм восстанавливает то состояние, которое было сохранено последним, т. е. перед тем, как подвыражение ⌜.*⌟ совпало с последним A. Отказ от этого совпадения позволит нам сравнить A с первым классом ⌜[0-9]⌟. Попытка оказывается неудачной.

Возвраты и проверки в цикле продолжаются до тех пор, пока механизм не отменит совпадение для цифры 2. В этот момент для первого класса ⌜[0-9]⌟ находится совпадение. Однако второму классу совпадать не с чем, поэтому возврат продолжается. При этом механизм забывает, что первый класс ⌜[0-9]⌟ совпал при предыдущей попытке — в восстановленном состоянии поиск продолжается с позиции перед первым классом ⌜[0-9]⌟. В результате возврата текущая позиция в строке перемещается перед цифрой 7, поэтому для первого класса ⌜[0-9]⌟ снова находится совпадение. Но на этот раз оно находится и для второго класса (цифра 2). Таким образом, в строке обнаруживается совпадение: ‘CA 95472,•USA’, а переменной $1 присваивается ‘72’.

ОТВЕТ

❖ *Ответ на вопрос со с. 215.*

Итак, будет ли учитываться сохраненное состояние ‘a•▲1234•num’ при поиске совпадения для выражения ⌜[0-9]*⌟ в строке ‘a•1234•num’?

Нет, не будет. Я задал этот вопрос, поскольку многие новички допускают эту ошибку. Вспомните: компонент, содержащий *, совпадает *всегда*. Если регулярное выражение не содержит других компонентов, то и для него всегда найдется совпадение. Конечно, этому правилу подчиняется и самая первая попытка, когда регулярное выражение применяется от начала строки. В этом случае механизм находит совпадение в позиции ‘▲a•1234•num’, и на этом все кончается — до цифр он даже не доходит.

Если вы упустили это обстоятельство из виду, могу вас немного утешить. Если бы после ⌜[0-9]*⌟ в регулярном выражении присутствовали какие-либо символы, это не позволило бы найти общее совпадение до их обработки:

в строке ‘a•▲1234…’	в выражении ⌜▲[0-9]*…⌟

В этом случае при попытке поиска в позиции перед ‘1’ будет создано следующее состояние:

в строке ‘a•▲1234…’	в выражении ⌜[0-9]*▲…⌟

Несколько замечаний. Во-первых, в процессе возврата также обновляется информация о тексте, совпавшем с подвыражениями в круглых скобках. В приведенном примере возврат всегда приводил к возобновлению поиска с позиции ⌜^.*▲([0-9][0-9])⌟. В том, что касается простого поиска совпадений, это выражение эквивалентно ⌜^.*▲([0-9][0-9])⌟, поэтому я использовал такие обороты, как «...продолжается с позиции перед первым классом ⌜[0-9]⌟». Однако при перемещении текущей позиции в круглые скобки и за их пределы обновляется информация о том, какой текст попадает в $1, а это влияет на эффективность.

Необходимо понимать, что для конструкции, к которой применяется * (или любой другой квантификатор), сначала находится как можно больше совпадений *независимо от того, что следует за ней в регулярном выражении*. В нашем примере ⌜.*⌟ не остановится у первой цифры, у предпоследней цифры или в любом другом месте до тех пор, пока поиск совпадения не завершится неудачей. Об этом уже упоминалось ранее, когда мы говорили о том, что выражение ⌜^.*([0-9]+)⌟ в части ⌜[0-9]⌟ никогда не совпадет более чем с одной цифрой (☞ 203).

Подробнее о максимализме и о возврате

Многие хорошие (и плохие) стороны принципа максимального захвата присущи как НКА, так и ДКА (минимальные квантификаторы в ДКА не поддерживаются, поэтому до настоящего момента речь шла о максимальных квантификаторах). Я хочу рассмотреть некоторые аспекты максимализма для обоих механизмов, но на примерах, относящихся к НКА. Все сказанное также относится и к ДКА, но по другим причинам. ДКА стремится к максимальному захвату, и точка — говорить здесь больше не о чем. Его работа отличается редкостным постоянством, но писать об этом скучно. С другой стороны, механизм НКА интересен именно творческими возможностями, которые обусловлены самой природой механизма, управляемого регулярным выражением. НКА позволяет автору регулярного выражения непосредственно управлять процессом поиска совпадений. При этом открывается немало полезных возможностей, но не обходится и без подвохов, связанных с эффективностью (проблемы эффективности обсуждаются в главе 6).

Несмотря на все отличия, поиск часто приносит одинаковые результаты. На нескольких ближайших страницах я буду говорить о механизмах обоих типов, но в примерах буду использовать более понятную терминологию поиска НКА, управляемого регулярными выражениями. К концу главы вы будете четко представлять себе, в каких случаях результаты могут различаться, а также почему это происходит.

Проблемы максимализма

Как было показано в последнем примере, `.*` всегда распространяет совпадение до конца строки[1]. Это связано с тем, что `.*` думает только о себе и стремится захватить все, что может. Впрочем, позднее часть захваченного может быть возвращена, если это необходимо для общего совпадения.

Иногда это поведение вызывает немало проблем. Рассмотрим регулярное выражение для поиска текста, заключенного в кавычки. На первый взгляд напрашивается простое `".*"`, но попробуйте на основании того, что нам известно о `.*`, предположить, какое совпадение будет найдено в строке:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Вообще говоря, поскольку вы понимаете механику поиска совпадений, предполагать не нужно — вы просто *знаете* правильный ответ. После совпадения первого

[1] В тех программах или режимах, где точка может совпадать с символом новой строки, а текстовый фрагмент может состоять из нескольких логических строк, совпадение всегда распространяется по всем логическим строкам до конца фрагмента.

символа " управление передается конструкции ⌜.*⌟, которая немедленно захватывает все символы до конца строки. Она начинает нехотя *отступать* (под нажимом механизма регулярных выражений), но только до тех пор, пока не будет найдено совпадение для последней кавычки. Если прокрутить происходящее в голове, вы поймете, что найденное совпадение будет выглядеть так:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Найден совсем не тот текст, который мы искали. Это одна из причин, по которым я не рекомендую злоупотреблять ⌜.*⌟, — если не учитывать проблем максимального совпадения, эта конструкция часто приводит к неожиданным результатам.

Как же ограничить совпадение строкой "McDonald's"? Главное — понять, что между кавычками должно находиться не «все, что угодно», а «все, что угодно, кроме кавычек». Если вместо ⌜.*⌟ воспользоваться выражением ⌜^"⌟, совпадение не пройдет дальше закрывающей кавычки.

При поиске совпадения для ⌜"[^"]*"⌟ механизм ведет себя практически так же. После совпадения первой кавычки ⌜[^"]*⌟ стремится захватить как можно большее потенциальное совпадение. В данном случае это совпадение распространяется до кавычки, следующей после McDonald's. На этом месте поиск прекращается, поскольку ⌜[^"]⌟ не совпадает с кавычкой, после чего управление передается закрывающей кавычке в регулярном выражении. Она благополучно совпадает, обеспечивая общее совпадение:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

На самом деле в этом примере возможен неожиданный поворот, связанный с тем, что в большинстве диалектов ⌜[^"]⌟ может совпадать с символом новой строки, а ⌜.⌟ — не может. Если вы хотите предотвратить возможный выход за границу логической строки, используйте выражение ⌜[^"\n]⌟.

Многосимвольные «кавычки»

В первой главе я упоминал о задаче поиска тегов HTML. Скажем, в последовательности <B>very</B> слово «very» выводится жирным шрифтом, если браузер на это способен. Задача поиска последовательности <B>…</B> на первый взгляд похожа на поиск строк, заключенных в кавычки. Правда, в роли «кавычек» на этот раз выступают многосимвольные последовательности <B> и </B>. Как и в предыдущем примере, повторяющиеся «кавычки» порождают проблемы при использовании выражения ⌜.*⌟:

```
…<B>Billions</B> and <B>Zillions</B> of suns…
```

Если воспользоваться выражением ⌜<B>.*</B>⌟, жадное подвыражение ⌜.*⌟ распространяет текущее совпадение до конца строки и отступает лишь до тех пор, пока не найдется совпадение для ⌜<B>⌟. В результате вместо тега, парного открывающему ⌜<B>⌟, совпадает последний тег </B>.

К сожалению, завершающий ограничитель состоит из нескольких символов, поэтому решение с инвертированным классом, использованное в примере с кавычками, здесь не подходит. Конечно, смехотворные попытки типа ⌜<B>[^</B>]*</B>⌟ не годятся. Символьный класс представляет только один символ, а нам нужна последовательность </B>. Заманчивая простота ⌜[^</B>]⌟ не должна сбить вас с толку. Это всего лишь класс, совпадающий с одним символом — любым символом, кроме <, >, / и B. С таким же успехом его можно было записать в виде [^/<>B]. Конечно, для поиска «всего, кроме </B>» он не годится. (Настоять на том, чтобы выражение </B> не совпадало в определенной позиции, можно, используя опережающую проверку; пример такого решения приводится в следующем разделе.)

Минимальные квантификаторы

Эта проблема возникает лишь в том случае, если стандартные квантификаторы работают по максимальному принципу. В некоторых вариантах НКА поддерживаются минимальные квантификаторы (☞ 190), при этом *? является минимальным аналогом квантификатора *. Учитывая сказанное, попробуем применить выражение ⌜<B>.*?</B>⌟ к тексту:

```
…<B>Billions</B> and <B>Zillions</B> of suns…
```

После совпадения начального ⌜<B>⌟ минимальный квантификатор немедленно решает, что обязательные совпадения не требуются, а раз так — не стоит и возиться с дальнейшими поисками, поэтому управление немедленно передается ⌜<⌟:

в строке '…<B>▲Billions…'	в выражении ⌜<B>.*?▲<B>⌟

На этой стадии ⌜<⌟ не совпадает, поэтому управление возвращается конструкции ⌜.*?⌟, у которой остались непроверенные варианты поиска совпадения (точнее говоря, многих совпадений). Квантификатор неохотно берется за дело, и метасимволу ⌜.⌟ предлагается совпадение с подчеркнутым символом B в строке …<B>Billions… . И снова *? может либо продолжить захват, либо остановиться. В нашем примере квантификатор стремится к минимальному захвату, поэтому сначала он останавливается. Сравнение следующего элемента ⌜<⌟ завершается неудачей, поэтому ⌜.*?⌟ снова переходит к непроверенному варианту совпадения. После восьми итераций ⌜.*?⌟ распространится на Billions, и в этот момент появится возможность совпадения последующего ⌜<⌟ (и всего подвыражения ⌜</B>⌟):

```
…<B>Billions</B> and <B>Zillions</B> of suns…
```

Как вы убедились, максимализм * и других квантификаторов в одних случаях приносит пользу, а в других — одни хлопоты. Минимальные, «ленивые» версии квантификаторов очень удобны, поскольку с их помощью можно решать задачи, которые другими средствами решаются с большим трудом (или вообще не решаются). Тем не менее я часто видел, как неопытные программисты используют минимальные квантификаторы в неподходящих ситуациях. Более того, в только что рассмотренном примере они тоже могут оказаться неуместными. Рассмотрим пример применения ⌜<B>.*?</B>⌟ к тексту:

…<B>Billions and <B>Zillions</B> of suns…

Совпадение выделено в тексте подчеркиванием. Наверное, все зависит от конкретной ситуации, однако вряд ли вы предполагали именно такой вариант совпадения. Тем не менее ничто не помешает ⌜.*?⌟ пройти мимо <B> в начале слова Zillions к завершающему тегу </B>.

Перед вами отличный пример того, почему минимальный квантификатор часто не обеспечивает достойной замены для инвертированного класса. В примере ⌜".*"⌟ замена точки на ⌜[^"]⌟ не позволяет механизму регулярных выражений пройти мимо ограничителя — именно это качество нам хотелось бы реализовать в текущем регулярном выражении.

Тем не менее если диалект поддерживает *ретроспективную опережающую проверку* (☞ 181), с ее помощью можно создать некое подобие инвертированного класса. Сама по себе проверка ⌜(?!<B>)⌟ успешна в том случае, если <B> не начинается с текущей позиции строки. Именно в таких позициях должна совпадать точка в выражении ⌜<B>.*?</B>⌟, поэтому замена точки на ⌜((?!<B>).)⌟ создает регулярное выражение, которое совпадает там, где нужно, и не совпадает в других местах. Если объединить все сказанное, получается весьма непростое выражение, поэтому я привожу его в режиме свободного форматирования (☞ 151) с комментариями:

```
<B>             # Совпадение для открывающего тега <B>
(               # Теперь как можно меньше экземпляров...
   (?! <B> )    # Если не <B>...
.               #               ... то подходит любой символ
)*?             #
</B>            # ... пока не совпадет закрывающий тег
```

Внеся небольшое исправление в опережающую проверку, можно вернуться к обычному максимальному квантификатору, с которым многие чувствуют себя более уверенно:

```
<B>               # Совпадение для открывающего тега <B>
(                 # Теперь как можно больше экземпляров...
```

```
   (?! </?B> )      # Если не <B> и не не <B/>...
.                   #   ... то подходит любой символ.
)*                  # (теперь максимальный)
</B>                # <ANNO>... пока не совпадет закрывающий тег
```

В этом случае опережающая проверка запрещает внутренние совпадения для `</B>` и `<B>`. Тем самым устраняется проблема, которую мы пытались решить при помощи минимальных квантификаторов, поэтому нам удастся обойтись без них. Однако и это выражение можно усовершенствовать; мы вернемся к нему во время обсуждения проблем эффективности в главе 6 (☞ 341).

Максимальные и минимальные конструкции всегда выбирают совпадение

Вспомните проблему с округлением денежных величин (☞ 82). В этой главе упомянутый пример будет подробно рассмотрен с разных точек зрения, поэтому напомню суть проблемы: из-за специфики представления вещественных чисел величины «1.625» или «3.00» иногда превращались в «1.625000000002828» или «3,00000000028822». Решение задачи выглядело так:

```
$price =~ s/(\.\d\d[1-9]?)\d*/$1/;
```

Команда удаляет все цифры, кроме первых двух или трех, из дробной части числа, хранящегося в переменной `$price`. Подвыражение ⌜`\.\d\d`⌟ всегда совпадает с первыми двумя цифрами дробной части, а ⌜`[1-9]?`⌟ совпадает с третьей цифрой лишь в том случае, если она отлична от нуля.

Затем я написал следующее:

> Все совпавшие до настоящего момента символы образуют часть, которую мы хотим *оставить*, поэтому они заключаются в круглые скобки для сохранения в переменной `$1`. Затем значение `$1` используется в строке замены. Если ничего больше не совпало, строка заменяется сама собой — не слишком интересно. Однако вне круглых скобок `$1` продолжается поиск других символов. Эти символы не включаются в строку замены, поэтому они фактически удаляются из числа. В данном случае «удаляемый» текст состоит из всех дальнейших цифр, т. е. ⌜`\d*`⌟ в конце регулярного выражения.

Все это, конечно, хорошо. Но давайте посмотрим, что произойдет, если содержимое переменной `$price` уже имеет правильный формат. Для числа `27.625` подвыражение ⌜`(\.\d\d[1-9]?)`⌟ совпадает со всей дробной частью. Поскольку завершающая конструкция ⌜`\d*`⌟ не совпадает ни с чем, подстановка заменяет '`.625`' строкой '`.625`' — в сущности, не происходит ровным счетом ничего.

Конечно, мы добиваемся нужного результата, но не лучше ли выполнять замену лишь в том случае, если она обеспечивает какой-то реальный эффект (т. е. когда ⌜\d*⌟ совпадает хотя бы с одной цифрой, которая должна удаляться из результата)? Но мы же знаем, как записать «хотя бы одну цифру»! Достаточно заменить ⌜\d*⌟ на ⌜\d+⌟:

```
$price =~ s/(\.\d\d[1-9]?)\d+/$1/
```

Для ненормальных чисел типа «1.625000000002828» все работает так же, как раньше, но для числа «9.43» завершающее подвыражение ⌜\d+⌟ совпасть не может, и подстановка в этом случае не выполняется. Отличная замена, да? *Нет!* Что происходит с числами, у которых дробная часть состоит из трех цифр (например, 27.625)? Эти числа изменяться не должны, но на практике они изменяются. Ненадолго задержитесь и попробуйте самостоятельно вычислить результат для числа 27.625. Обратите особое внимание на то, как регулярное выражение поступает с цифрой '5'.

На самом деле задача очень проста. Продолжив поиск с того момента, когда выражение ⌜(\.\d\d[1-9]?)\d+⌟ совпало с 27.625, мы выясняем, что для ⌜\d+⌟ совпадение не находится. Механизм регулярных выражений это вполне устраивает, поскольку совпадение ⌜[1-9]⌟ с 5 было *необязательным* и у него в запасе имеется сохраненное состояние, которое еще не было проверено. В этом варианте ⌜[1-9]?⌟ не совпадает ни с чем, а цифра 5 используется для обязательного совпадения ⌜\d+⌟. Таким образом, мы получаем совпадение, но только неправильное: .625 заменяется на .62, и значение становится неверным.

А если бы вместо ⌜[1-9]?⌟ использовалась конструкция с минимальным квантификатором? Мы получили бы то же самое совпадение, но без промежуточных этапов «совпадения 5 с последующим возвратом», поскольку минимальный квантификатор ⌜[1-9]??⌟ пропускает исходное совпадение. Следовательно, минимальный квантификатор проблемы не решает.

О сущности максимализма, минимализма и возврата

Вывод, который напрямую следует из предыдущего раздела: независимо от наличия в выражении минимальных или максимальных компонентов, общее совпадение имеет приоритет перед общим несовпадением. Для получения общего совпадения максимальные квантификаторы уступают символы (а минимальные квантификаторы их «принудительно получают»), поскольку при достижении «локального тупика» механизм возвращается к сохраненным состояниям и опробует непроверенные пути. И в максимальном, и минимальном случае *будут проверены все возможные пути, и только после этого механизм может решить, что совпадение в принципе недостижимо*.

Порядок перебора различается для максимальных и минимальных квантификаторов (собственно, для этого они и нужны!), но в итоге о невозможности совпадения становится известно лишь после проверки всех возможных путей.

С другой стороны, если существует только *одно* возможное совпадение, оно будет найдено регулярным выражением как с минимальным, так и с максимальным квантификатором, хотя последовательности перебора для его получения могут заметно различаться. В этих случаях выбор типа квантификатора влияет не на содержимое совпадения, а всего лишь на длину пути, по которому механизм регулярных выражений к нему придет (вопросы эффективности рассматриваются в главе 6).

Наконец, если имеются несколько возможных совпадений, хорошее понимание принципов работы минимальных и максимальных квантификаторов позволит определить, какой из вариантов будет выбран. Так, выражение ⌜`".*"`⌟ имеет три потенциальных совпадения в следующей строке:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Мы знаем, что максимальный квантификатор в ⌜`".*"`⌟ выберет самое длинное совпадение, а минимальный квантификатор в ⌜`".*?"`⌟ ограничится самым коротким совпадением.

Захватывающие квантификаторы и атомарная группировка

Приведенный выше пример с '`.625`' демонстрирует важные особенности поиска НКА. В этом конкретном примере наши наивные ожидания были опровергнуты. Некоторые диалекты поддерживают средства, способные помочь в решении этой задачи, но прежде чем знакомиться с ними, абсолютно необходимо в полной мере разобраться в предыдущем подразделе «О сущности максимализма, минимализма и возврата». Если у вас остаются сомнения, обязательно вернитесь к нему.

Итак, вернемся к примеру с '`.625`' и попробуем определить, чего же мы фактически добиваемся. Если совпадение успешно добралось до отмеченной позиции в выражении ⌜`(\.\d\d[1-9]?)\d+`⌟, оно не должно возвращаться назад. Иначе говоря, мы хотим, чтобы подвыражение ⌜`[1-9]`⌟ по возможности совпадало, и если оно совпало — символы из этого совпадения возвращаться уже не должны. Выражаясь категоричнее, мы хотим, чтобы вся попытка поиска совпадения завершалась неудачей, если потребуется уступить что-либо из совпадения ⌜`[1-9]`⌟ (если вы еще не забыли, проблема с этим выражением состояла как раз в том, что поиск *не завершался* неудачей, а возвращался для проверки последнего варианта с пропуском ⌜`[1-9]`⌟).

А нельзя ли как-то исключить этот вариант (ликвидировать состояние, сохраненное квантификатором ⌜?⌟ перед попыткой найти совпадение для ⌜[1-9]⌟)? Если не будет состояния, к которому можно вернуться, совпадение для ⌜[1-9]⌟ останется в неприкосновенности. Именно это нам и нужно! Но если не будет состояния с нулевым количеством экземпляров, к которому можно вернуться, то что произойдет в случае применения регулярного выражения к строке '.5000'? Совпадения для ⌜[1-9]⌟ нет, а в этом случае механизм должен вернуться и пропустить ⌜[1-9]⌟, чтобы следующее подвыражение ⌜\d+⌟ совпало с удаляемыми цифрами.

На первый взгляд кажется, что эти две цели исключают друг друга, но подумайте — в действительности мы хотим, чтобы вариант без совпадения ⌜[1-9]⌟ исключался только в том случае, если вариант с совпадением ⌜[1-9]⌟ возможен. То есть если ⌜[1-9]⌟ может совпасть, нужно избавиться от сохраненного состояния без совпадения ⌜[1-9]⌟, чтобы предотвратить возможную уступку символов. Такая возможность *существует* в диалектах регулярных выражений, поддерживающих атомарную группировку ⌜(?>…)⌟ (☞ 187) или захватывающие квантификаторы типа ⌜[1-9]?+⌟ (☞ 190). Начнем с рассмотрения атомарной группировки.

Атомарная группировка ⌜(?>…)⌟

Поиск совпадения для ⌜(?>…)⌟ проходит как обычно, но в тот момент, когда процесс поиска выходит за пределы конструкции (т. е. за закрывающую круглую скобку), все сохраненные состояния, созданные внутри скобок, удаляются. На практике это означает, что после выхода из атомарной группировки весь текст, совпавший внутри нее, считается одной неизменяемой единицей и включается или исключается из совпадения только как единое целое. Все сохраненные состояния, представляющие неопробованные варианты внутри скобок, уничтожаются, поэтому в процессе возврата никогда не отменяется ни одно решение, принятое в скобках (во всяком случае после его «закрепления» при выходе из конструкции).

Рассмотрим выражение ⌜\.\d\d(?>[1-9]?)\d+⌟. Внутри атомарных групп квантификаторы работают как обычно, поэтому если для ⌜[1-9]⌟ не удается найти совпадение, регулярное выражение возвращается к сохраненному состоянию, оставленному ⌜?⌟. Это позволяет процессу поиска выйти из атомарной группировки и продолжить подбор вариантов для ⌜\d+⌟. В этом случае при выходе управления за пределы атомарной группировки не остается ни одного сохраненного состояния, созданного внутри нее.

Тем не менее если совпадение для ⌜[1-9]⌟ *существует*, поиск может выйти за пределы атомарной группировки, но на этот раз сохраненное состояние, оставленное ⌜?⌟, остается. Поскольку оно было создано внутри атомарной группировки, за пределы которой выходит поиск, это состояние удаляется. Это произойдет как при поиске

в тексте '.625', так и, скажем, '.625000'. В последнем случае уничтожение сохраненного состояния ни на что не влияет, поскольку ⌜\d+⌟ совпадает с '.625000' после применения регулярного выражения. Что касается '.625', отсутствие совпадения для ⌜\d+⌟ приводит к попытке возврата, но возвращаться некуда — вариант без совпадения для ⌜[1-9]⌟ был уничтожен. Таким образом, все попытки совпадения завершатся неудачей, а строка '.625' остается без изменений, как мы и хотели.

Сущность атомарной группировки

В разделе «О сущности максимализма, минимализма и возврата» (с. 222) утверждается одно важное обстоятельство: максимализм и минимализм влияют не на то, *какие* варианты будут проверены в процессе поиска, а лишь на *порядок* их проверки. Если совпадения не существует, то и при минимальном, и при максимальном порядке решение будет принято лишь после проверки всех возможных вариантов.

Атомарная группировка принципиально отличается от них тем, что она *уничтожает потенциальные варианты*. Удаление сохраненных состояний может приводить к различным последствиям в зависимости от ситуации:

- **Без последствий.** Если совпадение будет найдено до того, как механизм мог бы обратиться к одному из удаленных состояний, удаление не влияет на результат. Мы только что встречались с подобным примером '.625000'. Совпадение было найдено до того, как удаленное состояние могло вступить в игру.
- **Запрет совпадения.** Удаление состояния может привести к тому, что возможное совпадение станет невозможным, а поиск завершится неудачей. Именно это произошло в примере '.625'.
- **Другое совпадение.** В некоторых случаях удаление состояния может привести к тому, что в результате поиска будет найдено *другое* совпадение.
- **Ускорение отказа.** Иногда удаление состояния не влияет ни на что и всего лишь позволяет механизму регулярных выражений быстрее прийти к выводу об отсутствии совпадений. Эта тема рассматривается ниже.

Простенький вопрос: что делает конструкция ⌜(?>.*?)⌟? Как вы думаете, с чем она совпадет? ❖ Проверьте свой ответ, перевернув страницу.

Некоторые состояния могут оставаться без изменений. Обратите внимание: когда механизм регулярных выражений *выходит за пределы атомарной группировки* в процессе поиска, удаляются только состояния, созданные *внутри атомарной группы*. Состояния, которые существовали до входа в нее, остаются без изменений, поэтому все атомарное подвыражение может быть исключено из совпадения как единое целое, если позднее возврат приведет к восстановлению одного из этих состояний.

ОТВЕТ

❖ *Ответ на вопрос со с. 225.*

С чем совпадает `(?>.*?)`?

Никогда и ни с чем! Это довольно замысловатый способ потратить время впустую! `*?` — это минимальный вариант квантификатора `*`, поэтому сначала будет опробован вариант с отсутствием совпадения для точки, а поиск совпадения будет отложен на будущее в виде сохраненного состояния. Но сразу же после сохранения это состояние будет немедленно уничтожено, поскольку поиск выходит за пределы атомарной группы, и вариант с наличием совпадения проверяться не будет. Если какой-то вариант всегда игнорируется, с таким же успехом можно считать, что он вовсе не существует.

Ускорение отказа при использовании атомарной группировки. Теперь рассмотрим применение выражения `^\w+:` к строке '`Subject`'. С первого взгляда можно сказать, что поиск завершится неудачей, потому что в тексте нет ни одного двоеточия, но механизм регулярных выражений придет к этому выводу лишь после полной проверки.

Итак, к моменту первой проверки `:` конструкция `\w+` распространится до конца строки. Тем самым порождается множество сохраненных состояний — по одному состоянию исключения для каждого символа, включаемого в потенциальное совпадение `\w+` (кроме первого, поскольку одно совпадение является обязательным). Когда попытка найти `:` в конце строки завершается неудачей, механизм регулярных выражений отступает к последнему из сохраненных состояний:

в строке '`Subjec▲t`'	в выражении `^\w+▲:`

Здесь попытка найти совпадение для `:` тоже завершается неудачей (при этом проверяется символ '`t`'). Цикл «возврат—проверка—неудача» повторяется до тех пор, пока механизм не вернется к самому первому состоянию:

в строке '`S▲ubject`'	в выражении `^\w+▲:`

Только после неудачи в этом последнем состоянии принимается решение о невозможности совпадения. Обработка возвратов требует немалого объема работы — особенно с учетом того, что мы с первого взгляда узнали об отсутствии совпадения.

Если двоеточие не совпадает за последней буквой, оно заведомо не совпадет ни с одной из букв, которую уступает квантификатор `+`.

Итак, если ни одно из состояний, порожденных `\w+`, заведомо не приводит к совпадению, проще избавить механизм регулярных выражений от хлопот с их проверкой: `^(?>\w+):`. Включение атомарной группировки основано на хорошем знании регу-

лярного выражения; мы повышаем локальную эффективность конструкции ⌜\w+⌟ за счет удаления ее сохраненных состояний (заведомо бесполезных). Если совпадение *существует*, атомарная группировка роли не играет, но при отсутствии совпадения уничтожение бесполезных состояний поможет быстрее прийти к нужному выводу (в современных реализациях эта разновидность оптимизации может применяться автоматически ☞ 319).

Как будет показано в главе 6 (☞ 340), этот прием является одним из самых полезных применений атомарной группировки. Вероятно, в будущем он станет и самым распространенным из ее применений.

Захватывающие квантификаторы ?+, *+, ++ и {max, min}+

Захватывающие квантификаторы имеют много общего с максимальными, но они никогда не расстаются даже с частью того, что им удалось захватить. Например, ⌜+⌟ в ходе поиска предварительного совпадения порождает несколько сохраненных состояний, как было показано в примере ⌜^\w+⌟. Его *захватывающий* аналог ⌜++⌟ просто уничтожает эти состояния (а скорее всего, просто не создает их).

Как нетрудно догадаться, захватывающие квантификаторы тесно связаны с атомарной группировкой. Конструкции типа ⌜^\w++⌟ почти ничем не отличаются от ⌜(?>\w+)⌟, кроме удобства записи[1]. При использовании захватывающего квантификатора выражение ⌜^(?>\w+):⌟ записывается в виде ⌜^\w++:⌟, а выражение ⌜(\.\d\d(?>[1-9]?))\d+⌟ — в виде ⌜(\.\d\d[1-9]?+)\d+⌟.

Обязательно разберитесь в том, чем ⌜(?>M)+⌟ отличается от ⌜(?>M+)⌟. Первая конструкция удаляет все неиспользованные состояния, созданные при поиске совпадения для ⌜M, что не приносит особой пользы, поскольку ⌜M таких состояний не создает. Вторая конструкция удаляет все неиспользованные состояния, созданные при поиске совпадения для ⌜M+⌟, что несомненно может принести пользу.

Вероятно, при сопоставлении ⌜(?>M)+ с ⌜(?>M+)⌟ достаточно очевидно, что вторая конструкция сравнима с ⌜M++⌟, но при переходе от захватывающих квантификаторов к атомарной группировке в более сложных выражениях (например, ⌜(\\"|[^"])*+⌟) возникает соблазн просто добавить '?>' к уже существующим круглым скобкам: ⌜(?>\\"|[^"])*⌟. Возможно, полученное выражение и справится с задачей, но следует хорошо понимать, что оно не эквивалентно исходной версии с захватывающим квантификатором; такая замена равносильна переходу от ⌜M++⌟ к ⌜(?>M)+⌟. Чтобы сохранить эквивалентность, уберите захватывающий + и заключите остаток в конструкцию атомарной группировки: ⌜(?>(\\"|[^"])*)⌟.

[1] В некоторых реализациях версия с захватывающим квантификатором обрабатывается чуть более эффективно, чем ее аналог с атомарной группировкой (☞ 319).

Возврат при позиционной проверке

Хотя на первый взгляд это не очевидно, но позиционная проверка (глава 2, ☞ 91) тесно связана с атомарной группировкой и захватывающими квантификаторами. Существуют четыре типа позиционных проверок: позитивная и негативная разновидности двух проверок — опережающей и ретроспективной. Все они просто проверяют, может ли указанное подвыражение совпасть/не совпасть так, чтобы оно начиналось с текущей позиции (опережающая проверка) или заканчивалось в ней (ретроспективная проверка).

Как же реализуется позиционная проверка в мире НКА с его сохраненными состояниями и возвратами? Подвыражения всех конструкций позиционной проверки проверяются отдельно от основного выражения. В процессе проверки по мере необходимости создаются сохраненные состояния и происходит возврат. Что же происходит, если для всего подвыражения удается найти совпадение? При *позитивной* проверке вся конструкция считается успешной, а при *негативной* — наличие совпадения является признаком неудачи. В любом случае нас интересует лишь сам факт наличия или отсутствия совпадения, поэтому весь контекст поиска, включая все оставшиеся непроверенные состояния, отбрасывается.

А если найти совпадение для подвыражения не удалось? Поскольку поиск производится в «своем отдельном мире», в нем доступны только состояния, созданные в процессе поиска совпадения для текущего подвыражения. Иначе говоря, если обнаруживается, что в результате возврата поиск должен выйти за пределы позиции, с которой начинается проверка, считается, что подвыражение совпасть не может. Для позитивной проверки это означает неудачу, а для негативной — наоборот, успех. В любом случае никаких сохраненных состояний не остается (иначе поиск совпадения для подвыражения еще не был бы закончен), поэтому и удалять ничего не требуется.

Итак, во всех случаях после обработки конструкции позиционной проверки от нее не остается никаких сохраненных состояний. Все состояния, которые могли остаться (например, в случае успешной позитивной проверки), удаляются. А где мы еще встречались с удалением сохраненных состояний? Конечно, при знакомстве с атомарной группировкой и захватывающими квантификаторами.

Имитация атомарной группировки на базе позитивной опережающей проверки

Для диалектов, поддерживающих атомарную группировку, эта тема представляет чисто академический интерес, но в других диалектах она способна принести практическую пользу: *если* язык поддерживает позитивную опережающую проверку и *если* в ней поддерживаются сохраняющие круглые скобки (в большинстве

диалектов это требование выполняется, но есть и исключения — например, Tcl), атомарную группировку и захватывающие квантификаторы можно имитировать другими средствами. Конструкция ⌜`?>`*выражение*⌟ имитируется выражением ⌜`(?=(`*выражение*`))\1`⌟ — например, сравните ⌜`^(?>\w+):`⌟ с ⌜`^(?=(\w+))\1:`⌟.

В варианте с опережающей проверкой ⌜`\w+`⌟ совпадает на максимальную длину текста и захватывает целое слово. Поскольку конструкция находится внутри условия опережающей проверки, сразу после совпадения все промежуточные состояния будут удалены (кстати говоря, то же самое произошло бы и при атомарной группировке). В отличие от атомарной группировки, совпавшее слово не включается в совпадение (в чем, собственно, и заключается главная особенность опережающей проверки), но остается сохраненным. На этом важном факте основано приведенное решение; выражение ⌜`\1`⌟ относится к тексту, который был получен при сохранении, поэтому совпадение для него заведомо существует. Дополнительное применение ⌜`\1`⌟ нужно просто для того, чтобы включить проверяемое слово в совпадение.

Представленный способ несколько уступает по эффективности настоящей атомарной группировке, что связано с затратами времени на повторный поиск совпадения для ⌜`\1`⌟. Но благодаря удалению состояний результат определяется быстрее, чем для простой конструкции ⌜`\w+`⌟ при отсутствии совпадения для ⌜`:`⌟.

Максимальна ли конструкция выбора?

Принцип обработки конструкции выбора играет важную роль, поскольку в разных механизмах она может обрабатываться совершенно различными способами. При передаче управления конструкции выбора в строке может совпасть любое количество из перечисленных альтернатив, но какой из них будет отдано предпочтение? Иначе говоря, если совпасть могут несколько альтернатив, какая из них в итоге совпадет? Если самая длинная, то можно сказать, что конструкция выбора максимальна. Если всегда совпадает самая короткая альтернатива, то можно сказать, что конструкция выбора минимальна. Так к какой из этих категорий относится конструкция выбора (и относится ли к какой-нибудь)?

Рассмотрим традиционный механизм НКА, используемый в Perl, пакетах Java, языках .NET и множестве других программ (☞ 194). Столкнувшись с конструкцией выбора, он последовательно проверяет все альтернативы в порядке их перечисления в выражении. Рассмотрим выражение ⌜`(Subject|Date):•`⌟. Когда механизм достигает конструкции выбора, он пытается найти совпадение для первой альтернативы, ⌜`Subject`⌟. Если совпадение будет найдено, проверяется оставшаяся часть регулярного выражения, ⌜`:•`⌟. Если выяснится, что общее совпадение невозможно, и при этом остаются другие непроверенные альтернативы (в данном случае ⌜`Date`⌟), механизм регулярных выражений возвращается и проверяет их. *Это еще*

один случай, когда механизм возвращается к точке с непроверенными вариантами. Перебор продолжается до тех пор, пока не будет найдено общее совпадение или пока не закончатся все варианты (в данном случае — альтернативы).

Итак, какой текст совпадет с выражением ⌜tour|to|tournament⌟, примененным к строке 'three•tournaments•won' в традиционном механизме НКА? Механизм пробует применить все альтернативы (неудачно) во всех позициях выражения, пока не будет достигнута позиция 'three•▲tournaments•won'. На этот раз совпадает первая альтернатива, ⌜tour⌟. Поскольку конструкция выбора завершает регулярное выражение, при нахождении совпадения для ⌜tour⌟ совпадает все регулярное выражение. Другие альтернативы даже не рассматриваются.

Итак, мы видим, что конструкция выбора не максимальна и не минимальна, но *упорядочена* — по крайней мере, в традиционном НКА. Упорядоченные конструкции выбора мощнее максимальных, поскольку они предоставляют автору регулярного выражения больше возможностей для управления поиском совпадения — он может действовать по принципу «попробовать сначала одно, затем другое и, наконец, третье, пока не будет найдено совпадение».

Упорядоченный выбор поддерживается не всеми диалектами. В ДКА и POSIX НКА поддерживается максимальный выбор, при котором всегда находится совпадение для альтернативы, совпадающей с текстом наибольшей длины (в данном примере ⌜tournament⌟). Но в Perl, PHP, в языках .NET, `java.util.regex` и в любой другой системе с традиционным механизмом НКА (☞ 194) конструкция выбора является упорядоченной.

Использование упорядоченного выбора

Вернемся к примеру ⌜(\.\d\d[1-9]?)\d*⌟ со с. 214. Если вы поймете, что ⌜\.\d\d[1-9]?⌟ фактически означает «либо ⌜\.\d\d⌟, либо ⌜\.\d\d[1-9]⌟», все выражение можно записать в виде ⌜(\.\d\d|\.\d\d[1-9])\d*⌟ (для подобных изменений особых причин нет — это всего лишь удобный пример). Но *действительно* ли оно эквивалентно первоначальному выражению? Для максимальной конструкции выбора это действительно так, но для упорядоченного выбора эти два выражения различаются.

Начнем с упорядоченной конструкции. Сначала проверяется первая альтернатива, ⌜\.\d\d⌟. Если она совпадает, управление передается подвыражению ⌜\d*⌟, следующему за ней. Если в тексте еще остаются цифры, ⌜\d*⌟ с ними совпадет, включая любую начальную цифру, в том числе и отличную от нуля, ставшую источником всех проблем в исходном примере (если вспомнить исходную задачу, это та самая цифра, которая должна совпадать только внутри круглых скобок, а не в подвыражении ⌜\d*⌟ после них). Также обратите внимание на то, что если первая альтернатива

не совпадает, то и вторая заведомо не совпадет, поскольку она начинается с копии первой альтернативы. Тем не менее механизм регулярных выражений потратит время на напрасные поиски.

Один интересный момент: если использовать выражение ⌜(\.\d\d[1-9]|\.\d\d)\d*⌟, полностью идентичное предыдущему, но с переставленными альтернативами, мы фактически получим копию исходного максимального выражения ⌜(\.\d\d[1-9]?)\d*⌟. В этом случае перестановка осмыслена — если первая альтернатива завершится неудачей из-за ⌜[1-9]⌟, у второй альтернативы еще остаются шансы. Конструкция выбора остается упорядоченной, но теперь альтернативы стоят в таком порядке, чтобы предпочтение отдавалось максимальному варианту.

Распределив ⌜[1-9]?⌟ по двум альтернативам и поставив более короткую альтернативу в начало, мы в каком-то отношении смоделировали минимальный квантификатор ⌜?⌟. В данном примере это все равно бессмысленно — если первая альтернатива не совпадает, то и вторая совпасть никак не сможет. Подобный «ложный выбор» мне встречается довольно часто, и неизменно он является ошибкой при использовании с традиционным НКА. В одной из книг, которые я читал, выражение ⌜a*((ab)*|b*)⌟ использовалось в качестве примера при объяснении чего-то, относящегося к круглым скобкам в регулярных выражениях. Поскольку совпадение для первой альтернативы, ⌜(ab)*⌟, существует всегда, все остальные альтернативы (в нашем случае только ⌜b*⌟) абсолютно бессмысленны. Можно добавить:

⌜a*((ab)*|b*|.*|partridge•in•a•pear•tree|[a-z])⌟

От этого смысл выражения нисколько не изменится. Мораль: если при упорядоченной конструкции выбора один и тот же текст может совпасть с несколькими альтернативами, будьте особенно внимательны при выборе порядка альтернатив.

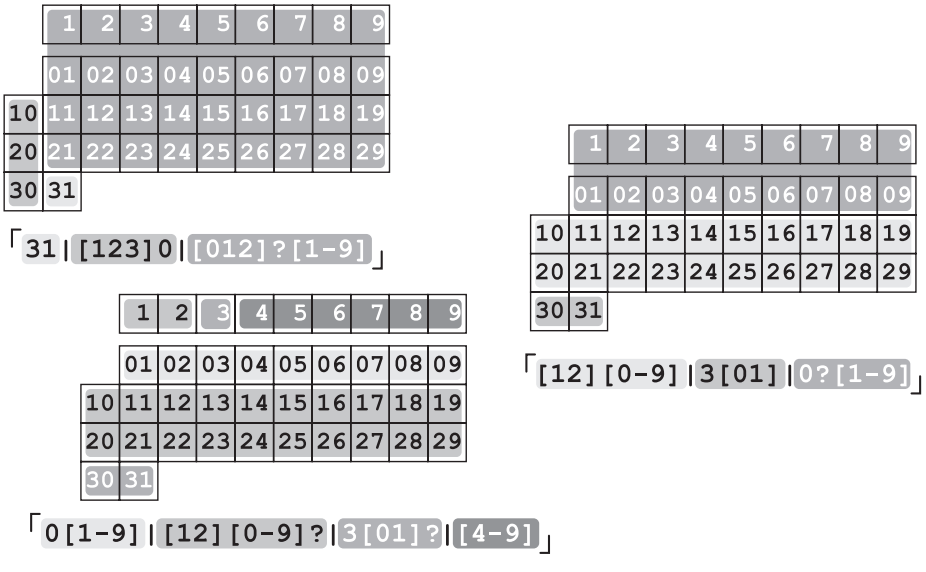
Проблемы упорядоченного выбора

Упорядоченную конструкцию выбора часто можно использовать с выгодой для себя и построить в точности такое выражение, которое вам нужно, однако для непосвященных она оборачивается неожиданными затруднениями. Допустим, вы ищете в строке январскую дату в формате ‘Jan 31’. Для этого потребуется что-нибудь более изощренное, чем ⌜Jan•[0123][0-9]⌟, поскольку при этом будут найдены «даты» ‘Jan•00’ и ‘Jan•39’, но потеряются вполне нормальные даты типа ‘Jan•7’.

Один из простых способов поиска даты заключается в том, чтобы разделить ее на секции. От 1 до 9 января используется подвыражение ⌜0?[1-9]⌟, допускающее начальный ноль. Подвыражение ⌜[12][0-9]⌟ обеспечивает поиск чисел с 10-го по 29-е, а ⌜3[01]⌟ находит оставшиеся числа месяца. Объединив эти подвыражения в конструкцию выбора, мы получаем ⌜Jan•(0?[1-9]|[12][0-9]|3[01])⌟.

РАЗНЫЕ СПОСОБЫ ПОИСКА ДАТЫ

На этом рисунке продемонстрированы разные решения задачи с поиском даты, описанной выше. Разными цветами на календаре обозначены числа, соответствующие каждой альтернативе.



Как вы думаете, с чем это выражение совпадет в строке '`Jan 31 is my dad's birthday`'? Конечно, мы хотим, чтобы оно совпало с '`Jan 31`', но минимальный выбор обеспечивает совпадение только с '`Jan 3`'. Удивлены? Во время проверки первой альтернативы, ⌜`0?[1-9]`⌟, начальное подвыражение ⌜`0?`⌟ не совпадает, но альтернатива в целом подходит, поскольку следующий элемент ⌜`[1-9]`⌟ совпадает с `3`. Выражение на этом заканчивается, и общее совпадение считается найденным.

Если бы порядок альтернатив был таким, что альтернатива с более коротким текстом стояла на последнем месте, такой проблемы не возникло бы. Следующее выражение работает нормально: ⌜`Jan•([12][0-9] |3[01]|0?[1-9])`⌟.

Другое решение задачи с поиском даты выглядит так: ⌜`Jan•(31|[123]0| [012]?[1-9])`⌟. Как и в первом решении, проблемы обходятся продуманным порядком альтернатив. Впрочем, существует и третье решение, ⌜`Jan•(0[1-9]|[12][0-9]?|3[01]?|[4-9])`⌟, которое работает правильно при любом порядке. Сравнительный анализ этих трех выражений — довольно интересное занятие (это упражнение остается читателю для самостоятельной работы, хотя приведенная выше врезка «Разные способы поиска даты» поможет вам в этом).

НКА, ДКА и POSIX

«Самое длинное совпадение, ближнее к левому краю»

Позвольте мне повторить: когда механизм ДКА ищет регулярное выражение с какой-либо позиции строки и совпадение вообще существует, то ДКА найдет самое длинное совпадение из всех возможных. Точка. Поскольку это самое длинное из всех возможных совпадений среди позиций, равноудаленных от левого края строки, это «самое длинное совпадение, ближнее к левому краю».

Самое длинное совпадение

Проблема поиска самого длинного совпадения не ограничивается конструкцией выбора. Давайте посмотрим, как НКА ищет совпадение для хитроумного выражения `⌜one(self)?(selfsufficient)?⌟` в строке `oneselfsufficient`. Сначала НКА находит совпадение для `⌜one⌟`, а потом — для максимального `⌜(self)?⌟`, после чего `⌜(selfsufficient)?⌟` сравнивается с оставшимися символами `sufficient`. Совпадения нет, но это не страшно, поскольку этот элемент является необязательным. Итак, традиционный механизм НКА возвращает `oneselfsufficient` и отбрасывает непроверенные состояния (в POSIX НКА дело обстоит иначе, но до этого мы вскоре доберемся).

С другой стороны, ДКА находит самое длинное совпадение `oneselfsufficient`. НКА тоже найдет его, если элемент `⌜(self)?⌟` почему-либо останется без совпадения, поскольку в этом случае сможет совпасть `⌜(selfsufficient)?⌟`. Традиционный механизм НКА этого не делает, но ДКА находит это совпадение всегда, поскольку оно имеет наибольшую длину. Такое становится возможным, так как механизм следит одновременно за всеми потенциальными совпадениями и постоянно располагает полной информацией о них.

Я выбрал этот глупый пример по соображениям наглядности, но вы должны понимать: эта проблема актуальна и в реальной жизни. Например, рассмотрим задачу поиска *строк продолжения*. В спецификациях данных одна логическая строка часто может распространяться на несколько «физических» строк. Признаком продолжения предыдущей строки является символ \, стоящий перед символом новой строки. Рассмотрим следующий пример:

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c \
        missing.c msg.c node.c re.c version.c
```

Для поиска строк в формате «переменная=значение» обычно используется выражение `⌜^\w+=.*⌟`, но в нем не учитываются возможные строки продолжения

(предполагается, что в используемой программе точка не совпадает с символом новой строки). Чтобы выражение находило строки продолжения, можно попытаться присоединить к нему ⌜(\\\n.*)*⌟, в результате чего получится ⌜^\w+=.*(\\\n.*)*⌟. Предполагается, что это выражение означает «любое количество дополнительных логических строк, следующих за комбинацией \+ символ новой строки». Такое решение выглядит разумно, однако в традиционном НКА оно не работает. К тому моменту, когда ⌜.*⌟ достигает символа новой строки, символ \ *уже остается позади*, и никакие компоненты выражения не заставляют механизм произвести возврат (☞ 203). Однако ДКА найдет самое длинное многострочное совпадение, если оно существует, — просто потому, что оно является самым длинным.

Если в вашем диалекте поддерживаются минимальные квантификаторы, возникает искушение воспользоваться ими — в этом случае выражение имеет вид ⌜^\w+=.*?(\\\n.*?)*⌟. Часть с экранированным символом новой строки проверяется до того, как первая точка с чем-нибудь совпадет, поэтому возникает предположение, что ⌜\\⌟ совпадет с символом \ перед новой строкой. Этот вариант тоже не работает. Минимальный квантификатор совпадает с чем-то необязательным только в том случае, если это будет абсолютно необходимо, но в нашем случае все элементы после ⌜=⌟ не являются обязательными, поэтому ничто не заставит минимальный квантификатор с чем-то совпасть. Приведенное выражение с минимальным квантификатором совпадет только с 'SRC='; конечно, это не является правильным ответом.

Существуют другие подходы к решению этой проблемы. Мы вернемся к этому примеру в следующей главе (☞ 243).

POSIX и правило «самого длинного совпадения, ближнего к левому краю»

Стандарт POSIX требует, чтобы при наличии нескольких возможных совпадений, начинающихся в одной и той же позиции, возвращалось совпадение, содержащее наибольшее количество символов.

В стандарте использовано выражение «самое длинное совпадение, ближнее к левому краю». В нем не сказано, что вы обязаны использовать ДКА. Что делать, если вы хотите реализовать в программе поддержку НКА? Если реализуется POSIX-совместимая версия НКА, вам придется обеспечить поиск всего текста oneselfsufficient и всех строк продолжения, хотя для традиционного НКА эти результаты выглядят «неестественно».

Традиционный механизм НКА останавливается с первым найденным совпадением. А что если заставить его перебрать все непроверенные варианты? Каждый раз,

когда механизм достигает конца регулярного выражения, он получает еще одно возможное совпадение. Когда будут перебраны *все* возможные варианты, механизм просто возвращает самое длинное из всех найденных совпадений. Так работает POSIX НКА.

В рассмотренном выше примере механизм НКА после найденного совпадения `(self)?` сохраняет состояние с информацией о том, что поиск совпадения для `one(self)?␣(selfsufficient)?` может быть продолжен с позиции `one␣selfsufficient`. Даже после нахождения `oneselfsufficient`, на котором традиционный механизм НКА останавливается, POSIX НКА продолжает исчерпывающий перебор остальных вариантов и со временем находит более длинное совпадение `oneselfsufficient`.

В главе 7 будет рассмотрен прием, который позволяет имитировать семантику POSIX в Perl и обеспечить совпадение наибольшей длины (☞ 421).

Скорость и эффективность

Если в традиционном НКА эффективность чрезвычайно важна из-за большого количества возвратов, то в POSIX НКА она играет еще более важную роль, поскольку возвратов становится еще больше. Механизм POSIX НКА должен каждый раз перебирать все возможные комбинации всех компонентов регулярного выражения. Примеры главы 6 наглядно показывают, что плохо написанное регулярное выражение обрабатывается гораздо медленнее.

Эффективность ДКА

Управляемый текстом механизм ДКА прекрасно справляется с проблемой неэффективности возвратов. Высокая скорость поиска достигается за счет отслеживания всех текущих потенциальных совпадений. Как же это происходит?

Перед тем как производить попытки поиска, механизм ДКА расходует дополнительное время и память на анализ регулярного выражения (более тщательный и проводимый по другому принципу, чем в НКА). Когда механизм переходит к фактическому поиску в строке, у него уже есть внутренняя карта с описанием типа «если сейчас я прочитаю такой-то символ, то он будет относиться к такому-то потенциальному совпадению». При проверке каждого символа строки механизм просто следует этой карте.

Построение карты иногда требует довольно значительных расходов времени и памяти, но после того как эта операция будет один раз выполнена для конкретного регулярного выражения, результат компиляции можно применять к тексту неограниченного объема. В нашей аналогии это напоминает зарядку аккумуляторов

НКА: ТЕОРИЯ И ПРАКТИКА

Настоящий математический смысл термина «НКА» отличается от того, что обычно называется «механизмом НКА» применительно к регулярным выражениям. Теоретически механизмы НКА и ДКА должны обеспечивать совпадение с одним и тем же текстом и обладать абсолютно одинаковыми возможностями. На практике необходимость в богатом, более выразительном синтаксисе привела к расхождению их семантик. Одним из характерных примеров является поддержка обратных ссылок.

Принцип работы механизма ДКА не позволяет реализовать обратные ссылки, но для полноценного (в математическом смысле) механизма НКА они реализуются элементарно. При этом программа обогащается новыми возможностями, но в математическом смысле она становится *нерегулярной*. Что это означает? Вероятно, то, что вы должны перейти к термину «нерегулярные выражения», поскольку с математической точки зрения он точнее описывает новую ситуацию. Использовать *термин* НКА по отношению к этому механизму тоже не совсем корректно. Но в действительности подобные формальности не соблюдаются, а термин «НКА» продолжает существовать, хотя с математических позиций реализация уже не отвечает принципам НКА.

Что все сказанное означает для пользователя? Абсолютно ничего. Пользователя не интересует, является ли выражение регулярным, нерегулярным или каким-нибудь еще. Если вы хорошо понимаете, как оно работает, то будете знать, за чем следует особо проследить.

Для тех, кому требуется более глубокое понимание теории регулярных выражений, можно порекомендовать прочитать третью главу книги «Compilers — Principles, Techniques, and Tools» (Addison-Wesley, 1996), написанной Альфредом Ахо (Alfred Aho), Рави Сези (Ravi Sethi) и Джеффри Ульманом (Jeffrey Ullman), которую из-за оригинального оформления обложки часто называют «Книгой дракона». Если быть более точным, это «Книга красного дракона», предшественницей которой была «Книга зеленого дракона» — «Principles of Compiler Design», написанная Ахо и Ульманом.

электромобиля. Сначала автомобиль некоторое время стоит в гараже, подзаряжаясь от сети, но потом он работает четко и безотказно.

Обработка регулярного выражения при его первом вхождении в программу (выполняемая всего один раз для каждого выражения) называется *компиляцией регулярного выражения*. Построение карты характерно для механизмов ДКА. Механизм НКА тоже создает внутреннее представление регулярного выражения, но оно больше напоминает мини-программу, которая затем выполняется механизмом регулярных выражений.

Сравнение ДКА и НКА

Механизмы ДКА и НКА обладают как сильными, так и слабыми сторонами.

Предварительная компиляция

Перед тем как применять регулярное выражение в процессе поиска, оба типа механизмов компилируют его во внутреннее представление в соответствии со своими алгоритмами поиска. В НКА компиляция обычно происходит быстрее и требует меньших затрат памяти. Не существует принципиальных различий между компиляцией в традиционном и POSIX-совместимом механизмах НКА.

Скорость поиска

Простой литеральный поиск в «обычных» ситуациях выполняется механизмами обоих типов с одинаковой скоростью. Скорость поиска в ДКА не связана с конкретным регулярным выражением, тогда как в НКА она напрямую зависит от него.

Чтобы традиционный механизм НКА сделал вывод об отсутствии совпадения, он должен перепробовать все возможные комбинации элементов регулярного выражения, поэтому я посвятил целую главу (глава 6) приемам написания регулярных выражений НКА, обеспечивающих быстрый поиск. Как будет показано, поиск в НКА иногда занимает целую вечность (или чуть меньше). Традиционный механизм НКА хотя бы может остановиться в тот момент, когда он найдет совпадение.

С другой стороны, POSIX НКА всегда проверяет все возможные комбинации и убеждается в том, что он нашел самое длинное возможное совпадение, поэтому успешный поиск обычно занимает столько же времени, сколько и неудачный. В POSIX НКА эффективность регулярных выражений приобретает еще большее значение.

Возможно, я немного сгустил краски — оптимизация часто сокращает объем работы, необходимой для получения ответа. Один из примеров оптимизации уже встречался нам, когда мы говорили о том, что поиск совпадений для регулярных выражений с якорным метасимволом ⌜^⌟ должен осуществляться только с начала строки (☞ 200). Множество других примеров будет рассмотрено в главе 6.

Вообще говоря, в механизме ДКА необходимость оптимизации не так уж велика (вследствие того, что он так быстро работает), но обычно дополнительная работа, выполняемая на стадии предварительной компиляции в ДКА, обеспечивает лучшую оптимизацию, чем в большинстве существующих механизмов НКА.

Современные механизмы ДКА часто пытаются сократить затраты времени и памяти на стадии компиляции за счет того, что часть работы откладывается до момента

непосредственного поиска. Большая часть затрат на стадии компиляции часто остается невостребованной из-за специфики проверяемого текста. Откладывая эту работу до того момента, когда в ней возникнет прямая необходимость, можно добиться заметной экономии времени и памяти. (Говоря техническим языком, это называется *отложенными вычислениями*.) С другой стороны, при этом возникают ситуации, когда скорость поиска в ДКА начинает зависеть от регулярного выражения и структуры проверяемого текста.

Совпадающий текст

ДКА (а также любой механизм, соответствующий стандарту POSIX) всегда находит самое длинное совпадение, ближнее к левому краю. Традиционный механизм НКА может вернуть тот же текст, но может найти и что-нибудь другое. Любой конкретный механизм всегда одинаково интерпретирует заданную комбинацию регулярного выражения с текстом, поэтому в этом смысле его поведение нельзя назвать «случайным», однако другие механизмы НКА могут работать несколько иначе. Практически все известные мне механизмы НКА работают именно так, как я описываю, однако это не гарантируется ни технологией, ни стандартами.

Возможности

Механизм НКА может поддерживать многие возможности, недоступные для ДКА, в том числе:

- сохранение текста, совпадающего с подвыражениями в круглых скобках. К этой же категории относятся обратные ссылки и полученная после поиска информация о том, *где именно* в целевом тексте были найдены частичные совпадения;
- позиционная проверка и другие сложные проверки с нулевой длиной совпадения[1] (☞ 181);
- минимальные квантификаторы и упорядоченная конструкция выбора. Механизм ДКА может легко обеспечить поиск общего совпадения минимальной длины (хотя почему-то такая возможность не поддерживалась ни в одной программе), но он не позволяет реализовать локальный минимализм и упорядоченный выбор, о чем говорилось выше;
- захватывающие квантификаторы (☞ 190) и атомарная группировка (☞ 187).

[1] В *lex* существует понятие *завершающего контекста* (trailing context), в точности эквивалентное позитивному опережению нулевой длины в конце регулярного выражения, однако оно не обобщается для использования внутри выражения.

СКОРОСТЬ ДКА С ВОЗМОЖНОСТЯМИ НКА: НИРВАНА В МИРЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ?

Я несколько раз упоминал о том, что механизм ДКА не поддерживает сохраняющих круглых скобок или обратных ссылок. Это действительно так, однако существуют комбинированные решения, в которых разные технологии объединяются для достижения нирваны. Во врезке на с. 236 говорится о том, что в стремлении к богатству возможностей механизмы НКА отклоняются от прямого и узкого пути, начертанного теорией. Вполне естественно, что то же самое происходит и с ДКА. Принцип работы ДКА усложняет задачу, но не делает ее невозможной.

В GNU *grep* было выбрано простое, но эффективное решение. Там, где это возможно, программа использует ДКА и переходит на НКА лишь при использовании обратных ссылок. Нечто похожее происходит и в GNU awk — для простых проверок принадлежности используется быстрый механизм поиска кратчайшего совпадения из GNU *grep*, а в тех ситуациях, когда необходимо знать действительные границы совпадения, задействуется другой механизм. Поскольку этот механизм построен по технологии НКА, GNU awk поддерживает сохранение текста совпадений в круглых скобках при помощи специальной функции `gensub`.

Механизм регулярных выражений Tcl относится к числу гибридных решений. Он был написан Генри Спенсером (который, как говорилось выше, сыграл важную роль на ранних стадиях разработки и популяризации регулярных выражений ☞ 124). Механизм Tcl имеет ряд общих черт с НКА — он поддерживает позиционную проверку, сохраняющие скобки, обратные ссылки и минимальные квантификаторы. Тем не менее он обеспечивает поиск «самого длинного совпадения, ближнего к левому краю» по стандарту POSIX (☞ 233), и ему не присущи некоторые недостатки НКА, о которых речь пойдет в главе 6. Пока результат выглядит весьма перспективно.

Простота реализации

Несмотря на некоторые ограничения, простейшие версии механизмов ДКА и НКА понятны и легко реализуемы. Стремление к сокращению затрат (как времени, так и памяти) и расширению возможностей все более и более усложняет реализацию.

Если взять за основу оценки объем программного кода, поддержка регулярных выражений НКА в *ed* версии 7 (январь 1979 года) занимала менее 350 строк программного кода C (кстати, весь исходный текст *grep* состоял всего из 478 строк). Бесплатно распространяемый пакет для работы с регулярными выражениями (версия 8, 1986 год), написанный Генри Спенсером, занимал почти 1900 строк на

языке C. Пакет *rx* с поддержкой POSIX НКА, написанный Томом Лордом (Tom Lord) и используемый в GNU sed, занимает уже 9700 строк.

Что касается реализаций ДКА, то механизм регулярных выражений в *egrep* версии 7 занимал чуть более 400 строк программного кода, а полноценный пакет POSIX ДКА Генри Спенсера (1992 год) содержит свыше 4500 строк.

Чтобы объединить все лучшие стороны обоих механизмов, в GNU *egrep* версии 2.0 используются два полноценных механизма (около 8300 строк программного кода), а гибридный механизм ДКА/НКА языка Tcl состоит из 9500 строк.

Впрочем, простота не всегда означает неполноценность. Недавно я захотел воспользоваться регулярными выражениями для обработки текста в Pascal. Я не программировал на Pascal со времен учебы в колледже, однако мне удалось довольно быстро написать несложный механизм НКА. Пусть он не обладает всевозможными «рюшечками и бантиками» и не оптимизирован для максимальной скорости работы, но даже этот простой пакет обладает относительно полными возможностями и приносит немалую пользу.

Итоги

Если вы с первого раза поняли все, о чем говорится в этой главе, вероятно, вам вообще незачем было ее читать. Речь идет о вещах, мягко говоря, нетривиальных. Мне понадобилось немало времени, чтобы понять этот материал, и еще больше — чтобы *осознать* его. Надеюсь, четкое, последовательное изложение поможет вам в этом. Я старался объяснять просто, но не увлекаться чрезмерным упрощением (к сожалению, это весьма распространенное явление, которое лишь мешает подлинному пониманию). Материал главы очень важен, поэтому в следующей сводке приводятся ссылки на конкретные страницы на случай, если вам потребуется быстро найти информацию по какой-либо теме.

При реализации механизмов регулярных выражений обычно используются две базовые технологии. Механизм НКА управляется регулярным выражением (☞ 204), а механизм ДКА управляется текстом (☞ 206). Расшифровки сокращений приведены на с. 204.

В сочетании со стандартом POSIX (☞ 234) и для практических целей механизмы делятся на три типа:

- Традиционный механизм НКА (аналог — бензиновый двигатель с широкими возможностями регулировки).
- Механизм POSIX НКА (аналог — бензиновый двигатель со стандартным поведением).

- Механизм ДКА, POSIX-совместимый или нет (аналог — электрический стабильно работающий двигатель).

Чтобы программа приносила максимальную пользу, необходимо знать, какой тип механизма в ней используется, и соответствующим образом строить регулярные выражения. Самым распространенным типом является традиционный механизм НКА, за ним по популярности следует ДКА. В табл. 4.1 (☞ 194) перечислены некоторые распространенные программы и типы их механизмов, а в разделе «Определение типа механизма» (☞ 196) было показано, как опытным путем определить тип механизма в незнакомой программе.

Существует универсальное правило, не зависящее от типа механизма: предпочтение отдается совпадениям, начинающимся с более ранней позиции. Это связано с последовательной проверкой регулярного выражения в каждой позиции строки (☞ 198).

Что происходит при попытке найти совпадение с произвольной позиции строки?

Механизм ДКА, управляемый текстом

Находит самое длинное совпадение. Точка. Говорить больше не о чем (☞ 233). Все очень предсказуемо, очень быстро и очень скучно (☞ 235).

Механизм НКА, управляемый регулярным выражением

«Прорабатывает» совпадение. Сердцем механизма НКА является принцип возврата (☞ 209, 214). Метасимволы управляют совпадением: одни, стандартные, квантификаторы (* и подобные) *максимальные* (☞ 203), другие — *минимальные* («ленивые») или захватывающие (☞ 213). Конструкция выбора в традиционном НКА обычно упорядочена (☞ 230), но в POSIX НКА она максимальна.

POSIX НКА всегда находит самое длинное совпадение. Однако вам не придется скучать, поскольку возникает проблема эффективности (тема главы 6).

Традиционный НКА — самый выразительный механизм регулярных выражений. Используя природу механизма, управляемого регулярным выражением, можно сделать так, чтобы найденное совпадение было именно тем, что вам нужно.

Понимание принципов и практических приемов, описанных в этой главе, закладывает основу для написания правильных и эффективных регулярных выражений. Этим двум темам и посвящаются две ближайшие главы.

5 Практические приемы построения регулярных выражений

Рассмотрев теоретические принципы построения регулярных выражений, мы применим этот материал на практике и перейдем к более сложным примерам по сравнению с теми, которые рассматривались выше. В каждом регулярном выражении необходимо выдержать баланс между двумя факторами: выражение должно совпадать там, где нужно, и не совпадать там, где не нужно. Вы уже неоднократно видели, что максимализм при грамотном применении может быть вашим другом, но при малейшей неосторожности он преподносит неприятные сюрпризы. В этой главе будет приведено еще немало примеров такого рода.

Для механизмов НКА необходимо учитывать и фактор эффективности, о котором речь пойдет в следующей главе. Плохо спроектированное регулярное выражение (даже не содержащее синтаксических ошибок) может вызвать сбои в процессе обработки.

Эта глава состоит в основном из практических задач. Я продемонстрирую ход своих размышлений при выборе подхода к их решению. Прочитайте эту главу даже в том случае, если ни один конкретный пример не связан напрямую с областью ваших интересов.

Например, даже если вы не работаете с HTML, я рекомендую разобраться в примерах обработки кода HTML. Написание хорошего регулярного выражения — нечто большее, чем простое ремесло; оно сродни искусству. Этому искусству невозможно ни научить, ни научиться одним перечислением строгих правил — требуется опыт! В представленных примерах я делюсь тем пониманием, которое приобретается лишь за годы практической работы с регулярными выражениями.

Конечно, в полной мере усвоить эту информацию можно лишь после получения собственного опыта, но изучение примеров этой главы станет хорошим первым шагом.

Балансировка регулярных выражений

В хорошо написанном регулярном выражении должны быть сбалансированы несколько факторов:

- Регулярное выражение должно совпадать там, где нужно, и нигде больше.
- Регулярное выражение должно быть понятным и управляемым.
- При использовании механизма НКА выражение должно быть эффективным (т. е. быстро приводить к совпадению или несовпадению, в зависимости от результата поиска).

Эти факторы часто зависят от контекста. Если вы работаете в режиме командной строки и хотите быстрее найти нужные строки при помощи *grep*, вероятно, вас не слишком огорчит пара лишних совпадений и вы вряд ли станете долго корпеть над построением нужного выражения. Для экономии времени следует немного поступиться качеством, поскольку полученные результаты можно будет быстро проверить. Но при работе над важным сценарием стоит затратить больше времени и усилий и сделать все на совесть: сложное регулярное выражение приносит пользу лишь в том случае, если оно работает, как было задумано. Между всеми перечисленными факторами приходится выдерживать баланс.

Впрочем, даже в программах эффективность зависит от контекста. Например, в НКА выражение ⌜`^-(display|geometry|cemap|…|quick24|random|raw)$`⌟, предназначенное для проверки аргументов командной строки, из-за большой конструкции выбора работает неэффективно. Но это выражение всего лишь проверяет аргументы командной строки, что обычно делается раз-другой в самом начале работы программы, поэтому не имеет значения, если выражение работает даже в 100 раз медленнее положенного. Просто это не тот случай, когда следует беспокоиться об эффективности. Однако если бы мы проверяли каждую строку большого файла, то неэффективность привела бы к более тяжелым последствиям.

Несколько коротких примеров

Снова о строках продолжения

Вернемся к примеру со строками продолжения (☞ 233). Как говорилось выше, выражение ⌜`^\w+=.*(\\\n.*)*`⌟ в традиционном НКА не обнаружит вторую строку фрагмента:

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c \
        missing.c msg.c node.c re.c version.c
```

Дело в том, что первое подвыражение ⌜*⌟ захватывает символ \ и «отнимает» его у подвыражения ⌜(\\\n.*)*⌟, в котором оно должно совпасть. Первый практический урок: если мы не хотим, чтобы совпадение распространялось за \, необходимо сообщить об этом в регулярном выражении, заменив каждую точку на ⌜[^\n\\]⌟ (обратите внимание на включение символа \n в инвертированный класс; предполагается, что точка не совпадает с символом новой строки, поэтому ее замена тоже не должна с ним совпадать (☞ 162)).

Выполнив необходимые изменения, мы получаем:

```
⌜^\w+=[^\n\\]*(\\\n[^\n\\]*)*⌟
```

Такое решение работает, но решив одну проблему, мы сами создали другую: теперь в строках не допускаются символы \ (кроме тех, которые завершают строку). Если такие символы встречаются внутри обрабатываемых данных, у нас возникнут проблемы. Предположить, что таких символов нет, мы не можем, поэтому регулярное выражение придется адаптировать.

До настоящего момента наше решение работало по принципу: «сначала найти совпадение для строки, а затем попытаться найти совпадение для строки продолжения, если она есть». Давайте воспользуемся другим принципом, который чаще подходит для общих случаев: сосредоточим внимание на том, что же может совпадать в каждой конкретной позиции. При поиске совпадения для строки нам нужны либо обычные символы (отличные от \ и \n), либо комбинации «\ + любой символ». Если представить эту комбинацию последовательностью ⌜\\⌟ и применить ее в режиме совпадения точки с любым символом, она также совпадет с комбинацией «\ + новая строка».

Итак, мы приходим к выражению ⌜^\w+=([^\n\\]|\\.)*⌟, применяемому в режиме совпадения точки с любыми символами. При наличии начального метасимвола ⌜^⌟ также может пригодиться расширенный режим привязки к границам строк (☞ 153); все зависит от специфики применения выражения.

Впрочем, анализ этого примера еще не закончен. Мы вернемся к нему в следующей главе и обсудим его эффективность (☞ 342).

Поиск IP-адреса

Следующий пример будет рассмотрен значительно подробнее. Наша задача — поиск адресов IP (Internet protocol), т. е. четырех чисел, разделенных точками (например, 1.2.3.4). Числа нередко дополняются нулями до трех цифр — 001.002.003.004. Если вы хотите проверить строку на присутствие IP-адреса, можно воспользоваться выражением ⌜[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*⌟, но это решение настолько неопреде-

ленное, что оно совпадает даже в строке ‘and then.....?’. Взгляните на регулярное выражение: оно даже *не требует* существования чисел — единственным требованием является наличие трех точек (между которыми нет *ничего*, кроме цифр, которых тоже может не быть).

Сначала мы заменяем звездочки плюсами, поскольку нам известно, что каждое число должно содержать хотя бы одну цифру. Чтобы гарантировать, что вся строка состоит только из IP-адреса, мы заключаем регулярное выражение в ⌜^...$⌟. Полученное выражение выглядит так:

```
⌜^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$⌟
```

Если заменить ⌜[0-9]⌟ метасимволом Perl ⌜\d⌟, выражение становится более наглядным[1]: ⌜^\d+\.\d+\.\d+\.\d+$⌟, но оно по-прежнему совпадает с конструкциями, которые не являются IP-адресами, например ‘1234.5678.9101112.131415’. (Каждое число в IP-адресе находится в интервале 0–255.) Для начала можно потребовать, чтобы каждое число состояло ровно из трех цифр, и воспользоваться выражением ⌜^\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d$⌟. Однако это требование *слишком* жесткое. Необходимо поддерживать числа, состоящие из одной и двух цифр (например, 1.234.5.67). Если в диалекте регулярных выражений поддерживается интервальный квантификатор {min, max}, можно воспользоваться выражением ⌜^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$⌟, а если нет — каждая часть легко заменяется на \d\d?\d?⌟ или ⌜\d(\d\d?)?⌟. Все эти выражения поддерживают от одной до трех цифр в каждом числе, но каждое делает это по-своему.

Возможно, в зависимости от ситуации вас устроят определенные нечеткости в построенном выражении. Если вы стремитесь к твердому соблюдению всех правил, придется учесть, что ⌜\d{1,3}⌟ может совпасть с числом 999, которое больше 255 и поэтому не является допустимым компонентом IP-адреса.

Чтобы в IP-адресе содержались только числа от 0 до 255, возможны несколько решений. Самое тупое выглядит так: ⌜0|1|2|3|...253|254|255⌟. На самом деле это решение не поддерживает дополнение чисел нулями, поэтому в действительности вам понадобится ⌜0|00|000|1|01|001|...⌟, что еще смешнее. Впрочем, для механизма ДКА вызывает смех только длина и неуклюжесть выражения — оно совпадает точно так же, как и любое регулярное выражение, описывающее этот текст. Для механизма НКА конструкция выбора делает выражение крайне неэффективным.

В более реалистичном решении нужно проследить за тем, какие цифры допускаются в числе и в каких позициях они находятся. Если число состоит всего

[1] А может, и наоборот — все зависит от того, к чему вы привыкли. В сложном регулярном выражении метасимвол ⌜\d⌟ мне кажется более наглядным, чем ⌜[0-9]⌟, но в некоторых системах их смысл может различаться. Например, в системах с поддержкой Юникода ⌜\d⌟ также может совпадать с цифрами, не входящими в набор ASCII (☞ 164).

из одной или двух цифр, принадлежность числа нужному интервалу проверять не нужно, поэтому для этих случаев вполне достаточно ⌜\d|\d\d⌟. Проверка не нужна и в том случае, если число из трех цифр начинается с 0 или 1, поскольку оно заведомо принадлежит интервалу 000–199. Следовательно, в конструкцию выбора можно добавить ⌜[01]\d\d⌟: получается ⌜\d|\d\d|[01]\d\d⌟. Полученное выражение похоже на те, которые использовались для поиска времени в главе 1 (☞ 56) и даты (☞ 232).

Число из трех цифр, начинающееся с 2, допустимо лишь в том случае, если оно равно 255 и меньше. Следовательно, если вторая цифра меньше 5, значит, число правильное. Если вторая цифра равна пяти, третья цифра должна быть меньше 6. Все сказанное выражается в форме ⌜2[0-4]\d|25[0-5]⌟.

На первый взгляд кажется, что наш анализ окончательно запутал задачу, но если поразмыслить, все становится на свои места. В результате получается выражение ⌜\d|\d\d|[01]\d\d|2[0-4]\d|25[0-5]⌟. Вообще говоря, первые три альтернативы можно объединить, и выражение придет к виду ⌜[01]?\d\d?|2[0-4]\d|25[0-5]⌟. В НКА такое решение работает более эффективно, поскольку любая неудачная альтернатива приводит к возврату. Обратите внимание: использование ⌜\d\d?⌟ в первой альтернативе вместо ⌜\d?\d⌟ немного ускоряет выявление неудачи в НКА при полном отсутствии цифр. Подробный анализ оставляю вам для самостоятельной работы — применение двух вариантов к простому примеру наглядно продемонстрирует их отличия. Возможны и другие шаги к тому, чтобы эта часть выражения работала более эффективно, но я оставлю этот аспект до следующей главы.

Итак, мы построили подвыражение, совпадающее с отдельным числом в интервале от 0 до 255. Его можно заключить в круглые скобки и подставить вместо подвыражений ⌜\d{1,3}⌟ в предыдущем примере. Окончательный результат выглядит так (выражение разбито на строки по ширине печатной страницы):

```
⌜^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.
  ([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$⌟
```

Ничего себе! А стоит ли игра свеч? Решайте сами в зависимости от своих потребностей. Это выражение все равно допускает строку 0.0.0.0, которая *семантически* неверна, поскольку в ней все цифры равны нулю. Используя опережающую проверку (☞ 181), можно запретить этот конкретный случай, разместив конструкцию ⌜(?!0+\.0+\.0+\.0+$)⌟ после начального ⌜^⌟, но в какой-то момент, в зависимости от конкретной ситуации, вы должны решить, стоит ли стремиться к дальнейшей точности — хлопоты начинают приносить все меньше пользы. Иногда бывает проще лишить регулярное выражение каких-то второстепенных возможностей. Например, можно вернуться к выражению ⌜^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$⌟, заключить каждый компонент в круглые скобки, чтобы числа сохранились в переменных $1, $2, $3 и $4, и проверить их при помощи других программных конструкций.

Знай свой контекст

Обратите внимание на два якорных метасимвола — ⌜^⌟ и ⌜$⌟. Их присутствие необходимо для правильной работы регулярного выражения. Без них оно запросто совпадет с ip=72123.3.21.9933 или (для традиционного НКА) даже с ip=123.3.21.223.

Во втором случае выражение даже не полностью совпадает с завершающим числом 223, как могло бы. На самом деле оно действительно *могло*, но в выражении нет ничего (разделительной точки или завершающего якоря), что бы форсировало это совпадение. Первая альтернатива последней группы, ⌜[01]?\d\d?⌟, совпадает с первыми двумя цифрами, и без завершающего ⌜$⌟ регулярное выражение на этом завершается. Как и в задаче с поиском даты в предыдущей главе (☞ 232), для достижения нужного эффекта можно изменить порядок альтернатив. На первое место ставится альтернатива с тремя цифрами, поэтому любое допустимое число из трех цифр будет полностью обнаружено до того, как проверка перейдет к двухцифровой альтернативе (конечно, в ДКА и POSIX НКА такая перестановка не требуется, поскольку они в любом случае выбирают самое длинное совпадение).

Но с перестановкой или без нее, первое ошибочное совпадение все равно остается. «Ага! — подумаете вы. — Проблема решается при помощи метасимволов границ слов». К сожалению, нет. Такое выражение будет находить совпадения типа 1.2.3.4.5.6. Чтобы исключить подобные внутренние совпадения, необходимо проверить окружающий контекст и убедиться в отсутствии по крайней мере алфавитно-цифровых символов и точек. Если доступна опережающая проверка, можно заключить регулярное выражение в конструкцию вида ⌜(?<![\w.])…(?![\w.])⌟ и запретить совпадения, которые начинаются сразу же после потенциального совпадения для ⌜[\w.]⌟ (или заканчиваются непосредственно перед ним). При отсутствии опережающей проверки иногда бывает достаточно заключить регулярное выражение в конструкцию ⌜(^|•)…(•|$)⌟.

Работа с именами файлов

Работа с именами файлов и каталогов (вида /usr/local/bin/perl в UNIX или \Program Files\Yahoo!\Messenger в Windows) дает множество хороших примеров регулярных выражений. По своему опыту знаю, что работать обычно интереснее, чем читать, поэтому я приведу несколько примеров, написанных на Perl, PHP, Java и VB.NET. Если эти конкретные языки вас не интересуют, можете спокойно пропустить соответствующие фрагменты кода — важны только концепции регулярных выражений, которые в них используются.

Удаление пути из полного имени

Для начала попробуем удалить путь из полного имени файла, например превратим /usr/local/bin/gcc в gcc. Толковая постановка задачи — половина решения.

В данном случае мы хотим удалить все символы до последнего / (или \ в системе Windows) включительно. Если в выражении нет ни одного символа /, значит, все нормально, и делать ничего не нужно. Я часто говорил о том, что конструкцией ⌜.*⌟ часто злоупотребляют, но ее максимализм в данном случае уместен. В выражении ⌜^.*/⌟ подвыражение ⌜.*⌟ поглощает всю строку, но затем отступает назад к последнему символу /, чтобы обеспечить совпадение.

Ниже приведены примеры удаления пути из полного имени файла, хранящегося в переменной f, на четырех языках. В первую очередь проанализируем путь к файлу в операционной системе Unix:

Язык	Код
Perl	`$f =~ s{^.*/}{};`
PHP	`$f = preg_replace('{^.*/}','',$f);`
java.util.regex	`f = f.replaceFirst("^.*/", "");`
VB.NET	`f = Regex.Replace(f, "^.*/", "")`

Регулярное выражение (или строка, интерпретируемая как регулярное выражение) подчеркнуто, а языковые компоненты выделены жирным шрифтом.

Для сравнения ниже приведены аналогичные команды для файлов Windows, в которых для разделения элементов пути к файлу используются символы \, а не /. В этих командах используется регулярное выражение ⌜^.*\\⌟. Как известно, для поиска совпадений с одиночным символом \ в целевом тексте регулярное выражение должно содержать комбинацию \\, однако во втором и третьем примерах свой отпечаток накладывают особенности используемого языка программирования — в некоторых языках необходимо использовать \ для экранирования символов, имеющих специальное назначение.

Язык	Код
Perl	`$f =~ s/^.*\\//;`
PHP	`$f = preg_replace('/^.*\\\/','',$f);`
java.util.regex	`f = f.replaceFirst("^.*\\\\", "");`
VB.NET	`f = Regex.Replace(f, "^.*\\", "")`

Интересно сравнить различия между двумя вариантами в каждом из языков. Особенно заметна последовательность \\\\, необходимая для представления \ в Java (☞ 98).

Всегда помните об очень важном обстоятельстве: что произойдет при отсутствии совпадения? В нашем случае если в строке нет ни одного символа /, подстановка не выполняется, а строка остается без изменений. Именно это нам и нужно.

При использовании механизма НКА следует учитывать и то, как механизм регулярных выражений выполняет свою работу, — от этого зависит эффективность поиска. Давайте разберемся, что произойдет, если забыть о начальном метасимволе ⌜^⌟ (а это весьма распространенное явление) и применить выражение к строке, не содержащей ни одного символа /. Как обычно, механизм регулярных выражений начинает поиск от начала строки. Конструкция ⌜.*⌟ распространяется до конца текста, но затем постепенно отступает в поисках символа / или \. Рано или поздно будут возвращены все символы, захваченные предварительным совпадением ⌜.*⌟, но совпадение так и не найдется. Механизм регулярных выражений приходит к выводу, *что совпадения не существует от начала строки*, — но это еще не все!

Происходит смещение текущей позиции, и все регулярное выражение проверяется заново с позиции второго символа. Весь процесс проверки и возврата приходится (во всяком случае, теоретически) повторять для всех новых начальных позиций в строке. Имена файлов обычно имеют ограниченную длину, поэтому в данном случае это не вызовет особых проблем, но этот принцип применим ко многим другим ситуациям — в длинных строках возникает потенциальная опасность большого количества возвратов (конечно, в ДКА такой проблемы не существует).

На практике нормально оптимизированная подсистема смещения текущей позиции знает, что почти любое регулярное выражение, начинающееся с ⌜.*⌟ и не совпадающее в начале строки, ни при каких условиях не совпадет с другой позицией, поэтому поиск совпадения производится только один раз с начала строки (☞ 311). И все же разумнее было бы с самого начала правильно сформулировать регулярное выражение, как это было сделано.

Выделение имени файла

Возможен и другой подход: просто обойти путь и ограничиться совпадением с завершающим именем файла, которое состоит из всех символов в конце строки, отличных от ⌜[^/]*$⌟. На этот раз якорный метасимвол предназначен не только для оптимизации, нам действительно нужна привязка к концу строки. Ниже показан пример реализации на Perl:

```
$WholePath =~ m{([^/]*)$}; # Применить регулярное выражение к переменной
                           $WholePath.
$FileName = $1;            # Сохранить совпавший текст
```

Обратите внимание: мы не проверяем, совпало ли регулярное выражение, поскольку совпадение *существует* всегда. Единственное обязательное *требование* заключается в совпадении знака ⌜$⌟ в конце строки, а оно выполняется даже для пустых строк. Следовательно, на совпадение подвыражения в круглых скобках можно ссылаться при помощи переменной $1, поскольку эта переменная заведо-

мо имеет определенное значение (хотя если текст завершается символом /, этим значением будет пустая строка).

Другое замечание по поводу эффективности — в НКА конструкция ⌜[^/] *$⌟ чрезвычайно неэффективна. Если внимательно разобраться в том, как механизм НКА ищет совпадение, вы увидите, что при этом происходит множество возвратов. Даже в коротком примере '/usr/local/bin/perl' окончательное совпадение находится лишь после сорока с лишним возвратов. Для примера рассмотрим попытку, начинающуюся с позиции …▲local/…. После того как ⌜[^/]*$⌟ распространяет совпадение до второй буквы l и не совпадает с символом /, метасимвол ⌜$⌟ последовательно проверяется для всех сохраненных состояний l, a, c, o и l (и каждая попытка завершается неудачей). Если этого недостаточно, большая часть этой работы повторяется с позиции …l▲ocal/…, затем с позиции …lo▲cal/… и т. д.

В данном примере это не так существенно, поскольку имена файлов обычно имеют небольшую длину, а 40 возвратов ничего не решают — вот если бы это было 40 миллионов!.. И все же вы должны принимать во внимание эти общие проблемы, чтобы потом применить их к своей конкретной ситуации.

Вероятно, стоит упомянуть еще об одном обстоятельстве. Хотя книга посвящена регулярным выражениям, это не значит, что регулярные выражения всегда обеспечивают лучшее решение — в большинстве языков программирования предусмотрены специальные функции для работы с каталогами и файлами. Но не будем останавливаться на достигнутом и двинемся дальше.

Путь и имя файла

Следующим логичным шагом станет разделение полного имени на два компонента: путь и имя файла. Существует много возможных решений; все зависит от того, чего вы добиваетесь. Прежде всего, можно воспользоваться выражением ⌜^(.*)/(.*)$⌟, чтобы присвоить соответствующие компоненты переменным $1 и $2. Выражение выглядит симметрично, но, зная, как работают максимальные квантификаторы, можно быть уверенными в том, что первое подвыражение ⌜.*⌟ никогда не оставит для $2 текст, в котором присутствует символ /. Первоначально ⌜.*⌟ захватывает все символы, после чего происходит возврат для нахождения следующего в регулярном выражении символа /. Для второй конструкции ⌜.*⌟ не остается только «возвращенная» часть. Таким образом, $1 будет содержать путь к файлу, а $2 — завершающее имя файла.

Обратите внимание: мы полагаемся на то, что первое подвыражение ⌜(.*)/⌟ не оставит второму ⌜(.*)⌟ текста, содержащего символ слэша. Принцип максимализма вам понятен, поэтому дополнительные разъяснения не понадобятся. И все же я стараюсь как можно точнее формулировать свои выражения, поэтому для имени файла предпочитаю использовать ⌜[^/]*⌟. Получается ⌜^(.*)/([^/]*)$⌟. Поскольку

это выражение наглядно показывает, что мы ищем, оно выполняет и документирующие функции.

Наше выражение обладает одним большим недостатком — оно требует, чтобы в строке присутствовал хотя бы один символ слэша. Если попытаться применить его к строке `file.txt`, совпадения не будет, а значит, не будет и информации. При правильном подходе это можно обратить себе на пользу:

```
if ( $WholePath =~ m!^(.*)/([^/]*)$! ){
    # Найдено совпадение - переменные $1 и $2 содержат информацию
    $LeadingPath = $1;
    $FileName = $2,
} else {
    # Совпадение не найдено, в полном имени отсутствует /
    $LeadingPath = "."; # Чтобы имя файла "file.txt" воспринималось как
                        # "./file.txt" ("." - текущий каталог)
    $FileName = $WholePath;
}
```

Поиск парных скобок

Поиск парных скобок (круглых, угловых и прочих) представляет определенную сложность. Задача поиска парных скобок часто встречается, например, при лексическом разборе конфигурационных файлов, исходных текстов программ и других текстов. Допустим, вы хотите каким-то образом обработать все аргументы функции в процессе анализа программы, написанной на языке C. Аргументы функции перечисляются в круглых скобках после ее имени. Они и сами могут содержать круглые скобки, обусловленные вложенными вызовами функций или группировкой операндов при выполнении математических операций. Если забыть о возможном присутствии вложенных скобок, возникает искушение воспользоваться выражением вида ⌜`\bfoo\([^)]*\)`⌟, но такое решение работать не будет.

По священной традиции программирования на C я присвоил функции-примеру имя `foo`. Предполагается, что подчеркнутая часть выражения совпадет с аргументами функции. В таких примерах, как `foo(2,•4.0)` и `foo(somevar,•3.7)`, все работает именно так, как предполагалось. К сожалению, в `foo(bar(somevar),•3.7)` возникают проблемы. Необходимо придумать что-нибудь поумнее ⌜`[^)]*`⌟.

Для поиска текста, заключенного в круглые скобки, можно предложить следующие регулярные выражения:

1. `\(.*\)` Скобки-литералы, между которыми находится произвольный текст.
2. `\([^)]*\)` От открывающей круглой скобки до следующей закрывающей круглой скобки.

3. `\([^()]*\)` От открывающей круглой скобки до следующей закрывающей круглой скобки, но с запретом других открывающих скобок между ними.

На рис. 5.1 показано, как эти выражения совпадают в тексте примера.

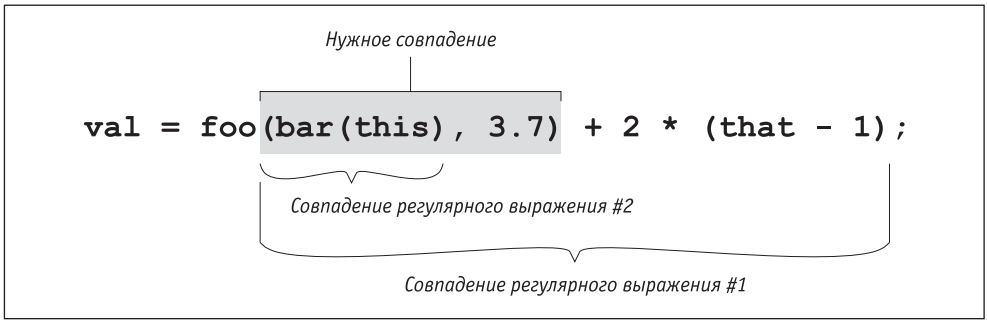


Рис. 5.1. Совпадения для приведенных вариантов регулярных выражений

Мы видим, что первое регулярное выражение забирает слишком много[1], а второе выражение довольствуется слишком малым. Третье выражение вообще не совпадает — само по себе оно бы совпало с '`(this)`', но поскольку оно должно следовать сразу же после `foo`, поиск оказывается неудачным. Итак, ни один из предложенных вариантов не работает.

В сущности, проблема в том, что в подавляющем большинстве систем *находить конструкции произвольной вложенности при помощи регулярных выражений невозможно*. В течение долгого времени это утверждение было абсолютной истиной, но недавно в Perl, .NET и PCRE/PHP появились конструкции, позволяющие реализовать подобные проверки (с. 410, 541 и 593 соответственно). Впрочем, даже без этих специальных конструкций можно построить регулярное выражение, которое будет находить вложенные конструкции до *определенного уровня вложенности, но не до произвольного уровня*. Всего один уровень вложенности требует чудовищного выражения:

```
⌜\([^()]*(\([^()]*\)[^()]*)*\)⌟
```

Одна мысль о том, чтобы взяться за следующие уровни вложенности, наводит на меня ужас. Иногда приходится прибегать к другим способам, не связанным с регулярными выражениями. Но следующий фрагмент на Perl по заданному уровню вложенности `$depth` генерирует регулярное выражение, совпадающее вплоть до

[1] ⌜`.*`⌟ говорит о том, что вы должны быть особенно внимательны и лишний раз проверить, действительно ли квантификатор * должен применяться к точке. В некоторых ситуациях именно это и требуется, но конструкция ⌜`.*`⌟ часто используется неправильно.

заданного уровня. В нем используется оператор Perl «*строка* x *число*», который повторяет *строку* в указанном количестве экземпляров:

```
$regex = '\(' . '(?:[^()]|\(' x $depth . '[^()]*' . '\))*' x $depth . '\)';
```

Читатель может проанализировать его самостоятельно.

Исключение нежелательных совпадений

Новички часто забывают о том, что происходит, если структура текста отличается от предполагаемой. Допустим, вы пишете фильтр для преобразования текстового файла в формат HTML и хотите, чтобы строка дефисов преобразовывалась в тег <HR> (этот тег рисует на странице горизонтальную линию). Если воспользоваться командой поиска и замены `s/-*/<HR>/`, она заменит все нужные последовательности, но лишь в том случае, если они находятся в начале строки. Удивлены? Это еще не все: команда `s/-*/<HR>/` добавит <HR> в начало *каждой* строки, независимо от того, начинается ли она с серии дефисов или нет!

Вспомните: все элементы, которые не являются обязательными, совпадают всегда. При попытке применить ⌜`-*`⌟ в начале строки это выражение совпадает со всеми найденными дефисами. Но если ни одного дефиса не найдено, выражение успешно совпадет с «ничем». Уж так устроен квантификатор *.

Рассмотрим похожий пример. Однажды мне попалась книга одного почтенного автора, в которой он описывает регулярное выражение для поиска чисел, целых и вещественных. В соответствии с условиями задачи число состоит из необязательного знака «минус», произвольного количества цифр в целой части, необязательной десятичной точки и произвольного количества цифр в дробной части. Приведенное решение выглядело так: ⌜`-?[0-9]*\.?[0-9]*`⌟.

В самом деле, это выражение успешно совпадает с такими примерами, как `1`, `-272.37`, `129238843.`, `.191919` и даже `-.0`. Вроде бы все хорошо.

А теперь подумайте, что произойдет, если применить это выражение к строкам '`this•text•has•no•number`', '`nothing•here`' и даже к пустой строке? Внимательно взгляните на регулярное выражение — все его элементы являются необязательными. *Если* там присутствует число и *если* оно находится в начале строки, то это число будет успешно найдено, однако ни один элемент *не является обязательным*. Выражение совпадает во всех трех случаях, причем каждый раз оно совпадает с «ничем» в начале строки. Более того, оно совпадет с «ничем» даже в начале такой строки, как '`num•123`', поскольку совпадение с «ничем» обнаруживается раньше, чем с числом!

Поэтому необходимо точно сформулировать, что же вы имеете в виду. Вещественное число *должно* содержать как минимум одну цифру, или это вообще не число.

В процессе конструирования регулярного выражения мы сначала предположим, что хотя бы одна цифра находится перед десятичной точкой. В этом случае для цифр целой части используется квантификатор +: `-?[0-9]+`.

При написании подвыражения для поиска необязательной десятичной точки и последующих цифр необходимо понять, что наличие дробной части полностью зависит от наличия самой десятичной точки. Если воспользоваться наивным выражением типа `\.?[0-9]*`, то `[0-9]*` сможет совпасть независимо от наличия десятичной точки.

Как и прежде, необходимо точно сформулировать, что же мы хотим. Десятичная точка необязательна (и цифры в дробной части, если она есть): `(\.[0-9]*)?`. Здесь вопросительный знак квантифицирует не одну десятичную точку, а комбинацию десятичной точки с последующими цифрами. *Внутри* этой комбинации десятичная точка должна присутствовать обязательно: если ее нет, `[0-9]*` вообще не получит шанса на совпадение.

Объединяя все сказанное, мы получаем `-?[0-9]+(\.[0-9]*)?`. Это выражение все еще не находит числа вида `.007`, поскольку оно требует наличия хотя бы одной цифры перед десятичной точкой. Если изменить левую часть и разрешить нулевое количество цифр, придется изменять и правую часть, потому что нельзя допустить, чтобы *все* цифры выражения были необязательными (собственно, именно эту проблему мы и пытались решить).

Одно из возможных решений — добавить для таких ситуаций специальную альтернативу: `-?[0-9]+(\.[0-9]*)?|-?\.[0-9]+`. Теперь выражение также находит числа, состоящие из десятичной точки, за которой следуют одна или несколько цифр. Подробности, подробности... Вы обратили внимание на то, что во второй альтернативе также предусмотрено наличие необязательного минуса? Об этом легко забыть. Конечно, `-?` можно вынести за пределы конструкции выбора, и тогда получится выражение `-?([0-9]+(\.[0-9]*)?|\.[0-9]+)`.

Хотя такое решение работает лучше оригинала, оно все равно найдет совпадение в строке '`2003.04.12`'. Хорошее знание контекста, в котором будет применяться регулярное выражение, помогает выдержать основное требование — обеспечить совпадения там, где нужно, и исключить их там, где они излишни. Если наше выражение будет использоваться внутри другого регулярного выражения, его следует определенным образом ограничить — например, заключить в конструкцию `^...$` или `num\s*=\s*...$`.

Поиск текста в ограничителях

Рассмотренные выше выражения для поиска IP-адресов и строк, заключенных в кавычки, демонстрируют лишь два примера из целой категории распространенных

задач. Речь идет о поиске совпадения для текста, ограниченного (или, возможно, разделяемого) другими текстовыми элементами. В числе других примеров:

- Поиск комментариев в программах на языке C, расположенных между ограничителями '`/*`' и '`*/`'.
- Поиск тегов HTML, заключенных в угловые скобки `<...>`, например `<CODE>`.
- Выборка текста, находящегося *между* тегами HTML, например '`super exciting`' в '`a <I>super exciting</I> offer!`'.
- Поиск строк в файле *.mailrc*. В этом файле определяются псевдонимы электронной почты, а каждая строка имеет формат:

  `alias` *псевдоним полный_адрес*

  Пример: '`alias jeff jfriedl@regex.info`'. В данном случае ограничителями являются пропуски между символами, а также конец строки.
- Поиск строк, заключенных в кавычки, которые могут содержать внутренние экранированные кавычки, например '`a passport needs a "2\"x3\" likeness" of the holder.`'.
- Анализ файлов в формате CSV (comma-separated values — значения, разделенные запятыми).

Обобщенное решение всех этих задач выглядит так:

1. Найти открывающий ограничитель.
2. Найти основной текст (фактически это означает «всё, что не является закрывающим ограничителем»).
3. Найти закрывающий ограничитель.

Как упоминалось выше, «поиск всего, что не является закрывающим ограничителем» значительно усложняется в том случае, если закрывающий ограничитель состоит из нескольких символов или может встречаться в основном тексте.

Поддержка экранированных кавычек

Вернемся к примеру **`2\"x3\"`**. Искомая строка заключена в кавычки, однако в тексте могут встречаться экранированные кавычки. Найти открывающую и закрывающую кавычки нетрудно, сейчас речь идет о поиске основного текста.

Давайте разберемся, что же может входить в основной текст. Мы знаем, что если символ не является кавычкой (т. е. совпадает с выражением `⌈[^"]⌋`), его следует принять. Но даже если символ *является* кавычкой, он тоже подойдет, если перед

ним находится экранирующий символ \. На языке регулярных выражений понятие «если перед ним находится» представляется конструкцией ретроспективной проверки (☞ 181). Таким образом, мы получаем выражение ⌜**"([^"]|(?<=\\)")*"**⌟, которое действительно правильно совпадает с примером **2\"x3\"**.

Перед вами идеальный пример того, как внешне правильное регулярное выражение может преподнести сюрпризы в виде нежелательных совпадений. Все выглядит правильно, но работает не всегда. Мы хотим, чтобы выражение совпадало с подчеркнутой частью следующего нелепого примера:

```
Darth Symbol: "/-|-\\" or "[^-^]"
```

В действительности будет получено совсем другое совпадение:

```
Darth Symbol: "/-|-\\" or "[^-^]"
```

Перед закрывающей кавычкой в первой строке действительно стоит символ \, но этот символ сам экранируется, поэтому он *не экранирует* следующую за ним кавычку; следовательно, кавычка *завершает* совпадение. Наша ретроспективная проверка не распознает экранирование предшествующих символов \, а если учесть, что количество вхождений '\\' может быть произвольным, такое решение создает множество потенциальных проблем. Настоящая проблема заключается в том, что символ \, экранирующий кавычку, не распознается как экранирующий префикс при первоначальной обработке. Попробуем сменить подход и взглянуть на происходящее под новым углом.

Давайте попытаемся сформулировать, какие символы могут совпадать между открывающей и закрывающей кавычкой. Допускаются любые экранированные символы (⌜\\⌟), а также любые символы, отличные от закрывающей кавычки (⌜[^"]⌟); следовательно, мы получаем выражение ⌜"(\\.|[^"])*"⌟. Ура, задача решена!.. К сожалению, не совсем. В результат по-прежнему могут вкрасться нежелательные совпадения, как показывает следующий пример, в котором совпадения быть не должно из-за отсутствия закрывающей (не экранированной) кавычки:

```
"You need a 2\"x3\" photo.
```

Почему же находится совпадение? Вспомните, о чем говорилось в разделе «Максимальные и минимальные конструкции всегда выбирают совпадение» (☞ 221). Даже несмотря на то, что регулярное выражение изначально проходит мимо последней кавычки, как и требовалось, не обнаружив завершающей кавычки, оно возвращается к следующей позиции:

в строке '...2\"x3▲\"•…'	в выражении ⌜\\.|▲[^"])⌟

В той позиции ⌜[^"]⌟ совпадает с символом \, что приводит к ложному обнаружению завершающей кавычки.

Из этого примера следует важный вывод.

> Если возврат может привести к нежелательным совпадениям из-за конструкции выбора, это с большой долей вероятности указывает на то, что любой успех является простым везением, обусловленным порядком альтернатив.

Если поменять местами альтернативы в исходном выражении, оно будет неправильно совпадать во *всех* строках, содержащих экранированные кавычки. Проблема в том, что одна альтернатива совпадает с текстом, который должен обрабатываться другой альтернативой.

Что же делать? Как и в примере со строками продолжения на с. 243, мы должны исключить возможность посторонних совпадений для символа \, для чего конструкция ⌜[^"]⌟ преобразуется в ⌜[^\\"]⌟. Такое выражение указывает на то, что кавычка и \ имеют специальную интерпретацию в этом контексте и поэтому должны обрабатываться соответствующим образом. Полученное выражение ⌜"(\\.|[^\\"])*⌟ работает нормально. Впрочем, и его можно улучшить и добиться значительного повышения эффективности в механизме НКА; мы вернемся к этому примеру в следующей главе (☞ 284).

Отсюда следует очень важный вывод.

> Всегда учитывайте, что может произойти в особых ситуациях, когда ваше регулярное выражение не должно совпадать (например, при «неправильных» данных).

Наше исправление работает, но стоит заметить, что при наличии поддержки захватывающих квантификаторов (☞ 190) или атомарной группировки (☞ 187) это выражение можно записать в виде ⌜"(\\.|[^"])*+"⌟ или ⌜"(?>(\\.|[^"])*)"⌟ соответственно. Эти варианты не столько решают проблему, сколько маскируют ее, запрещая механизму регулярных выражений возвращаться к позициям, в которых эта проблема может проявиться. Впрочем, так или иначе, они делают именно то, что вам требовалось.

Понимать роль захватывающих квантификаторов и атомарной группировки в этой ситуации чрезвычайно полезно, но я все же остановил свой выбор на предыдущем решении, поскольку в нем проще разобраться. Кстати говоря, в данном случае следовало бы использовать захватывающие квантификаторы и атомарную группировку *в сочетании* с предыдущим решением — не для того, чтобы решить проблему, а по соображениям эффективности (чтобы отсутствие совпадений обнаруживалось быстрее).

Данные и предположения

Я хочу еще раз особо выделить один общий принцип, относящийся к конструированию и использованию регулярных выражений, о котором я уже несколько раз кратко упоминал. Необходимо учитывать все предположения, относящиеся к типу используемых данных и тем условиям, для которых предназначено ваше регулярное выражение. Даже такое простое выражение, как ⌜a⌟, предполагает, что целевые данные относятся к той же кодировке (☞ 145), которую использует автор. Это понятно на уровне здравого смысла, поэтому я не надоедал вам напоминаниями.

Однако многие предположения, которые выглядят очевидными для одних, вовсе не кажутся очевидными другим. Скажем, решение из предыдущего раздела предполагает, что экранированные символы новой строки не должны включаться в совпадение или что выражение будет применяться в режиме совпадения точки со всеми символами (☞ 152). Если вы хотите гарантировать совпадение точки с символом новой строки, воспользуйтесь записью ⌜(?s:.)⌟, если она поддерживается диалектом.

Другое предположение, сделанное в предыдущем разделе, относится к типу данных, к которым будет применяться регулярное выражение. Предполагается, что посторонние кавычки в строке отсутствуют. Если попытаться применить это выражение к исходным текстам, написанным практически на любом языке программирования, окажется, что выражение не работает из-за кавычек внутри комментариев.

Нет ничего плохого в том, чтобы делать предположения о структуре данных или тех условиях, в которых будет использоваться ваше выражение. Все возникающие проблемы обычно связаны с чрезмерно оптимистичными предположениями, а также расхождениями между намерениями автора и реальным использованием регулярного выражения. Вероятно, такие предположения стоит документировать.

Удаление пропусков в начале и конце строки

Задача удаления пропусков в начале и конце строки не так уж сложна, но она часто встречается на практике. До настоящего времени самым лучшим и универсальным решением остается простая последовательность двух подстановок:

```
s/^\s+//;
s/\s+$//;
```

По соображениям эффективности вместо ⌜*⌟ в них используется ⌜+⌟, поскольку замена имеет смысл лишь при наличии хотя бы одного пропуска.

Многие программисты почему-то упорно ищут способ сделать все в одном выражении, поэтому ниже для сравнения приводятся несколько других решений. Я не

рекомендую пользоваться ими, но будет весьма познавательно разобраться в том, как они работают и в чем заключаются их недостатки.

```
s/\s*(.*?)\s*$/$1/s
```

Раньше это выражение приводилось как замечательный пример применения минимальных квантификаторов, но сейчас оно утратило популярность из-за того, что оно работает гораздо медленнее простого решения (в Perl — примерно в 5 раз). Замедление связано с тем, что перед *каждым* применением квалифицированной точки приходится проверять совпадение для ⌜\s*$⌟, что требует большого количества возвратов.

```
s/^\s*((?:.*\S)?)\s*$/$1/s
```

По сравнению с предыдущим примером это выражение выглядит более сложным, но поиск ведется прямолинейнее, а по скорости оно всего в два раза уступает простому решению. После того как исходная конструкция ⌜^\s*⌟ обойдет все начальные пропуски, ⌜.*⌟ в середине совпадает со всем текстом до конца строки. Следующее подвыражение ⌜\S⌟ заставляет механизм регулярных выражений отступить до последнего символа, не являющегося пропуском, а завершающие пропуски совпадают с последним подвыражением ⌜\s*$⌟ после круглых скобок.

Вопросительный знак необходим для того, чтобы выражение правильно работало в строках, состоящих из одних пропусков. Без него совпадение не обнаруживается, а строка остается без изменений.

```
s/^\s+|\s+$//g
```

Весьма распространенная идея. Выражение не является формально неправильным (как и все остальные приведенные выражения), но оно содержит высокоуровневую конструкцию выбора, поэтому для него не могут использоваться многие оптимизации, о которых речь пойдет в следующей главе.

Модификатор /g обеспечивает совпадение обеих альтернатив и удаление как начальных, *так и* конечных пропусков. Тем не менее было бы слишком расточительно использовать /g, когда количество совпадений заведомо не больше двух, причем с разными подвыражениями. Такое решение работает примерно в 4 раза медленнее простого.

Приведенные цифры основаны на результатах тестирования, но на практике действительные соотношения скоростей зависят от специфики конкретных программ и данных. Например, при обработке очень длинного текста, на концах которого находится небольшое количество пропусков, второе решение может работать чуть быстрее варианта с двумя заменами. Лично я использую языковой эквивалент фрагмента

```
s/^\s+//; s/\s+$//;
```

поскольку это решение почти всегда быстрее работает, а по наглядности превосходит все остальные варианты.

Работа с HTML

В главе 2 был рассмотрен пример преобразования неструктурированного текста в формат HTML (☞ 101), в котором использовались регулярные выражения для извлечения адресов электронной почты и URL. В этом разделе будут рассмотрены другие задачи, встречающиеся при работе с кодом HTML.

Поиск тегов HTML

Для поиска тегов HTML очень часто используется выражение `⌜<[^>]+>⌟`. Обычно оно работает нормально — например, следующая команда Perl используется для удаления тегов:

```
$html =~ s/<[^>]+>//g;
```

Тем не менее это выражение дает сбой, если тег содержит внутренние символы '>', как в следующей вполне допустимой команде HTML: `<input name=dir value=">">`. Спецификация HTML разрешает использовать неэкранированные символы '<' и'>' в атрибутах тегов, заключенных в кавычки или апострофы, хотя это и не рекомендуется делать. В нашем простом выражении `⌜<[^>]+>⌟` такая возможность не предусмотрена, поэтому выражение необходимо усовершенствовать.

Внутри конструкции '<...>' могут находиться последовательности, заключенные в кавычки или апострофы, а также «прочие» символы — любые, кроме >, кавычек и апострофов. Строки в HTML могут заключаться как в апострофы, так и в кавычки. Внутренние экранированные кавычки и апострофы не допускаются, поэтому для их поиска можно использовать простые выражения `⌜"[^"]*"⌟` и `⌜'[^']*'⌟`.

Объединяя все сказанное с выражением для представления «прочих» символов `⌜[^'">]⌟`, мы получаем:

```
⌜<("[^"]*"|'[^']*'|[^'">])*>⌟
```

Выражение выглядит весьма запутанным, поэтому его лучше представить в режиме свободного форматирования с комментариями:

```
<                # Открывающий символ "<"
    (            #    Произвольное количество ...
        "[^"]*"  #      строк, заключенных в кавычки,
        |        #      или...
```

```
        '[^']*'    #       строк, заключенных в апострофы,
        |          #       или...
        [^'">]     #       "прочих" символов.
    )*             #
>                  # Закрывающий символ ">"
```

Общая идея решения весьма элегантна. Каждая часть, заключенная в кавычки или апострофы, обрабатывается как единое целое, а выражение ясно показывает, какие символы могут присутствовать в каждой из совпадающих частей. Совпадение текста с более чем одной альтернативой в принципе невозможно, поэтому нам не нужно беспокоиться о возможном появлении непреднамеренных совпадений, как в предыдущих примерах.

Вы обратили внимание на то, что в первых двух альтернативах вместо ⌜+⌟ используется квантификатор ⌜*⌟? Дело в том, что строка, заключенная в кавычки или апострофы, вполне может быть пустой (например, '**alt=""**'). Но в третьей альтернативе не используется ни ⌜+⌟, ни ⌜*⌟, поскольку подвыражение ⌜[^'">]⌟ квантифицируется через внешние круглые скобки ⌜(…)*⌟. Добавление второго квантификатора (например, ⌜([^'">]+)*⌟) порождает крайне неприятные сюрпризы, суть которых вы пока не поймете; эта тема более подробно обсуждается в следующей главе (☞ 289).

Замечание по поводу эффективности, актуальное для механизмов НКА: поскольку текст, совпадающий с подвыражением в круглых скобках, не используется, их можно заменить несохраняющими круглыми скобками (☞ 185). А если учесть невозможность совпадения текста с разными альтернативами, то, когда завершающий символ ⌜>⌟ не совпадет, нет смысла возвращаться и испытывать другие альтернативы. Если одна из альтернатив совпала, начиная с некоторой позиции, никакая другая альтернатива с этой позиции уже не совпадет. Значит, будет вполне логично уничтожить сохраненные состояния, чтобы ускорить принятие решения об отсутствии совпадения. Задача решается заменой несохраняющих скобок атомарной группировкой ⌜(?>…)⌟ (или захватывающим квантификатором *, относящимся к используемым скобкам).

Поиск ссылок HTML

Предположим, вы хотите извлечь из документа URL и тексты ссылок. В следующем фрагменте нужные данные выделены подчеркиванием:

```
<a href="http://www.oreilly.com">O'Reilly Media</a>
```

Содержимое тега <A> может оказаться весьма сложным, поэтому задача будет решаться в два этапа. На первом этапе из тега <A> будут извлечены все «внутренности», включая текст ссылки, а на втором этапе в полученных данных выделяется URL.

Упрощенное решение первого этапа основано на применении минимального квантификатора * в выражении `⌜<a\b([^>]+)>(.*?)</a>⌟`, при этом поиск производится без учета регистра и в режиме совпадения точки со всеми символами. Содержимое тега `<A>` сохраняется в переменной `$1`, а текст ссылки — в переменной `$2`. Конечно, вместо `⌜[^>]+⌟` следует использовать выражение, созданное в предыдущем разделе. Однако в дальнейшем я буду использовать упрощенную версию — просто, чтобы сделать эту часть регулярного выражения более короткой и наглядной.

После того как содержимое `<A>` будет сохранено в строке, его можно будет проанализировать при помощи отдельного регулярного выражения. Адрес URL представлен значением атрибута `href=`*значение*. HTML допускает наличие пробелов по обеим сторонам от знака равенства, а *значение* может быть заключено в кавычки/апострофы (о чем говорилось в предыдущем разделе). Ниже приводится наше решение задачи с выделением ссылок, оформленное в виде мини-программы на языке Perl, которая помещает информацию о ссылке в переменную `$Html`:

```
# Регулярное выражение в цикле while(...) намеренно упрощено -
# о чем уже говорилось выше
while ($Html =~ m{<a\b(^>]+)>(.*?)</a>}ig)
{
    my $Guts = $1; # Сохранить результаты найденного совпадения…
    my $Link = $2; # …в именованных переменных.

    if ($Guts =~ m{
                  \b HREF          # Атрибут "href"
                  \s* = \s*        # "=" допускаются пропуски
                  (?:              # Значение представляет собой...
                      "([^"]*)"    # строку, заключенную в кавычки.
                      |            # или...
                      '([^']*)'    # строку, заключенную в апострофы,
                      |            # или...
                      ([^'">\s]+)  # "прочее"
                   )               #
                  }xi)             #
    {
        my $Url = $+; # Переменная с наибольшим номером из серии $1, $2,
                      # которой было присвоено фактическое значение.
        print "$Url with link text: $Link\n";
    }
}
```

Несколько замечаний по поводу приведенного фрагмента.

- На этот раз каждая альтернатива заключена в круглые скобки для сохранения фактического совпадения.

- Некоторые круглые скобки используются для сохранения, поэтому в остальных случаях я использую несохраняющие круглые скобки — как для наглядности, так и для повышения эффективности поиска.
- На этот раз в альтернативе «прочее» наряду с кавычками, апострофами и '>' исключаются пропуски, поскольку пропуски разделяют пары «атрибут=значение».
- В последней альтернативе используется квантификатор `⌜+⌟`, необходимый для сохранения всего значения `href`. Не приводит ли это к «неприятным сюрпризам», которые возникали при использовании `⌜+⌟` в альтернативе «прочее» на с. 260? Нет, потому что выражение не содержит другого квантификатора, напрямую относящегося к повторяемому классу. Напомню, что эта тема подробно рассматривается в следующей главе.

В зависимости от текста URL может оказаться в одной из переменных — `$1`, `$2` или `$3`. Остальные переменные пусты или имеют неопределенные значения. В Perl поддерживается специальная переменная `$+`, содержащая значение переменной с наибольшим номером из серии `$1`, `$2`..., которой было присвоено фактическое значение. В нашем примере именно это значение соответствует найденному URL. Специальная переменная Perl `$+` удобна, но в других программах существуют другие средства выделения сохраненного URL. Вы всегда можете проанализировать сохраненные группы при помощи обычных программных конструкций и использовать ту из них, которой было присвоено значение. Для этой цели идеально подходит именованное сохранение (☞ 186), если оно поддерживается программой, — пример для языка VB.NET приведен на с. 264 (хорошо, что в .NET поддерживается именованное сохранение, потому что функциональность `$+` на этой платформе реализована неверно, ☞ 528).

Анализ HTTP URL

Получив предполагаемый URL, мы должны проверить правильность его структуры, и если все верно, разобрать его на компоненты (имя хоста и путь). Задача проверки предполагаемого URL значительно проще, чем *идентификация* URL в произвольном тексте; эта гораздо более сложная задача будет рассмотрена позднее в этой главе.

Итак, у нас имеется строка URL, и мы хотим разделить ее на составляющие. Имя хоста состоит из всех символов, расположенных после `⌜^http://⌟`, но до следующего символа / (если он присутствует в строке), а все остальные символы образуют путь: `⌜^http://([^/]+)(/.*)?$⌟`.

Следует учесть, что между именем хоста и путем в URL может указываться и необязательный номер порта, перед которым ставится двоеточие: `⌜^http://([^/:]+)(:(\d+))?(/.*)?$⌟`.

ПРОВЕРКА ССЫЛОК В VB.NET

Следующая программа выводит информацию о ссылках в HTML-коде, хранящемся в переменной Html:

```
Imports System.Text.RegularExpressions
  ⋮
' Подготовка регулярных выражений для последующего  использования в цикле
Dim A_Regex as Regex = New Regex(                    _
            "<a\b(?<guts>[^>]+)>(?<Link>.*?)</a>",  _
            RegexOptions.IgnoreCase)

Dim GutsRegex as Regex = New Regex( _
   "\b HREF                 (?#   атрибут 'href'                    )" & _
   "\s* = \s*               (?#   '=' с необязательными пробелами   )" & _
   "(?:                     (?#   Значение представляет собой...    )" & _
   "  ""(?<url>[^""]*)""    (?#   строку в кавычках,                 )" & _
   "  |                     (?#   или...                            )" & _
   "  '(?<url>[^']*)'       (?#   строку в апострофах,               )" & _
   "  |                     (?#   или...                            )" & _
   "  (?<url>[^'""">\s]+)   (?#   прочие символы                     )" & _
   ")                       (?#                                     )" , _
RegexOptions.IgnoreCase OR RegexOptions.IgnorePatternWhitespace)

' Проверка переменной 'Html...
Dim CheckA as Match = A_Regex.Match(Html)

' Для каждого найденного совпадения...
While CheckA.Success
   ' Найден тег <a>, теперь ищем URL.
   Dim UrlCheck as Match = _
       GutsRegex.Match(CheckA.Groups("guts").Value)
   If UrlCheck.Success
      ' Найдено совпадение, вывести пару URL/ссылка
      Console.WriteLine("Url " & UrlCheck.Groups("url").Value & _
                        " WITH LINK " & CheckA.Groups("Link").Value)
   End If
  CheckA = CheckA.NextMatch
   End While
```

В этом листинге имеется ряд обстоятельств, заслуживающих внимания.

- Программы VB.NET, использующие регулярные выражения, начинаются с директивы `Imports`, которая передает компилятору информацию об используемых библиотеках объектов.

- Я использовал комментарии в стиле ⌜(?#...)⌟, потому что в строки VB.NET неудобно включать символы новой строки, а обычные комментарии '#' продолжаются до следующего символа новой строки или до конца всего текста (поэтому первый же символ # превратил бы остаток регулярного выражения в комментарий). Если вы предпочитаете обычные комментарии ⌜#...⌟, завершите каждую логическую строку конструкцией `&chr(10)` (☞ 509).

- Каждая кавычка в регулярном выражении представляется двумя кавычками в строковом литерале (☞ 142).
- В обоих выражениях используется именованное сохранение, что позволяет использовать более содержательные ссылки вида `Groups("url")` вместо `Groups(1)`, `Groups(2)` и т. д.

Следующий фрагмент на языке Perl выводит информацию об URL:

```
if ($url =~ m{^http://([^/:]+(:(\d+))?)(/.*)?$}i)
{
    my $host = $1;
    my $port = $3 || 80;   # Использовать значение $3, если оно существует;
                           # в противном случае используется стандартный
                           # порт с номером 80.
    my $path = $4 || "/";  # Использовать значение $4, если оно существует;
                           # по умолчанию используется значение "/".
    print "Host: $host\n";
    print "Port: $port\n";
    print "Path: $path\n";
} else {
    print "Not an HTTP url\n";
}
```

Проверка имени хоста

В предыдущем примере совпадение с именем хоста определялось выражением `⌜[^/:]+⌟`. С другой стороны, в главе 2 (☞ 111) использовалось более сложное выражение `⌜[-a-z]+(\.[-a-z]+)*\.(com|edu|…|info)⌟`. Чем объясняются такие различия, если речь идет практически об одном и том же?

Хотя оба выражения обеспечивают совпадение с именем хоста, они используются по-разному. Одно дело — извлечь что-то из готового текста (например, из строки, которая заведомо содержит URL), и совсем другое — точно и безошибочно выбрать информацию того же типа из произвольного текста. В предыдущем примере мы решили, что после '`http://`' следует имя хоста, поэтому простая выборка с выражением `⌜[^/:]+⌟` с учетом такого допущения выглядит вполне разумно. Но в примере из главы 2 регулярное выражение применяется для поиска имени хоста в произвольном тексте, поэтому оно должно быть гораздо более точным.

Перейдем к третьей задаче, связанной с именами хостов, — их проверке по регулярным выражениям. На этот раз нужно убедиться в том, что строка содержит нормально сформированное, синтаксически правильное имя хоста. Официально имя хоста

представляет собой набор компонентов, разделенных точками, причем компоненты состоят из символов ASCII, цифр и дефисов, но не могут начинаться с дефисов. Следовательно, для поиска отдельного компонента можно воспользоваться регулярным выражением ⌜[a-z0-9]|[a-z0-9][-a-z0-9]*[a-z0-9]⌟, примененным без учета регистра символов. Как упоминалось в главе 2, суффиксы ('com', 'edu', 'uk' и прочие) принимают значения из ограниченного набора. Таким образом, мы получаем следующее регулярное выражение для проверки синтаксически правильного имени хоста:

```
^
  (?i) # Выражение должно применяться без учета регистра символов.
  # Один или более компонентов, разделенных точками...
  (?: [a-z0-9]\. | [a-z0-9][a-z0-9]*[a-z0-9]\. )+
  # За которыми следует завершающий суффикс...
  (?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )
$
```

Совпадение с этим регулярным выражением еще не гарантирует правильности имени хоста, поскольку существует дополнительное ограничение: длина каждого компонента не может превышать 63 символов. Это означает, что подвыражение ⌜[a-z0-9]*⌟ следует заменить на ⌜[a-z0-9]{0,61}⌟.

Остается внести последнее изменение, которое носит формальный характер. Формально имена, состоящие только из суффиксов ('com', 'edu' и т. д.), тоже считаются синтаксически правильными. На практике у этих «имен» обычно не существует нормальной интерпретации, но это не относится к двухбуквенным кодам стран. Например, домен верхнего уровня Ангильи, 'ai', обслуживается веб-сервером: при обращении по адресу *http://ai/* в браузере загружается страница. Среди других примеров подобного рода можно упомянуть cc, co, dk, mm, ph, tj, tv, tw.

Чтобы обеспечить поддержку вырожденных случаев, замените центральное подвыражение ⌜(?:…)+⌟ на ⌜(?:…)*⌟. В результате мы получаем следующий фрагмент:

```
^
  (?i) # Выражение должно применяться без учета регистра символов.
  # Ноль или более компонентов, разделенных точками...
  (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[-a-z0-9]\. )*
  # За которыми следует завершающий суффикс...
  (?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )
$
```

Приведенное выражение отлично справляется с проверкой строки, содержащей имя хоста. Из всех трех регулярных выражений, предназначенных для проверки имен хостов, оно специализировано в наибольшей степени, поэтому возникает вопрос: нельзя ли удалить якорные метасимволы? Может, полученное выражение лучше подойдет для извлечения имен хостов из произвольного текста? К сожалению, нет. Регулярное выражение совпадает с любым двухбуквенным словом, поэтому более общее регулярное выражение из главы 2 на практике работает

лучше. Впрочем, как показано в следующем разделе, в некоторых ситуациях и оно оказывается недостаточно хорошим.

Поиск URL на практике

Работая на Yahoo! Finance, я писал программы для обработки поступающих финансовых новостей и данных. Статьи обычно принимались в простом текстовом формате, а мои программы преобразовывали текст в HTML, чтобы он лучше смотрелся (если вы интересовались финансовыми новостями на сайте *http://finance.yahoo.com* в последние 10 лет, то наверняка имели возможность увидеть, как я справился с этой задачей).

Задача часто оказывалась весьма нетривиальной, что объяснялось непредсказуемым «форматированием» (или его отсутствием) в полученных данных, а также тем, что *находить* в произвольном тексте такие конструкции, как имена хостов и URL, гораздо сложнее, чем *проверять* их после обнаружения. Предыдущий раздел можно рассматривать как вступление, а в этом разделе я продемонстрирую код, который реально использовался в Yahoo! для решения поставленных задач.

Программа должна извлекать из текста URL нескольких видов — `mailto`, `http`, `https` и `ftp`. Если в тексте находится префикс '`http://`', скорее всего, он отмечает начало URL, поэтому для выделения имени хоста может использоваться простое выражение типа ⌜`http://[-\w]+(\.-\w[\w]*)+`⌟. Знание контекста применения (обычного английского текста в виде ASCII-символов) позволяет заменить ⌜`[-a-z0-9]`⌟ на ⌜`-\w`⌟. Метасимвол ⌜`\w`⌟ также совпадает с символом подчеркивания, а в некоторых системах — со всеми буквами Юникода, но мы знаем, что в этой конкретной ситуации это несущественно.

Однако адреса URL часто задаются без префиксов `http://` или `mailto:`:

```
visit us at www.oreilly.com or mall to orders@oreilly.com
```

В таких случаях необходимо действовать осторожнее. Регулярное выражение обладает некоторым сходством с выражением из предыдущего раздела, но в некоторых аспектах отличается от него:

```
(?i: [a-z0-9] (?:[a-z0-9]*[a-z0-9])? \. )+ # Домены нижних уровней
# Окончания .com и прочие должны записываться строчными буквами
(?i: com\b
   | edu\b
   | biz\b
   | org\b
   | gov\b
   | in(?:t|fo)\b # .int или .Info
   | mil\b
   | net\b
   | name\b
```

```
   | museum\b
   | coop\b
   | aero\b
   | [a-z][a-z]\b # Двухбуквенные коды стран
)
```

В этом выражении конструкции ⌜(?i:…)⌟ и ⌜(?i:…)⌟ управляют режимом поиска без учета регистра символов в заданных частях регулярного выражения (☞ 181). Наше выражение должно находить URL вида ‘www.OReilly.com’, но не последовательности вида ‘NT.TO’ (биржевое обозначение Nortel Networks на фондовой бирже Торонто) — не забывайте, мы обрабатываем финансовые новости и данные, в которых часто встречаются биржевые котировки. Теоретически суффикс URL (например, ‘.com’) может записываться в верхнем регистре, но наше выражение такие URL не распознает. Здесь проявляется компромисс между поиском нужных совпадений (практически все URL, встречающиеся на практике), исключением лишних совпадений (обозначения биржевых котировок и т. д.) и простотой выражения. Вероятно, подвыражение ⌜(?-i:…)⌟ можно было бы переместить так, чтобы оно относилось только к кодам стран, но в нашем примере запись URL в верхнем регистре не используется, поэтому оставим все без изменений.

Ниже приведена заготовка для поиска URL в произвольном тексте, в которую вставляется подвыражение для имени хоста:

```
\b
# Начало URL (префикс://имя_хоста или просто имя_хоста)
(
    # ftp://, http:// или https://
    (ftp|https?)://[-\w]+(\.\w[-\w]*)+
  |
    # Или попытаться найти имя хоста по уточненному подвыражению
    полное_регулярное_выражене_для_имени_хоста
)
# Разрешить необязательный номер порта
( : \d+ )?

# Остаток URL не является обязательным и начинается с / ...
(
   / путь
)?
```

Мы еще не обсуждали подвыражение для поиска пути, следующего за именем хоста (т. е. подчеркнутой части http://www.oreilly.com/catalog/ regex/). Оказывается, поиск правильного совпадения для пути — довольно сложная задача, требующая ряда неформальных допущений. Как упоминалось в главе 2, символы, следующие *после* URL в тексте, также могут присутствовать в URL. Пример:

```
Read his comments at http://www.oreilly.com/ask_tim/index.html. He...
```

Точка после '`index.html`' является обычным знаком препинания и в URL не входит, но точка *внутри* '`index.html`' *является* частью URL.

Человек легко различит эти два случая, но для программы это довольно трудно, поэтому нам придется разработать систему эвристических правил, которые по мере возможности справляются с этой задачей. В примере из главы 2 использовалась ретроспективная проверка, которая гарантировала, что URL не может завершаться стандартными знаками препинания. Во время написания программ Yahoo! Finance ретроспективная проверка еще не поддерживалась, поэтому программа получается более сложной, но приводит к тому же результату (листинг приведен ниже).

Регулярное выражение для извлечения URL из финансовых новостей

```
\b
# Начало URL (префикс://имя_хоста или просто имя_хоста)
(
    # ftp://, http:// или https://
    (ftp|https?)://[-\w]+(\.\w[-\w]*)+
  |
    # Или попытаться найти имя хоста по уточненному подвыражению
    (?i: [a-z0-9] (?:[-a-z0-9]*[a-z0-9])? \. )+ # Домены нижних уровней
     # Окончания .com и прочие должны записываться строчными буквами
     (?-i: com\b
         | edu\b
         | biz\b
         | gov\b
         | in(?:t|fo)\b # .int или .info
         | mil\b
         | net\b
         | org\b
         | [a-z][a-z]\b # двухбуквенные коды стран
    )
)

# Разрешить необязательный номер порта
( : \d+ )?

# Остаток URL не является обязательным и начинается с / ...
(
    /
    # Эвристические правила, которые хорошо работают в нашем случае
    [^.!,?;"'<>()\[\]{}\s\x7F-\xFF]*
    (?:
       [.!,?]+ [^.!,?;"'<>()\[\]{}\s\x7F-\xFF]+
    )*
)?
```

ПОСТРОЕНИЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ ИЗ ПЕРЕМЕННЫХ JAVA

```
String SubDomain   = "(?i:[a-z0-9]|[a-z0-9][-a-z0-9]*[a-z0-9])";
String TopDomains  = "(?x-i:com\\b          \n" +
                     "     |edu\\b          \n" +
                     "     |biz\\b          \n" +
                     "     |in(?:t|fo)\\b   \n" +
                     "     |mil\\b          \n" +
                     "     |net\\b          \n" +
                     "     |org\\b          \n" +
                     "     |[a-z][a-z]\\b   \n" + // коды стран
                     "                     \n";
String Hostname  = "(?:" + SubDomain + "\\.)+" + TopDomains;

String NOT_IN    = ";\"'<>()\\[\\]{}\\s\\x7F-\\xFF";
String NOT_END   = "!.,?";
String ANYWHERE  = "[^" + NOT_IN + NOT_END + "]";
String EMBEDDED  = "[" + NOT_END + "]";
String UrlPath   = "/"+ANYWHERE + "*("+EMBEDDED+"+"+ANYWHERE+"+)*";
String Url =
  " (?x:                                                    \n"+
  "  \\b                                                    \n"+
  "  ## поиск имени хоста                                   \n"+
  "  (                                                      \n"+
  "    (?: ftp ; http s? ): // [-\\w]+(\\.\\w[-\\w]+)+      \n"+
  "  |                                                      \n"+
  "    " + Hostname + "                                     \n"+
  "  )                                                      \n"+
  "  # Разрешить необязательный номер порта                 \n"+
  "  (?: :\\d+ )?                                           \n"+
  "                                                         \n"+
  "  # Остаток URL не является обязательным и начинается с /\n"+
  " (?: " + UrlPath + ")?                                   \n"+
  ")";

// Преобразовать строки, созданные выше, в объект регулярного выражения
Pattern UrlRegex = Pattern.compile(Url);
// Теперь выражение может применяться к тексту для поиска url...
```

Для поиска пути выбран новый подход, отличающийся от старого по целому ряду параметров; сравните его с приведенным в главе 2 (на с. 110), это довольно интересно. В частности, версия на языке Java (во врезке ниже) дает представление о том, как строилось это выражение.

На практике я бы не стал создавать подобных «монстров», а построил бы библиотеку регулярных выражений и пользовался ими по мере надобности. Примеры такого подхода встречаются на с. 111 (переменная `$HostnameRegex`) и во врезке на этой странице.

Нетривиальные примеры

В оставшейся части этой главы будут продемонстрированы некоторые важные приемы из области работы с регулярными выражениями. Примеры снабжаются более подробными комментариями, а основное внимание в них уделяется общей логике мышления и типичным ошибкам, часто встречающимся на пути к правильному решению.

Синхронизация

Рассмотрим довольно длинный пример, который на первый взгляд кажется несколько неестественным, но на самом деле убедительно доказывает важность синхронизации при поиске (а также предлагает несколько вариантов ее практического применения).

Предположим, данные представляют собой сплошную последовательность почтовых индексов США, каждый из которых состоит из 5 цифр. Вы хотите выбрать из потока все индексы, начинающиеся, допустим, с цифр 44. Ниже приведен пример данных, в котором нужные индексы выделены жирным шрифтом:

`03824531449411615213`**`44182`**`95035`**`44272`**`752010217443235`

Для начала рассмотрим возможность многократного применения выражения ⌜`\d\d\d\d\d`⌟ для выделения отдельных индексов во входном потоке данных. В Perl задача решается простой командой **`@zips = m/ \d\d\d\d\d/g;`**. Команда создает список, каждый элемент которого соответствует одному индексу (чтобы пример был менее громоздким, предполагается, что данные находятся в стандартной переменной Perl **`$_`**; ☞ 113). В других языках обычно применяется иной подход — метод «find» или его аналог вызывается в цикле. Нас в данном случае интересует регулярное выражение, а не детали реализации в конкретных языках, поэтому в дальнейших примерах будет использоваться Perl.

Вернемся к выражению ⌜`\d\d\d\d\d`⌟. Важное обстоятельство, смысл которого вскоре станет очевидным, — применение регулярного выражения никогда не завершается неудачей до завершения обработки всего списка; возвраты и повторные попытки начисто отсутствуют (предполагается, что все данные имеют правильный формат; в реальном мире такое иногда случается, но довольно редко).

Итак, замена ⌜`\d\d\d\d\d`⌟ на ⌜`44\d\d\d`⌟ для нахождения индексов, начинающихся с 44, является очевидной глупостью — после того как попытка совпадения завершится неудачей, механизм смещается на один символ, в результате чего теряется синхронизация ⌜`44...`⌟ с началом очередного индекса. При использовании ⌜`44\d\d\d`⌟ первое совпадение будет ошибочно обнаружено в последовательности '...`531`**`44947`**`16...'.

Конечно, регулярное выражение можно начать с ⌜\A⌟, но тогда индекс будет найден лишь в том случае, если он стоит на первом месте в строке. Нам нужна возможность ручной синхронизации механизма регулярных выражений, при которой регулярное выражение пропускало бы отвергнутые индексы. Главное — чтобы индекс пропускался полностью, а не по одному символу, как при автоматическом смещении текущей позиции поиска.

Синхронизация совпадений

Ниже приведены различные варианты пропуска ненужных индексов в регулярном выражении. Чтобы добиться желаемого эффекта, достаточно вставить их перед нужной конструкцией (⌜(44\d\d\d)⌟). Совпадения для неподходящих индексов заключены в несохраняющие круглые скобки ⌜(?:…)⌟, что позволяет игнорировать их и сохранить подходящий индекс в переменной $1:

⌜(?:[^4]\d\d\d\d|\d[^4]\d\d\d)*…⌟

Это решение можно назвать «методом грубой силы»: мы активно пропускаем все индексы, начинающиеся с чего-либо, кроме 44 (вероятно, вместо ⌜[^4]⌟ следовало бы использовать ⌜[1235-9]⌟, но, как говорилось выше, я предполагаю, что мы работаем с правильно отформатированными данными). Кстати говоря, мы не можем использовать конструкцию ⌜(?:[^4][^4]\d\d\d)*⌟, поскольку она не пропускает ненужные индексы типа 43210.

⌜(?:(?!44)\d\d\d\d\d)*…⌟

Аналогично решение на базе негативной опережающей проверки, которое также активно пропускает индексы, не начинающиеся с 44. Описание очень похоже на то, которое приведено выше, но на языке регулярных выражений оно выглядит совсем иначе. Сравните два описания и выражения. В нашем случае нужный индекс (начинающийся с 44) приводит конструкцию ⌜(?!44)⌟ к ложному результату, вследствие чего пропускание индексов прекращается.

⌜(?:\d\d\d\d\d)*?…⌟

В этом решении, основанном на применении минимального квантификатора, индексы пропускаются лишь при необходимости (т. е. когда дальнейшее подвыражение, которое описывает, что же нам нужно, не совпадает). В соответствии с принципом минимализма при отсутствии совпадения для последующего выражения ⌜(?:\d\d\d\d\d)⌟ проверка даже не проводится. Квантификатор * обеспечивает многократное применение выражения до тех пор, пока совпадение не будет найдено; в результате выражение пропускает те индексы, которые должны пропускаться по условиям задачи.

Объединяя последнее выражение с ⌜(44\d\d\d)⌟, мы получаем:

```
@zips = m/(?:\d\d\d\d\d)*?(44\d\d\d)/g;
```

Это выражение извлекает из строки интересующие нас индексы '44*xxx*', активно пропуская промежуточные ненужные группы (в контексте '`@array = m/…/g`' Perl заполняет массив текстом, совпадающим с подвыражениями в сохраняющих круглых скобках при каждом совпадении; ☞ 390). Полученное регулярное выражение может многократно применяться к строке, поскольку мы знаем, что при каждом совпадении «текущая позиция» остается в начале следующего индекса и поиск следующего совпадения начнется от начала очередного индекса, как и предполагает регулярное выражение.

Поддержание синхронизации при отсутствии совпадения

Действительно ли мы гарантируем, что регулярное выражение применяется только от начала очередного индекса? *Нет*! Мы пропускаем *промежуточные* ненужные индексы, но как только в тексте не останется ни одного нужного индекса, поиск совпадения завершится неудачей. Как всегда, при этом вступает в действие механизм смещения текущей позиции, и поиск продолжится с позиции *внутри* очередного индекса — а наше решение основано именно на том, что подобная ситуация никогда не возникает.

Вернемся к нашим примерным данным:

03824 53144 94116 15213 **44182** 95035 **44272**␣7␣5␣2␣0 10217 **44323**5

Совпадающие индексы выделены жирным шрифтом (третье совпадение является нежелательным). Активно пропущенные индексы подчеркнуты, а символы, пропущенные вследствие смещения текущей позиции поиска, помечены. После совпадения `44272` в строке не остается совпадающих индексов, поэтому следующая попытка поиска завершается неудачей. Но завершается ли при этом весь поиск? Конечно, нет. Механизм смещает текущую позицию и пытается применить регулярное выражение к следующему символу, нарушая тем самым синхронизацию с началом индексов. После четвертого смещения регулярное выражение пропускает `10217` и ошибочно находит «индекс» `44323`.

Любое из трех регулярных выражений прекрасно работает, пока оно применяется от начала индекса, но смещение текущей позиции нарушает все планы. Эта проблема может быть решена за счет предотвращения возврата или за счет создания регулярного выражения, в котором возврат не вызывает проблем.

Один из способов предотвращения возврата (по крайней мере, для первых двух выражений, приведенных выше) основан на том, чтобы сделать подвыражение `(44\d\d\d)` необязательным, присоединив к нему квантификатор `?`. Мы знаем, что предшествующее подвыражение `(?:(?!44)\d\d\d\d\d)*…` или `(?:[^4]\d\d\d\d|\d[^4]\d\d\d)*…` завершается только на границе действительного индекса или если индексов больше нет (поэтому данный способ не применим к третьему, не

максимальному варианту). В итоге ⌜(44\d\d\d)?⌟ совпадает с искомым индексом, если он обнаружен, но не приводит к возврату при его отсутствии.

У этого решения есть свои недостатки. В частности, регулярное выражение теперь может совпадать даже при отсутствии искомого индекса, что несколько усложняет код обработки. С другой стороны, оно быстро работает из-за малого количества возвратов и отсутствия смещений текущей позиции.

Синхронизация с использованием метасимвола \G

Существует и другое, универсальное решение — поставить перед любым из трех выражений метасимвол ⌜\G⌟ (☞ 177). Поскольку выражение строилось таким образом, чтобы совпадение заканчивалось на границе индекса, все последующие совпадения при отсутствии промежуточных смещений текущей позиции также будут начинаться с границы индекса. А если промежуточное смещение все-таки *произошло*, начальный метасимвол ⌜\G⌟ немедленно приводит к неудаче, потому что в большинстве диалектов он успешно совпадает лишь при отсутствии промежуточных смещений (исключение составляют Ruby и другие диалекты, в которых ⌜\G⌟ означает «начало текущего совпадения», а не «конец предыдущего совпадения»; ☞ 179).

Итак, для второго выражения можно воспользоваться конструкцией

```
@zips = m/\G(?:(?!44)\d\d\d\d\d)*(44\d\d\d)/g;
```

и обойтись без каких-либо специальных проверок после совпадения.

Взгляд со стороны

Готов признать, что рассмотренный пример весьма сложен, однако он демонстрирует целый ряд ценных приемов синхронизации регулярного выражения с данными. Если бы мне пришлось решать подобную задачу на практике, вероятно, я бы не стал ограничиваться одними регулярными выражениями. Проще выделить в строке очередной индекс при помощи выражения ⌜\d\d\d\d\d⌟ и занести его в массив, если он начинается с цифр '44'. На языке Perl это выглядит примерно так:

```
@zips = ( ); # Очистка массива
while (m/(\d\d\d\d\d)/g) {
    $zip = $1;
    if (substr($zip, 0, 2) eq "44") {
        push @zips, $zip;
    }
}
```

Особенно интересный пример использования ⌜\G⌟ приведен во врезке на с. 178, хотя на момент написания книги такая возможность существует только в Perl.

Разбор данных, разделенных запятыми

Любой, кто хоть раз занимался разбором данных, разделенных запятыми (CSV — Comma Separated Values), скажет, что эта задача не из простых. Главная проблема заключается в том, что каждая программа, генерирующая данные в формате CSV, руководствуется собственными представлениями о том, каким должен быть этот формат. В этом разделе мы начнем с задачи разбора данных CSV в формате Microsoft Excel, а затем перейдем к другим разновидностям[1]. Формат Microsoft является одной из самых простых разновидностей CSV. Перечисляемые значения либо задаются в непосредственном виде (т. е. как текст, находящийся между запятыми), либо заключаются в кавычки (в этом случае символ " представляется последовательностью "").

Пример:

```
Ten Thousand,10000, 2710 ,,"10,000","It's ""10 Grand"", baby",10K
```

Приведенная строка представляет семь полей данных:

```
Ten•Thousand
10000
•2710•
пустое поле
10,000
It's•"10•Grand",•baby
10K
```

Итак, выражение для выборки данных из строки должно различать две разновидности полей. С полями без кавычек все просто — они представляют собой последовательность произвольных символов, кроме запятых и кавычек, и поэтому задаются выражением ⌜`[^",]+`⌟.

Поля в кавычках могут содержать запятые, пробелы и любые другие символы, кроме кавычек. Также они могут содержать последовательность "", которая представляет один символ « в итоговом значении. Следовательно, поле в кавычках определяется как произвольное количество экземпляров ⌜`[^",]|""`⌟ между ⌜`"…"`⌟ — ⌜`"(?:[^"]|"")*"`⌟ (вообще говоря, по соображениям эффективности вместо ⌜`(?:…)`⌟ следовало бы воспользоваться атомарной группировкой ⌜`(?>…)`⌟, но мы вернемся к этой теме в следующей главе; ☞ 327).

Объединяя эти два выражения, мы получаем следующее представление для одного поля: ⌜`[^",]+|"(?:[^"]|"")*"`⌟. Чтобы выражение лучше читалось, его стоит записать в режиме свободного форматирования (с. 151):

[1] Окончательная версия кода для обработки данных CSV в формате Microsoft приводится в главе 6 (☞ 342), где она рассматривается с позиций эффективности.

```
# Произвольная последовательность символов, кроме запятых и кавычек...
[^",]+
# ...или...
  |
# ...поле в кавычках (внутри поля разрешаются удвоенные кавычки)
" # открывающая кавычка
  (?: [^"] | "" )*
" # закрывающая кавычка
```

Полученное выражение многократно применяется к строке, содержащей поля CSV. Но чтобы сделать что-нибудь полезное с полученными данными, необходимо знать, какая из двух альтернатив совпала. Для второй альтернативы из совпадения следует удалить внешние кавычки, а также заменить внутренние последовательности «""» на «"».

В голову приходят два возможных подхода. Можно просто просмотреть полученный текст и определить, является ли первый символ кавычкой. В этом случае из строки удаляется первый и последний символ (кавычки-ограничители), а все внутренние последовательности '""' заменяются на '"'. Предлагаемое решение достаточно просто, но существует и другой вариант, основанный на грамотном применении сохраняющих скобок. Если заключить каждое из подвыражений, совпадающих с фактическим содержимым поля, в сохраняющие круглые скобки, мы можем просмотреть их после совпадения и узнать, какая из групп содержит данные:

```
# Произвольная последовательность символов, кроме запятых и кавычек...
( [^",]+ )
# ...или...
  |
# ...поле в кавычках (внутри поля разрешаются удвоенные кавычки)
" # открывающая кавычка
  ( (?: [^"] | "" )* )
" # закрывающая кавычка
```

Если выясняется, что сохранена первая группа, ее значение используется без всяких изменений. Если сохранена вторая группа, перед использованием следует заменить '""' на '"'.

Ниже приведен код примера на Perl, а немного позднее (после устранения кое-каких ошибок) будут приведены реализации на Java и VB.NET (кроме того, в главе 10 будет продемонстрирован пример на языке PHP; ☞ 600). Предполагается, что входные данные хранятся в переменной `$line`, из которой удален завершающий символ новой строки (мы не хотим, чтобы он вошел в последнее поле!).

```
while ($line =~ m{
           # Произвольная последовательность символов,
           # кроме запятых и кавычек..
            ( [^",]+ )
```

```
            # ...или...
              |
            # ...поле в кавычках (внутри поля разрешается "")
            " # открывающая кавычка
              ( (?: [^"] | "" )* )
            " # закрывающая кавычка
          }gx)
{
    if (defined $1) {
        $field = $1;
    } else {
        $field = $2;
        $field =~ s/""/"/g;
    }
    print "[$field]"; # Вывод содержимого поля для отладочных целей
    # Теперь можно работать с переменной $field...
}
```

Для тестовых данных, приведенных выше, результат выглядит так:

```
[Ten•Thousand][10000][•2710•][10,000][It's•"10•Grand",•baby][10K]
```

В целом неплохо, но, к сожалению, мы потеряли пустое четвертое поле. Если программа сохраняет содержимое `$field` в массиве, то обращение к пятому элементу массива должно возвращать значение пятого поля («`10,000`»). Это не произойдет, если в массиве не будет создан пустой элемент для каждого пустого поля.

Первое, что приходит в голову, — заменить ⌜`[^",]+`⌟ на ⌜`[^",]*`⌟. Решение действительно очевидное, но работает ли оно?

Давайте проверим. Результат выглядит так:

```
[Ten•Thousand][][10000][][•2710•][][][][10,000][][][It's•"10•Grand",...
```

Сюрприз — откуда-то появилось множество лишних полей! Впрочем, при некотором размышлении все встает на свои места. Использование ⌜`(…)*`⌟ для поиска фактически означает, что совпадение не является обязательным. Для пустых полей это удобно, но после нахождения совпадения с первым полем следующее применение регулярного выражения начнется с позиции ‘`Ten•Thousand▲,10000…`’. Если никакая часть регулярного выражения не совпадает с запятой (как в нашем случае), но пустое совпадение считается допустимым, оно будет сочтено допустимым *в этой позиции*. Более того, количество пустых совпадений могло бы оказаться бесконечным, если механизм регулярных выражений не распознает такую ситуацию и не выполнит принудительное смещение текущей позиции, чтобы предотвратить два совпадения нулевой длины подряд (☞ 177). Именно поэтому каждые два «правильных» совпадения разделяются одним «неправильным» (а также появляется пустое совпадение в конце, хотя оно и не показано).

Контроль за смещением текущей позиции

Проблема обусловлена тем, что преодоление разделяющих запятых было доверено механизму смещения текущей позиции. Чтобы справиться с ней, нам придется взять контроль за смещением в свои руки. В голову приходят два возможных решения.

1. Самостоятельно находить совпадения для запятых. Если мы пойдем по этому пути, запятая включается в выражение и используется для «самостоятельного» продвижения текущей позиции.

2. Организовать дополнительную проверку и убедиться в том, что поиск совпадения начинается с позиции, с которой может начинаться поле. Поля начинаются либо с начала строки, либо после запятых.

Возможно, еще правильнее было бы объединить эти два решения. Если начать с первого решения (самостоятельный поиск запятых), мы просто требуем обязательного наличия запятой перед каждым полем, кроме первого (также можно требовать обязательного наличия запятой после каждого поля, кроме последнего). Для этого в регулярное выражение включается префикс `(?:^|,)` или суффикс `(?:$|,)` и круглые скобки, определяющие его область действия.

Возможная реализация варианта с префиксом выглядит так:

```
(?:^|,)
(?:
    # Произвольная последовательность символов,
    # кроме запятых и кавычек...
    ( [^",]* )
  # ...или...
  |
    # ...поле в кавычках (внутри поля разрешаются удвоенные кавычки)
    " # открывающая кавычка
      ( (?: [^"]| "")* )
    " # закрывающая кавычка
)
```

Вроде бы все должно быть нормально, но запуск тестовой программы приносит обескураживающий результат:

```
[Ten•Thousand][10000][•2710•][][][000][][•baby][10K]
```

Ожидалось несколько иное:

```
[Ten•Thousand][10000][•2710•][][10,000][It's•"10•Grand",•baby][10K]
```

Почему наш алгоритм не работает? Похоже, неправильно обрабатываются поля в кавычках, поэтому проблема связана со второй альтернативой... Верно? Нет, проблема возникает на более высоком уровне. Вспомните мораль со с. 231: «*...если*

при упорядоченной конструкции выбора один и тот же текст может совпасть с несколькими альтернативами, будьте особенно внимательны при выборе порядка альтернатив». Поскольку для успешного совпадения первой альтернативы, ⌜`[^",]*`⌟, ничто не является обязательным, вторая альтернатива вообще не будет применяться, если этого не потребуют последующие элементы регулярного выражения. Наше выражение завершается конструкцией выбора, так что в его текущем виде вторая альтернатива вообще остается недоступной.

Давайте поменяем альтернативы местами и попробуем снова:

```
(?:^|,)
(?: # Поле в кавычках (внутри поля разрешаются удвоенные кавычки)
    " # открывающая кавычка
      ( (?: [^"] | "" )* ) )
    " # закрывающая кавычка
|
    # ...или произвольная последовательность символов,
    # кроме запятых и кавычек.
    ( [^",]* )
)
```

На этот раз все работает!.. Во всяком случае, для наших тестовых данных. А как насчет других данных? Не забывайте, что этот раздел называется «Контроль за смещением текущей позиции». Хотя при построении регулярных выражений ничто не заменит размышлений, подкрепленных грамотным тестированием, мы можем воспользоваться метасимволом ⌜`\G`⌟ и гарантировать, что каждый поиск совпадения начинается в точности с той позиции, на которой завершилось предыдущее совпадение. Мы полагаем, что соблюдение этого условия уже обеспечивается тем, как конструировалось и применялось наше регулярное выражение. Если начать выражение с ⌜`\G`⌟, мы запретим любые совпадения после вынужденного смещения текущей позиции. Внешне такое изменение не должно ни на что влиять, но если применить его к первому варианту регулярного выражения, который давал результат

```
[Ten•Thousand][10000][•2710•][][][000][][•baby][10K]
```

мы получим

```
[Ten•Thousand][10000][•2710•][][]
```

Ошибка становится еще более очевидной, хотя в первый раз мы ее упустили.

Другой подход

В начале этого раздела упоминаются два способа сохранения выравнивания по границам полей. В первом способе мы убеждаемся в том, что совпадение начина-

ется только с тех позиций, с которых может начинаться поле. На первый взгляд оно напоминает решение с префиксом `(?<=^|,)`, если не считать ретроспективной проверки `(?<=^|,)`.

ОБРАБОТКА ДАННЫХ В ФОРМАТЕ CSV НА JAVA

Ниже приведен пример с разбором данных CSV, реализованный с использованием пакета Sun java.util.regex. Программа ориентирована на наглядность и простоту — более эффективная версия приведена в главе 8 (☞ 499).

```
import java.util.regex.*;
        ⋮
String regex = // Поля в кавычках будут помещаться в group(1)
               // Поля не в кавычках - в group(2)
"\\G(?:^|,)                                                    \n"+
"(?:                                                           \n"+
"    # Для поля в кавычках...                                  \n"+
"    \"   # открывающая кавычка                                \n"+
"     ( (?: [^\"]++ | \"\" )*+ )                               \n"+
"    \"   # закрывающая кавычка                                \n"+
"  | # ...или...                                               \n"+
"       # любой текст, не содержащий кавычки и запятые...     \n"+
"     ( [^\",]* )                                              \n";
")

// Создать объект для вышеприведенного регулярного выражения.
Matcher mMain = Pattern.compile(regex, Pattern.COMMENTS).matcher("");

// Создать объект для регулярного выражения И""°
Matcher mQuote = Pattern.compile("\"\"").matcher(""); .
        ⋮

// Подготовка закончена, теперь можно выполнить поиск
mMain.reset(line); // Строка с текстом в формате CSV
                   // находится в переменной line
while (mMain.find())
{
    String field;
    if (mMain.start(2) >= 0)
        field = mMain.group(2);   // Поле не в кавычках
                                  // используется как есть
    else
        // Поле в кавычках, нужно удалить лишние кавычки
        field = mQuote.reset(mMain.group(1)).replaceAll("\"");
    // Теперь можно выполнить необходимые действия с полем...
    System.out.println("Field [" + field + "]");
}
```

К сожалению, как объясняется в главе 3 (☞ 181), даже при поддержке ретроспективной проверки иногда не поддерживается ретроспективная проверка переменной длины, поэтому такое решение работает не всегда. Если возникнут трудности с переменной длиной, ⌜`(?<=^|,)`⌟ можно заменить на ⌜`(?:^|(?<=,))`⌟, однако это чрезмерно усложняет решение, особенно если учесть, что у нас уже имеется работоспособный первый вариант. Кроме того, преодоление запятых снова возлагается на механизм смещения текущей позиции, и если где-то будет допущена ошибка, совпадение может начаться в позиции типа '…`"10,␣000"`…'. Если учесть все сказанное, первый подход начинает казаться более надежным.

Впрочем, к проблеме можно подойти несколько иначе и потребовать, чтобы совпадение завершалось перед запятой (или концом строки). Включение конструкции ⌜`(?=$|,)`⌟ в конец регулярного выражения помогает убедиться в отсутствии посторонних совпадений. Стал бы я включать подобную проверку на практике? Честно говоря, я вполне уверен в надежности предложенного выше решения, так что в этой конкретной ситуации я бы обошелся без дополнительных проверок, но в целом это полезный прием, о котором стоит вспомнить в случае необходимости.

Повышение эффективности

Тема эффективности будет подробно рассмотрена в следующей главе, но я все же упомяну об одном изменении, повышающем эффективность поиска в системах с поддержкой атомарной группировки (☞ 187). В таких системах альтернатива для полей в кавычках ⌜`(?:[^"] |"")*`⌟ заменяется на ⌜`(?>[^"]+|"")*`⌟. Пример на языке VB.NET приводится во врезке на с. 282.

Если диалект поддерживает захватывающие квантификаторы (☞ 190), как пакет регулярных выражений для Java от Sun, вы можете воспользоваться ими. Пример приведен выше на врезке с кодом Java.

Причины этих изменений будут рассмотрены в следующей главе. В итоге мы придем к самой эффективной версии, приведенной на с. 342.

Другие форматы CSV

Описанный выше формат CSV широко распространен, потому что он используется в продуктах Microsoft, но это вовсе не означает, что он поддерживается всеми программами. Ниже описаны некоторые из известных мне разновидностей.

- В качестве разделителя вместо запятой может использоваться другой символ — например, символ табуляции или '`;`' (хотя возникает резонный вопрос — почему такие данные называются «разделенными *запятыми*»?).
- После разделителей допускаются пробелы, которые не включаются в значение.

ОБРАБОТКА ДАННЫХ В ФОРМАТЕ CSV НА VB.NET

```
Imports System.Text.RegularExpressions
     ⋮
Dim FieldRegex as Regex = New Regex( _
       "(?:^|,)                                                    " & _
       "(?:                                                        " & _
       "    (?# Для поля в кавычках...)                            " & _
       "    ""  (?# открывающая кавычка )                          " & _
       "    (  (?> [^""]+ | """" )* )                              " & _
       "    ""  (?# закрывающая кавычка )                          " & _
       "  (?# ... или ...)                                         " & _
       "  |                                                        " & _
       " (?# ...любой текст, не содержащий кавычки и запятые...)   " & _
       "    ([^"",]*)                                              " & _
       " )", RegexOptions.IgnorePatternWhitespace)

Dim QuotesRegex as Regex = New Regex(" "" "" ") 'Строка, содержащая
                                                'повторяющиеся кавычки
     ⋮
Dim FieldMatch as Match = FieldRegex.Match(Line)
While FieldMatch.Success
    Dim Field as String
    If FieldMatch.Groups(1).Success
        Field = QuotesRegex.Replace(FieldMatch.Groups(1).Value, """")
    Else
        Field = FieldMatch.Groups(2).Value
    End If

    Console.WriteLine("[" & Field & "]")
    'Теперь можно выполнить необходимые действия с полем...

    FieldMatch = FieldMatch.NextMatch
End While
```

- Кавычки могут экранироваться символом \ (т. е. для включения кавычки в строку вместо '""' используется последовательность '\"'). Обычно это означает, что символ \ может находиться перед любым символом (и игнорируется).

Все перечисленные изменения достаточно просто реализуются. В первом случае запятые в регулярном выражении заменяются новым символом-разделителем; во втором случае после первого разделителя добавляется конструкция ⌜**\s***⌟, т. е. начальное подвыражение принимает вид ⌜**(?:^|,\s***⌟.

В третьем случае можно воспользоваться предыдущими результатами (☞ 257) и заменить ⌜**[^"]+|""**⌟ на ⌜**[^\\"]+|\\.**⌟. Разумеется, следующую команду **s/""/"/g** придется заменить более общей командой **s/\\(.)/$1/g** или ее аналогом в вашем языке.

6 Построение эффективных регулярных выражений

Управляемый регулярным выражением механизм НКА встречается в Perl, пакетах Java, языках .NET, Python и PHP (список далеко не полон, за дополнительной информацией обращайтесь к таблице на с. 194). Природа этого механизма такова, что незначительные изменения в регулярном выражении могут кардинально изменить результат и время поиска. Проблемы, которых в механизме ДКА попросту нет, в НКА выходят на первый план. Возможности точной регулировки механизма НКА позволяют *творить* выражения, хотя для непосвященных это порой вызывает немало проблем. Настоящая глава поможет вам овладеть этим искусством.

Наша цель — правильность и эффективность. Это означает, что выражение должно находить все нужное, не находить ничего лишнего, и притом быстро. Правильность рассматривалась в главах 4 и 5, а в этой главе будут рассмотрены вопросы эффективности механизма НКА и то, как обратить их в свою пользу (там, где это уместно, будет приведена и информация о ДКА, но эта глава в первую очередь посвящена механизмам НКА). Главное, что для этого нужно, — доскональное понимание возврата и умение избегать его там, где это возможно. Мы рассмотрим некоторые практические приемы написания эффективных выражений, которые не только ускоряют их работу, но и при хорошем понимании механики их обработки, помогут вам создавать более сложные выражения.

Глава начинается с подробного примера, который демонстрирует, насколько важными могут быть эти проблемы. Затем, чтобы подготовиться к восприятию более сложных приемов, описанных далее, мы снова рассмотрим базовую процедуру возврата, описанную в предыдущей главе, с упором на эффективность и глобальные последствия возврата. Далее рассматриваются некоторые стандартные приемы внутренней оптимизации, способные довольно заметно влиять на эффективность, и особенности построения выражений для тех реализаций, в которых эти приемы используются. Наконец, все сказанное объединяется в нескольких «убойных» приемах, которые помогут вам конструировать чрезвычайно эффективные регулярные выражения НКА.

Проверки и возвраты

Приведенные примеры демонстрируют общие ситуации, возникающие при использовании регулярных выражений. Анализируя эффективность конкретного примера, я часто привожу число отдельных проверок, используемых механизмом регулярных выражений при поиске совпадения. Например, при поиске ⌜`marty`⌟ в строке `smarty` происходит шесть отдельных проверок: сначала ⌜`m`⌟ сравнивается с `s` (неудача), затем ⌜`m`⌟ сравнивается с `m`, ⌜`a`⌟ — с `a` и т. д. (все эти проверки проходят успешно). Я также часто сообщаю количество возвратов (в данном примере ноль, хотя неявный возврат для повторного применения регулярного выражения со второго символа можно посчитать за один).

Я привожу эти конкретные величины не потому, что здесь так важна точность, а скорее для того, чтобы избежать использования туманных слов «много», «мало», «лучше», «терпимо» и т. д. Не подумайте, что использование регулярных выражений в НКА сводится к подсчету проверок или возвратов; я просто хочу, чтобы вы представляли себе порядок этих величин.

Еще одно важное замечание: вы должны понимать, что эти «точные» числа, вероятно, в разных программах будут разными. Я привожу лишь базовые показатели для тех примеров, которые, как я надеюсь, вам еще пригодятся. Однако приходится учитывать и другой важный фактор — оптимизацию, выполняемую конкретной программой. Достаточно «умная» реализация может полностью устранить поиск конкретного регулярного выражения, если она заранее решит, что оно в любом случае не совпадет с имеющейся строкой (например, из-за отсутствия в строке некоторого символа, который обязательно должен присутствовать в возможном совпадении). Мы рассмотрим некоторые приемы оптимизации в этой главе, но общие принципы важнее частных случаев.

Традиционный НКА и POSIX НКА

Анализируя эффективность выражения, следует учитывать тип механизма используемой программы (традиционный НКА или POSIX НКА). Как будет показано в следующем разделе, некоторые проблемы относятся лишь к одному из этих типов. Иногда изменение, не влияющее на один механизм, сильно отражается на работе другого. Еще раз подчеркну, что понимание базовых принципов поможет вам правильно оценивать все ситуации по мере их возникновения.

Убедительный пример

Начнем с примера, который наглядно продемонстрирует, какими важными могут быть проблемы возврата и эффективности. На с. 257 мы построили выражение

⌜"(\\.|[^\\"])*"⌟ для поиска строк, заключенных в кавычки. В строке могут присутствовать внутренние кавычки, экранированные символом \. Это регулярное выражение работает, но в механизме НКА конструкция выбора, применяемая к каждому символу, работает крайне неэффективно. Для каждого «обычного» символа в строке (не кавычки и не экранированного символа) механизм должен проверить ⌜\\.⌟ обнаружить неудачу и вернуться, чтобы в результате найти совпадение для ⌜[^\\"]⌟. Если выражение используется в ситуации, когда важна эффективность, конечно, хотелось бы немного ускорить обработку этого выражения.

Простое изменение — начинаем с более вероятного случая

Поскольку в средней строке, заключенной в кавычки, обычных символов больше, чем экранированных, напрашивается простое изменение — сменить порядок альтернатив и поставить ⌜[^\\"]⌟ на первое место, а ⌜\\.⌟ — на второе. Если ⌜[^\\"]⌟ стоит на первом месте, то возврат происходит лишь при обнаружении экранированного символа в строке (и, конечно, при несовпадении *, поскольку конструкция выбора не совпадает лишь в том случае, если не совпадают все альтернативы). Рисунок 6.1 наглядно демонстрирует отличия между этими двумя выражениями. Уменьшение количества стрелок в нижней половине означает, что для первой альтернативы совпадения находятся чаще. Это приводит к уменьшению количества возвратов.

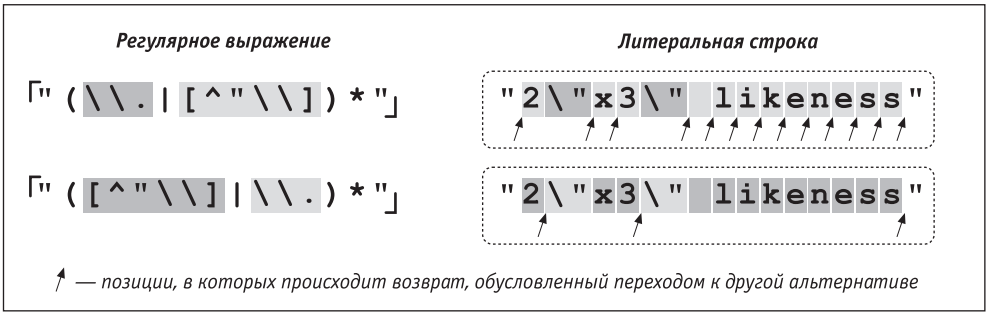


Рис. 6.1. Изменение порядка альтернатив (для традиционного НКА)

Оценивая последствия такого изменения для эффективности, необходимо задать себе несколько ключевых вопросов:

- Какой механизм выиграет от этих изменений — традиционный НКА, POSIX НКА или оба?

- Когда изменение приносит наибольшую пользу — когда текст совпадает, когда текст не совпадает или в любом случае?

❖ Подумайте над этими вопросами, затем проверьте свои ответы на с. 288. Прежде чем переходить к следующему разделу, убедитесь в том, что вы хорошо понимаете смысл ответов и их обоснование.

Эффективность и правильность

Самый важный вопрос, который необходимо себе задать при любых попытках повышения эффективности: а не повлияет ли изменение на правильность совпадения? Изменение порядка альтернатив допустимо лишь в том случае, если порядок не влияет на успешность совпадения. Рассмотрим выражение `"(\\.|[^"])*"`, более раннюю (☞ 255) и ошибочную версию выражения из предыдущего раздела. В этом выражении инвертированный символьный класс не содержит символ \\ и поэтому может совпасть в случаях, в которых он совпадать не должен. Если регулярное выражение применяется только к «правильным» данным, с которыми оно *должно* совпадать, проблема останется незамеченной. Полагая, что выражение работает нормально, а перестановка альтернатив повысит эффективность поиска, вы попадете в беду. Перестановка, после которой `[^"]` оказывается на первом месте, приводит к неверному совпадению во всех случаях, когда целевой текст содержит экранированную кавычку:

```
"You need a 2\"3\" photo."
```

Итак, прежде чем беспокоиться об эффективности, обязательно убедитесь в том, что выражение работает правильно.

Следующий шаг — локализация максимального поиска

Из рис. 6.1 ясно видно, что в обоих случаях квантификатор * должен последовательно перебрать все нормальные символы, при этом он снова и снова входит в конструкцию выбора (и круглые скобки) и выходит из нее. Все эти действия сопряжены с лишней работой, от которой хотелось бы по возможности избавиться.

Однажды, работая над аналогичным выражением, я вдруг понял, что выражение можно оптимизировать, если учесть, что когда выражение `[^"\\]` относится к «нормальным» (не кавычки и не экранированные символы) случаям, то при его замене на `[^\\"]+` одна итерация `(…)*` прочитает все последовательно стоящие обычные символы. При отсутствии экранированных символов будет прочитана вся строка. Это позволяет найти совпадение практически без возвратов и сокращает

многократное повторение * до абсолютного минимума. Я был очень доволен своим открытием.

Этот пример будет подробно рассмотрен ниже, но даже беглый взгляд на статистику наглядно демонстрирует преимущества нового выражения. На рис. 6.2 показано, как происходит поиск в традиционном НКА. Модификация исходного выражения `"(\\.|[^\\"])*"` (верхняя пара на рис. 6.2) уменьшает количество возвратов, связанных с конструкцией выбора, а также число итераций квантификатора *. Нижняя пара на рис. 6.2 показывает, что объединение этой модификации с изменением порядка альтернатив приводит к еще большему повышению эффективности.

Рис. 6.2. Последствия добавления плюса (традиционный механизм НКА)

Добавление квантификатора + уменьшает количество возвратов, обусловленных конструкцией выбора, что, в свою очередь, приводит к уменьшению количества итераций *. Квантификатор * относится к подвыражению в круглых скобках, а каждая итерация сопряжена с немалыми затратами при входе в круглые скобки и выходе из них, поскольку механизм должен хранить информацию о том, какой текст совпадает с подвыражением в скобках (эта тема подробно рассматривается ниже).

Таблица 6.1 аналогична таблице, приведенной во врезке, но в ней рассматривается меньшее количество примеров и имеются дополнительные столбцы для количества итераций *. При модификации выражения количество проверок и возвратов увеличивается незначительно, но количество итераций уменьшается очень заметно. Налицо заметное повышение эффективности.

ПОСЛЕДСТВИЯ ПРОСТЫХ ИЗМЕНЕНИЙ

❖ *Ответ на вопрос со с. 286.*

Для какого типа механизма? Изменение практически никак не повлияет на работу механизма POSIX НКА. Поскольку этот механизм в любом случае должен опробовать все комбинации элементов регулярного выражения, порядок проверки альтернатив не важен. Однако в традиционном НКА порядок альтернатив, ускоряющий поиск совпадения, является преимуществом, поскольку механизм может остановиться сразу же после того, как будет найдено первое совпадение.

Для какого результата? Изменение приводит к ускорению поиска лишь при наличии совпадения. НКА может сделать вывод о неудаче только после того, как будут проверены все возможные комбинации (повторяю, POSIX НКА проверяет их в любом случае). Следовательно, если попытка окажется неудачной, значит, были опробованы все комбинации, поэтому порядок не важен.

В следующей таблице перечислено количество проверок и возвратов для некоторых случаев (чем меньше число, тем лучше).

	Традиционный НКА				**POSIX НКА**	
	`「"(\\.|[^"])*"」`		`「"([^"]|\\.)*"」`		**Оба выражения**	
Текст примера	**Пров.**	**Возвр.**	**Пров.**	**Возвр.**	**Пров.**	**Возвр.**
`"2\"x3\" likeness"`	32	14	22	4	48	30
`"makudonarudo"`	28	14	16	2	40	26
`"very...`*99 символов*`...long"`	218	109	111	2	325	216
`"No \"match\" here`	124	86	124	86	124	86

Как видите, в POSIX НКА оба выражения дают одинаковые результаты, а в традиционном НКА для нового выражения быстродействие возрастает (уменьшается количество возвратов). В ситуации без совпадения (последний пример в таблице) оба механизма проверяют все возможные комбинации, поэтому и результаты оказываются одинаковыми.

Таблица 6.1. Эффективность совпадений для традиционного НКА

	`「"([^\\"]|\\.)*"」`			`「"([^\\"]+|\\.)*"」`		
Текст примера	**Пров.**	**Возвр.**	**Итер**	**Пров.**	**Возвр.**	**Итер.**
`"makudonarudo"`	16	2	13	17	3	2
`"2\"x3\" likeness"`	22	4	15	25	7	6
`"very...`*99 символов*`...long"`	111	2	108	112	3	2

Возвращение к реальности

Да, я был очень доволен своим открытием. Но как бы замечательно ни выглядело мое «усовершенствование», я своими руками создал потенциальную катастрофу. Обратите внимание: расписывая достоинства этого решения, я не привел статистику для механизма POSIX НКА. Вероятно, вы бы сильно удивились, узнав, что пример `"very•…•long"` требует выполнения свыше *трехсот тысяч миллионов миллиардов триллионов возвратов* (324 518 553 658 426 726 783 156 020 576 256, или около 325 ниллионов). Мягко говоря, это ОЧЕНЬ много работы. На моем компьютере это заняло бы свыше 50 *квинтиллионов* лет... плюс-минус несколько сотен триллионов тысячелетий[1].

Ничего себе сюрприз! Почему же это происходит? В двух словах: потому что к некоторой части нашего выражения применяется как непосредственный квантификатор +, так и внешний квантификатор *, и механизм никак не может определить, какой из этих квантификаторов относится к конкретному символу текста. Подобная неопределенность оборачивается катастрофой. Позвольте пояснить чуть подробнее.

«Экспоненциальный» поиск

Прежде чем в выражении появился +, ⌜`[^\\"]`⌟ относилось только к *, и количество вариантов совпадения в тексте выражения ⌜`([^\\"])*`⌟ было ограниченным. Выражение могло совпасть с одним символом, двумя символами и т. д. до тех пор, пока каждый символ целевого текста не будет проверен максимум один раз. Конструкция с квантификатором могла и не совпадать со всей целевой строкой, но в худшем случае количество совпавших символов было прямо пропорционально длине целевого текста. Потенциальный объем работы возрастал теми же темпами, что и длина целевого текста.

У «эффективного» выражения ⌜`([^\\"]+)*`⌟ количество вариантов, которыми + и * могут поделить между собой строку, растет с экспоненциальной скоростью. Возьмем строку `makudonarudo`. Следует ли рассматривать ее как 12 итераций *, когда каждое внутреннее выражение ⌜`[^\\"]+`⌟ совпадает лишь с одним символом (‘`m a k u d o n a r u d o`’)? А может, одну итерацию *, при которой внутреннее выражение ⌜`[^\\"]+`⌟ совпадает со всей строкой (‘`makudonarudo`’)? А может, три итерации *, при которых внутреннее выражение ⌜`[^\\"]+`⌟ совпадает соответственно с 5, 3 и 4 символами (‘`makud ona rudo`’)? Или 2, 2, 5 и 3 символами (‘`ma ku donar udo`’)? Или...

В общем, вы поняли: возможностей очень много (4096 в 12-символьной строке). Для каждого нового символа в строке количество возможных комбинаций удваи-

[1] Приведенное время вычислено на основании других эталонных тестов; я не проверял его на практике.

вается, и механизм POSIX НКА должен перепробовать все варианты, прежде чем вернуть ответ, поэтому подобная ситуация называется «экспоненциальным поиском» (также иногда встречается термин «*суперлинейный* поиск»).

Впрочем, как ни называй, суть от этого не изменится — в процессе поиска происходят возвраты, и очень много![1] 4096 комбинаций для 12 символов обрабатываются быстро, но обработка миллиона с лишним комбинаций для 20 символов занимает уже несколько секунд. Для 30 символов миллиард с лишним комбинаций обрабатывается несколько часов, а для 40 символов обработка займет уже больше года. Конечно, это неприемлемо.

«Ага! — скажете вы. — Но механизм POSIX НКА встречается не так уж часто. Я знаю, что в моей программе используется традиционный НКА, поэтому все нормально». Главное отличие между POSIX и традиционным НКА заключается в том, что последний останавливается при первом найденном совпадении. Если полное совпадение отсутствует, то даже традиционный НКА должен перебрать все возможные комбинации, чтобы узнать об этом. Даже в коротком примере `"No•\"match\"•here` из приведенной выше врезки сообщение о неудаче поступает лишь после проверки 8192 комбинаций.

В процессе выполнения экспоненциального поиска может показаться, что программа «зависает». Впервые столкнувшись с проблемой, я решил, что в программе обнаружилась какая-то ошибка. Оказывается, она всего лишь занималась бесконечным перебором комбинаций. Теперь, когда я это понял, подобные выражения вошли в набор тестов для проверки типа механизма:

- Если выражение обрабатывается быстро даже в случае несовпадения, это, вероятно, ДКА.
- Если выражение обрабатывается быстро только при наличии совпадения, это традиционный НКА.
- Если выражение всегда обрабатывается медленно, это POSIX НКА.

Присутствие слова «вероятно» в последнем пункте объясняется тем, что хорошо оптимизированный механизм НКА может обнаруживать экспоненциальный поиск и избегать его (подробности приводятся на ☞ 317). Вскоре будут рассмотрены некоторые усовершенствованные варианты этого выражения, ускоряющие как поиск совпадений, так и принятие решений о неудаче.

Как видно из приведенного списка, по относительной скорости обработки регулярного выражения можно определить тип механизма регулярных выражений

[1] Для любознательных: количество возвратов, выполняемых для строки длины n, равно 2^{n+1}. Количество проверок равно $2^{n+1} + 2^n$.

(по крайней мере, при отсутствии некоторых мощных оптимизаций). Теперь становится ясно, почему это выражение приводилось в разделе «Определение типа механизма» в главе 4 (☞ 196).

Конечно, не каждое мелкое изменение приводит к таким катастрофическим последствиям. Но если вы не разбираетесь в механике обработки выражений, то просто не будете знать о проблеме до тех пор, пока не столкнетесь с ней. В этой главе проблема эффективности и ее последствия поясняются многочисленными примерами. Как обычно, понимание базовых принципов абсолютно необходимо для восприятия более сложных концепций, поэтому прежде чем искать решение проблемы бесконечного перебора, я хочу более подробно описать процесс возврата.

Возврат с глобальной точки зрения

На локальном уровне возврат — это обратный переход к непроверенному варианту. На глобальном уровне дело обстоит сложнее. В этом разделе мы подробно проанализируем ход возврата при найденном и ненайденном совпадениях, а также попытаемся найти общие закономерности в возникающих ситуациях.

Начнем с более подробного рассмотрения некоторых приемов из предыдущей главы. Процесс поиска совпадения ⌜".*"⌟ в строке

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

наглядно продемонстрирован на рис. 6.3.

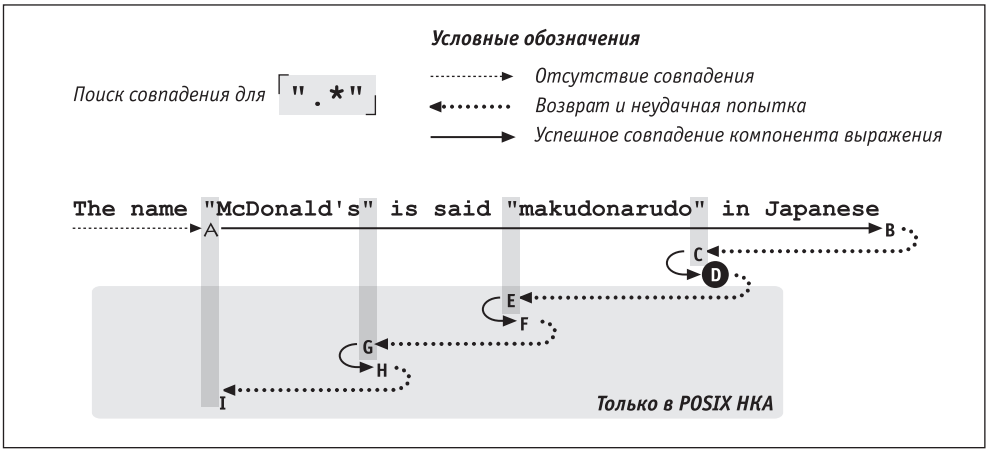


Рис. 6.3. Успешный поиск ⌜".*"⌟

Поиск регулярного выражения осуществляется в каждой позиции строки, начиная с первой. Поскольку несовпадение обнаруживается при проверке первого элемента (кавычка), ничего интересного не происходит до тех пор, когда поиск начнется с позиции **A**. В этот момент проверяется оставшаяся часть выражения, однако подсистема смещения текущей позиции (☞ 198) знает, что если попытка приведет к тупику, все регулярное выражение будет проверено заново со следующей позиции.

⌈.*⌋ распространяется до самого конца строки, когда для точки не находится совпадения и квантификатор * завершает свою работу. Ни один из 46 символов, совпавших с ⌈.*⌋, не является обязательным, поэтому в процессе поиска механизм накапливает 46 сохраненных состояний, к которым он может вернуться в случае необходимости. После остановки ⌈.*⌋ механизм возвращается к последнему сохраненному состоянию — «поиск ⌈".*▲"⌋ в позиции ...**anese▲**».

Это означает, что механизм пытается найти совпадение для завершающей кавычки в конце текста. Разумеется, кавычка с «ничем» не совпадает (как и точка), поэтому проверка завершается неудачей. Механизм отступает и пытается найти совпадение для завершающей кавычки в позиции `Japanese`. Попытка снова оказывается неудачной.

Сохраненные состояния, накопленные при поиске совпадения от **A** до **B**, последовательно проверяются в обратном порядке при перемещении от **B** к **C**. После дюжины возвратов проверяется состояние «поиск ⌈".*▲"⌋ в позиции ...**arudo▲"•in•Japa**..." (состояние **C**). На этот раз совпадение *может быть* найдено, в результате чего мы переходим в состояние **D** и получаем общее совпадение:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Так работает традиционный механизм НКА. Остальные непроверенные состояния попросту игнорируются, и механизм возвращает найденное совпадение.

POSIX НКА — работа продолжается

В POSIX НКА найденное выше совпадение запоминается как «самое длинное совпадение, найденное до настоящего момента», однако механизм должен исследовать остальные состояния и убедиться в том, что он не сможет найти более длинное совпадение. Мы знаем, что в данном примере это невозможно, но механизм регулярных выражений должен убедиться в этом.

Механизм перебирает и немедленно отвергает все сохраненные состояния за исключением двух оставшихся ситуаций — когда в строке находится кавычка, которая может совпасть с завершающей кавычкой. Таким образом, последовательности **D–E–F** и **F–G–H** аналогичны **B–C–D**, за исключением того, что совпадения **F** и **H** отвергаются как уступающие по длине ранее найденному совпадению **D**.

При возврате к состоянию **I** остается всего одна возможность — перейти к следующей позиции в строке и попробовать заново. Но поскольку в результате попытки, начинающейся с позиции **A**, *было* найдено совпадение (даже целых три совпадения), механизм POSIX НКА завершает свою работу и выдает совпадение **D**.

Работа механизма при отсутствии совпадения

Остается выяснить, что происходит при отсутствии совпадений. Рассмотрим выражение `"".*"!`, для которого в нашем примере не существует совпадения. Однако в процессе поиска оно довольно близко подходит к совпадению, что, как вы сейчас увидите, приводит к существенному возрастанию объема работы.

Происходящее показано на рис. 6.4. Последовательность **A-I** похожа на рис. 6.3. Первое отличие состоит в том, что на этот раз она не совпадает в точке **D** (из-за отсутствия совпадения для завершающего восклицательного знака). Второе

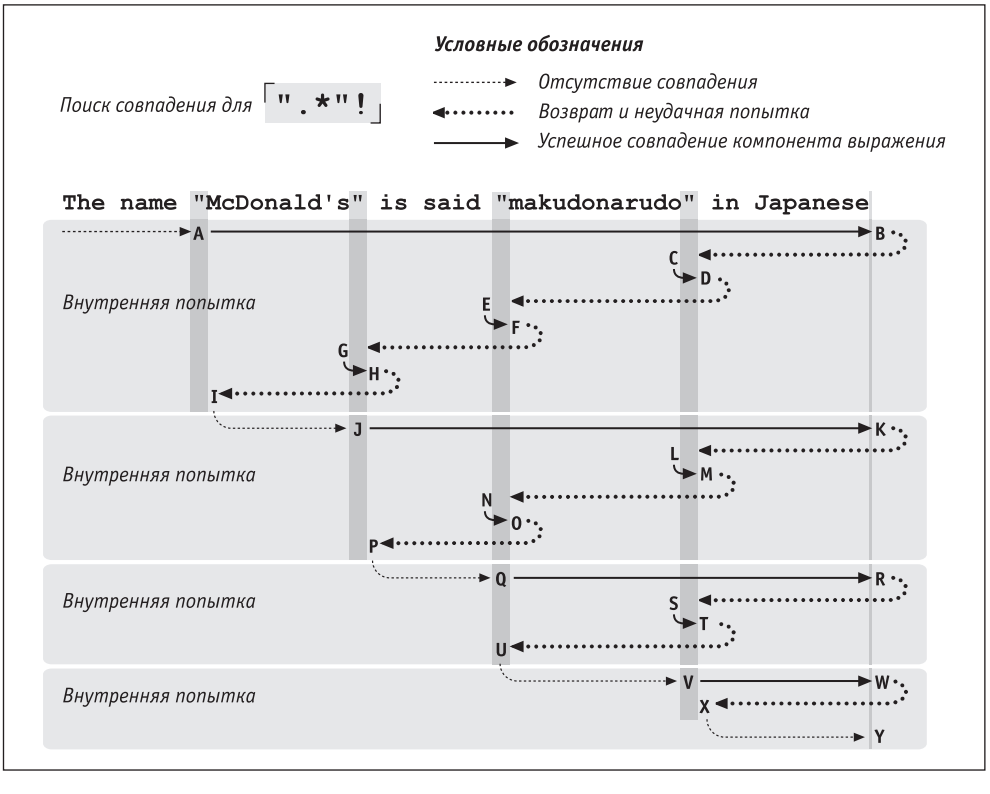


Рис. 6.4 Неудачный поиск совпадения для `"".*"!`

отличие состоит в том, что вся последовательность операций на рис. 6.4 относится как к традиционному НКА, так и к POSIX НКА: при отсутствии совпадения традиционный НКА должен перепробовать те же варианты, что и POSIX НКА, т. е. все возможные варианты.

Поскольку при общей попытке, начинающейся с точки **A** и заканчивающейся в точке **I**, совпадение не найдено, механизм переходит к следующей позиции в строке. Попытки, начинающиеся в позициях **J**, **Q** и **V**, выглядят перспективными, но все они завершаются неудачей в точке **A**. Наконец, в точке **Y** завершается перебор всех начальных позиций в строке, поэтому вся попытка завершается неудачей. Как видно из рис. 6.4, для получения этого результата пришлось выполнить довольно большую работу.

Уточнение

Для сравнения давайте заменим точку выражением ⌜`[^"]`⌟. Как было показано в предыдущей главе, это улучшает общую картину, поскольку выражение становится более точным и работает эффективнее. В выражении ⌜`"[^"]*"!`⌟ конструкция ⌜`[^"]*`⌟ не проходит мимо завершающей кавычки, что устраняет многие попытки и последующие возвраты.

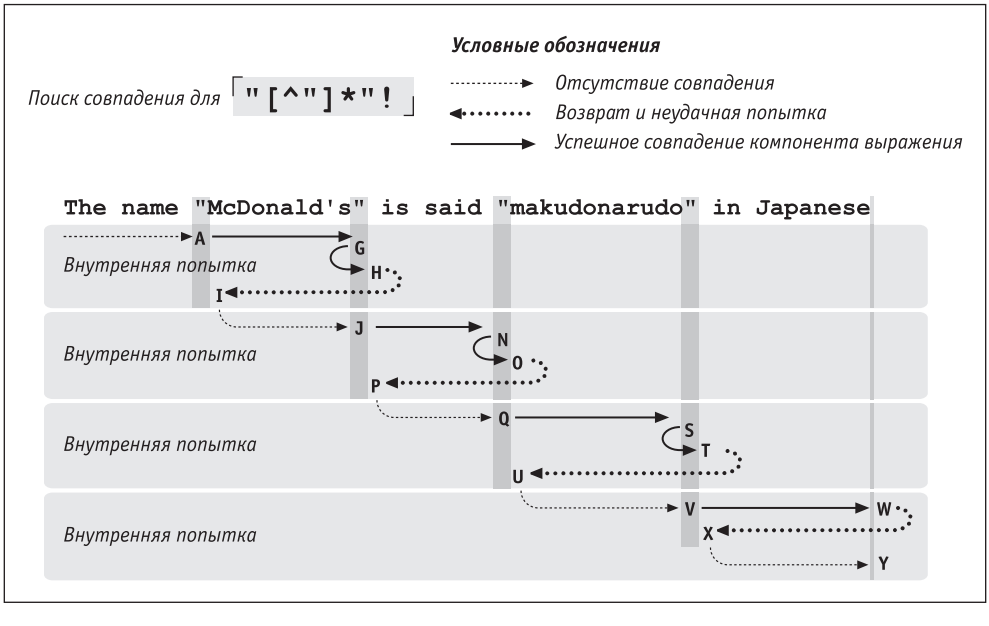


Рис. 6.5. Неудачный поиск совпадения для ⌜`"[^"]*"!`⌟

На рис. 6.5 показано, что происходит при неудачной попытке (ср. с рис. 6.4). Как видите, количество возвратов значительно уменьшилось. Если новый результат вам подходит, то сокращение возвратов можно считать положительным побочным эффектом.

Конструкция выбора может дорого обойтись

Конструкция выбора является одной из главных причин, порождающих возвраты. В качестве простого примера мы воспользуемся знакомым текстом `makudonarudo` и сравним, как организуется поиск совпадений для выражений `⌜u|v|w|x|y|z⌟` и `⌜[uvwxyz]⌟`. Символьный класс проверяется простым сравнением[1], поэтому `⌜[uvwxyz]⌟` «страдает» только от возвратов перехода к следующей позиции в строке (всего 34) до обнаружения совпадения:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

Использование выражения `⌜u|v|w|x|y|z⌟` требует шести возвратов для каждой начальной позиции, поэтому то же совпадение будет найдено только после 204 возвратов. Конечно, далеко не каждую конструкцию выбора удается заменить каким-нибудь эквивалентом, но даже в этом случае замена не всегда происходит так просто, как в этом примере. Впрочем, мы рассмотрим некоторые приемы, которые в ряде ситуаций кардинально сокращают количество возвратов, необходимых для поиска совпадения.

Вероятно, хорошее понимание возврата можно считать важнейшим аспектом эффективности в НКА, но оно остается всего лишь одной из частей уравнения. Оптимизации механизма регулярных выражений способны оказать *очень сильное* влияние на эффективность. Позднее в этой главе мы подробно рассмотрим, что должен сделать механизм регулярных выражений и как оптимизируется его быстродействие.

Хронометраж

В этой главе много говорится о скорости и эффективности, а также часто упоминаются результаты эталонных тестов, поэтому я хочу кратко рассказать об основных принципах хронометража. Также будут приведены простые примеры хронометража на нескольких языках.

[1] Реализации различаются по эффективности, но в общем случае можно смело предположить, что класс всегда работает быстрее эквивалентной конструкции выбора.

Простейший хронометраж сводится к измерению времени, потраченного на выполнение некоторых действий. Мы запрашиваем системное время, выполняем нужные действия, снова получаем системное время и вычисляем разность. Для примера давайте сравним время обработки выражений ⌜^(a|b|c|d|e|f|g)+$⌟ и ⌜^[a-g]+$⌟. Сначала методика хронометража поясняется примерами на языке Perl, но ниже будут представлены и другие языки. Простой (и как вы вскоре убедитесь, несовершенный) сценарий Perl выглядит так:

```
use Time::HiRes 'time'; # Функция time() возвращает значение
                        # с высокой степенью точности.
$StartTime = time();
"abababdedfg" =~ m/^(a|b|c|d|e|f|g)+$/;
$EndTime = time();
prlntf("Alternation takes %.3f seconds.\n",$EndTime - $StartTime);

$StartTime = time();
"abababdedfg" =~ m/^[ag]+$/;
$EndTime = time();
printf("Character class takes %.3f seconds.\n", $EndTime - $StartTime);
```

Сценарий выглядит просто (и внешнее впечатление вас не обманывает), но при разработке хронометражных тестов необходимо учитывать некоторые важные обстоятельства:

- **В измерения должны включаться только «интересные» операции.** Постарайтесь исключить из хронометража все «посторонние» операции. Например, инициализацию или другие подготовительные операции следует выполнять перед тем, как фиксируется начальная точка интервала. Завершающие операции, соответственно, должны выполняться после фиксации конца интервала.

- **Позаботьтесь о том, чтобы при хронометраже выполнялось достаточное количество операций.** Нередко измеряемая операция выполняется очень быстро, и ограниченная точность системных часов просто не позволяет добиться осмысленных результатов.

  При выполнении приведенного сценария Perl на моем компьютере результат выглядит так:

  ```
  Alternation takes 0.000 seconds
  Character class takes 0.000 seconds
  ```

  Полученные данные говорят только об одном: продолжительность обеих операций меньше минимального промежутка времени, измеряемого системными часами. Следовательно, быстрые операции необходимо выполнить дважды, 10 или 10 000 000 раз — сколько потребуется для того, чтобы это количество стало «достаточным». Конкретное число зависит от точности системных часов.

В большинстве современных систем часы отсчитывают интервалы с точностью до 1/100 секунды, при этом даже полусекунды суммарного времени хватит для получения осмысленных результатов.

- **Правильно организуйте хронометраж.** Если очень быстрая операция выполняется 10 миллионов раз, в полученное время будут включены затраты на 10 миллионов обновлений переменной счетчика в тестируемом блоке. Старайтесь увеличивать объем *реальной* работы так, чтобы при этом не увеличивался объем *лишней* работы. В примере на Perl регулярные выражения применяются к относительно коротким строкам; в длинных строках доля «реальной» работы будет увеличиваться.

Итак, с учетом всего сказанного мы приходим к новой версии:

```
use Time::HiRes 'time' # Функция time() возвращает значение
                       # с высокой степенью точности
$TimesToDo = 1000;                  # Простая инициализация
$TestString = "abababdedfg" x 1000; # ОЧЕНЬ большая строка

$Count = $TimesToDo;
$StartTime = time();
while ($Count-- > 0) {
    $TestString =~ m/^(a|b|c|d|e|f|g)+$/;
}
$EndTime = time();
printf("Alternation takes %.3f seconds.\n", $EndTime  $StartTime);

$Count = $TimesToDo;
$StartTime = time();
while ($Count-- > 0) {
    $TestString =~ m/^[ag]+$/;
}
$EndTime = time();
printf("Character class takes %.3f seconds.\n", $EndTime - $StartTime);
```

Обратите внимание: переменные `$TestString` и `$Count` инициализируются перед началом хронометража. Переменная `$TestString` инициализируется при помощи удобного оператора **x**, который дублирует строку в заданном количестве экземпляров. На моем компьютере с Perl 5.8 программа выводит следующий результат:

```
Alternation takes 7.276 seconds
Character class takes 0.333 seconds
```

Как видите, второй вариант работает почти в 22 раза быстрее первого. Тесты рекомендуется проводить несколько раз и выбирать результат с минимальным временем, чтобы уменьшить возможное влияние системных операций, проводимых в фоновом режиме.

Зависимость результатов хронометража от данных

Интересно посмотреть, что произойдет при инициализации переменных другими значениями:

```
$TimesToDo = 1000000:
$TestStrlng = "abababdedfg";
```

Тестовая строка стала в 1000 раз короче, но мы проводим в 1000 раз больше тестов. Логика подсказывает, что объем «работы» должен остаться прежним, но выходные данные выглядят совершенно иначе:

```
Alternation takes 18.167 seconds
Character class takes 5.231 seconds
```

Оба теста выполняются медленнее, чем раньше. Чем это объясняется? После внесенных изменений возросла относительная доля «лишней» работы, т. е. обновление и проверка `$Count` и подготовка механизма регулярных выражений производятся в 1000 раз чаще, чем прежде.

Новые тесты выполняются на 11 и 5 секунд медленнее старых. Вероятно, основная часть этого времени тратится на операции с сохраняющими круглыми скобками (которые требуют дополнительного времени для своей работы) и потому вполне естественно, что после увеличения числа тестов в 1000 раз мы получили такое снижение производительности.

Таким образом, полученные результаты наглядно демонстрируют, насколько сильно могут влиять непроизводительные расходы на результаты хронометража.

Хронометраж в языке PHP

Ниже приводится пример хронометража на языке PHP, где используется механизм preg:

```
$TimesToDo = 1000;
/* Подготовка тестовой строки */
$TestString = "";
for ($i = 0; $i < 1000; $i++)
    $TestString .= "abababdedfg";

/* Выполнить первый тест */
$start = gettimeofday();
for ($i = 0; $i < $TimesToDo; $i++)
    preg_match('/^(a|b|c|d|e|f|g)+$/', $TestString);
$final = gettimeofday();
```

```
$sec = ($final['sec'] + $final['usec']/1000000) -
       ($start['sec'] + $start['usec']/1000000);
printf("Alternation takes %.3f seconds\n", $sec);

/* И второй тест */
$start = gettimeofday();
for ($i = 0; $i < $TimesToDo; $i++)
     preg_match('/^[a-g]+$/', $TestString);
$final = gettimeofday();
$sec = ($final['sec'] + $final['usec']/1000000) -
       ($start['sec'] + $start['usec']/1000000);
printf("Character class takes %.3f seconds\n", $sec);
```

На моем компьютере получились следующие результаты:

```
Alternation takes 27.404 seconds
Character class takes 0.288 seconds
```

Если при проведении тестов на языке PHP будет получено сообщение «not being safe to rely on the system's timezone settings» («небезопасно полагаться на параметры настройки часового пояса в системе»), тогда в начало сценария следует добавить строки:

```
if (phpversion() >= 5)
    date_default_timezone_set("GMT");
```

Хронометраж в языке Java

Хронометраж в программах на языке Java затрудняется рядом обстоятельств. Сначала мы рассмотрим наивный пример, а затем разберемся, почему он наивен и как его усовершенствовать:

```
import java.util.regex.*;
public class JavaBenchmark {
  public static void main(String [] args)
  {
    Matcher regex1 = Pattern.compile("^(a|b|c|d|e|f|g)+$").matcher("");
    Matcher regex2 = Pattern.compile("^[a-g]+$").matcher("");
    long timesToDo = 1000;

    StringBuffer temp = new StringBuffer();
    for (int i = 1000; i > 0; i --)
      temp.append("abababdedfg");
    String testString = temp.toString();

    // Хронометраж первого варианта...
    long count = timesToDo;
    long startTime = System.currentTimeMillis();
```

```
    while (-- count > 0)
      regex1.reset(testString).find();
    double seconds = (System.currentTimeMillis() - startTime)/1000.0;
    System.out.println("Alternation takes " + seconds + " seconds");

    // Хронометраж второго варианта...
    count = timesToDo;
    startTime = System.currentTimeMillis();
    while (-- count > 0)
      regex2.reset(testString).find();
    seconds = (System.currentTimeMillis() - startTime)/1000.0;
    System.out.println("Character class takes " + seconds + " seconds");
  }
}
```

Обратите внимание: регулярные выражения компилируются во время инициализации программы. Мы хотим измерять скорость поиска, а не скорость компиляции.

Скорость также зависит от того, какая виртуальная машина (VM) при этом используется. Стандартная среда JRE от Sun содержит две виртуальные машины: *клиентскую* виртуальную машину, оптимизированную для быстрого запуска, и *серверную* виртуальную машину, оптимизированную для интенсивной работы.

На моем компьютере при проведении тестов на клиентской машине были получены следующие данные:

```
Alternation takes 19.318 seconds
Character class takes 1.685 seconds
```

На серверной машине результат выглядит иначе:

```
Alternation takes 12.106 seconds
Character class takes 0.657 seconds
```

Я назвал этот пример наивным, а сам процесс хронометража — делом нетривиальным, потому что результаты сильно зависят от качества автоматической компиляции и от того, как компилятор времени выполнения взаимодействует с тестируемым кодом. На некоторых виртуальных машинах используется JIT-компиляция (Just-In-Time, т. е. «своевременная компиляция», или «компиляция во время исполнения»), при которой код компилируется на стадии работы программы, непосредственно перед использованием.

В реализации Java используется компилятор, который я бы назвал BLTN-компилятором (Better Late Than Never — «лучше поздно, чем никогда»), который работает непосредственно *во время* выполнения, компилируя и оптимизируя интенсивно используемый код. Природа BLTN-компилятора такова, что он начинает работать лишь после того, как определит, что некоторый код интенсивно использу-

ется. Виртуальная машина, работавшая в течение некоторого времени (например, на сервере), находится в «разогретом» состоянии, тогда как в наших тестах был задействован «холодный» сервер (т. е. BLTN-оптимизации еще не проводились).

Один из способов получения данных для «разогретого» состояния основан на выполнении хронометража в цикле:

```
// Хронометраж первого варианта
for (int i = 4; i > 0; i--)
{
    long count = timesToDo;
    long startTime = System.currentTimeMil1is();
    while (-- count > 0)
        regex1.reset(testString).find();
    double seconds = (System.currentTimeMillis() - startTime)/1000.0;
    System.out.println("Alternation takes " + seconds + " seconds");
}
```

Если дополнительный цикл выполняется достаточное количество раз (скажем, в течение 10 секунд), BLTN оптимизирует активно используемый код, и последнее возвращаемое время дает представление о повышении производительности в «разогретой» системе. При повторном тестировании на серверной виртуальной машине достигается ускорение соответственно на 8 и 25%:

```
Alternation takes 11.151 seconds
Character class takes 0.483 seconds
```

Хронометраж в Java также усложняется непредсказуемым характером сборки мусора и выделения времени программным потокам. Повторюсь, проведение тестов в течение достаточно долгого времени помогает компенсировать воздействие этих непредсказуемых факторов.

Хронометраж в языке VB.NET

Ниже приводится пример хронометража на языке VB.NET:

```
Option Explicit On
Option Strict On

Imports System.Text.RegularExpressions

Module Benchmark
Sub Main()
    Dim Regex1 as Regex = New Regex("^(a|b|c|d|e|f|g)+$")
    Dim Regex2 as Regex = New Regex("^[a-g]+$")
    Dim TimesToDo as Integer = 1000
```

```
    Dim TestString as String = ""
    Dim I as Integer
    For I = 1 to 1000
        TestString = TestString & "abababdedfg"
    Next

    Dim StartTime as Double = Timer()
    For I = 1 to TimesToDo
        Regex1.Match(TestString)
    Next
    Dim Seconds as Double = Math.Round(Timer()  StartTime, 3)
    Console.WriteLine("Alternation takes " & Seconds & " seconds")

    StartTime = Timer()
    For I = 1 to TimesToDo
        Regex2.Match(TestString)
    Next
    Seconds = Math.Round(Timer() - StartTime, 3)
    Console.WriteLine("Character class takes " & Seconds & " seconds")
End Sub
End Module
```

На моем компьютере он выдает следующий результат:

```
Alternation takes 13.311 seconds
Character class takes 1.680 seconds
```

.NET Framework позволяет компилировать выражения в более эффективную форму, для чего конструктору `Regex` передается второй аргумент `RegexOptions.Compiled` (☞ 511). В новом варианте тесты выполняются значительно быстрее:

```
Alternation takes 5.499 seconds
Character class takes 1.157 seconds
```

Параметр `Compiled` ускоряет оба теста, но особенно заметный выигрыш достигается в первом случае (почти в 3 раза, тогда как второй тест ускоряется всего в 1,5 раза). Из этого можно сделать вывод, что конструкция выбора выигрывает от более эффективной компиляции в большей степени, чем символьный класс.

Хронометраж в языке Ruby

Ниже приводится пример хронометража на языке Ruby:

```
TimesToDo=1000
testString=""
for i in 1..1000
```

```
    testString += "abababdedfg"
end

Regex1 = Regexp::new("^(a|b|c|d|e|f|g)+$");
Regex2 = Regexp::new("^[a-g]+$");

startTime = Time.new.to_f
for i in 1..TimesToDo
    Regex1.match(testString)
end
print "Alternation takes %.3f seconds\n" % (Time.new.to_f - startTime);

startTime = Time.new.to_f
for i in 1..TimesToDo
    Regex2.match(testString)
end
print "Character class takes %.3f seconds\n" % (Time.new.to_f - startTime);
```

На моем компьютере он выдает следующий результат:

```
Alternation takes 16.311 seconds
Character class takes 3.479 seconds
```

Хронометраж в языке Python

Ниже приводится пример хронометража в языке Python:

```
import re
import time
import fpformat

Regex1 = re.compile("^(a|b|c|d|e|f|g)+$")
Regex2 = re.compile("^[a-g]+$")

TimesToDo = 1250;
TestString = ""
for i in range(800):
    TestString += "abababdedfg"

StartTime = time.time()
for i in range(TimesToDo):
    Regex1.search(TestString)
Seconds = time.time() - StartTime
print "Alternation takes " + fpformat.fix(Seconds,3) + " seconds"

StartTime = time.time()
for i in range(TimesToDo):
```

```
    Regex2.search(TestString)
Seconds = time.time() - StartTime
print "Character class takes " + fpformat.fix(Seconds,3) + " seconds"
```

Из-за специфики механизма регулярных выражений Python мне пришлось немного урезать размер строки, поскольку исходная версия вызывала внутреннюю ошибку («превышение максимальной глубины рекурсии»).

Для компенсации этого сокращения тест был проведен большее (пропорционально) число раз. На моем компьютере он выдает следующий результат:

```
Alternation takes 10.357 seconds
Character class takes 0.769 seconds
```

Хронометраж в языке Tcl

Ниже приводится пример хронометража в языке Tcl:

```
set TimesToDo 1000
set TestString ""
for {set i 1000} {$i > 0} {incr i -1} {
    append TestString "abababdedfg"
}

set Count $TimesToDo
set StartTime [clock clicks -milliseconds]
for {} {$Count > 0} {incr Count -1} {
    regexp {^(a|b|c|d|e|f|g)+$} $TestString
}
set EndTime [clock clicks -milliseconds]
set Seconds [expr ($EndTime - $StartTime)/1000.0]
puts [format "Alternation takes %.3f seconds" $Seconds]

set Count $TimesToDo
set StartTime [clock clicks -milliseconds]
for {} {$Count > 0} {incr Count -1} {
     regexp {^[a-g]+$} $TestString

}
set EndTime [clock clicks -milliseconds]
set Seconds [expr ($EndTime - $StartTime)/1000.0]
puts [format "Character class takes %.3f seconds" $Seconds]
```

На моем компьютере он выдает следующий результат:

```
Alternation takes 0.362 seconds
Character class takes 0.352 seconds
```

Оба варианта работают практически с одинаковой скоростью! Впрочем, это легко объяснимо: как указано в таблице на с. 194, в Tcl используется гибридный механизм НКА/ДКА, и с точки зрения механизма ДКА эти регулярные выражения эквивалентны. Большая часть из того, о чем говорится в этой главе, к Tcl попросту не относится. Дополнительная информация приводится во врезке на с. 309.

Стандартные оптимизации

Грамотно реализованный механизм регулярных выражений может разными способами оптимизировать получение желаемых результатов. Оптимизации обычно делятся на две категории:

- **Ускорение операций.** Некоторые конструкции (такие, как `\d+`) встречаются так часто, что для них может быть предусмотрена особая обработка, превосходящая по скорости общую обработку регулярных выражений.
- **Предотвращение лишних операций.** Если механизм решает, что некоторая операция не нужна для достижения правильного результата или может применяться к меньшему объему текста, чем указано в исходном выражении, он может отказаться от выполнения этих операций и ускорить обработку выражения. Например, регулярное выражение, начинающееся с `\A` (начало текста), может совпадать только от начала строки, поэтому при отсутствии совпадения в этой позиции механизму смещения не нужно проверять другие позиции.

В этом разделе описаны многие известные мне оптимизации, весьма разнообразные и хитроумные. Ни один язык или программа не поддерживает весь набор оптимизаций (или хотя бы такого же, как в другой программе или языке), но после чтения этой главы вы будете лучше ориентироваться в этом вопросе и сможете лучше воспользоваться оптимизациями, которые поддерживаются вашей программой.

Ничто не дается бесплатно

Оптимизации часто ускоряют обработку выражений, но такой результат не гарантирован. Выигрыш по времени достигается лишь в том случае, если экономия времени превышает затраты хотя бы на проверку самой возможности применения оптимизации. Если механизм проверяет, можно ли применить оптимизацию, и приходит к отрицательному выводу, обработка выражения *замедляется* из-за включения бесполезной проверки в общее время стандартной обработки. Итак, существует некий баланс между затратами времени на оптимизацию, достигнутой экономией и — что самое важное — вероятностью, с которой эта оптимизация будет задействована.

Рассмотрим пример. Выражение ⌜\b\B⌟ (*граница слова* в позиции, которая одновременно *не является границей слова*) явно не совпадает ни при каких условиях. Если механизм регулярных выражений поймет, что подвыражение ⌜\b\B⌟ обязательно для достижения общего совпадения, он сделает вывод о невозможности совпадения и вообще не будет применять регулярное выражение. Вместо этого он сразу сообщает о неудаче. Для длинных строк экономия может быть весьма существенной.

Однако ни в одном из известных мне механизмов эта оптимизация не применяется. Почему? Прежде всего, потому, что иногда бывает трудно решить, применима ли она к конкретному выражению. Выражение может содержать ⌜\b\B⌟ и при этом совпадать, поэтому для полной уверенности механизму придется заранее выполнять лишнюю работу. С другой стороны, экономия может оказаться весьма существенной, и если бы конструкция ⌜\b\B⌟ встречалась часто, такая оптимизация имела бы смысл.

Но она встречается крайне редко, поэтому, несмотря на большой потенциальный выигрыш, замедление обработки всех регулярных выражений неоправданно[1].

Универсальных истин не бывает

Помните об этом, когда будете знакомиться с различными видами оптимизаций, представленными в этой главе. Хотя я постарался подобрать для каждой оптимизации простое, понятное название, может оказаться, что в разных механизмах она реализуется с существенными различиями. Внешне незаметное изменение в регулярном выражении способно заметно ускорить его работу в одной реализации и серьезно замедлить ее в другой реализации.

Механика применения регулярных выражений

Прежде чем рассматривать приемы оптимизации в сложных системах и возможности их практического использования, необходимо сначала разобраться в основных принципах применения регулярных выражений. Мы успели подробно рассмотреть процесс возврата, а в этом коротком разделе мы попробуем окинуть взглядом общую картину.

---

[1] Раньше я использовал конструкцию ⌜\b\B⌟, чтобы обеспечить гарантированное несовпадение некоторой части выражения на стадии тестирования. Например, ее можно было бы вставить в отмеченную позицию выражения ⌜…(this▲|this other)…⌟, чтобы первая альтернатива никогда не совпадала. Сейчас я использую в «заведомо несовпадающих» компонентах подвыражение ⌜(?!)⌟. Интересный, хотя и специфический для Perl пример на эту тему приведен на с. 418.

Ниже перечислены основные этапы применения регулярного выражения к целевой строке:

1. **Компиляция регулярного выражения.** Регулярное выражение проверяется на наличие ошибок. Если выражение имеет правильный синтаксис, оно компилируется во внутреннее представление.

2. **Начальное позиционирование.** Подсистема смещения текущей позиции «устанавливает» текущую позицию в начале целевого текста.

3. **Проверка компонентов.** Механизм обрабатывает целевой текст и регулярное выражение, перемещаясь между компонентами регулярного выражения, как было описано в главе 4. Процесс возврата в НКА достаточно подробно описан выше, здесь следует упомянуть лишь некоторые дополнительные обстоятельства:

   - Последовательные компоненты (например, подвыражения ⌜`S`⌟, ⌜`u`⌟, ⌜`b`⌟, ⌜`j`⌟, ⌜`e`⌟, ... строки ⌜`Subject`⌟) обрабатываются друг за другом. Их обработка прерывается только в том случае, если для какого-то компонента не удается найти совпадение;

   - При обработке квантификаторов управление передается между квантификатором (по которому механизм узнает, нужно ли продолжать поиск совпадений) и компонентом, содержащим квантификатор (проверка совпадения);

   - Вход в сохраняющие круглые скобки и выход из них сопряжен с некоторыми затратами. Текст, совпавший с подвыражением в скобках, сохраняется, чтобы работали переменные `$1`, `$2` и их аналоги. Поскольку выход из круглых скобок может произойти из-за возврата, состояние круглых скобок является частью сохраненного состояния, используемого при возврате, поэтому вход и выход из скобок требует модификации этого состояния.

4. **Обнаружение совпадения.** Обнаружив совпадение, традиционный механизм НКА «фиксирует» текущее состояние и сообщает об успехе поиска в целом. С другой стороны, механизм POSIX НКА всего лишь запоминает потенциальное совпадение, если оно имеет максимальную длину из всех найденных до настоящего момента, и продолжает поиск по всем оставшимся сохраненным состояниям. Когда состояний больше не остается, механизм возвращает самое длинное совпадение.

5. **Смещение начальной позиции.** Если совпадение не найдено, начальная позиция смещается к следующему символу текста, после чего регулярное выражение применяется заново (начиная с шага 3).

6. **Общая неудача.** Если после применения регулярного выражения с каждой позиции символа в целевой строке, в том числе и за последним символом, совпадение так и не будет найдено, механизм сообщает об общей неудаче.

В нескольких ближайших разделах описаны различные пути сокращения объема работы в «умных» реализациях, а также возможности их применения умными пользователями.

Предварительные оптимизации

Хорошая реализация способна уменьшить объем работы, выполняемой на стадии фактического поиска. Но в некоторых ситуациях она может быстро определить, что регулярное выражение не совпадет никогда, поэтому проводить поиск вообще не нужно.

Кэширование при компиляции

Вспомните почтовую мини-программу из главы 2 (☞ 85). Общая структура главного цикла, обрабатывавшего каждую строку заголовка, выглядела так:

```
while (…) {
    if ($line =~ m/^\s*$/ ) …
    if ($line =~ m/^Subject: (.*)/) …
    if ($line =~ m/^Date: (.*)/) …
    if ($line =~ m/^Reply-To: (\S+)/) …
    if ($line =~ m/^From: (\S+) \(([^()]*)\)/) …
      ⋮
}
```

Перед использованием регулярное выражение должно быть проверено и откомпилировано во внутреннее представление. После компиляции внутренняя форма может использоваться при проверке разных строк. Конечно, было бы крайне неэффективно заново перекомпилировать регулярное выражение при каждой итерации цикла. Разумнее было бы сохранить (*кэшировать*) внутреннее представление после первой компиляции и использовать его для всех последующих применений в цикле.

Возможность применения этой оптимизации зависит от типа интерфейса для работы с регулярными выражениями, поддерживаемого приложением. Как упоминалось на с. 131, существуют три разновидности интерфейса: *интегрированный*, *процедурный* и *объектно-ориентированный*.

Кэширование при компиляции для интегрированного интерфейса

Интегрированный интерфейс, используемый в Perl и awk, позволяет легко реализовать кэширование при компиляции. Во внутреннем представлении каждое регулярное выражение связывается с некоторым фрагментом программного кода;

ДКА, TCL И РУЧНАЯ НАСТРОЙКА РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Большинство оптимизаций, описанных в этой главе, просто не относится к ДКА. Оптимизация *кэширования при компиляции*, описанная выше, поддерживается всеми типами механизмов, но ни один прием «ручной настройки» выражений, рассматриваемый в этой главе, не относится к ДКА. Как показано в главе 4 (☞ 208), логически эквивалентные выражения (например, `⌈this|that⌋` и `⌈th(is|at)⌋`) *полностью* эквивалентны с точки зрения ДКА. С другой стороны, с точки зрения НКА они не всегда эквивалентны, иначе эта глава оказалось бы лишней.

Но что можно сказать о языке Tcl с его гибридным механизмом ДКА/НКА? Этот механизм разрабатывался специально для Tcl Генри Спенсером, живой легендой мира регулярных выражений (☞ 124). Генри удалось совершенно фантастическим образом объединить лучшие черты механизмов ДКА и НКА. Как он сам написал в апреле 2000 года в своем сообщении в Usenet:

> «...В общем случае механизм регулярных выражений Tcl зависит от конкретной формулировки выражения в гораздо меньшей степени, чем традиционные механизмы. Если выражение обрабатывается быстро, оно обрабатывается быстро всегда, как бы оно ни было сформулировано; медленные выражения тоже обрабатываются медленно независимо от формулировки. Старые принципы ручной оптимизации регулярных выражений в данном случае просто не действуют».

Механизм регулярных выражений Tcl стал важным шагом на пути прогресса. Если эта технология получит широкое распространение, большая часть материала этой главы окажется ненужной.

откомпилированное представление ассоциируется с кодом только при первом выполнении, а в дальнейшем просто используется готовая ссылка. Тем самым обеспечивается максимальная оптимизация по скорости за счет затрат памяти на хранение всех кэшированных выражений.

Интерполяция переменных в операндах (т. е. подстановки значения переменной в регулярное выражение) способна нарушить ход кэширования. При интерполяции переменных в выражениях вида **`m/^Subject:•\Q$DesiredSubject`**`\E\s*$/` регулярное выражение может изменяться между итерациями, поскольку оно зависит от значения переменной. Если регулярное выражение изменяется, его приходится каждый раз компилировать заново, поэтому использование готовых результатов невозможно.

Точнее говоря, регулярное выражение *может* изменяться при каждой итерации, но это не означает, что его следует каждый раз компилировать заново. Возможен промежуточный вариант оптимизации с проверкой результата интерполяции

(фактического значения, используемого в регулярном выражении) и перекомпиляции выражения только в том случае, если оно отличается от предыдущего. Если значение изменяется каждый раз, никакие оптимизации невозможны и регулярное выражение действительно приходится компилировать заново. Но если выражение изменяется относительно редко, в большинстве случаев выполняется проверка без последующей компиляции, что заметно ускоряет обработку выражения.

Кэширование при компиляции для процедурного интерфейса

При интегрированном интерфейсе использование регулярного выражения связывается с определенной точкой программы, что позволяет кэшировать откомпилированную версию выражения и использовать ее при следующей передаче управления в указанную точку. Однако при процедурном подходе существует общая функция «применения регулярного выражения», которая вызывается по мере надобности. Это означает, что в программе не существует позиции, с которой связывается откомпилированное представление, поэтому при следующем вызове функции выражение должно компилироваться заново. Во всяком случае так должно быть в теории, но на практике было бы слишком неэффективно полностью отказываться от кэширования. Вместо этого программы обычно кэшируют последние использованные регулярные выражения и связывают каждый шаблон с его откомпилированным представлением.

При вызове функция применения регулярного выражения сравнивает свой аргумент-выражение с кэшированными выражениями и использует готовое внутреннее представление, если оно существует. Если аргумент отсутствует в кэше, функция компилирует выражение и сохраняет его в кэше (возможно, с удалением старых выражений, если размер кэша ограничен). При переполнении кэша и необходимости удаления откомпилированного представления обычно удаляется самый старый элемент (т. е. тот, с момента последнего использования которого прошло больше всего времени).

GNU Emacs кэширует 20 выражений, в Tcl кэш рассчитан на 30 выражений, а PHP способен хранить до 4000 выражений. Платформа .NET по умолчанию хранит в кэше всего 15 выражений, однако это значение можно увеличить или уменьшить в ходе исполнения программы.

Увеличение размеров кэша в некоторых ситуациях играет очень важную роль: если количество регулярных выражений, используемых в цикле, превышает размер кэша, то к началу новой итерации первое выражение будет удалено из кэша, следовательно, при каждой итерации *все* выражения необходимо будет перекомпилировать.

Кэширование при компиляции для объектно-ориентированного интерфейса

В объектно-ориентированном интерфейсе компиляция находится в полном распоряжении программиста. Регулярные выражения компилируются в конструкторах объектов (`New Regex`, `re.compile` и `Pattern.compile` соответственно для .NET, Python и `java.util.regex`). В простых примерах главы 3 (начиная со с. 133) регулярные выражения компилируются непосредственно перед использованием, но ничто не мешает откомпилировать их раньше (например, перед началом цикла и даже во время инициализации программы) и затем свободно использовать по мере необходимости. В частности, это делается в примерах хронометража на с. 299, 301 и 302.

Объектно-ориентированный подход также позволяет программисту управлять уничтожением откомпилированной формы выражения с помощью деструктора объекта. Возможность немедленного уничтожения ненужных откомпилированных форм снижает затраты памяти.

Предварительная проверка обязательных символов/подстрок

Поиск в строке определенного символа (или подстроки) связан в НКА с существенно меньшими затратами, чем полноценное применение регулярного выражения, поэтому некоторые системы проводят дополнительный анализ выражения во время компиляции и проверяют наличие символов или подстрок, присутствие которых *обязательно* для совпадения. Перед фактическим применением регулярного выражения строка быстро проверяется на наличие обязательных символов или подстрок — если обязательные элементы не найдены, регулярное выражение применять не обязательно.

Например, в выражении ⌜`^Subject:•(.*)`⌟ строка '`Subject:•`' обязательна. Программа анализирует целевой текст, возможно, с применением алгоритма *Бойера—Мура* (*Boyer—Moore*), обеспечивающего быстрый поиск литералов в тексте (чем длиннее текст, тем выше эффективность поиска). Даже если программа не реализует алгоритм Бойера–Мура, все равно она может оптимизировать поиск — достаточно взять обязательный символ и проверить его наличие в целевом тексте. Для улучшения результата следует выбрать символ, реже встречающийся в целевом тексте (например, в примере с '`Subject:•`' выбор символа '`:`' лучше выбора '`t`').

Механизм регулярных выражений без труда определит ту часть ⌜`^Subject:•(.*)`⌟, которая представляет собой фиксированный литерал, обязательный в любом совпадении. С другой стороны, определить, что любое совпадение ⌜`this|that|other`⌟ содержит подстроку '`th`', несколько сложнее, и большинство реализаций этого не делает. Впрочем, и здесь не все однозначно — даже если реализация не понимает,

что подстрока 'th' является обязательной, она вполне может понять, что обязательны отдельные символы 't' и 'h', и выполнит проверку хотя бы на уровне символов.

Возможности реализаций по распознаванию обязательных символов и подстрок значительно различаются. Многие реализации отказываются от подобных проверок при наличии конструкции выбора, в таких системах выражение ⌜th(is|at)⌟ может обрабатываться быстрее ⌜this|that⌟. За дополнительной информацией обращайтесь к разделу «Исключение по первому символу/классу/подстроке» на с. 313.

Учет длины текста

Выражение ⌜^Subject:␣(.*)⌟ может совпасть с текстом произвольной длины, но длина совпадения в любом случае не может быть меньше 9 символов. Следовательно, для целевого текста меньшей длины даже не стоит активизировать механизм регулярных выражений. Конечно, преимущество такой оптимизации нагляднее проявляется для выражений с большей обязательной длиной, таких как ⌜:\d{79}:⌟ (минимальная длина совпадения равна 81 символу).

За дополнительной информацией обращайтесь к разделу «Учет длины текста при смещении» на с. 314.

Оптимизации при смещении текущей позиции

Даже если механизм регулярных выражений не может заранее решить, что некоторая строка ни при каких условиях не обнаружит совпадения, в некоторых случаях ему удается сократить количество начальных позиций, с которых ищется совпадение для регулярного выражения.

Привязка к началу текста/логической строки

Эта разновидность оптимизации понимает, что регулярные выражения, начинающиеся с ⌜^⌟, могут совпасть только от начала строки, поэтому их не следует применять с других позиций.

Все, что говорилось в разделе «Предварительная проверка обязательных символов/подстрок» о проверке применимости оптимизации, также относится к этому разделу. Любая реализация, в которой задействована эта оптимизация, должна понимать, что выражение ⌜^(this|that)⌟ может совпадать, только начиная с позиции, с которой совпадает ⌜^⌟, но лишь немногие реализации придут к тому же выводу для выражения ⌜^this|^that⌟. В подобных ситуациях формулировка выражения в виде ⌜^(this|that)⌟ или, еще лучше, ⌜^(?:this|that)⌟ способна заметно ускорить поиск.

Аналогичные оптимизации применяются для метасимвола ⌜\A⌟, а также для ⌜\G⌟ (при повторном поиске).

Косвенная привязка к началу строки

Механизм, в котором задействована эта оптимизация, понимает, что если выражение начинается с ⌜`.*`⌟ или ⌜`.+`⌟ и не содержит глобальной конструкции выбора, к нему можно присоединить неявный префикс ⌜`^`⌟. Это позволит использовать оптимизацию привязки *к началу текста/логической строки* (см. предыдущий раздел), что может обеспечить значительную экономию.

Более совершенная система поймет, что аналогичная оптимизация может применяться и в том случае, когда начальная конструкция ⌜`.*`⌟ или ⌜`.+`⌟ заключена в круглые скобки, но если скобки являются сохраняющими, необходима особая осторожность. Например, выражение ⌜`(.+)X\1`⌟ находит повторяющиеся строки слева и справа от 'X', и появление неявного префикса ⌜`^`⌟ приведет к тому, что выражение не совпадет в строке '1234X2345'[1].

Привязка к концу текста/логической строки

Эта разновидность оптимизации основана на том, что совпадения некоторых регулярных выражений, завершающихся метасимволом ⌜`$`⌟ или другими якорями конца строки (☞ 175), должны отделяться от конца строки определенным количеством байтов. Например, для выражения ⌜`regex(es)?$`⌟ совпадение должно начинаться не более чем за 8[2] символов от конца строки, поэтому механизм может сразу перейти к этой позиции. При большой длине целевого текста это существенно сокращает количество начальных позиций поиска.

Исключение по первому символу/классу/подстроке

Более общий вариант *предварительной проверки обязательных символов/подстрок*. Данный вид оптимизации руководствуется той же исходной информацией (любое совпадение должно начинаться с конкретного символа или подстроки), на основании которой производится быстрая проверка и применение регулярного выражения только с соответствующих позиций строки. Например, выражение ⌜`this|that|other`⌟ может совпадать только в позициях, начинающихся с ⌜`[ot]`⌟; механизм проверяет все символы строки и применяет выражение только с соответствующих позиций, что может привести к огромной экономии времени. Чем

[1] Любопытная деталь: в Perl эта ошибка из области «чрезмерной оптимизации» оставалась незамеченной в течение 10 лет, пока один из разработчиков Perl Джефф Пиньян (Jeff Pinyan) не обнаружил и не исправил ее в начале 2002 года. Конечно, регулярные выражения типа ⌜`(.+)X\1`⌟ встречаются нечасто, иначе ошибка была бы обнаружена гораздо раньше.

[2] Восемь, а не семь, потому что во многих диалектах ⌜`$`⌟ может совпадать перед завершающим символом новой строки (☞ 175).

длиннее проверяемая строка, тем меньше вероятность ложной идентификации начальных позиций.

Проверка внутренних литералов

Этот тип оптимизации имеет много общего с *исключением по начальной подстроке*, но он рассматривает литеральные подстроки, находящиеся на заданном расстоянии от начала при любом совпадении. Например, в любом совпадении выражения ⌜\b(perl|java)\.regex\.info\b⌟ после первых 4 символов следует подстрока '.regex.info'. Умная реализация воспользуется алгоритмом Бойера—Мура, найдет подстроку '.regex.info' и применит регулярное выражение на четыре позиции раньше.

В общем случае эта оптимизация работает, только если подстрока имеет одинаковое смещение во всех вариантах совпадения. Она неприменима к выражению ⌜\b(vb|java)\.regex\.info\b⌟, в совпадениях которого имеется общая подстрока, но находящаяся либо в двух, либо в четырех символах от начала. Кроме того, она неприменима к выражению ⌜\b(w+)\.regex\.info\b⌟, поскольку подстрока может быть смещена от начала на произвольное количество символов.

Учет длины текста при смещении

Данная разновидность оптимизации, напрямую связанная с оптимизацией *учета длины текста* (о которой рассказывалось на с. 312), позволяет механизму регулярных выражений прекратить дальнейшие попытки поиска, если из-за малого расстояния текущей позиции от конца строки совпадение стало невозможным.

Оптимизации на уровне регулярных выражений

Конкатенация подстрок

Вероятно, самая очевидная оптимизация заключается в том, чтобы интерпретировать подвыражение ⌜abc⌟ как единое целое, а не как совокупность трех частей «⌜a⌟, затем ⌜b⌟, затем ⌜c⌟». В этом случае подвыражение применяется за одну итерацию, что позволяет обойтись без затрат на три отдельные итерации.

Упрощение квантификаторов

Квантификаторы *, + и другие метасимволы, применимые к простым конструкциям (например, отдельным символам или символьным классам), часто оптимизируются таким образом, чтобы предотвратить большую часть затрат на последовательный перебор в стандартном механизме НКА. Основной цикл обработки должен быть

достаточно универсальным, чтобы справиться со всеми поддерживаемыми конструкциями. В программировании термин «универсальный» часто означает «медленный», поэтому эта важная оптимизация сводит простые квантифицированные конструкции ⌜.*⌟ в одну «часть» и заменяет общую механику обработки квантификаторов быстрой специализированной обработкой. Таким образом, проверка осуществляется без участия общего механизма регулярных выражений.

Например, конструкции ⌜.*⌟ и ⌜(?:.)*⌟ логически эквивалентны, но в системах с этим видом оптимизации ⌜.*⌟ работает гораздо быстрее ⌜(?:.)*⌟. Приведу несколько примеров: в пакете регулярных выражений для Java от Sun оно работает всего на 10% быстрее, но в Ruby и языках .NET скорость возрастает в 2,5 раза. В Python оптимизированное выражение работает в 50, а в PCRE/PHP — в 150 раз быстрее! В Perl реализована оптимизация, описанная в следующем подразделе, поэтому ⌜.*⌟ и ⌜(?:.)*⌟ работают с одинаковой скоростью (во врезке далее рассказано, как интерпретировать эти числа).

Исключение лишних круглых скобок

Если реализация понимает, что выражение ⌜(?:.)*⌟ в точности эквивалентно ⌜.*⌟, она использует оптимизацию из предыдущего раздела.

Исключение лишних символьных классов

Символьный класс, состоящий из одного символа, выглядит нелепо — он требует затрат, связанных с обработкой символьного класса, но не предоставляет никаких преимуществ. Умная реализация в процессе обработки выражения заменяет конструкции вида ⌜[.]⌟ на ⌜\.⌟.

Проверка символа, следующего за минимальным квантификатором

При использовании минимальных квантификаторов в конструкциях вида ⌜"(.*?)"⌟ механизм должен постоянно переходить между проверками совпадения квантифицированной конструкции (точки) и тем, что следует после нее (⌜"⌟). По этой и по ряду других причин минимальные квантификаторы обычно работают гораздо медленнее максимальных (особенно оптимизированных с применением *упрощения квантификаторов*, о котором говорилось пару разделов назад). Если минимальный квантификатор заключен в сохраняющие круглые скобки, управление приходится постоянно передавать внутрь скобок и за их пределы, что приводит к дополнительным затратам времени.

Данный вид оптимизации основан на простом наблюдении: если за минимальным квантификатором следует литерал, то минимальный квантификатор может

ИНТЕРПРЕТАЦИЯ РЕЗУЛЬТАТОВ ТЕСТОВ

В основном данные хронометража приводятся в виде отношения двух показателей для конкретного языка. Например, на с. 315 я упоминал о том, что некоторая оптимизированная конструкция в пакете регулярных выражений для Java от Sun работает на 10% быстрее неоптимизированной. В .NET Framework оптимизированный вариант работает в 2,5 раза быстрее неоптимизированного, а в PCRE оптимизированный вариант работает в 150 раз быстрее. В Perl это отношение равно единице (т. е. обе конструкции работают с одинаковой скоростью).

Какие же выводы можно сделать об относительной скорости перечисленных языков? Абсолютно никаких. 150-кратное ускорение оптимизированного варианта в PCRE может означать как то, что данный вид оптимизации особенно хорошо реализован по сравнению с другими языками, так и то, что неоптимизированный вариант работает слишком медленно. Как правило, я стараюсь избегать сравнения разных языков, чтобы не разводить споры на тему «чей язык быстрее».

Но как бы то ни было, будет интересно повнимательнее присмотреться к таким разным результатам, как 10-процентный выигрыш в Java и 150-кратное возрастание в PCRE. Неоптимизированное выражение ⌜`(?:.)*`⌟ работает в 11 раз *медленнее*, чем в Java, а оптимизированное выражение ⌜`.*`⌟ работает в 13 раз *быстрее*! Оптимизированные версии в Java и Ruby работают примерно с одинаковой скоростью, а неоптимизированная Ruby-версия примерно в 2,5 раза медленнее неоптимизированной Java-версии. С другой стороны, неоптимизированная Ruby-версия всего на 10% медленнее неоптимизированной Python-версии, но оптимизированная Python-версия в 20 раз быстрее оптимизированной Ruby-версии.

Все эти версии работают медленнее, чем аналогичные выражения в языке Perl. Как оптимизированная, так и неоптимизированная версия Perl работает на 10% быстрее самой быстрой Python-версии. Следует при этом помнить, что у каждого языка есть свои сильные стороны и приведенные цифры относятся лишь к конкретному тесту.

работать как обычный максимальный квантификатор до тех пор, пока механизм не достигнет этого символа. Реализация переключается на специализированную обработку минимального квантификатора, в процессе которой целевой текст быстро проверяется до нужного литерала в обход стандартной процедуры, пока проверяемая позиция не достигнет указанного литерала.

Среди разновидностей этой оптимизации стоит упомянуть предварительную проверку символьных классов вместо проверки конкретных литералов (например, предварительная проверка для класса ⌜`['"]`⌟ в выражении ⌜`['"](.*?)["']`⌟,

которая отчасти схожа с оптимизацией *исключения по первому символу*, описанной на с. 315).

Обнаружение «лишних» возвратов

Как было показано в разделе «Возвращение к реальности», на с. 289, некоторые комбинации квантификаторов (например, ⌜`(.+)*`⌟) генерируют возвраты в экспоненциальном количестве. Простой способ предотвращения экспоненциальных возвратов основан на их подсчете и отмене поиска, когда количество возвратов станет «слишком большим». Несомненно, такая возможность полезна, но она накладывает искусственные ограничения на объем текста, с которым могут использоваться некоторые регулярные выражения.

Например, если установить предел в 10 000 возвратов, конструкция ⌜`.*?`⌟ никогда не совпадет с текстом, длина которого превышает 10 000 символов, потому что совпадение каждого символа порождает потенциальный возврат. Тексты такого объема встречаются не так редко, особенно при работе с веб-страницами, поэтому подобные ограничения нежелательны.

По иным соображениям в некоторых реализациях ограничивается размер стека возвратов (количество одновременно сохраненных состояний). Например, Python позволяет одновременно хранить до 10 000 состояний. Ограничение размера стека, как и ограничение числа возвратов, уменьшает объем текста, с которым могут работать некоторые регулярные выражения.

Это обстоятельство несколько затрудняло построение хронометражных тестов во время работы над книгой. Для получения наилучших результатов «измеряемая» часть теста должна выполнять как можно больше полезной работы, поэтому я создавал огромные строки и сравнивал время, затраченное на обработку, скажем ⌜`"(.*)"`⌟, ⌜`"(.)*"`⌟, ⌜`"(.)*?"`⌟ и ⌜`"([^" ])*?"`⌟. Для получения осмысленных результатов мне приходилось ограничивать длину строк, чтобы не споткнуться об ограничения количества возвратов или размера стека. Пример приведен на с. 304.

Предотвращение экспоненциального поиска

С экспоненциальным поиском можно бороться и другим способом — своевременно обнаружить его. После этого можно потратить дополнительные усилия на отслеживание позиции, в которой применялось подвыражение каждого квантификатора, и предотвратить повторы.

Обнаружить факт экспоненциального поиска относительно просто — обычно количество итераций квантификатора не превышает количество символов в целевом тексте. Если вы уже знаете, что в вашем случае поиск может продолжаться вечно, встает гораздо более сложная задача — обнаружить и ликвидировать лишние по-

тенциальные совпадения. Впрочем, если учесть, что альтернатива — очень, очень долгий поиск, эти усилия окупаются.

Одно из отрицательных последствий обнаружения экспоненциального поиска и быстрого возврата признака неудачи заключается в том, что неэффективность регулярного выражения остается скрытой от программиста. Даже при предотвращенном экспоненциальном поиске обработка выражения ведется гораздо медленнее обычного, но не настолько, чтобы вы это заметили. Ждать до захода солнца не нужно, проходит всего 1/100 секунды или около того — мгновение для нас, но целая вечность в компьютерном мире.

И все же игра стоит свеч. Многие программисты вообще не думают об эффективности; они опасаются регулярных выражений и хотят только одного — поскорее добиться желаемого результата. Возможно, раньше вы принадлежали к их числу, но я надеюсь, что после чтения книги все изменится.

Подавление состояний при использовании захватывающих квантификаторов

При нахождении потенциального совпадения для обычного квантификатора создается несколько сохраненных состояний (по одному на каждую итерацию квантификатора). Захватывающие квантификаторы (☞ 190) таких состояний не оставляют. Уничтожить лишние состояния можно после выполнения работы квантификатора, но существует более эффективное решение — удалять состояние предыдущей итерации во время текущей итерации (в процессе поиска одно состояние всегда должно присутствовать, чтобы при отсутствии совпадения для квантифицированного элемента можно было продолжить поиск общего совпадения).

Удаление состояний «на ходу» повышает эффективность, поскольку оно снижает затраты памяти. Применение `.*` оставляет по одному состоянию на каждый символ, что при большой длине строки может приводить к заметным затратам памяти.

Эквивалентность малых квантификаторов

Одни программисты предпочитают запись `\d\d\d\d`, другие используют малые квантификаторы и выбирают конструкцию `\d{4}`. Отличаются ли эти конструкции по эффективности? В НКА ответ почти наверняка будет положительным, но какой из вариантов работает быстрее? Ответ на этот вопрос зависит от программы. Если квантификаторы в программе оптимизированы, то версия `\d{4}` с большой долей вероятности будет работать быстрее, если только не окажется, что версия без квантификатора по какой-либо причине была оптимизирована еще сильнее. Звучит несколько запутанно? Так оно и есть.

АВТОМАТИЧЕСКИЙ «ЗАХВАТ»

Вспомните пример из главы 4 (☞ 225), где выражение ⌜^\w+:⌟ применялось к тексту «Subject». После того как ⌜\w+⌟ распространяется до конца строки, двоеточие совпасть уже не может, и механизму регулярных выражений приходится тратить усилия на поиск ⌜:⌟ во всех позициях, пока возврат заставляет ⌜\w+⌟ отдавать символы. Затем был сделан вывод, что для избежания лишней работы можно было воспользоваться атомарной группировкой ⌜^(?>\w+):⌟ или захватывающими квантификаторами ⌜^\w++:⌟.

Умная реализация сделает это за вас. При первой компиляции регулярного выражения механизм видит, что элементы, *следующие за квантификатором*, не могут совпасть с *квантифицируемым* текстом, поэтому квантификатор автоматически преобразуется в захватывающий.

Хотя я не знаю ни одной системы, в которой эта оптимизация была бы реализована в настоящее время, разработчикам определенно стоит обратить на нее внимание. Полагаю, она может иметь значительный положительный эффект.

Мои тесты показывают, что в Perl, Python, PHP/PCRE и .NET выражение ⌜\d{4}⌟ работает примерно на 20 % быстрее. С другой стороны, в Ruby и в пакете регулярных выражений для Java от Sun ⌜\d\d\d\d⌟ работает быстрее (иногда в несколько раз). Из сказанного можно сделать вывод, что в одних случаях малый квантификатор подходит лучше, в других — хуже. Но и это еще не все!

Сравните ⌜====⌟ с ⌜={4}⌟. Этот пример отличается от предыдущего, поскольку в нем квантифицируется простой литерал, а также потому, что ⌜====⌟ позволит механизму регулярных выражений быстрее распознать литеральную подстроку. В этом случае может быть задействована чрезвычайно эффективная оптимизация *исключения по первому символу/подстроке* (☞ 315), если она поддерживается. Именно так обстоит дело в Python и в пакете регулярных выражений для Java от Sun, где версия ⌜====⌟ превосходит по скорости версию ⌜={4}⌟ в 100 и более раз!

С другой стороны, Perl, Ruby и .NET распознают возможность этой оптимизации как в ⌜====⌟, так и в ⌜={4}⌟, поэтому обе версии работают с одинаковой быстротой (причем в обоих случаях в сотни и тысячи раз быстрее аналогов ⌜\d\d\d\d⌟ и ⌜\d{4}⌟).

Учет необходимых элементов

Один из простых видов оптимизации заключается в следующем: если механизм понимает, что некоторый аспект результата совпадения в дальнейшем не используется (например, функция сохранения сохраняющих круглых скобок), он может исключить работу по их поддержке. Возможность обнаружения таких ситуаций во многом зависит от специфики языка, но зато подобная оптимизация легко

реализуется при помощи дополнительных параметров, позволяющих обойти затратные операции.

Среди примеров систем, в которых поддерживается такая оптимизация, следует особо упомянуть Tcl. Сохраняющие круглые скобки в Tcl ничего не сохраняют, если этого специально не потребовать. И наоборот, в регулярных выражениях .NET Framework имеется ключ, при помощи которого программист может отменить сохранение.

Приемы построения быстрых выражений

В предыдущем разделе были описаны некоторые оптимизации, встречавшиеся мне в традиционных механизмах НКА. Ни в одной программе не реализован весь набор, и какую бы программу вы ни выбрали для себя, в будущем состав ее оптимизаций наверняка изменится. Но хорошее понимание оптимизаций поможет вам создавать более эффективные выражения. В сочетании с хорошим знанием принципов работы традиционного механизма НКА эта информация может применяться по трем направлениям.

- **Формулировка выражений с учетом возможных оптимизаций.** Выражение записывается таким образом, чтобы при его обработке использовались уже существующие оптимизации (или те, которые могут быть реализованы в будущем). Например, замена ⌜`x+`⌟ на ⌜`xx*`⌟ обеспечивает возможность применения многочисленных оптимизаций, в том числе проверки обязательного символа или строки (☞ 313) и исключения по первому символу (☞ 313).

- **Имитация оптимизаций.** Иногда вы знаете, что некая оптимизация не поддерживается программой, но можете самостоятельно имитировать оптимизацию и добиться большого выигрыша. Приведу лишь один пример, который будет рассмотрен более подробно: включение конструкции ⌜`(?=t)`⌟ в начало выражения ⌜`this|that`⌟ в какой-то степени имитирует исключение по первому символу (☞ 313) в системах, которые не могут самостоятельно определить, что любое совпадение должно начинаться с символа ‘`t`’.

- **Ускоренное достижение совпадения.** Руководствуясь знанием принципов работы традиционного механизма НКА, можно привести механизм к совпадению по ускоренному пути. Рассмотрим пример ⌜`this|that`⌟. Каждая альтернатива начинается с ⌜`th`⌟, если у первой альтернативы не найдется совпадение для ⌜`th`⌟, то ⌜`th`⌟ второй альтернативы тоже заведомо не совпадет, поэтому такая попытка заведомо завершится неудачей. Чтобы время не тратилось даром, можно сформулировать то же выражение в виде ⌜`th(?:is|at)`⌟. В этом случае ⌜`th`⌟ проверяется всего один раз, а относительно затратная конструкция выбора откладывается до момента, когда она становится действительно необходимой. Кроме того,

в выражении ⌜th(?:is|at)⌟ проявляется начальный литерал ⌜th⌟, что позволяет задействовать ряд других оптимизаций.

Вы должны хорошо понимать, что эффективность и оптимизации — дело деликатное. При знакомстве с материалом оставшейся части этого раздела помните о следующих обстоятельствах.

- Даже однозначно полезные изменения иногда замедляют работу программы, потому что они мешают применению других оптимизаций, о которых вы не подозревали.
- При имитации оптимизаций может оказаться, что затраченная на обработку имитации работа больше отнимает время, чем экономит.
- Имитация оптимизаций, отсутствующих в настоящий момент, может помешать применению полноценных оптимизаций, поддержка которых добавится в следующей версии программы (или продублирует их).
- Подгонка выражения под оптимизацию одного типа может помешать применению других, более выгодных оптимизаций, поддержка которых добавится в следующей версии программы.
- Преобразование выражения для достижения максимальной эффективности может усложнить выражение, затруднить его понимание и сопровождение.
- Выигрыш (или потеря) от конкретного изменения почти всегда сильно зависит от данных, с которыми работает выражение. Изменение, благоприятное для одного набора данных, может оказаться вредным для других данных.

Приведу безумный пример: в конструкции ⌜(000|999)$⌟ сценария Perl вы решаете заменить сохраняющие круглые скобки несохраняющими. По идее, это должно несколько ускорить работу сценария за счет затрат времени на сохранение. Но вас ожидает сюрприз — маленькое и вроде бы однозначно полезное изменение замедляет обработку выражения *на несколько порядков* (в десятки тысяч раз). *Что?!* Оказывается, применение несохраняющих скобок не позволяет использовать оптимизацию привязки к *концу текста/логических строк* (☞ 313). Я не собираюсь отговаривать вас от применения несохраняющих скобок в Perl — в подавляющем большинстве случаев они приносят пользу, но в данной конкретной ситуации приводят к катастрофическим последствиям.

Итак, тестирование и хронометраж на примере характерных данных помогут определить, насколько благотворным или вредным окажется то или иное изменение, но вам по-прежнему придется самостоятельно оценивать все факторы. После всего сказанного я представлю некоторые приемы, которые помогут выжать из механизма регулярных выражений всю возможную эффективность до последней капли.

Приемы, основанные на здравом смысле

Многие из полезных приемов оптимизации не требуют ничего, кроме здравого смысла.

Избегайте повторной компиляции

Старайтесь как можно реже компилировать или определять регулярные выражения. При объектно-ориентированном интерфейсе (☞ 132) программист в полной мере управляет компиляцией. Например, если регулярное выражение должно применяться в цикле, создайте объект выражения *за пределами* цикла и затем многократно *используйте* его внутри цикла.

При процедурном интерфейсе (например, в GNU Emacs и Tcl) постарайтесь, чтобы количество регулярных выражений, используемых в цикле, не превышало размер кэша соответствующей программы (☞ 311).

При интегрированном интерфейсе (например, в Perl) постарайтесь избежать интерполяции переменных в регулярных выражениях внутри цикла, поскольку в лучшем случае регулярное выражение будет заново оцениваться при каждой итерации, даже если оно заведомо не изменяется. Впрочем, в Perl предусмотрены эффективные средства для решения этой проблемы (☞ 436).

Используйте несохраняющие круглые скобки

Если вы не используете текст, совпадающий с подвыражениями в круглых скобках, используйте несохраняющие скобки ⌜`(?:…)`⌟ (☞ 75). Помимо прямого выигрыша, из-за отсутствия затрат на сохранение появляется побочная экономия — состояния, используемые при возврате, становятся менее сложными и поэтому быстрее восстанавливаются. Также открывается возможность для дополнительных оптимизаций, таких как исключение лишних скобок (☞ 315).

Исключите из выражения лишние круглые скобки

Используйте круглые скобки по мере надобности, но помните: их присутствие может предотвратить применение некоторых оптимизаций. Если вам не нужно знать последний символ, совпавший с ⌜`.*`⌟, не используйте ⌜`(.)*`⌟. На первый взгляд такая рекомендация выглядит очевидной, но не забывайте — мы рассматриваем приемы, основанные на здравом смысле!

Исключите из выражения лишние символьные классы

Следующая рекомендация тоже вполне тривиальна, но я неоднократно видел, как новички используют выражения вида ⌜`^.*[:]`⌟. Не представляю, зачем определять символьный класс из одного символа, — это добавляет затраты на обработку класса,

но не дает абсолютно никаких преимуществ, присущих многосимвольным классам. Возможно, при поиске символов с особой интерпретацией в конструкциях вида ⌜[.]⌟ и ⌜[*]⌟ новички просто не знают о возможности экранирования (⌜\.⌟ и ⌜*⌟). Чаще всего подобные перлы встречаются в режиме свободного форматирования (☞ 152).

Встречается и другая разновидность этой проблемы: пользователи Perl, знакомые с первым изданием этой книги, иногда используют выражения вида ⌜^[Ff][Rr][Oo][Mm]:⌟ вместо ⌜^from⌟ в режиме поиска без учета регистра. В старых версиях Perl поиск без учета регистра был реализован крайне неэффективно, поэтому в некоторых случаях я рекомендовал использовать символьные классы. В настоящее время эта рекомендация стала неактуальной, поскольку проблема с эффективностью поиска была решена несколько лет назад.

Используйте привязку к началу строки

Выражение, начинающееся с ⌜.*⌟, практически всегда должно начинаться с ⌜^ ⌟ или с ⌜\A⌟ (☞ 175). Если такое выражение не совпадет от начала строки, вряд ли поиск даст лучшие результаты после смещения ко второму, третьему символу и т. д. Добавление якорных метасимволов (явное или косвенное в результате оптимизации ☞ 313) позволяет задействовать стандартную оптимизацию привязки к началу строки и сэкономить массу времени.

Выделение литерального текста

Многие стандартные оптимизации, о которых говорилось в этой главе, основаны на простом факте: механизм регулярных выражений осознает, что некоторый интервал литерального текста должен входить в любое успешное совпадение. Одни механизмы понимают это лучше, чем другие, поэтому существует особая категория приемов ручной оптимизации, помогающих «выделить» литеральный текст в выражении, повысить вероятность того, что механизм сможет опознать больше текста и применить соответствующие оптимизации.

Выделение обязательных компонентов в квантификаторах

Использование ⌜xx*⌟ вместо ⌜x+⌟ подчеркивает обязательное присутствие хотя бы одного экземпляра ‘x’. По аналогичным соображениям ⌜-{5,7}⌟ записывается в виде ⌜------{0,2}⌟.

Выделение обязательных компонентов в начале конструкции выбора

Выражение ⌜th(?:is|at)⌟ вместо ⌜(?:this|that)⌟ указывает на то, что элемент ⌜th⌟ является обязательным. Литеральный текст может выделяться и в правой части,

когда общий текст следует за вариативным: ⌜(?:optim|standard)ization⌟. Как объясняется в следующем разделе, подобные приемы особенно эффективны в том случае, если выделяемая часть содержит якорные метасимволы.

Выделение якорей

К числу наиболее эффективных внутренних оптимизаций относятся те, в которых используются якорные метасимволы, привязывающие выражение к одному из концов целевой строки (в том числе ⌜^⌟, ⌜$⌟ и ⌜\G⌟). Одни механизмы лучше распознают возможность применения такой оптимизации, другие — хуже. Рекомендации, приведенные ниже, помогут вам направить такие механизмы на правильный путь.

Выделяйте ^ и \G в начале выражений

Выражения ⌜^(?:abc|123)⌟ и ⌜^abc|^123⌟ логически эквивалентны, но оптимизация *привязки к началу текста/логической строки* (☞ 312) значительно чаще применяется в первом случае. Следовательно, в общем случае первый вариант формулировки выражения превосходит второй по эффективности. PCRE (и те программы, в которых он используется) обрабатывает оба выражения с одинаковой эффективностью, но большинство других программ на базе НКА распознают возможность оптимизации только в версии с выделенным якорем.

Еще одно отличие можно заметить при сравнении ⌜(^abc)⌟ и ⌜^(abc)⌟. Первое выражение лишено возможности применения многих оптимизаций, так как оно «прячет» якорный элемент и вынуждает механизм регулярных выражений входить в сохраняющие круглые скобки до того, как появится возможность проверить соответствие якорному элементу. В некоторых системах это может приводить к существенному снижению эффективности. В одних системах (таких, как PCRE, Perl и языки .NET) снижения эффективности не наблюдается, но в других (таких, как Ruby или пакет регулярных выражений от Sun в Java) возможность оптимизации определяется только при наличии выделенных якорных элементов.

Похоже, в языке Python оптимизация привязки отсутствует, поэтому этот прием к нему не относится. И конечно, большая часть оптимизаций этой главы не распространяется на Tcl (☞ 309).

Выделяйте $ в конце выражения

На концептуальном уровне эта оптимизация очень близка к предыдущей. Выражения ⌜abc$|123$⌟ и ⌜(?:abc|123)$⌟ логически эквивалентны, но механизмы регулярных выражений могут обрабатывать их по-разному. В настоящее время различия

проявляются только в Perl, поскольку лишь в этом языке сейчас поддерживается оптимизация *привязки к концу текста/логической строки* (☞ 313). Оптимизация проявляется в выражениях вида ⌜(…|…)$⌟, но не ⌜(…$|…$)⌟.

Выбор между минимальными и максимальными квантификаторами

Как правило, выбор между минимальными и максимальными квантификаторами определяется спецификой регулярного выражения. Скажем, выражение ⌜^.*:⌟ принципиально отличается от ⌜^.*?:⌟; в первом случае совпадение распространяется до последнего двоеточия, а во втором — только до первого. А если вы точно знаете, что в целевых данных присутствует только одно двоеточие? В этом случае семантика обоих выражений одинакова («совпадение до двоеточия»), поэтому будет разумно выбрать то выражение, которое быстрее работает.

Правильный выбор не всегда очевиден. При большой длине целевого текста, если вы ожидаете найти двоеточие где-то в начале, используйте минимальный квантификатор — он поможет быстрее найти совпадение. Если двоеточие должно находиться где-то в конце, выбирайте максимальный квантификатор. Если вам абсолютно ничего не известно о целевых данных, выбирайте максимальный квантификатор, поскольку он обычно оптимизируется чуть лучше минимального, особенно если следующий элемент регулярного выражения не позволяет использовать оптимизацию *по символу, следующему за минимальным квантификатором* (☞ 315).

При малой длине целевого текста выбирать еще сложнее. В общем-то, оба варианта работают достаточно быстро, но если вам нужна вся возможная скорость, проведите хронометраж по характерным данным.

Похожая проблема возникает в ситуации, когда могут использоваться и минимальный квантификатор, и инвертированный класс (например, ⌜^.*?:⌟ или ⌜^[^:]*:⌟). Какому из двух вариантов отдать предпочтение? Выбор также зависит от данных и языка, но в большинстве случаев инвертированный класс работает гораздо эффективнее минимального квантификатора. В настоящее время существует единственное исключение — Perl, в котором реализована оптимизация *по символу, следующему за минимальным квантификатором*.

Разделение регулярных выражений

Иногда бывает быстрее применить несколько маленьких выражений вместо одного большого. Рассмотрим несколько неестественный пример: если вы хотите прове-

рить большую строку и узнать, содержатся ли в ней названия месяцев, вероятно, отдельные проверки для ⌜`January`⌟, ⌜`February`⌟, ⌜`March`⌟ и т. д. будут работать гораздо быстрее, чем комбинированное выражение ⌜`January|February|March|…`⌟. Во втором варианте отсутствует литеральный текст, обязательный для каждого совпадения, поэтому оптимизация *проверки внутренних литералов* (☞ 314) в этом случае неприменима. Учитывая, что все проверки осуществляются в одной конструкции выбора, проверка каждого подвыражения от каждой позиции текста выполняется относительно медленно.

Во время работы над этим разделом я столкнулся с одной любопытной ситуацией. Используя модуль Perl, предназначенный для обработки данных, я обнаружил, что в некоторых ситуациях клиентская программа вместо нормальных данных передавала фиктивную информацию вида '`HASH(0x80f60ac)`'. Я решил усовершенствовать модуль, чтобы он обнаруживал фиктивные данные и сообщал об ошибке. Прямолинейное решение, которое находило нужный текст, выглядело так:

⌜`\b(?:SCALAR|ARRAY|…|HASH)\(0x[0-9a-fA-F]+\)`⌟.

Это была одна из тех ситуаций, в которых особенно важна эффективность решения. Будет ли оно достаточно быстрым? В Perl существует отладочный режим, который позволяет получить информацию о некоторых оптимизациях, применяемых к регулярному выражению (☞ 451), и я им воспользовался. Меня интересовало, задействована ли *предварительная проверка обязательных строк* (☞ 313), поскольку достаточно умный механизм мог определить, что подстрока '`(0x`' присутствует в любом совпадении. Я знал, что в нормальных данных эта подстрока встречается крайне редко, поэтому предварительная проверка отклонит практически все строки. К сожалению, Perl эту оптимизацию не распознал, поэтому мое регулярное выражение искало совпадение для нескольких альтернатив на каждом символе целевой строки. Такой вариант работал медленнее, чем я рассчитывал.

В то время я как раз занимался исследованиями и работал над главой, посвященной оптимизации. Оставалось придумать, в каком виде нужно записать регулярное выражение, чтобы оно лучше оптимизировалось. Первое, что пришло мне в голову, — ⌜`\(0x(?<=(?:SCALAR|…|HASH)\(0x)[0-9a-fA-F]+\)`⌟ (довольно сложная конструкция). Если для ⌜`\(0x`⌟ находится совпадение, позитивная ретроспективная проверка (подчеркнута для наглядности) сначала проверяет предшествующий текст, а затем — дальнейший текст. Вся эта эквилибристика нужна для того, чтобы выделить обязательный литеральный текст ⌜`\(0x`⌟, обеспечивающий хороший потенциал для оптимизации. В частности, должна быть задействована оптимизация *предварительной проверки обязательной строки*, а также *исключения по первому символу/подстроке* (☞ 313). Несомненно, эти оптимизации сильно ускорили бы работу выражения, но Perl не поддерживает ретроспективу переменной длины (☞ 181), поэтому все усилия привели в тупик.

Тем не менее я осознал, что если Perl не выполняет предварительную проверку `\(0x` за меня, я могу сделать это сам:

```
if ($data =~ m/\(0x/
    and
    $data =~ m/(?: SCALAR|ARRAY|…|HASH)\(0x[0-9a-fA-F]+\)/)
{
    # Предупредить о недействительных данных…
}
```

Проверка `\(0x` исключала практически все строки исходного текста, поэтому относительно медленное полное регулярное выражение применялось только в том случае, когда вероятность совпадения была достаточно большой. Тем самым достигался замечательный баланс между эффективностью (очень высокой) и наглядностью (тоже очень высокой)[1].

Имитация исключения по первому символу

Если оптимизация *исключения по первому символу* (☞ 313) не поддерживается вашей реализацией, вы можете имитировать ее за счет включения в начало выражения соответствующей опережающей проверки (☞ 129). Опережающая проверка позволяет убедиться в наличии нужного символа, прежде чем переходить к поиску совпадения для основного выражения.

Например, для выражения `Jan|Feb|…|Dec` можно воспользоваться опережающей проверкой `(?=[JFMASOND])(?:Jan|Feb|…|Dec)`. Начальное условие `[JFMASOND]` содержит первые буквы английских названий месяцев. Тем не менее при подобных проверках необходима осторожность, поскольку лишние затраты на опережающую проверку могут перевесить экономию. В нашем конкретном примере, где опережающая проверка способна заранее отклонить многие несовпадающие альтернативы, этот прием приносит пользу в большинстве систем, в которых он тестировался (Java, Perl, Python, Ruby, языки .NET). Похоже, ни одна из этих систем не смогла самостоятельно вывести проверку `[JFMASOND]` по выражению `Jan|Feb|…|Dec`.

В PHP/PCRE по умолчанию этот вид оптимизации не используется, однако она доступна в функции `pcre_study` из библиотеки PCRE (ghb использовании модификатора `S` ☞ 597). Разумеется, Tcl с этой задачей прекрасно справляется (☞ 309).

Конечно, незаметная проверка `[JFMASOND]`, используемая нормальной оптимизацией, работает быстрее, чем аналогичная проверка, включенная программистом в регулярное выражение. Можно ли модифицировать регулярное выражение так,

[1] При желании можете убедиться сами. Модуль, о котором идет речь, `DBIx::DWIW`, доступен в архиве CPAN. Этот модуль обеспечивает очень простой доступ к базе данных MySQL. Джереми Заводни (Jeremy Zawodny) и я создали его для Yahoo!.

чтобы в нем использовалась стандартная проверка? В большинстве систем можно воспользоваться невероятно запутанным выражением

```
⌜[JFMASOND](?:(?<=J)an|(?<=F)eb|…|(?<=D)ec)⌟
```

Вряд ли вы разберетесь в этом выражении с первого взгляда, но я рекомендую потратить немного времени и разобраться, что и как оно делает, — это полезное упражнение. Простой класс, с которого начинается выражение, в большинстве систем распознается оптимизацией *исключения по первому символу*, благодаря чему сам механизм может проверить ⌜`[JFMASOND]`⌟ с повышенной эффективностью. Если целевая строка содержит небольшое количество совпадающих символов, результат работает существенно быстрее оригинала ⌜`Jan|Feb|…|Dec`⌟ или нашего варианта с опережающей проверкой. Но при большом количестве совпадений для первого символа в целевом тексте суммарные затраты на дополнительные ретроспективные проверки способны сильно замедлить работу выражения. Не стоит и говорить, что регулярное выражение становится *гораздо* менее понятным. И все же его стоит проанализировать, это полезно и познавательно.

Не делайте это в Tcl!

Предыдущий пример убеждает в том, что ручная настройка регулярных выражений иногда только ухудшает дело. Во врезке на с. 309 упоминается о том, что скорость обработки регулярных выражений в Tcl в основном не зависит от формулировки, поэтому попытки ручной оптимизации обычно ни на что не влияют. В данном примере ручная оптимизация *влияет* на скорость, и еще как! При включении предварительной проверки ⌜`(?=[JFMASOND])`⌟ в моих тестах скорость обработки выражения в Tcl уменьшилась примерно в 100 раз.

Не делайте это в PHP!

Как уже упоминалось выше, такой подход не следует использовать в PHP, потому что в PHP существует возможность использовать параметр «study» (модификатор S). Более подробно об этом будет рассказываться в главе 10 на с. 597.

Использование атомарной группировки и захватывающих квантификаторов

Во многих случаях атомарная группировка (☞ 187) и захватывающие квантификаторы (☞ 190) сильно повышают скорость поиска, хотя они никак не отражаются на совпадающем тексте. Например, если выражение ⌜`^[^:]+:`⌟ не находит совпадения для двоеточия при первой попытке, оно заведомо не совпадет и после возврата ⌜`[^:]+`⌟, потому что символы, «уступаемые» в ходе возврата, по опреде-

лению не могут совпадать с двоеточием. Атомарная группировка ⌜`^(?>[^:]+):`⌟ и захватывающий квантификатор ⌜`^[^:]++:`⌟ уничтожают состояния, порожденные квантификатором `+` (или предотвращают их создание). Поскольку у механизма не остается сохраненных состояний, он не будет пытаться выполнять бесполезный возврат (как упоминается во врезке на с. 319, достаточно умный механизм регулярных выражений может применять подобную оптимизацию автоматически).

Тем не менее я должен подчеркнуть, что злоупотребление этими конструкциями может привести к случайному изменению совпадающего текста, поэтому ими необходимо пользоваться крайне осторожно. Например, их применение к подвыражению ⌜`^.*:`⌟ — скажем, ⌜`^(?>.*):`⌟ — приводит к катастрофе. Вся строка совпадает с ⌜`.*`⌟, в том числе и двоеточие, которое позднее понадобится для ⌜`:`⌟. Атомарная группировка отменяет возможность возврата, необходимую для поиска совпадения с ⌜`:`⌟, поэтому неудача гарантирована.

Руководство процессом поиска

Следующая концепция в немалой степени решает вопрос с построением более эффективных регулярных выражений в НКА. Речь идет о том, чтобы по возможности отложить «управляющие» проблемы на как можно более позднюю стадию. Пример уже встречался выше, когда мы заменяли ⌜`this|that`⌟ на ⌜`th(?:is|at)`⌟. В первом выражении конструкция выбора находится на верхнем уровне, а во втором она рассматривается лишь после того, как будет найдено совпадение для ⌜`th`⌟.

В следующем разделе, «Раскрутка цикла», будет описана более сложная форма этой методики. Здесь я упомяну лишь несколько простых приемов.

Ставьте наиболее вероятную альтернативу на первое место

В этой книге нам неоднократно встречались примеры ситуаций, когда порядок перечисления альтернатив в конструкции выбора оказывал огромное влияние на ход поиска (☞ 56, 232, 247, 278). В подобных ситуациях правильность совпадения важнее оптимизации, но в остальном, если порядок альтернатив не отражается на правильности, размещение наиболее вероятных альтернатив на первых местах позволяет улучшить эффективность поиска.

Например, при построении регулярного выражения для имени хоста (☞ 265) возникает естественное желание перечислить имена доменов в алфавитном порядке — ⌜`(?:aero|biz|com|coop|…)`⌟. Но некоторые из имен, находящихся в начале алфавита, появились недавно и не пользуются особой популярностью, так стоит ли тратить время и проверять их раньше других, если эта проверка с большой вероятностью завершится неудачей? Вероятно, при расположении более популярных доменов на первых местах выражение ⌜`(?:com|edu|org|net|…)`⌟ будет приходить к совпадению быстрее.

Разумеется, эта тема актуальна только для традиционного механизма НКА и только при *наличии* совпадения. В POSIX НКА и при неудаче проверяются все альтернативы, поэтому порядок их перечисления не важен.

Распределение общих элементов по альтернативам

Продолжим рассмотрение удобного примера. Давайте сравним два выражения: ⌜`(?:com|edu|…|[a-z][a-z])\b`⌟ и ⌜`com\b|edu\b|…\b|[a-z][a-z]\b`⌟. Во втором случае метасимвол ⌜`\b`⌟, следующий после конструкции выбора, распределяется по всем альтернативам. Потенциальный выигрыш от такого решения связан с тем, что совпавшая альтернатива, которая будет отвергнута из-за отсутствия совпадения для ⌜`\b`⌟ после конструкции выбора, во втором варианте отвергается чуть быстрее. Неудача распознается без затрат, связанных с выходом из конструкции выбора.

Возможно, это не лучший пример для демонстрации этой методики. Его достоинства проявляются только в специфической ситуации, когда альтернатива совпадает, а то, что следует после нее, — нет. Более удачный пример будет приведен ниже, при описании `$OTHER*` на с. 353.

Учтите, что эта оптимизация бывает рискованной

При применении ручных оптимизаций очень важно позаботиться о том, чтобы они не противоречили более выгодным внутренним оптимизациям. Например, если «распределяемое» подвыражение содержит литеральный текст, как при распределении символа ‘`:`’, преобразующего ⌜`(?:this|that):`⌟ в ⌜`this:|that:`⌟, возникает прямое противоречие с некоторыми принципами, изложенными в разделе «Выделение литерального текста» (☞ 323). При прочих равных условиях внутренние оптимизации обычно приносят больше пользы, поэтому будьте осторожны и не отказывайтесь от них в пользу ручной настройки.

Аналогичное предупреждение относится к распределению завершающего метасимвола ⌜`$`⌟ в системах, оптимизирующих выделенные якорные метасимволы конца строки (☞ 324). В таких системах выражение ⌜`(?:com|edu|…)$`⌟ работает гораздо быстрее, чем распределенный вариант ⌜`com$|edu$|…$`⌟ (из всех протестированных мной систем эта оптимизация поддерживалась только в Perl).

Раскрутка цикла

Какие бы внутренние оптимизации ни поддерживались системой, вероятно, наибольший выигрыш может принести хорошее понимание базовых принципов работы механизма регулярных выражений и формулировка выражений, ускоряющая поиск. От подробного описания базовых принципов мы переходим к методике,

которую я называю «раскруткой цикла». Данная методика хорошо ускоряет работу некоторых распространенных выражений. В частности, она позволяет прервать бесконечный поиск, описанный в начале главы (☞ 289), и быстро прийти к выводу об отсутствии совпадения. Кроме того, она ускоряет поиск совпадения, если оно существует.

Цикл, о котором идет речь, создается квантификатором * в выражении, построенном по шаблону ⌜(*альтернатива1**альтернатива2*\...)*⌟. Возможно, вы увидели, что наше выражение «бесконечного перебора», ⌜`"(\\.|[^\\"]+)*"`⌟, подходит под этот шаблон. Учитывая, что для сообщения о несовпадении приходится ждать целую вечность или около того, было бы неплохо как-нибудь ускорить поиск!

При реализации этой методики можно пойти по одному из двух путей.

1. Можно проанализировать, какие части ⌜`(\\.|[^\\"]+)*`⌟ фактически совпадают при поиске в разных текстах. После этого можно заново сконструировать эффективное выражение на основании шаблонов, выявленных в результате анализа. Я представляю себе происходящее так: большой шар, изображающий ⌜`(...)*`⌟, прокатывается по тексту. Элементы, находящиеся внутри `(...)`, «прилипают» к совпавшему тексту, оставляя за собой цепочку из совпавших подвыражений (как от грязного мяча, который катится по ковру).

2. В другом варианте используется высокоуровневый анализ той конструкции, для которой находится совпадение. После этого мы принимаем обоснованное допущение относительно вероятного целевого текста, что позволяет нам смоделировать то, что, по нашему мнению, является стандартной ситуацией. Располагая этими сведениями, можно сконструировать эффективное регулярное выражение.

Выражения, полученные в обоих случаях, будут идентичными. Я начну с первого способа, а затем покажу, как прийти к тому же результату с позиций высокоуровневого анализа.

Чтобы примеры были по возможности компактными и универсальными, я буду использовать обычные круглые скобки ⌜`(...)`⌟. Если в системе поддерживаются несохраняющие круглые скобки ⌜`(?:...)`⌟, возможно, они позволят немного повысить эффективность поиска. Позднее также будет рассмотрено применение атомарной группировки (☞ 187) и захватывающих квантификаторов (☞ 190).

Метод 1: построение регулярного выражения по результатам тестов

При работе с выражением ⌜`"(\\.|[^\\"]+)*"`⌟ стоит проанализировать некоторые примеры совпадений и выяснить, какие подвыражения использовались в процессе поиска общего совпадения. Например, в строке ‘`"hi"`’ фактически используется

только выражение `"[^\\"]+"`. Другими словами, в общем совпадении используется только начальная кавычка `"`, один экземпляр альтернативы `[^\\"]` и завершающая кавычка `"`. В тексте

```
"he said \"hi there\" and left"
```

фактически используется конструкция `"[^\\"]+\\.[^\\"]+ \\.[^\\"]+"`. В этих примерах, а также в табл. 6.2 я пометил выражения, чтобы шаблоны выглядели более наглядно. Нам хотелось бы построить специализированное регулярное выражение для каждой входной строки. Конечно, сделать это невозможно, однако мы можем выделить стандартные шаблоны и сконструировать более эффективное, но при этом достаточно общее регулярное выражение.

Пока давайте ограничимся первыми четырьмя примерами из табл. 6.2. В них подчеркнуты те части, которые обозначают «экранированный элемент, за которым следуют обычные символы». Ключевой момент: в каждом случае выражение между кавычками начинается с `[^\\"]+`, после чего следует некоторое количество повторений `\\.[^\\"]+`. Перефразируя сказанное на языке регулярных выражений, мы получаем `[^\\"]+(\\.[^\\"]+)*`. Перед вами частный случай общего шаблона, используемого при конструировании многих полезных выражений.

Таблица 6.2. Примеры раскрутки

Целевая строка	Фактическое выражение
`"hi there"`	`"[^\\"]+"`
`"just one \" here"`	`"[^\\"]+ \\.[^\\"]+"`
`"some \"quoted\" things"`	`"[^\\"]+ \\.[^\\"]+ \\ "[^\\"]+"`
`"with \"a\" and \"b\"."`	`"[^\\"]+ \\.[^\\"]+\\.[^\\"]+ \\.[^\\"]+\\.[^\\"]+"`
`"\"ok\"\n"`	`"\\.[^\\"]+\\. \\."`
`"empty \"\" quote"`	`"[^\\"]+\\. \\.[^\\"]+"`

Построение общего шаблона «раскрутки цикла»

При поиске строк, заключенных в кавычки, сама кавычка и обратный слэш являются «специальными» символами. Кавычка — потому что она может завершить строку. Обратный слэш — потому что он означает, что следующий символ не является завершением строки. Все остальные символы, `[^\\"]`, являются «нормальными». Если внимательнее присмотреться к структуре выражения `[^\\"]+(\\.[^\\"]+)*`, можно заметить, что оно соответствует общему шаблону `нормальный+(специальный нормальный+)*`.

Добавляя кавычки, начальную и завершающую, получаем `"[^\\"]+(\\.[^\\"]+)*"`. К сожалению, это выражение не совпадает с двумя последними примерами

в табл. 6.2. Проблема состоит в том, что два ⌜[^\\"]⌟ в этом выражении *требуют* присутствия нормального символа в начале строки и после любого специального символа. Как видно из примеров, это не всегда возможно — строка может начинаться или заканчиваться экранированным символом или в ней могут идти два экранированных символа подряд.

Можно попытаться заменить плюсы звездочками: ⌜"[^\\"]*(\\.[^\\"]*)*"⌟. Приведет ли это к желаемому результату? И что еще важнее, не возникнут ли при этом нежелательные побочные эффекты?

Что касается положительных эффектов, видно, что все примеры теперь совпадают. Более того, совпадает даже строка "\"\"\"". Это хорошо, но при внесении столь принципиальных изменений нужно быть абсолютно уверенным в том, что это не вызовет отрицательных последствий. Не совпадет ли выражение с чем-нибудь, кроме допустимой строки, заключенной в кавычки? Может ли допустимая строка в кавычках не совпасть? И как насчет эффективности?

Присмотритесь к ⌜"[^\\"]*(\\.[^\\"]*)*"⌟ повнимательнее. Начальное подвыражение ⌜"[^\\"]*⌟ совпадает всего один раз и выглядит безвредным; оно соответствует обязательной начальной кавычке и всем нормальным символам, следующим непосредственно за ней. Никакой опасности в этом нет. Следующее подвыражение ⌜(\\.[^\\"]*)*"⌟ заключено в **(…)***, поэтому оно может совпасть ноль раз. Это означает, что при его исключении должно остаться правильное выражение. В самом деле, исключив его, мы получаем выражение ⌜"[^\\"]*"⌟, отражающее ситуацию, при которой строка не содержит ни одного экранированного элемента.

С другой стороны, если ⌜(\\.[^\\"]*)*⌟ совпадает один раз, мы фактически приходим к выражению ⌜"[^\\"]*(\\.[^\\"]*"⌟. Даже если завершающее подвыражение ⌜[^\\"]*⌟ не совпадает ни с чем (при этом выражение фактически превращается в ⌜"[^\\"]*\\."⌟), проблем не возникает. Продолжая аналогичные рассуждения (насколько я помню школьный курс логики, это называется «индукцией»), мы приходим к выводу, что предложенные изменения действительно не вызывают никаких проблем.

Итак, мы получаем окончательный вариант выражения для поиска строк в кавычках:

⌜"[^\\"]*(\\.[^\\"]*)*"⌟

Общий шаблон «раскрутки цикла»

Объединяя все сказанное, мы приходим к окончательному варианту выражения для поиска строк в кавычках: ⌜"[^\\"]*(\\.[^\\"]*)*"⌟. Это выражение совпадает точно с теми же строками, что и старое выражение с конструкцией выбора, и не

находит совпадения в тех же строках, где их не находит старое выражение. Но «раскрученная» версия обладает дополнительным преимуществом: она завершит свою работу еще при вашей жизни, потому что работает гораздо эффективнее и избегает проблемы «бесконечного перебора».

Обобщенный шаблон подобных выражений выглядит так:

⌜*начало норм** (*спец норм**)* *конец*⌟

Как избежать бесконечного перебора

Бесконечный перебор в выражении ⌜`"[^\\"]*(\\.[^\\"]*)*"`⌟ предотвращается тремя правилами, имеющими чрезвычайно большое значение:

1. **Начала спец и норм ни в коем случае не должны пересекаться.**

Во-первых, подвыражения *спец* и *норм* должны быть написаны так, чтобы они никогда не совпадали в одной начальной позиции. В нашем примере, где *норм* соответствует выражению ⌜`[^\\"]`⌟, а *спец* — выражению ⌜`\\.`⌟, это требование выполняется автоматически — второе выражение требует начального символа \, а в первом этот символ явно запрещен.

С другой стороны, ⌜`\\.`⌟ и ⌜`[^"]`⌟ могут совпадать, начиная с позиции '`"Hello␣\n"`', поэтому они не подходят для пары *спец* и *норм*. Если в какой-то ситуации они могут совпасть, начиная с одной позиции, будет непонятно, какое из подвыражений должно использоваться в этом случае, и неопределенность приведет к бесконечному перебору. В примере '`makudonarudo`' (☞ 291) это продемонстрировано наглядно. При отсутствии совпадения (или при любой попытке поиска в механизме POSIX НКА) механизму придется проверить все возможные комбинации элементов. Допустить этого никак нельзя, поскольку выражение строилось заново как раз для того, чтобы избежать подобного перебора.

Если вы проследите за тем, чтобы *спец* и *норм* никогда не совпадали в одной позиции, подвыражение *спец* будет использоваться для разрешения ситуаций, когда несколько экземпляров *норм* в разных итерациях цикла ⌜`(…)*`⌟ могут совпасть с одним и тем же текстом. Если *спец* и *норм* никогда не совпадают в одной позиции, всегда будет существовать ровно одна возможная «последовательность» подвыражений *спец* и *норм*, которая может совпасть с конкретной строкой. Одна последовательность проверяется гораздо быстрее, чем 100 миллионов последовательностей; таким образом, нам благополучно удается избежать бесконечного перебора.

2. **Выражение *спец* не может совпадать с «ничем».**

Второе важное правило состоит в том, что выражение *спец* всегда должно совпадать хотя бы с одним символом (если оно вообще с чем-нибудь совпадает). Если

это выражение может совпадать без поглощения символов, соседние нормальные символы смогут разделяться разным количеством итераций ⌜(*спец* *норм**)*⌟, и мы вернемся к прежней проблеме (…*)*.

Например, выбор в качестве *спец* выражения ⌜(\\.)*⌟ нарушает это правило. При попытке найти злосчастное выражение ⌜"[^\\"]* (*(\\.)**[^\\"]**)*"⌟ в строке '"Tubby' (при отсутствии совпадения) механизм должен проверить все комбинации совпадения нескольких экземпляров ⌜[^\\"]*⌟ в 'Tubby' и только после этого прийти к выводу о неудаче. Если выражение *спец* может совпадать с пустой строкой, оно утрачивает свои функции «устранения неоднозначности».

3. **Выражение *спец* должно быть атомарным.**

Текст, совпавший в результате одного применения *спец*, не может совпасть в результате нескольких применений *спец*. Рассмотрим задачу поиска последовательности, состоящей из необязательных комментариев Pascal {…} и пробелов. Регулярное выражение для поиска комментариев имеет вид ⌜\{[^}]*\}⌟, поэтому при бесконечном переборе все выражение выглядит так: ⌜(\{[^}]*\}|•+)*⌟. На первый взгляд хочется выбрать следующие выражения *спец* и *норм*:

спец	*норм*
⌜•+⌟	⌜\{[^}]*\}⌟

Подставляя эти выражения в построенный ранее шаблон ⌜*норм**(*спец* *норм**)*⌟, мы получаем: ⌜(\{[^}]*\})*(•+(\{[^}]*\})*)*⌟. А теперь рассмотрим строку:

```
{comment}•••{another}••
```

Последовательность из нескольких пробелов может совпадать с одним подвыражением ⌜•+⌟, с несколькими подвыражениями ⌜•+⌟ (каждое из которых совпадает с одним пробелом) или с различными комбинациями ⌜•+⌟, совпадающими с разными количествами пробелов. Наблюдается прямая аналогия с нашим примером 'makudonarudo'.

Корень проблемы заключается в том, что *спец* может совпасть с меньшим фрагментом текста *внутри* большего фрагмента, с которым это выражение тоже может совпасть, причем из-за (…)* это может происходить неоднократно. Подобная неопределенность порождает уже знакомую проблему «разных вариантов совпадения одного текста».

Если общее совпадение существует, вероятно, все пробелы будут отнесены к внутренней конструкции ⌜•+⌟, но при отсутствии совпадения (например, если выражение используется внутри большего регулярного выражения, для которого

поиск может оказаться неудачным) механизм должен рассмотреть все комбинации совпадения ⌜(•+)*⌟ в серии из нескольких пробелов. На это тратится время, но без малейшей надежды на совпадение. Поскольку подвыражение *спец* само должно устранять неопределенность, нет ничего, что позволило бы избавиться от неопределенности внутри этого выражения.

Решение проблемы: позаботиться о том, чтобы выражение *спец* совпадало только с фиксированным количеством пробелов. Поскольку оно должно совпадать по крайней мере один раз (но может и больше), мы просто выбираем ⌜•⌟ и разрешаем совпадение нескольких экземпляров *спец* с несколькими пробелами через внешний квантификатор ⌜(…)*⌟.

Приведенный пример хорошо подходит для анализа, но если бы мне действительно потребовалось такое выражение, я бы переставил *спец* и *норм*:

⌜•*(\{[^}]*\}•*)*⌟

Это связано с тем, что программа на языке Pascal содержит больше пробелов, чем комментариев, а выражение *норм* должно относиться к более распространенному случаю.

Общие признаки неэффективных выражений

После того как вы усвоите эти правила (возможно, для этого придется несколько раз перечитать их и немного поэкспериментировать), вы сможете выделить общие признаки регулярных выражений, подверженных «бесконечному перебору». Многоуровневые квантификаторы (такие, как ⌜(…*)*⌟) часто предупреждают об опасности, но они встречаются и во многих вполне допустимых выражениях:

- ⌜(Re:•*)*⌟ — выделение цепочек префиксов ‘Re:’ произвольной длины (например, выражение может использоваться для «очистки» строки темы ‘Subject:•Re:•Re:•Re:•hey’).
- ⌜(•*\$[0-9]+)*⌟ — поиск денежных сумм в долларах (возможно, разделенных пробелами).
- ⌜(.*\n)+⌟ — поиск одной или нескольких логических строк. Обратите внимание: если точка может совпадать с символом новой строки, а после этого подвыражения следует нечто, приводящее к неудаче, снова возникает ситуация «бесконечного перебора».

Все эти выражения вполне допустимы, поскольку в каждом из них присутствует «маркер», предотвращающий опасную ситуацию «разных вариантов совпадения одного текста». В первом примере это ⌜Re:⌟, во втором — ⌜\$⌟, а в третьем (если точка не совпадает с символом новой строки) — ⌜\n⌟.

Метод 2: структурный анализ

Как я говорил выше, к одному и тому же выражению можно прийти двумя путями. Второй путь мы начнем с рассмотрения типичных случаев, а затем включим в анализ то, что необходимо для обработки особых ситуаций. Давайте попробуем разобраться, чего добивается выражение ⌜(\\.|[^\\"]+)*⌟ и в какой ситуации оно будет в основном использоваться. Вероятно, строка в кавычках в основном состоит из обычных, а не экранированных символов, поэтому основная работа достанется подвыражению ⌜[^\\"]+⌟. Подвыражение ⌜\\.⌟ необходимо лишь для того, чтобы разобраться с редкими экранированными символами. Конструкция выбора учитывает оба случая и позволяет использовать это выражение на практике, но все же не хочется снижать эффективность всего поиска ради редких (а то и вовсе отсутствующих) экранированных символов.

Если мы полагаем, что ⌜[^\\"]+⌟ обычно совпадает с большей частью символов в строке, то после его совпадения должна следовать либо кавычка, либо обратный слэш. Если это обратный слэш, мы добавляем еще один символ (каким бы он ни был) и находим новую порцию основного текста ⌜[^\\"]+⌟. Каждый раз, когда совпадение ⌜[^\\"]+⌟ завершается, мы оказываемся в той же ситуации — следующим символом является либо завершающая кавычка, либо очередной обратный слэш.

Выражая все сказанное на языке регулярных выражений, мы приходим к тому же, что уже встречалось нам в методе 1:

⌜"[^\\"]+(▲\\.[^\\"]+)*"⌟

Каждый раз, когда поиск достигает позиции, обозначенной ▲, мы знаем, что следующим символом будет либо обратный слэш, либо кавычка. Если обратный слэш совпадает, мы берем его, следующий символ и текст до следующей точки перед кавычкой или обратным слэшем.

Как и в предыдущем методе, необходимо предусмотреть ситуации, при которых начальный сегмент или сегменты между кавычками пусты. Для этого плюсы заменяются звездочками, и мы приходим к уже знакомому выражению.

Метод 3: имена хостов Интернета

Я обещал описать два способа построения шаблона *«раскрутки цикла»*, но в дополнение к ним я представлю похожий метод, который можно считать третьим. Я столкнулся с ним во время работы над регулярным выражением для поиска имен хостов (таких, как `www.yahoo.com`), которые, в сущности, представляют собой списки имен доменов нижних уровней, разделенных точками. Точное описание имени домена нижнего уровня — весьма непростая задача (☞ 265), поэтому в данном

примере для его идентификации будет использоваться простое (хотя и неполное) выражение ⌜[a-z]+⌟.

Если домен нижнего уровня определяется выражением ⌜[a-z]+⌟ и мы хотим найти список доменов нижнего уровня, разделенных точками, сначала необходимо найти какое-то одно имя домена нижнего уровня. После него может следовать перечень необязательных имен других доменов нижних уровней, начинающихся с точки. Если выразить все сказанное на языке регулярных выражений, мы получим ⌜[a-z]+(\.[a-z]+)*⌟. А если записать это выражение в виде ⌜[a-z]+(\.[a-z]+)*⌟, оно становится очень знакомым!

Чтобы продемонстрировать сходство, мы проведем аналогию с выражением для поиска строк в кавычках. Если рассматривать строку как последовательность *нормальных* символов ⌜[^\\"]⌟, разделенных *специальными* символами ⌜\\.⌟, то все, что находится внутри '"..."', можно подставить в шаблон «раскрутки цикла». Получается ⌜"[^\\"]+(\\.[^\\"]+)*"⌟ — в точности то же самое, что мы получили на определенной стадии при обсуждении метода 1. На концептуальном уровне можно воспользоваться представлением имени хоста как последовательности символов, разделенной специальными символами, и применить его к строкам в кавычках; в результате строка в кавычках представляется как «последовательность неэкранированных символов, разделенных экранированными символами». Подобная интерпретация выглядит несколько неестественно, но она открывает новый интересный путь к уже знакомому результату.

Сходство интересно, но интересны и различия. В методе 1 регулярное выражение изменялось таким образом, чтобы оно допускало пустые экземпляры *норм* до и после каждого экземпляра *спец*, но в данном случае это нежелательно, поскольку имя домена нижнего уровня не может быть пустым. Итак, даже при том, что этот пример не является прямым аналогом предыдущих примеров, он принадлежит к тому же классу и демонстрирует мощь *и* гибкость методики раскрутки цикла.

Между этими двумя примерами (строки в кавычках и доменные имена) существуют два принципиальных отличия.

- Доменные имена не заключаются во внешние ограничители.
- Подвыражение *норм* в доменном имени, т. е. имя домена нижнего уровня, никогда не бывает пустым (иначе говоря, точки не могут стоять подряд и не могут начинать или заканчивать совпадение). В строке, заключенной в кавычки, наличие хотя бы одного совпадения *норм* вообще не требуется, хотя такие совпадения весьма вероятны, учитывая наши предположения относительно данных. Вот почему мы заменили ⌜[^\\"]+⌟ на ⌜[^\\"]*⌟. В примере с доменами нижних уровней это невозможно, поскольку в *спец* должен быть представлен обязательный разделитель.

Замечания

Подведем итоги. Можно заметить, что наше выражение ⌜"[^\\"]*(\\.[^\\"]*)*"⌟ обладает как достоинствами, так и недостатками.

Недостатки:

- **Неудобочитаемость.** Вероятно, самый большой недостаток заключается в том, что исходное выражение ⌜"([^\\"]|\\.)*"⌟ выглядело более понятно. Наглядностью пришлось отчасти пожертвовать ради эффективности.
- **Сложность сопровождения.** Выражение ⌜"[^\\"]*(\\.[^\\"]*)*"⌟ труднее поддерживать, поскольку любые изменения приходится синхронизировать в двух экземплярах ⌜[^\\"]⌟. Усложнение сопровождения искупается повышением эффективности.

Достоинства:

- **Исключение зависаний.** Новое выражение не приводит к зависанию при отсутствии совпадений (или в POSIX НКА). Благодаря тщательно продуманной структуре выражения, допускающей лишь один вариант совпадения с конкретным фрагментом, механизм быстро приходит к выводу, что несовпадающий текст действительно не совпадает.
- **Скорость.** Полученное регулярное выражение обрабатывается «плавно» — эта тема рассматривается в разделе «Исключение случайных совпадений» (☞ 349). В серии тестов, проведенных мной для традиционного НКА, раскрученная версия стабильно работала быстрее старой версии с конструкцией выбора. Это относится даже к успешным совпадениям, где в старой версии не возникало проблем с зависанием.

Применение атомарной группировки и захватывающих квантификаторов

Главный недостаток исходного выражения ⌜"(\\.|[^\\"]+)*"⌟ заключался в том, что оно увязало в переборе вариантов при отсутствии совпадения. Если совпадение есть, выражение работает довольно быстро; это объясняется тем, что компонент ⌜[^\\"]+⌟ совпадает с большей частью содержимого целевой строки (*норм* в предыдущем описании). Подвыражение ⌜[…]+⌟ обычно оптимизируется по скорости (☞ 314), а поскольку этот компонент обрабатывает большую часть символов, затраты на конструкцию выбора и внешний квантификатор ⌜(…)*⌟ значительно снижаются.

Итак, выражение ⌜"(\\.|[^\\"]+)*"⌟ при несовпадении «буксует», снова и снова перебирая варианты, которые заведомо не приводят к успеху.

Мы это знаем, потому что во всех случаях проверяются разные сочетания одного и того же (если ⌜abc⌟ не совпадает с 'foo', не совпадут также варианты ⌜abc⌟ и ⌜abc⌟, и, если уж на то пошло, то и ⌜abc⌟, ⌜abc⌟ и ⌜abc⌟). Следовательно, мы можем удалить эти состояния, чтобы регулярное выражение быстрее пришло к выводу об отсутствии совпадения.

Существует два способа уничтожения (или игнорирования) промежуточных состояний: атомарная группировка (☞ 187) и захватывающие квантификаторы (☞ 190).

Прежде чем переходить к исключению возвратов, стоит изменить порядок альтернатив: вместо ⌜"(\\.|[^\\"]+)*"⌟ выражение приводится к виду ⌜"([^\\"]+|\\.)*"⌟. Альтернатива, совпадающая с «нормальным» текстом, ставится на первое место. Как неоднократно упоминалось в предыдущих главах, если две альтернативы теоретически могут совпасть в одной позиции, необходимо тщательно выбрать порядок их перечисления, поскольку он может повлиять на то, какая из альтернатив будет включена в совпадение. Но если, как в нашем примере, альтернативы являются взаимоисключающими (ни одна не может совпасть в позиции, в которой может совпасть другая), порядок не важен с точки зрения правильности и поэтому выбирается с позиций наглядности или эффективности.

Предотвращение бесконечного поиска с использованием захватывающих квантификаторов

Наше выражение ⌜"([^\\"]+|\\.)*"⌟ содержит два квантификатора. Мы можем сделать захватывающим только первый, только второй или оба квантификатора. На что это повлияет? Основные проблемы с возвратом возникали из-за состояний, оставленных ⌜[…]+⌟, поэтому моим первым побуждением было преобразование ⌜[…]+⌟ к захватывающему варианту. В результате мы получим выражение, которое работает достаточно быстро даже при отсутствии совпадений. Тем не менее перевод внешней конструкции ⌜(…)*⌟ на захватывающий квантификатор уничтожает все состояния в круглых скобках, в том числе состояния ⌜[…]+⌟ и конструкции выбора, поэтому если выбирать что-то одно, я бы выбрал это подвыражение.

Но мы не обязаны ограничиваться одним квантификатором, поскольку в захватывающую форму можно преобразовать оба. Какой из трех вариантов работает быстрее? Все зависит от степени оптимизации захватывающих квантификаторов. В настоящее время они поддерживаются только пакетом регулярных выражений для Java от Sun, поэтому мое тестирование было ограниченным, но я проверил все три варианта и нашел примеры, в которых на первое место выходили разные варианты. Теоретические рассуждения наводили на мысль, что вариант с двумя захватывающими квантификаторами должен быть самым быстрым, поэтому из

этих результатов можно сделать вывод, что в реализации Sun захватывающие квантификаторы были оптимизированы не полностью.

Предотвращение бесконечного поиска с использованием атомарной группировки

Рассмотрим возможности применения атомарной группировки в выражении `"([^\\"]+|\\.)*"`. На первый взгляд, возникает искушение просто заменить обычные круглые скобки атомарными: `"(?>[^\\"] +|\\.)*"`. Тем не менее вы должны хорошо понимать, что в отношении уничтожения промежуточных состояний конструкция `(?>…|…)*` принципиально отличается от захватывающей конструкции `(…|…)*+` из предыдущего раздела.

Захватывающее подвыражение `(…|…)*+` после завершения обработки вообще не оставляет состояний. С другой стороны, атомарная группировка `(?>…|…)*` просто удаляет состояния, оставшиеся от каждой альтернативы и от самой конструкции выбора. Квантификатор `*` находится *за пределами* атомарной группировки, поэтому не подчиняется ее действию и оставляет все свои сохраненные состояния. Это означает, что частичные совпадения по-прежнему могут отменяться в результате возврата. Мы хотим, чтобы состояния внешнего квантификатора тоже уничтожались, поэтому нам понадобится дополнительная конструкция атомарной группировки. Теперь становится ясно, почему захватывающая конструкция `(…|…)*+` должна имитироваться конструкцией `(?>(…|…)*)`.

Конечно, обе конструкции, `(…|…)*+` и `(?>…|…)*`, помогают в решении проблемы бесконечного поиска, но состав уничтожаемых состояний и время их уничтожения различаются (дополнительная информация приведена на с. 228).

Примеры раскрутки цикла

Разобравшись с основными принципами раскрутки, давайте вернемся к некоторым примерам, приводимым ранее, и посмотрим, как применить к ним новые знания.

Раскрутка многосимвольных ограничителей

В главе 4 на с. 220 был приведен следующий пример:

```
<B>                 # Совпадение для открывающего тега <B>
(                   # Теперь как можно больше экземпляров...
    (?! </?B> )     # Если не <B> и не </B>...
    .               # ...то подойдет любой символ
)*                  # (теперь максимальный)
</B>                # ...пока не совпадет закрывающий тег
```

Если за *норм* принять ⌜[^<]⌟, а за *спец* — ⌜(?!</?B>)<⌟, раскрученная версия будет выглядеть так:

```
<B>                # Совпадение для открывающего тега <B>
  (?> [*<]* )      # Найти совпадение для "норм"...
  (?>              # Любое количество экземпляров...
  (?! </?B> )      # если не <B> и не </B>,
  <                # один экземпляр "спец"
  [^<]*            # и любое количество экземпляров "норм"
  )*               #
</B>  # Совпадение для закрывающего тега </B>
```

Атомарная группировка в данном случае не обязательна, но она ускоряет работу выражения при частичном совпадении.

Раскрутка в примере со строками продолжения

В примере со строками продолжения, приведенном в начале предыдущей главы (☞ 243), мы остановились на выражении ⌜^\w+=([^\n\\]| \\.)*⌟. Конечно, оно прекрасно подходит для раскрутки:

```
^ \w+ =   # Начальное имя поля и '='
# Прочитать (и сохранить) значение ...
(
 (?> [^ \n\\]* ) # "норм"*
 (?> \\. [^\n\\]* )* # ( "спец" "норм"* )*
)
```

Как и в предыдущих примерах, атомарная группировка не является строго обязательной, но помогает механизму регулярных выражений быстрее прийти к выводу об отсутствии совпадения.

Раскрутка регулярных выражений CSV

В главе 5 подробно обсуждалась задача разбора данных в формате CSV. В итоге мы пришли к следующему фрагменту со с. 279:

```
(?:^|,)
(?: # Поле в кавычках (внутри поля разрешаются удвоенные кавычки)
        " # (открывающая кавычка)
        ( (?: [^"]| "")* )
        " # (закрывающая кавычка)
|
    # ...или произвольная последовательность символов, кроме
         запятых и кавычек...
        ( [^",]* )
)
```

Затем в тексте рекомендуется включить в выражение префикс ⌜\G⌟ просто для того, чтобы во время работы примера заведомо не возникало никаких проблем со смещением текущей позиции, а также по соображениям эффективности. Теперь, когда вы знакомы с методикой раскрутки, давайте посмотрим, где ее можно применить в этом примере.

Несомненно, подвыражение для строк CSV в формате Microsoft, ⌜(?:[^"]|"")*⌟ выглядит заманчиво. Более того, в самом выражении уже представлены части *норм* и *спец*: ⌜[^"]⌟ и ⌜""⌟. Ниже показано, как выглядит раскрученное выражение в исходном фрагменте Perl с дальнейшей обработкой полей:

```
while ($line =~ m{
          \G(?:^|,)
          (?:
              # Либо поле в кавычках (каждая кавычка представляется
              # как "")..." открывающая кавычка поля
               ( (?> [^"]* ) (?> "" [^"]* )* )
              " # закрывающая кавычка поля
            # ..или …
            |
                # … некоторый текст, не содержащий кавычек и запятых....
                ( [^",]* )
            )
        }gx)
{
    if (defined $2) {
        $field = $2;
    } else {
        $field = $1;
        $field =~ s/""/"/g;
    }
    print "[$field]"; # вывод содержимого поля для отладочных целей
    Теперь можно работать с переменной $field...
}
```

Как и в других примерах, атомарная группировка не обязательна, но способствует повышению эффективности.

Раскрутка комментариев C

Рассмотрим пример раскрутки цикла для более сложного целевого текста. В языке C комментарии начинаются с символов /*, завершаются символами */ и могут распространяться на несколько строк (однако вложение комментариев не допускается). Выражение, совпадающее с таким комментарием, может использоваться в раз-

нообразных ситуациях, например при написании программы-фильтра для удаления комментариев. Работая над этой задачей, я впервые пришел к идее раскрутки цикла, и с тех пор она заняла почетное место в моем арсенале регулярных выражений.

Раскручивать или не раскручивать...

Регулярное выражение, описанное в этом разделе, было написано мной в начале 90-х годов. До этого поиск комментариев C при помощи регулярного выражения считался делом в лучшем случае крайне сложным, если не невозможным, поэтому работоспособное решение стало стандартным способом поиска комментариев на языке C. Но с появлением в языке Perl минимальных квантификаторов появилось другое, более простое решение: `/\*.*?\*/` в режиме совпадения точки со всеми символами.

Если бы минимальные квантификаторы поддерживались в то время, когда я впервые занялся методикой раскрутки, вероятно, я бы не стал с ней возиться. Впрочем, до какого-то времени это решение приносило пользу, поскольку в первой версии Perl с поддержкой минимальных квантификаторов раскрученная версия значительно опережает по скорости версию с минимальными квантификаторами (в разных тестах, проведенных мной, превышение составляло от 50% до 3,6 раза).

В современных версиях Perl поддерживаются многочисленные оптимизации, из-за которых картина в корне меняется — версия с минимальными квантификаторами работает от 50% до 5,5 раза быстрее. Итак, в современной версии Perl я совершенно спокойно использую выражение `/\*.*?\*/`.

Означает ли это, что методика раскрутки цикла утратила свою ценность при поиске комментариев C? Если механизм регулярных выражений не поддерживает минимальные квантификаторы, возможность применения раскрутки приносит несомненную пользу. Кроме того, не все механизмы регулярных выражений обладают одинаковыми наборами оптимизаций: во всех остальных языках раскрученное выражение работало быстрее — в моих тестах до 60 раз! Прием раскрутки цикла однозначно полезен, поэтому в оставшейся части этой главы будет показано, как применить его при поиске комментариев на языке C.

В комментариях на языке C не существует последовательностей, которые бы интерпретировались особым образом, как, например, экранированные кавычки внутри строки, заключенной в кавычки. Вроде бы это должно упростить задачу, однако поиск комментариев на языке C затрудняется тем, что «завершающая кавычка» `*/` состоит из нескольких символов. Простое решение `/\*[^*]*\*/` не работает, поскольку оно не совпадет со вполне допустимым комментарием `/** some comment here **/`, содержащим внутренние символы '*'. Нам потребуется более сложное решение.

Упрощение записи

Вероятно, вы обратили внимание, что выражение ⌜`/\*[^*]*\*/`⌟ очень плохо читается — к сожалению, один из символов-ограничителей комментария, '*', также является метасимволом регулярного выражения. От обилия префиксов \ начинает рябить в глазах. Чтобы выражение выглядело более понятно, мы будем считать, что комментарий заключается в ограничители `/x...x/`, а не `/*...*/`. Это искусственное изменение позволит записать ⌜`/\*[^*]*\*/`⌟ в наглядной форме ⌜`/x[^x]*x/`⌟. В процессе анализа этого примера выражение станет еще более сложным, и вы оцените это упрощение по достоинству.

Прямолинейный подход

В главе 5 (☞ 255) я привел стандартный алгоритм поиска текста, заключенного между ограничителями:

1. Найти открывающий ограничитель.
2. Найти основной текст (фактически это означает «все, что не является закрывающим ограничителем»).
3. Найти закрывающий ограничитель.

Похоже, наши псевдокомментарии `/x` и `x/` подходят под этот шаблон. Трудности начинаются, когда вы попытаетесь найти «все, что не является закрывающим ограничителем». Если закрывающий ограничитель представляет собой одиночный символ, можно воспользоваться инвертированным символьным классом, совпадающим со всеми символами, кроме ограничителя. Для многосимвольных подвыражений нельзя использовать символьный класс, но если диалект поддерживает негативную опережающую проверку, вы можете воспользоваться выражением вида ⌜`(?:(?!x/).)*`⌟. В сущности, оно эквивалентно выражению ⌜`(все, кроме x/)*`⌟.

Таким образом, для поиска комментариев можно воспользоваться выражением ⌜`/x(?:(?!x/).)*x/`⌟. Оно прекрасно работает, но его скорость оставляет желать лучшего (в некоторых тестах в сотни раз медленнее, чем то выражение, которое мы создадим позднее в этом разделе). Такое решение работает, но в нашем конкретном случае оно бесполезно — диалект, поддерживающий опережающую проверку, почти наверняка поддерживает минимальные квантификаторы. Следовательно, если эффективность не критична, воспользуйтесь выражением ⌜`/x.*?x/`⌟ и не ломайте голову.

Но вернемся к нашему прямолинейному алгоритму из трех шагов. Существует ли другой способ найти совпадение до первого `x/`? В голову приходят два возможных решения. Первое — рассматривать `x` как начало закрывающего ограничителя. В этом случае мы ищем все, что не совпадает с `x`, и допускаем `x` в том случае, если

за ним следует что-то отличное от символа /. Таким образом, «все, что не является закрывающим ограничителем» может быть:

- любым символом, кроме x: ⌜[^x]⌟;
- x, если за ним не следует символ /: ⌜x[^/]⌟.

В результате основному тексту соответствует выражение ⌜([^x]|x[^/])*⌟, а всему псевдокомментарию — ⌜/x([^x]|x[^/])*x/⌟. Как вы вскоре убедитесь, это решение не работает.

Другой способ заключается в том, чтобы рассматривать / как закрывающий ограничитель, но лишь в том случае, если ему предшествует x. Таким образом, «все, что не является закрывающим ограничителем» может быть:

- любым символом, кроме /: ⌜[^/]⌟;
- символом /, если ему не предшествует x: ⌜[^x]/⌟.

В результате основному тексту соответствует выражение ⌜([^/]|[^x]/)*⌟, а всему комментарию — ⌜/x([^/]|[^x]/)*x/⌟.

К сожалению, это решение тоже не работает.

Начнем с ⌜/x([^x]|x[^/])*x/⌟. Рассмотрим строку '/xx•foo•xx/' — после совпадения с 'foo•' первый символ x совпадает с ⌜x[^/]⌟, что вполне нормально. Но затем совпадает с xx/, а этот символ x должен входить в закрывающий ограничитель комментария. В результате совпадение продолжится и после x/ (до конца следующего комментария, если он существует).

Что касается ⌜/x([^/]|[^x]/)*x/⌟, то это выражение не совпадает с '/x/•foo•/x/' (вполне нормальным комментарием, который должен совпадать). В других случаях это выражение может выйти за конец комментария, за которым немедленно следует / (по аналогии с первым способом). Возврат, происходящий в таких случаях, несколько запутывает ситуацию, поэтому вам стоит разобраться, почему ⌜/x([^/]|[^x]/)*x/⌟ совпадает в строке

```
years = days /x divide x//365; /x assume non-leap year x/
```

с подчеркнутым текстом (эта задача остается читателю для самостоятельной работы).

Работа над ошибками

Давайте попробуем исправить эти регулярные выражения. В первом выражении, где ⌜x[^/]⌟ непреднамеренно совпадает с …xx/ в завершении комментария, рассмотрим новый вариант ⌜/x([^x]|x+[^/])*x/⌟. Предполагается, что благодаря

ПЕРЕВОД НА ЯЗЫК РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

На с. 344, при описании двух вариантов поиска в комментариях C «всего, что не является закрывающим ограничителем» я представил две идеи:

«x, если за ним не следует символ /: ⌜`x[^/]`⌟»

и

«символ /, если ему не предшествует x: ⌜`[^x]/`⌟»

При этом я выражался неформально — описания отличаются от приведенных регулярных выражений. Вы понимаете, о чем речь?

Чтобы разобраться, в чем заключаются отличия, примените первое описание к строке '`regex`'. В ней присутствует символ x, за которым не следует символ слэша, однако эта строка *не* совпадет с ⌜`x[^/]`⌟. Символьный класс совпадает с символом, и хотя этот символ не может быть косой чертой, он все равно должен быть чем-то другим, а не «*ничем*», как в строке '`regex`'. Во второй ситуации дело обстоит аналогично.

Если диалект поддерживает опережающую проверку, формулировка "`x`, если за ним не следует символ `/`" соответствует простому выражению ⌜`x(?!/)`⌟. Если опережающая проверка недоступна, попробуйте использовать выражение ⌜`x([^/]|$)`⌟. Оно по-прежнему совпадает с символом, следующим за x, но также может совпадать и с концом строки. Если поддерживается ретроспективная проверка, «символ `/`, которому не предшествует `x`» превращается в ⌜`(?<!x)/`⌟. В противном случае то же самое можно сделать с помощью ⌜`(^|[^x])/`⌟.

При работе с комментариями C эти выражения не используются, но я рекомендую хорошенько разобраться в них.

дополнительному `+`⌜`x+[^/]`⌟ совпадает с цепочкой `x`, после которой следует символ, отличный от /. И это действительно так, но из-за возврата «символ, отличный от /» может оказаться все тем же `x`. Сначала максимальный квантификатор ⌜`x+`⌟ совпадает с лишним `x`, как мы и хотели, но вследствие возврата этот символ может быть возвращен, если это необходимо для получения общего совпадения. К сожалению, выражение по-прежнему захватывает слишком много:

```
/xx A xx/ foo() /xx B xx/
```

Чтобы прийти к правильному решению, нужно вспомнить то, что я говорил раньше: *формулируйте выражение как можно точнее*. Если мы хотим определить «цепочку `x`, после которой следует символ, отличный от `/`», и при этом подразумевается, что «символ, отличный от `/`», также отличен и от `x`, об этом нужно сообщить явно: ⌜`x+[^/x]`⌟. Как и требовалось, эта запись предотвращает поглощение '`…xxx/`' — по-

следнего x в цепочке, завершающей комментарий. В качестве побочного эффекта предотвращается совпадение со всеми символами x, завершающими комментарий, поэтому мы оказываемся в позиции '...▲xxx/' перед закрывающим ограничителем. Поскольку часть выражения, относящаяся к завершающему ограничителю, допускает всего один символ x, в нее необходимо добавить квантификатор +: ⌜x+/⌟.

В результате получается следующее выражение: ⌜/x([^x]|x+[^/x])*x+/⌟.

Уф! Весьма запутанные рассуждения, не правда ли? Выражение для настоящих комментариев, со звездочками вместо x, выглядит еще хуже: ⌜/*([^*]|*+[^/*])**+/⌟. Чтобы прочитать такое выражение, вам придется изрядно пошевелить мозгами.

Раскрутка выражения для поиска комментариев C

Попробуем повысить эффективность выражения, избавившись от конструкции выбора. В табл. 6.3 приведены выражения, которые должны подставляться в шаблон раскрутки цикла.

Как и в примере с доменными именами, ⌜*норм**⌟ не может совпадать с «ничем». В предыдущем примере это было связано с тем, что «нормальная» часть (имя домена нижнего уровня) не могла быть пустой. В данном случае это объясняется особенностями обработки двухсимвольного закрывающего ограничителя. Любая последовательность *норм* должна завершаться с первым символом закрывающего ограничителя, позволяя *спец* «перехватить» совпадение лишь в том случае, если следующий символ не завершает ограничитель.

Таблица 6.3. Компоненты раскрутки цикла для комментариев C

⌜начало норм*(спец норм*)* конец⌟		
Компонент	**Аналог в тексте**	**Регулярное выражение**
начало	Начало комментария	/x
норм*	Текст комментария до одного или нескольких x включительно	[^x]*x+
спец	Символ, отличный от символа обратного слэша (и не являющегося x)	[^/x]
конец	Завершающий символ обратного слэша	/

Подставляя эти компоненты в общий шаблон раскрутки цикла, мы получаем:

⌜/x[^x]*x+(▲[^/x][^x]*x+)*⌟

Обратите внимание на помеченный фрагмент. Механизм регулярных выражений может прийти к нему двумя путями (как и в выражении на с. 337): либо продвиже-

нием через начальную конструкцию ⌜/x[^x]*x+⌟, либо циклическим перебором (…)*. В любом случае, оказавшись в этой позиции, мы знаем, что был найден символ x и текущая позиция является критической точкой — возможно, в конце комментария. Если следующим символом является символ слэша, поиск закончен. Если это любой другой символ (конечно, кроме x), мы знаем, что тревога была ложной и можно вернуться к поиску совпадений для компонента *норм* в ожидании следующего x. После того как символ будет найден, мы снова оказываемся в критической точке.

Практическая сторона

Выражение ⌜/x[^x]*x+([^/x][^x]*x+)*⌟ не совсем готово к практическому использованию. Во-первых, конечно, комментарии обозначаются ограничителями /*…*/, а не /x…x/. Проблема легко решается заменой каждого x на экранированную звездочку * (в символьных классах — простой заменой x на *):

```
⌜/\*[^*]*\*+([^/*][^*]*\*+)*/⌟
```

Также следует учесть тот факт, что комментарии часто распространяются на несколько строк. Если искомый текст содержит весь многострочный комментарий, это выражение будет работать. Однако в строчно-ориентированных программах (таких, как *egrep*) не существует возможности применить регулярное выражение к полному комментарию. Но в большинстве программ, упомянутых в книге, это возможно, поэтому данное выражение может применяться, например, для удаления комментариев.

На практике возникает еще одна большая проблема. Наше регулярное выражение понимает комментарии C, но ничего не знает о других важных аспектах синтаксиса C. Например, оно может совпасть и при отсутствии комментария:

```
const char *cstart = "/*". *cend = "*/";
```

Наше выражение будет усовершенствовано в следующем разделе.

Исключение случайных совпадений

Мы потратили некоторое время на конструирование регулярного выражения, предназначенного для поиска комментариев C, и остановились на проблеме случайных совпадений, по своей структуре напоминающих комментарии. Например, в Perl для удаления комментариев можно попытаться использовать следующий фрагмент:

```
$prog =~ s{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/}{}g; # удаление комментариев C
                                               # (и не только!)
```

Фрагменты содержимого переменной $prog, соответствующие нашему регулярному выражению, заменяются «ничем» (т. е. удаляются). Проблема заключается в том, что в процессе перемещения начальной позиции начало совпадения может быть случайно обнаружено внутри строки, как в следующем фрагменте C:

```
char *CommentStart = "/*": /* Начало комментария */
char *CommentEnd   = "*/"; /* Конец комментария */
```

В этом фрагменте жирным шрифтом выделено то, что мы хотим найти, а подчеркиванием — то, что *будет* найдено в действительности. В процессе поиска механизм пытается найти совпадение от каждой позиции строки. Поскольку поиск оказывается успешным лишь в позиции начала комментария (или похожей конструкции), в большинстве позиций совпадения не обнаруживаются. Но в какой-то момент смещение текущей позиции подходит к строке, заключенной в кавычки, содержимое которой напоминает начало комментария. Наша задача — сделать так, чтобы механизм регулярных выражений в процессе поиска игнорировал строки, заключенные в кавычки... И такая возможность существует.

Управление поиском совпадения

Рассмотрим следующий фрагмент:

```
$COMMENT = qr{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/}; # Регулярное выражение
                                               # для комментария
$DOUBLE = qr{"(?:\\.|[^\\"])*"};               # Регулярное выражение
                                               # для строки в кавычках
$text =~ s/$DOUBLE|$COMMENT//g;
```

Обратите внимание на два новых обстоятельства. Во-первых, регулярное выражение-операнд $DOUBLE|$COMMENT состоит из двух переменных, каждая из которых определяется при помощи оператора Perl qr/…/. Как подробно обсуждалось в главе 3 (☞ 140), при создании строк, интерпретируемых как регулярные выражения, часто допускаются ошибки. Для решения этой проблемы в Perl был создан оператор qr/…/, который интерпретирует свой операнд как регулярное выражение, но не применяет его к тексту. Вместо этого он возвращает «объект регулярного выражения», который позднее может использоваться для построения больших регулярных выражений. В главе 2 (☞ 111) вы уже убедились, что этот оператор чрезвычайно удобен. Как и при использовании операторов m/…/ и s/…/…/, ограничители можно выбирать по своему усмотрению (☞ 107); в данном примере были выбраны фигурные скобки.

Во-вторых, стоит обратить внимание на применение части $DOUBLE для поиска строк в кавычках. Когда в результате смещения механизм переходит к позиции, в которой может совпасть компонент $DOUBLE, он назначает совпадение и обходит

внутреннюю строку в кавычках. Возможность включения обеих альтернатив в поиск обусловлена полным отсутствием неоднозначности между ними. Начиная с левого края, любая начальная позиция в строке:

- является началом комментария, что приводит к немедленному пропуску символов до конца комментария; или...
- является началом строки в кавычках, что также приводит к немедленному пропуску до конца строки; или...
- не обеспечивает совпадения ни для одного из этих выражений. В этом случае механизм смещает начальную позицию только на один символ.

В этом случае *начальная* позиция совпадения никогда не будет находиться *внутри* строки или комментария — в этом и заключается секрет успеха. Пока этот фрагмент остается бесполезным, поскольку вместе с комментариями он удаляет и строки, заключенные в кавычки. Впрочем, небольшое изменение ставит все на свои места:

```
$COMMENT = qr{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/}; # Регулярное выражение
                                               # для комментария
$DOUBLE = qr{"(?:\\.|[^\\"])*"};               # Регулярное выражение
                                               # для строки в кавычках
$text =~ s/($DOUBLE)|$COMMENT/$1/g;
```

В этот фрагмент были внесены некоторые изменения:

- Добавлены круглые скобки для заполнения `$1` в том случае, если обнаруживается строка в кавычках. Если будет найдена альтернатива-комментарий, переменная `$1` остается пустой.
- В качестве заменяющей строки была подставлена та же переменная `$1`. Если теперь будет найдена строка в кавычках, она заменяется той же самой строкой — происходит тождественная замена, поэтому строки в кавычках из текста не удаляются (но в качестве побочного эффекта механизм пропускает внутреннюю строку в кавычках, для чего все и затевалось). С другой стороны, при совпадении альтернативы комментария переменная `$1` остается пустой, поэтому комментарий, как и было запланировано, заменяется пустой строкой[1].

[1] Если переменная `$1` в языке Perl не заполняется в процессе поиска, ей присваивается специальное «неопределенное» значение `undef`. В строке замены `undef` интерпретируется как пустая строка, поэтому все работает именно так, как требуется. Но если включить в Perl режим выдачи предупреждений, как поступают все порядочные программисты, при подобном использовании `undef` будет выдано предупреждение. Чтобы избежать выдачи предупреждения, включите директиву '`no warnings;`' перед поиском по регулярному выражению или воспользуйтесь специальной формой подстановки оператора Perl: `$text =~ s/($DOUBLE)|$COMMENT/defined($1) ? $1 : ""/ge;`

Наконец, мы должны позаботиться о константах C, заключенных в апострофы ('\t' и т. д.). Задача решается просто — включением в круглые скобки новой альтернативы. Если вы захотите удалять и комментарии C++/Java/C# (вида //), для этого достаточно добавить четвертую альтернативу `//[^\n]*`, не заключая ее в круглые скобки:

```
$COMMENT = qr{/\*[^*]*\*+(?:[^/*][^*]*\*+)+/}; # выражение для комментария
$COMMENT2 = qr{//[^\n]*};              # для комментария C++ в стиле //
$DOUBLE = qr{"(?:\\.|[^\\"])*"};       # для строк в кавычках
$SINGLE = qr{'(?:\\.|[^'\\])*'};       # для строк в апострофах

$text =~ s/($DOUBLE|$SINGLE)|$COMMENT|$COMMENT2/$1/g;
```

В основу этого решения заложена довольно изящная идея: во время проверки текста механизм быстро находит (и в нужных ситуациях — удаляет) эти специальные конструкции. Я провел небольшой тест: для удаления всех комментариев из тестового файла объемом 16 Мбайт, состоящего из 500 000 строк, на моем компьютере этот фрагмент Perl выполнялся всего 16,4 секунды. Быстро? Да, но мы существенно ускорим его работу.

Управление поиском = скорость

Если приложить некоторые дополнительные усилия, процесс поиска совпадений можно значительно ускорить. Рассмотрим длинные последовательности символов обычного кода C между комментариями и строками в кавычках. Для каждого такого символа механизм регулярных выражений должен опробовать все четыре альтернативы и определить, не относится ли он к одному из четырех «поглощаемых» фрагментов. Только если проверка всех четырех альтернатив завершается неудачей, символ отвергается как «неинтересный» и механизм переходит к следующей позиции. Таким образом, выполняется большая работа, которую на самом деле выполнять не обязательно.

Например, мы знаем, что для совпадения какой-либо из четырех альтернатив начальный символ должен быть символом слэша, апострофом или кавычкой. Выполнение этого условия еще не гарантирует совпадения, но его *невыполнение* заведомо говорит о том, что совпадения нет. Итак, вместо того чтобы заставлять механизм выполнять медленный и мучительный перебор, мы прямо укажем на этот факт и включим символьный класс `[^'"/]` в качестве отдельной альтернативы. Более того, в файле эти «неинтересные» символы могут следовать подряд, поэтому мы воспользуемся выражением `[^'"/]+`. Если вы помните пример с бесконечным перебором, возможно, вас обеспокоит появление нового плюса. В самом деле, плюс внутри конструкции (…)* способен причинить массу хлопот, но как самостоятельная конструкция он вполне допустим (за ним нет ничего, что могло бы принудить

механизм к возврату и стать причиной бесконечного перебора). В новом варианте наш тест выглядит так:

```
$OTHER = qr{[^"'/]}; # Символы, с которых не могут начинаться
                     # другие альтернативы
   ⋮
$text =~ s/($DOUBLE|$SINGLE|$OTHER+)|$COMMENT|$CQMMENT2/$1/g;
```

По причинам, которые будут объяснены ниже, квантификатор + ставится после `$OTHER` (вместо того, чтобы включить его в `$OTHER`).

Я заново запускаю свой тест. Изумленная публика ликует — всего одно изменение сократило время обработки на 75%! В построенном нами выражении исключена большая часть непроизводительных затрат по частому перебору альтернатив. Остаются еще относительно редкие ситуации, при которых не совпадает ни одна из альтернатив (например, '`c ▲/ 3.14`'). В этом случае для перебора приходится довольствоваться стандартным механизмом смещения текущей позиции.

Но даже сейчас работа еще не закончена — механизм можно еще ускорить.

- Как правило, чаще всего будет совпадать альтернатива `⌜$OTHER+⌟`, поэтому мы поставим ее на первое место в круглых скобках. Для механизма POSIX НКА это несущественно, поскольку он всегда проверяет все альтернативы, но в традиционном механизме НКА поиск прекращается сразу же после нахождения совпадения. Стоит ли заставлять его сначала долго разыскивать то, что встречается редко, вместо того, чтобы поставить вперед самую частую, по нашему мнению, альтернативу?

- Если совпадет строка, заключенная в апострофы или кавычки, то за ней с большой вероятностью последуют повторения `$OTHER`, после чего последует другая строка или комментарий. Если добавить `⌜$OTHER*⌟` после каждой строки, мы тем самым сообщим механизму, что он может перейти немедленно к поиску совпадения для `$OTHER` без обработки цикла `/g`.

  Происходящее напоминает методику раскрутки цикла. В сущности, выигрыш в скорости от раскрутки цикла в значительной степени обусловлен тем, что мы «направляем» механизм регулярных выражений к совпадению, используя общие сведения о целевом тексте для создания локальных оптимизаций. Механизм регулярных выражений получает именно те сведения, которые необходимы для его быстрой работы.

  Очень важно, чтобы выражение `$OTHER`, которое добавляется после каждого подвыражения, совпадающего со строкой в кавычках или апострофах, квантифицировалось звездочкой, а начальное выражение `$OTHER` (то, которое мы переместили в начало конструкции выбора) квантифицировалось плюсом. Если вам это покажется непонятным, подумайте, что произойдет, если суффиксное

выражение $OTHER будет квантифицироваться плюсом и в тексте встретятся, скажем, две строки в кавычках подряд. А если в начальном выражении $OTHER будет использоваться квантификатор *, оно будет совпадать всегда!

Описанные изменения приводят к следующему результату:

```
「($OTHER+|$DQUBLE$OTHER*|$SINGLE$OTHER*)|$COMMENT|$COMMENT2」
```

Это выражение, передаваемое в качестве аргумента regsub, уменьшает время обработки еще примерно на 5%.

Давайте вернемся на шаг назад и проанализируем два последних изменения. Поскольку мы «вычерпываем» $OTHER* после каждой строки в апострофах или кавычках, исходное подвыражение $OTHER+ (поставленное нами на первое место в конструкции выбора) может совпасть только в двух случаях:

1) в самом начале выполнения команды s/.../.../g, до того как была найдена хотя бы одна строка в кавычках или апострофах;

2) после любого комментария.

Возникает соблазнительная мысль: почему бы нам не разобраться с пунктом 2, добавив $OTHER* и после комментариев? На первый взгляд все хорошо, если не считать того, что весь оставляемый текст должен находиться внутри круглых скобок: располагая его после комментариев, мы вместе с водой (т. е. комментариями) выплескиваем из ванны и ребенка (код).

Итак, если исходное выражение $OTHER+ используется в основном после комментариев, стоит ли перемещать его на первое место? Вероятно, ответ на этот вопрос зависит от данных — если комментариев больше, чем строк в кавычках и апострофах, то такое перемещение оправданно. В противном случае я не стал бы выносить эту альтернативу вперед. Как показали мои тесты, при расположении этого выражения на первом месте были показаны лучшие результаты. При перемещении его в конец выражения была потеряна примерно половина выигрыша, достигнутого на предыдущем шаге.

Свертка

Но и это еще не все! Не забывайте о том, что каждое выражение для строк, заключенных в кавычки и апострофы, прекрасно подходит для раскрутки — методики, которой в этой главе был посвящен длинный раздел. Заменим выражения SINGLE и DOUBLE раскрученными версиями:

```
$DOUBLE = qr{"[^\\"]*(?:\\.[^\\"]*)*"};
$SINGLE = qr{'[^'\\]*(?:\\.[^'\\]*)*'};
```

Данное изменение добавляет еще 15% выигрыша. Небольшие изменения уменьшили время обработки выражения с 16,4 до 2,3 секунды — семикратное ускорение!

Последнее изменение также показывает, что переменные заметно облегчают построение регулярных выражений. Отдельные компоненты (такие, как `$DOUBLE`) относительно независимы, поэтому их можно изменять без модификации всего выражения. Конечно, при использовании переменных приходится учитывать некоторые второстепенные факторы (например, нумерацию круглых скобок), но в целом эта методика чрезвычайно полезна.

В данном случае она особенно удобна благодаря оператору Perl `qr/…/`, который создает строковый объект, предназначенный для работы с регулярными выражениями. В других языках отсутствуют прямые аналоги этого оператора, но в них имеются строковые типы, подходящие для построения регулярных выражений. За дополнительной информацией обращайтесь к разделу «Строки как регулярные выражения» главы 3 на с. 140.

Чтобы оценить подобный подход к построению регулярных выражений, достаточно взглянуть на полное выражение. Вот как оно выглядит после разбиения на строки:

```
([^"'/]+|"[^\\"]*(?:\\.[^\\"]*)*"[^"'/]*|'[^'\\]*
(?:\\.[^'\\]*)*'[^"'/]*)|/\*[^*]*\*+(?:[^/*][^*]*\*+)*/|//[^\n]*
```

Вывод: думайте!

Я хочу закончить эту главу примером, который наглядно показывает, какую пользу может принести творческое мышление при работе с регулярными выражениями НКА. Однажды при работе в GNU Emacs мне понадобилось регулярное выражение, которое находило бы сокращения типа «don't», «I'm», «we'll» и т. д., но не совпадало в других ситуациях. У меня получилось регулярное выражение, в котором за обозначением слова `\<\w+` следовал Emacs-эквивалент выражения `'([tdm]|re|ll|ve)`. Это решение работало, но я вдруг понял, что использовать `\<\w+` глупо — вполне достаточно `\w`. Ведь если апострофу непосредственно предшествует `\w`, то и `\w+` там заведомо присутствует; лишняя проверка не добавит никакой новой информации, если нас не интересуют точные границы совпадения (а меня они не интересовали — я хотел лишь найти строку). Использование `\w` ускорило работу выражения более чем в 10 раз.

Как видите, чтобы ускорить работу выражения, иногда достаточно просто подумать. Надеюсь, в этой главе вы нашли достаточно пищи для размышлений.

7 Perl

Perl часто упоминается на страницах книги, и не без оснований. Это популярный язык, обладающий исключительно богатыми возможностями в области регулярных выражений, бесплатный и доступный, понятный для начинающих и существующий на множестве платформ, включая все разновидности Windows, UNIX и Mac.

Некоторые программные конструкции Perl напоминают C и другие традиционные языки программирования, но на этом все сходство и кончается. Подход к решению задач на Perl — *путь Perl* — сильно отличается от традиционных языков. В Perl-программах используются традиционные концепции структурного и объектно-ориентированного программирования, но при обработке данных часто применяются регулярные выражения. В сущности, можно без преувеличения заявить, что **регулярные выражения играют ключевую роль практически в любой Perl-программе**. Это утверждение справедливо как для гигантской системы из 100 000 строк, так и для простых однострочных программ типа:

```
% perl-pi -e's{([-+]?\d+(\.\d*)?)F\b}{sprint "%.0fC",($1-32)*5/9)eg' *.txt
```

Эта программа просматривает файлы `*.txt` и преобразует температуру из шкалы Фаренгейта в шкалу Цельсия (вспомните первый пример из главы 2).

В этой главе

В этой главе рассматриваются практически все аспекты регулярных выражений в языке Perl[1], и в ней вы найдете информацию о диалекте и операторах, предназначенных для работы с регулярными выражениями. Материал, относящийся к регулярным выражениям, излагается с нуля, но предполагается, что вы, по крайней мере в общих чертах, знакомы с Perl (после чтения главы 2, вероятно, ваших

[1] Материал книги относится к Perl версии 5.8.8.

познаний хватит хотя бы на то, чтобы приступить к изучению этой главы). Я часто мимоходом использую концепции, которые до этого не рассматривались подробно, и не уделяю пристального внимания аспектам языка, не имеющим прямого отношения к регулярным выражениям. Возможно, вам стоит держать под рукой руководство по Perl или приобрести книгу «*Programming Perl*», выпущенную издательством O'Reilly[1].

Однако познания в области Perl не главное. Вероятно, еще важнее ваше *стремление узнать больше*. Эта глава ни в коем случае не является «легким чтивом». Поскольку я не собираюсь учить вас Perl с самого начала, у меня появляется возможность, которой нет у авторов общих учебников Perl, — мне не придется опускать важные детали, чтобы добиться связного повествования, плавно разворачивающегося на протяжении всей главы. Если что-то и остается постоянным, так это стремление к пониманию общей картины. Некоторые проблемы довольно сложны и загромождены обилием деталей. Не огорчайтесь, если вам не удастся усвоить все сразу. Я рекомендую один раз прочитать главу, чтобы получить общее представление о теме, а затем возвращаться к ней в будущем по мере надобности.

Чтобы вам было проще разобраться в материале, я приведу краткую сводку по структуре этой главы:

- В разделе «Диалект регулярных выражений Perl» (☞ 360) описан богатый набор метасимволов, поддерживаемых регулярными выражениями Perl, а также некоторые дополнительные возможности регулярных выражений, оформленных в виде литералов.
- В разделе «Perl'измы из области регулярных выражений» (☞ 369) рассматриваются некоторые аспекты Perl, имеющие особенно важное значение для регулярных выражений. Здесь подробно анализируются такие концепции, как *динамическая видимость* и *контекст выражения*, при этом особое внимание уделяется их связи с регулярными выражениями.
- Регулярные выражения приносят пользу лишь при наличии средств для их применения. В следующих разделах подробно описаны операторы Perl для работы с регулярными выражениями:
  - «Оператор `qr/…/` и объекты регулярных выражений» (☞ 381)
  - «Оператор поиска» (☞ 385)
  - «Оператор подстановки» (☞ 400)
  - «Оператор разбиения» (☞ 403)

[1] *Уолл Л., Кристиансен Т., Орвант Д.* Программирование на Perl». 3-е изд. / Пер. с англ. — СПб.: Символ-Плюс, 2002.

- В разделе «Специфические возможности Perl» (☞ 410) описаны некоторые средства из арсенала регулярных выражений Perl, в том числе возможность выполнения произвольного кода Perl в процессе поиска совпадения.
- Раздел «Проблемы эффективности в Perl» (☞ 435) затрагивает тему, близкую сердцу любого программиста. В Perl используется традиционный механизм НКА, поэтому вы можете смело начинать эксперименты со всеми приемами, описанными в главе 6. Конечно, существует немало факторов, специфических для Perl и сильно влияющих на то, как и насколько быстро применяются регулярные выражения в Perl; эти факторы будут рассмотрены в данном разделе.

Perl в предыдущих главах

Perl неоднократно упоминается практически во всех главах книги:

- **Глава 2** содержит введение в Perl с примерами регулярных выражений.
- **В главе 3** описана история появления Perl (☞ 124), а также упоминаются многочисленные аспекты регулярных выражений, относящиеся к Perl, например проблемы кодировок, включая Юникод (☞ 144), режимы поиска (☞ 150) и длинный обзор метасимволов (☞ 154).
- **Глава 4** срывает покров тайны с традиционного механизма НКА, реализованного в Perl, и потому чрезвычайно важна для пользователей Perl.
- **Глава 5** содержит множество примеров, рассматриваемых в свете главы 4. Многие примеры написаны на Perl, но даже примеры на других языках в той или иной степени применимы к Perl.
- **Глава 6** представляет несомненный интерес для пользователей Perl, интересующихся вопросами эффективности.

Чтобы книга стала более понятной для читателей, не имеющих опыта программирования на Perl, я часто упрощал примеры в предыдущих главах и старался по возможности использовать псевдокод, поясняющий назначение отдельных фрагментов. В этой главе приводимые примеры в большей степени соответствуют стилю Perl.

Регулярные выражения как компонент языка

К числу несомненных достоинств Perl относится поддержка регулярных выражений, интегрированная на языковом уровне. Вместо отдельных функций для применения регулярных выражений Perl предоставляет программисту специальные

операторы, тесно связанные с другими операторами и конструкциями, образующими язык Perl.

С учетом богатейших возможностей Perl в области регулярных выражений можно подумать, что в языке существует множество разных операторов, но в действительности Perl поддерживает всего четыре оператора для работы с регулярными выражениями и несколько вспомогательных конструкций, перечисленных в табл. 7.1.

Таблица 7.1. Общая поддержка регулярных выражений в Perl

Операторы	**Модификаторы**			**Modyfy How...**
`m`/*выражение*/*модификаторы* (☞ 385)	`/x`	`/o`		Интерпретация регулярного выражения (☞ 368, 438)
`s`/*выражение*/*замена*/*модификаторы* (☞ 400)	`/s`	`/m`	`/i`	Интерпретация целевого текста (☞ 368)
`qr`/*выражение*/*модификаторы* (☞ 381)	`/g`	`/c`	`/e`	Прочее (☞ 391, 396, 401)
`split(…)` (☞ 403)	**Переменные с информацией о совпадении (☞ 377)**			
Директивы	`$1,`	`$2`	и т. д.	Сохраненный текст
`use charnames ':full';` (☞ 365)	`$^N`	`$+`		Последнее/старшее из присвоенных значений `$1, $2...`
`use overload;` (☞ 428)	`@-`	`@+`		Массивы индексов в целевом тексте
`use re 'eval';` (☞ 422)	`$``	`$&`	`$´`	Предшествующий текст, текст совпадения и текст после совпадения
`use re 'debug';` (☞ 451)			(использовать не рекомендуется — см. «Проблемы эффективности в Perl», ☞ 435)	
Вспомогательные функции	**Вспомогательные переменные**			
`lc lcfirst uc ucfirst` (☞ 365)	`$_`			Целевой текст по умолчанию (☞ 388)
`pos` (☞ 394) `quotemeta` (☞ 366)	`$^R`			Результат выполнения встроенного кода (☞ 377)
`reset` (☞ 387) `study` (☞ 449)				

Perl обладает исключительно богатыми возможностями, но при воплощении в относительно малом наборе операторов эта мощь может стать «палкой о двух концах».

Самая сильная сторона Perl

Разнообразие возможностей и выразительных средств в операторах и функциях является, вероятно, самой сильной стороной Perl. Поведение операторов и функций изменяется в зависимости от контекста, в котором они используются, и довольно часто в каждой специфической ситуации делается именно то, что и предполагал программист. В книге «*Programming Perl*» даже утверждается, что «операторы Perl делают именно то, что вам нужно...». Например, оператор поиска `m/`*выражение*`/` обладает множеством разных функциональных возможностей в зависимости от того, где, как и с какими модификаторами он используется.

Самая слабая сторона Perl

Огромная концентрация разнообразных возможностей в выразительных средствах Perl также является одной из отталкивающих сторон языка. Существуют бесчисленные особые случаи, условия и контексты, которые неожиданно возникают при внесении небольшого исправления в программу, — просто вы сталкиваетесь с очередным особым случаем, о существовании которого и не подозревали[1]. Конечно, в программировании настоящим произведением искусства нередко считается скучный, последовательный, предсказуемый интерфейс. В книге «*Programming Perl*» цитата, что приводилась в предыдущем параграфе, имеет следующее продолжение: «...если вас не волнуют проблемы надежности». В руках опытного пользователя Perl превращается в оружие огромной разрушительной силы, но в процессе приобретения опыта вы неоднократно разрядите это оружие в себя.

Диалект регулярных выражений Perl

В табл. 7.2 приведено краткое описание диалекта регулярных выражений Perl. Раньше Perl поддерживал ряд метасимволов, отсутствующих в других системах, но за прошедшие годы другие системы позаимствовали многое из нововведений Perl. Эти общие возможности были представлены в главе 3, однако в Perl остались некоторые специфические возможности, о которых речь пойдет ниже (в табл. 7.2 приведены ссылки на соответствующие страницы).

Таблица 7.2. Общие сведения о диалекте регулярных выражений Perl

Сокращенные обозначения символов ①	
☞ 156 (C)	`\a [\b] \e \f \n \r \t` *восьм* **x***шестн* **x{***шестн***}** **c***символ*

[1] Несмотря на великое множество особых случаев, в этой главе мы попытаемся рассмотреть их все!

Символьные классы и аналогичные конструкции	
☞ 161	Обычные классы: `[…]` и `[^…]` (допускаются конструкции стандарта POSIX `[:alpha:]` ☞ 172)
☞ 162	Любой символ, кроме символа новой строки: *точка* (с модификатором `/s` — любой символ)
☞ 163	Комбинационные последовательности Юникода: `\X`
☞ 164	Принудительное однобайтовое совпадение (может быть рискованным): `\C`
☞ 164 (C)	Сокращенные обозначения классов:② `\w \d \s \W \D \S`
☞ 164 (C)	Свойства, алфавиты и блоки Юникода:③ `\p{`*свойство*`}, \P{`*свойство*`}`
Якорные метасимволы и другие проверки с нулевой длиной совпадения	
☞ 175	Начало строки/логической строки: `^ \A`
☞ 175	Конец строки/логической строки: `$ \z \Z`
☞ 395	Конец предыдущего совпадения: `\G`
☞ 180	Границы слов:④ `\b\B`
☞ 181	Позиционная проверка:⑤ `(?=…) (?!…) (?<=…) (?<!…)`
Комментарии и модификаторы режимов	
☞ 182	Модификаторы режимов:⑥ `(?`*модификатор*`)`. Допустимые модификаторы: `x s m i` (☞ 367)
☞ 183	Интервальное изменение режима `(?`*модификатор*`:…)`
☞ 183	Комментарии: `(?#…)` и `#…` (с модификатором `/x`, а также от символа `#` до новой строки или конца регулярного выражения)
Группировка, сохранение, условные и управляющие конструкции	
☞ 184	Сохраняющие круглые скобки: (…) `\1 \2 ...`
☞ 185	Группирующие круглые скобки: `(?:…)`
☞ 187	Атомарная группировка: `(?>…)`
☞ 187	Конструкция выбора: `|`
☞ 188	Условная конструкция: `(?`*if then*`|`*else*`)` — в части *if* может находиться встроенный фрагмент кода, позиционная проверка или (*число*)
☞ 189	Максимальные квантификаторы: `* + ? {n} {n,} {min,max}`
☞ 190	Минимальные квантификаторы: `*? +? ?? {n}? {n,}? {min,max}?`
☞ 415	Встроенный код: `(?{…})`
☞ 412	Динамическое регулярное выражение: `(??{…})`
Только в литералах регулярных выражений	
☞ 364 (C)	Интерполяция переменных: `$`*имя* `@`*имя*
☞ 365 (C)	Преобразование регистра следующего символа: `\l \u`

Таблица 7.2 (*окончание*)

☞ 365 (C)	Интервальное преобразование регистра: `\U \L … \E`
☞ 365 (C)	Литеральный текст: `\Q … \E`
☞ 365 (C)	Именованный символ Юникода: `\N`{*имя*}
(C) — может использоваться в символьном классе ①...⑥ — дополнительные замечания приводятся в тексте	

Ниже приводятся некоторые замечания по поводу таблицы.

① `\b` представляет символ Backspace (забой) только в символьных классах; за их пределами он представляет границу слова (☞ 180).

Восьмеричные коды могут состоять как из двух, так и из трех цифр.

Метапоследовательность ⌜`\xшестн`⌟ может состоять из двух цифр (а также из одной цифры, но в этом случае выводится предупреждение, если разрешена их выдача). Синтаксис ⌜`\x{шестн}`⌟ позволяет задавать шестнадцатеричные коды произвольной длины.

② Метасимволы `\w`, `\d` и `\s` полностью поддерживают Юникод.

Метасимвол `\s` не совпадает с ASCII-символом вертикальной табуляции (☞ 158).

③ Perl ориентируется на стандарт Юникода версии 4.1.0.

Алфавиты Юникода поддерживаются. В именах алфавитов и свойств допускается использование префикса '`Is`', но он необязателен (☞ 171). Имена блоков могут начинаться с префикса '`In`', но этот префикс обязателен только в том случае, если имя блока конфликтует с именем алфавита.

Поддерживается псевдосвойство ⌜`\p{L&}`⌟, а также псевдосвойства ⌜`\p{Any}`⌟, ⌜`\p{All}`⌟, ⌜`\p{Assigned}`⌟ и ⌜`\p{Unassigned}`⌟.

Поддерживаются длинные имена свойств (такие, как ⌜`\p{Letter}`⌟). Слова, образующие имя, разделяются пробелами или символами подчеркивания, а также могут писаться слитно — например, свойство ⌜`\p{Lowercase_Letter}`⌟ также может быть записано в виде ⌜`\p{LowercaseLetter}`⌟ или ⌜`\p{Lowercase•Letter}`⌟. Для соблюдения единства стиля я рекомендую использовать длинные имена из таблицы на с. 167.

Выражение ⌜`\p{^…}`⌟ дает тот же результат, что и ⌜`\p{…}`⌟.

④ Метасимволы границ слов полностью поддерживают Юникод.

⑤ Конструкции позиционной проверки могут содержать сохраняющие круглые скобки.

Ретроспективная проверка ограничивается подвыражениями, всегда совпадающими с текстом фиксированной длины.

⑥ Модификатор `/x` распознает только пропуски из набора ASCII.

Модификатор `/m` влияет только на символ новой строки, но не на полный список завершителей строк Юникода.

Модификатор `/i` корректно работает с символами Юникода.

Не все метасимволы обладают равными правами. Некоторые «метасимволы регулярных выражений» не поддерживаются механизмом регулярных выражений, а обрабатываются на стадии предварительной обработки литералов регулярных выражений.

Регулярные выражения — операнды и литералы

Последняя группа элементов регулярных выражений в табл. 7.2 озаглавлена «Только в литералах регулярных выражений». *Литералом регулярного выражения* называется часть команды `m/…/`, помеченная многоточием. Хотя в обыденной речи ее просто называют «регулярным выражением», в действительности часть между символами '/' обрабатывается по особым правилам. На жаргоне Perl говорится, что она интерпретируется как «строка в кавычках с учетом специфики регулярного выражения», а полученный результат передается механизму регулярных выражений. Стадия промежуточной обработки предоставляет программисту особые возможности при построении регулярных выражений.

Например, литералы регулярных выражений поддерживают *интерполяцию переменных*. Если переменная `$num` равна 20, при обработке фрагмента `m/:.{$num}:/` будет получено регулярное выражение `「:.{20}:」`. Интерполяция позволяет строить регулярные выражения во время работы программы. Другая возможность литералов регулярных выражений связана с автоматическим преобразованием регистра символов; например, все символы в конструкции `\U…\E` автоматически становятся прописными. Глупый пример: команда `m/abc\Uxyz\E/` работает с регулярным выражением `「abcXYZ」`. Я назвал этот пример глупым, потому что если вам понадобится выражение `「abcXYZ」`, его проще ввести вручную, но смысл преобразования регистра становится очевиден в сочетании с интерполяцией переменных: если переменная `$tag` содержит строку `"title"`, то команда `m{</\U$tag\E>}` сгенерирует выражение `「</TITLE>」`.

Какие же еще возможны варианты, кроме литералов? В качестве операнда могут использоваться строка или произвольное выражение. Например:

```
$MatchFleld = "^Subject:"; # Обычное присваивание строки
   ⋮
if ($text =~ $MatchFleld) {
     ⋮
```

Если переменная `$MatchField` используется в качестве операнда **`=~`**, ее содержимое интерпретируется как регулярное выражение. В этом случае «интерпретация» производится по стандартным правилам, поэтому, в отличие от выражений-литералов, интерполяция переменных и конструкции типа `\Q…\E` не поддерживается.

А теперь интересная подробность: если заменить

```
$text =~ $MatchFleld
```

на

```
$text =~ m/$MatchF1eld/
```

результат совершенно не изменится. В данном случае также используется литерал регулярного выражения, но он состоит из единственного элемента — интерполяции переменной `$MatchField`. *Содержимое* переменной, интерполируемой в литерале, *не* интерпретируется как литерал регулярного выражения, поэтому конструкции **`\U…\E`** и **`$`***var* в нем *не* распознаются (процесс обработки литералов регулярных выражений подробно описан на ☞ 367).

При многократном использовании регулярного выражения структура операндов (простой строковый литерал или интерполяция переменных) заметно влияет на эффективность программы. Эта тема рассматривается на с. 435.

Возможности литералов регулярных выражений

Литералы регулярных выражений обладают указанными ниже возможностями.

- **Интерполяция переменных.** Переменные, имена которых начинаются с **`$`** и **`@`**, интерполируются в регулярное выражение. Первые подставляются в виде простого скалярного значения, а вторые — в виде массива или среза, элементы которого разделяются пробелами (точнее, содержимым переменной **`$»`**, которая по умолчанию содержит пробел).

  В Perl хеши имеют префикс '**`%`**', однако вставка содержимого хеша в строку не имеет особого смысла, поэтому интерполяция переменных с префиксом **`%`** не поддерживается.

- **Именованные символы Юникода.** Если в программе присутствует директива «`use charnames ':full';`», на символы Юникода можно ссылаться по имени в синтаксисе `\N{`*имя*`}`. Например, последовательность `\N{LATIN SMALL LETTER SHARP S}` совпадает с «ß». Список символов Юникода, поддерживаемых Perl, находится в файле *UnicodeData.txt* каталога *unicore*:

  ```
  use Config;
  print "$Config{privlib}/unicore/UnicodeData.txt\n";
  ```

  Многие забывают включить директиву «`use charnames ':full';`» или поставить двоеточие перед '`full`' — в этом случае конструкция `\N{…}` работать не будет. Кроме того, `\N{…}` не работает при использовании перегрузки регулярных выражений (см. ниже).

- **Префиксы изменения регистра символа.** Специальные последовательности **`\l`** и **`\u`** преобразуют следующий символ к нижнему или верхнему регистру соответственно. Чаще всего они используются перед интерполяцией для приведения первого символа переменной к нужному регистру. Например, если переменная `$title` содержит строку «`mr.`», команда `m/…\u$title…/` создает выражение `/…Mr.…/`. Аналогичные возможности предоставляются функциями Perl `lcfirst()` и `ucfirst()`.

- **Интервальное преобразование регистра.** Специальные последовательности **`\L`** и **`\U`** преобразуют все дальнейшие символы соответственно к нижнему или верхнему регистру до конца литерала или до специальной метапоследовательности **`\E`**. Например, для приведенной выше переменной `$title` команда `m/…\U$title\E…/` создает регулярное выражение `/…MR.…/`. Аналогичные возможности предоставляются функциями Perl `lc()` и `uc()`.

  Префикс изменения регистра может объединяться с интервальным преобразованием регистра: команда `m/…\L\u$title\E…/` гарантирует, что строка будет выведена в виде `/…Mr.…/` независимо от регистра символов в исходной строке.

- **Блоки литерального текста.** Последовательность **`\Q`** включает режим «экранирования» метасимволов *регулярных выражений* (т. е. включения префикса `\`) до конца строки или до специальной метапоследовательности **`\E`**. Обратите внимание: экранируются метасимволы *регулярных выражений*, но не *литералов регулярных выражений* (интерполируемые переменные, **`\U`** и, конечно, завершающая метапоследовательность **`\E`**). Как ни странно, в этом режиме не экранируются символы `\` в неизвестных комбинациях (например, `\F` или `\H`). Даже в блоках `\Q…\E` для таких последовательностей выдаются предупреждения «unrecognized escape».

  На практике подобные ограничения не очень важны, поскольку конструкция `\Q…\E` чаще всего используется для экранирования интерполируемого текста, где она правильно экранирует *все* метасимволы. Например, если переменная

`$title` содержит строку «`Mr.`», команда `m/…\Q$title\E…/` создает выражение `/…Mr\.…/`, именно это нам и нужно, если в переменной `$title` хранится *текст* совпадения, а не *регулярное выражение*.

Блоки литерального текста особенно удобны при включении пользовательского ввода в регулярное выражение. Например, команда `m/…\Q$UserInput\E/i` выполняет поиск без учета регистра символов по тексту, хранящемуся в переменной `$UserInput` (именно по тексту, а не по регулярному выражению).

Возможности, аналогичные `\Q…\E`, также предоставляются функцией Perl `quotemeta()`.

- **Перегрузка.** Механизм *перегрузки* (*overloading*) позволяет выполнить предварительную обработку всех литеральных частей литерала регулярного выражения по вашему усмотрению. Концепция весьма любопытная, но в текущей реализации ее возможности сильно ограничены. Перегрузка подробно рассматривается, начиная со с. 428.

Выбор ограничителей

Одна из самых экстравагантных (но при этом полезных) особенностей синтаксиса Perl заключается в том, что программист может выбрать ограничители литералов регулярных выражений по своему усмотрению. Традиционно в качестве ограничителей используются символы слэша (`m/…/`, `s/…/…/` и `qr/…/`), но вместо них можно выбрать любой символ, не являющийся алфавитно-цифровым и не относящийся к категории пропусков. Приведу некоторые распространенные примеры.

```
m!…!      m{…}
m,…,      m<…>
s|…|…|    m[…]
qr#…#     m(…)
```

В правом столбце перечислены особые парные ограничители:

- При использовании парных ограничителей открывающий ограничитель отличается от закрывающего, причем допускается их вложение (т. е. внутри ограничителей могут находиться другие пары при условии правильного соответствия между открывающими и закрывающими ограничителями). Круглые и квадратные скобки очень часто встречаются в регулярных выражениях, поэтому конструкции `m(…)` и `m[…]` не так удобны, как остальные. В частности, с модификатором `/x` возможны фрагменты вида:

```
m{
    регулярное # комментарии
    выражение  # комментарии
}x;
```

Если регулярное выражение заключено в одну пару ограничителей, заменяющая строка заключается в другую пару (такую же, как первая, или другую — на ваше усмотрение). Примеры:

```
s{…}{…}
s{…}!…!
s<…>(…)
s[…]/…/
```

Эти две пары ограничителей могут отделяться друг от друга пропусками и комментариями. Дополнительная информация об операнде замены в операторе подстановки приводится на с. 400.

- (Только для оператора поиска.) Вопросительный знак в качестве ограничителя имеет специальное значение (подавление дополнительных совпадений). На практике этот синтаксис используется редко, но дополнительная информация о нем приводится в соответствующем разделе (☞ 387).

- Как упоминалось на с. 363, литерал регулярного выражения обрабатывается по правилам «строки в кавычках с учетом специфики регулярного выражения». Тем не менее если в качестве ограничителей используются апострофы, правила обработки меняются. В команде `m'…'` переменные *не* интерполируются, а конструкции оперативного изменения текста (например, `\Q…\E`) не работают, как и конструкция `\N{…}`. Синтаксис `m'…'` удобен при работе с регулярными выражениями, содержащими много символов `@`, чтобы их не приходилось специально экранировать.

Если в качестве ограничителя используется символ `/` или `?`, оператор поиска может записываться без буквы `m`. Другими словами, следующие две команды эквивалентны:

```
$text =~ m/…/;
$text =~ /…/;
```

Лично я предпочитаю всегда использовать букву `m`.

Порядок обработки литералов регулярных выражений

Большинство программистов «просто использует» только что описанные возможности литералов регулярных выражений, не вдаваясь в подробности их преобразования механизмами Perl в собственно регулярные выражения. Для них язык Perl хорош своей интуитивностью, но некоторые ситуации требуют более глубокого понимания. Далее приводится порядок обработки литералов регулярных выражений:

1. Поиск закрывающего ограничителя и чтение модификаторов (`/i` и т. д.). Если используется модификатор `/x`, об этом становится известно на дальнейших этапах обработки.
2. Интерполяция переменных.
3. Если используется перегрузка, каждая часть литерала передается перегружающей функции для обработки. Части разделяются интерполируемыми переменными; интерполированные значения в перегрузке не участвуют.

   Если перегрузка не используется, на этом этапе обрабатываются метапоследовательности `\N{…}`.
4. Обработка конструкций изменения регистра и т. д. (в том числе `\Q…\E`).
5. Результат передается механизму регулярных выражений.

Данное описание является руководством для программиста; оно не включает в себя подробностей внутренней обработки литералов регулярных выражений в Perl. Даже шаг 2 требует интерпретации метасимволов регулярных выражений, чтобы, например, подчеркнутая часть `this$|that$` не интерпретировалась как ссылка на переменную.

Модификаторы регулярных выражений

В операторы регулярных выражений Perl могут включаться модификаторы, расположенные сразу же после закрывающего ограничителя (например, модификатор **`i`** в операторах `m/…/i`, `s/…/…/i` и `qr/…/i`). Существует пять базовых модификаторов, поддерживаемых всеми операндами регулярных выражений Эти модификаторы перечислены в табл. 7.3.

Таблица 7.3. Базовые модификаторы, доступные во всех операторах регулярных выражений

`/i`	☞ 150	Игнорировать регистр символов при поиске
`/x`	☞ 151	Режим свободного форматирования
`/s`	☞ 152	Режим совпадения точки со всеми символами
`/m`	☞ 153	Расширенный режим привязки к границам строк
`/o`	☞ 435	Режим однократной компиляции

Первые четыре модификатора, описанные в главе 3, также могут использоваться внутри регулярных выражений в конструкциях общей модификации режима (☞ 182) и интервальной модификации (☞ 183). Если в регулярном выражении

используются «внутренние» модификаторы, а оператор содержит «внешние» модификаторы, «внутренние» модификаторы обладают более высоким приоритетом в той части регулярного выражения, которую они контролируют (иначе говоря, если модификатор был применен к некоторой части регулярного выражения, ничто не сможет отменить его действие).

Пятый базовый модификатор, `/o`, связан в основном с эффективностью. Он рассматривается позднее в этой главе, начиная со с. 435.

Если в операторе потребуется использовать сразу несколько модификаторов, сгруппируйте буквы и разместите их в произвольном порядке после завершающего ограничителя, каким бы он ни был[1]. Помните, что символ / не является частью модификатора — команда может записываться как в виде **`m/<title>/i`**, так и в виде **`m|<title>|i`**, **`m{<title>}i`** и даже **`m<<title>>i`**. Тем не менее во всех описаниях модификаторы обычно записываются с префиксом `/`, например «модификатор `/i`».

Perl'измы из области регулярных выражений

Существует множество общих концепций Perl, которые представляют интерес для нашего изучения регулярных выражений. В нескольких ближайших разделах рассматриваются две темы:

- **Контекст.** Многие функции и операторы Perl учитывают *контекст*, в котором они используются. Например, Perl ожидает, что в условии цикла `while` задается скалярная величина, а аргументы команды `print` задаются списком значений. Поскольку Perl позволяет выражениям «реагировать» на контекст их применения, идентичные выражения иногда порождают совершенно разные результаты.
- **Динамическая видимость.** Во многих языках программирования существуют концепции локальных и глобальных переменных. В Perl картина дополняется так называемой *динамической видимостью*. Динамическая область видимости временно «защищает» глобальную переменную; для этого сохраняется копия переменной, которая позднее автоматически восстанавливается. Эта любопытная концепция важна для нас, поскольку она влияет на `$1` и на другие связанные с поиском совпадений переменные.

[1] Поскольку модификаторы оператора поиска могут следовать в любом порядке, программисты часто тратят немало времени на то, чтобы добиться наибольшей выразительности. Например, `learn/by/osmosis` является допустимой командой (при условии, что у вас имеется функция `learn`). Слово `osmosis` составлено из модификаторов — повторение модификаторов оператора поиска (но не модификатора `/e` оператора подстановки!) допускается, хотя и не имеет смысла.

Контекст выражения

Понятие контекста играет важную роль в языке Perl, и в частности при использовании оператора поиска. Каждое выражение может относиться к одному из трех контекстов: *списковому*, *скалярному* или *неопределенному*. Контекст определяет тип значения, порождаемого выражением. Как нетрудно догадаться, в *списковом контексте* выражение создает список значений, а в *скалярном контексте* ожидается одна величина. Эти контексты встречаются очень часто и представляют основной интерес для нашего изучения регулярных выражений. В *неопределенном контексте* значение не создается.

Рассмотрим две команды присваивания:

```
$s  = выражение_1;
@a   = выражение_2;
```

Поскольку `$s` является простой скалярной переменной (т. е. содержит одну величину, а не список), *`выражение_1`*, каким бы оно ни было, принадлежит к скалярному контексту. Аналогично, поскольку переменная `@a` представляет собой массив и содержит список значений, *`выражение_2`* принадлежит к списковому контексту. Хотя выражения могут быть одинаковыми, в зависимости от контекста они могут возвращать абсолютно разные значения и вызывать различные побочные эффекты. Подробности зависят от специфики выражения.

Например, функция `localtime` в списковом контексте возвращает список значений, представляющих текущий год, месяц, дату, час и т. д. В скалярном контексте та же функция вернет текстовое представление текущего времени вида '`Mon Jan 20 22:05:15 2003`'.

Другой пример: оператор файлового ввода-вывода (например, `<MYDATA>`) в скалярном контексте возвращает следующую строку, а в списковом контексте — список всех (оставшихся) строк файла.

Многие конструкции Perl обрабатываются в соответствии с контекстом, и операторы регулярных выражений не являются исключением. Например, оператор `m/…/` в одних ситуациях возвращает простую логическую величину «истина/ложь», а в других — список совпадений. Подробности будут приведены ниже.

Преобразование типа

Не все выражения учитывают контекст своего применения, поэтому в Perl действуют особые правила, которые определяют, что должно происходить при использовании выражения в контексте, не полностью соответствующем типу порождаемого

значения. Если вдруг потребуется «заткнуть круглое отверстие квадратной пробкой», Perl преобразует в соответствии с необходимостью тип выражения. Если обработка выражения в списковом контексте дает скалярный результат, автоматически создается список, состоящий из одного элемента. Таким образом, команда **`@a = 42`** эквивалентна **`@a = (42)`**.

С другой стороны, общих правил преобразования списка в скаляр не существует. Например, для литерального списка:

```
$var = ($this, &is, 0xA, 'list');
```

переменной `$var` присваивается последний элемент, `'list'`. В команде вида **`$var = @array`** переменной `$var` присваивается длина массива.

Динамическая видимость и последствия совпадения регулярных выражений

Два уровня видимости переменных Perl (глобальные и закрытые), а также понятие *динамической видимости* достаточно важны сами по себе, но они представляют особый интерес для изучения регулярных выражений. Это связано с тем, каким образом программа получает доступ к информации о совпадении. В следующих разделах описаны эти концепции и их связь с регулярными выражениями.

Глобальные и закрытые переменные

В Perl существует два типа переменных: *глобальные* и *закрытые* (private). Закрытые переменные объявляются директивой `my(…)`. Глобальные переменные вообще не объявляются, а просто начинают существовать с момента применения. Глобальные переменные доступны в любой точке программы, а закрытые переменные с точки зрения лексики остаются доступными до конца содержащего их (объемлющего) блока. Другими словами, с закрытыми переменными может работать только код Perl, расположенный между соответствующим объявлением `my` и концом программного блока, внутри которого расположено объявление `my`.

Использовать глобальные переменные обычно нежелательно, кроме особых случаев, к которым относятся бесчисленные специальные переменные типа `$1`, `$_` и `@ARGV`. Обычные пользовательские переменные являются глобальными, если они не были объявлены с ключевым словом `my`, даже если они «кажутся» закрытыми. Perl позволяет делить имена глобальных переменных на группы, называемые *пакетами*, но сами переменные все равно остаются глобальными. Для ссылок на глобальную переменную `$Debug` из пакета `Acme::Widget` может использоваться *полностью уточ-*

ненное имя `$Acme::Widget::Debug`, но независимо от вида ссылки она остается той же глобальной переменной. Если в программу включена директива **`use strict;`**, на все (не специальные) глобальные переменные необходимо ссылаться либо по полностью уточненным именам, либо по именам, объявленным с ключевым словом `our` (ключевое слово `our` объявляет *имя*, а не новую переменную; за подробностями обращайтесь к документации Perl).

Значения переменных с динамической видимостью

Динамическая видимость — интересная концепция, отсутствующая во многих языках программирования. О том, какое отношение она имеет к регулярным выражениям, будет рассказано ниже. Речь идет о том, что Perl может сохранить значение глобальной переменной, которая должна измениться внутри блока, и автоматически восстановить исходное значение копии в момент завершения блока. Сохранение копии называется *созданием новой динамической области видимости*, или *локализацией*.

В частности, данная возможность часто используется для временного изменения некоего глобального состояния, хранящегося в глобальной переменной. Допустим, вы используете пакет `Acme::Widget` с флагом режима отладки, устанавливаемым при помощи глобальной переменной `$Acme::Widget::Debug`. Временное включение отладочного режима может осуществляться конструкциями вида:

```
⋮
{
    local($Acme::Widget::Debug) = 1; # Включить отладочный режим
    # работать с Acme::Widget в отладочном режиме
    ⋮
}
# Переменная $Acme::Widget::Debug возвращается к предыдущему состоянию
⋮
```

Имя функции `local` было выбрано на редкость неудачно — эта функция создает только новую динамическую область видимости. Давайте сразу договоримся, что *вызов* `local` *не создает новую переменную*. Если у вас имеется глобальная переменная, `local` выполняет три операции:

1. Сохранение внутренней копии значения переменной.
2. Копирование нового значения в переменную (`undef` или значения, указанного при вызове `local`).
3. Восстановление исходного значения переменной при выходе за пределы блока, в котором находится вызов `local`.

Таким образом, «локальность» в данном случае относится лишь к времени, в течение которого будут существовать изменения, внесенные в переменную. Локализованное значение существует лишь во время выполнения блока. Если вызвать в этом блоке какую-то функцию, эта функция «увидит» локализованное значение (ведь переменная по-прежнему остается глобальной). Единственное отличие от использования нелокализованной глобальной переменной заключается в том, что после завершения объемлющего блока автоматически восстанавливается предыдущее значение переменной.

Автоматическое сохранение и восстановление значения переменной — вот, в сущности, и все, что относится к вызову `local`. Хотя при использовании local часто возникают недоразумения, на самом деле эта функция эквивалентна фрагменту, приведенному в правом столбце табл. 7.4.

Для удобства допускается присваивание значений конструкции `local($SomeVar)`; это в точности эквивалентно присваиванию значения `$SomeVar` вместо присваивания `undef`. Кроме того, можно опустить круглые скобки, чтобы форсировать скалярный контекст.

Таблица 7.4. Смысл функции local

Обычный код Perl	Эквивалентный фрагмент

`{`
`  local($SomeVar); # Сохранить копию`
 	
`  $SomeVar = 'My Value';`
`  ⋮`
`} # Автоматическое восстановление`
`  # SomeVar`
	`{`
	`  my $TempCopy = $SomeVar;`
	`  $SomeVar = undef;`
	
	`  $SomeVar = 'MyValue';`
	`  ⋮`
	`  $SomeVar = $TempCopy;`
	`}`

Предположим, вам приходится вызывать функцию из небрежно написанной библиотеки. Функция генерирует множество предупреждений «Use of uninitialized warnings». Вы, как и все порядочные программисты Perl, используете ключ `-w`, но автор библиотеки этого, видимо, не сделал. Предупреждения вызывают у вас нарастающее раздражение, но что делать, если изменить библиотеку невозможно, — полностью отказаться от использования `-w`? Можно воспользоваться программным флагом выдачи предупреждений `$^W` (имя переменной `^W` может состоять из двух символов, «крышка» и `'W'`, или из одного символа `Control+W`):

```
{
    local $^W = 0; # Отключить выдачу предупреждений.
```

```
    UnrulyFunction(…);
}
# При выходе из блока восстанавливается исходное значение $^W.
```

Вызов `local` сохраняет внутреннюю копию предыдущего значения глобальной переменной `$^W`, каким бы оно ни было. Затем той же переменной `$^W` присваивается новое нулевое значение. При выполнении `Unru lyFunction` Perl проверяет переменную `$^W`, находит присвоенный ей ноль и не выдает предупреждений. При возвращении из функции переменная по-прежнему равна нулю.

Пока все идет так, словно никакого вызова `local` не было. Но при выходе из блока после возвращения из функции `UnrulyFunction` восстанавливается сохраненное значение `$^W`. Наше изменение было временным и действовало лишь *на время* выполнения блока. Вы можете вручную добиться того же эффекта, сохраняя и восстанавливая значение этой переменной (см. табл. 7.4), но функция `local` делает это за вас.

Для полноты картины давайте посмотрим, что произойдет, если вместо `local` использовать `my`[1]. Ключевое слово `my` создает *новую переменную*, которая первоначально имеет неопределенное значение. Эта переменная видна только в том лексическом блоке, в каком она была объявлена (т. е. в программном коде между `my` и концом объемлющего блока). Однако появление новой переменной никак не сказывается на других переменных, в том числе и на существующих глобальных переменных с тем же именем. Вновь созданная переменная останется невидимой для программы, в том числе и внутри `UnrulyFunction`. В приведенном фрагменте новой переменной `$^W` немедленно присваивается ноль, но эта переменная нигде не используется, поэтому все усилия оказываются напрасными (во время выполнения `UnrulyFunction` и принятия решения о выдаче предупреждений Perl обращается к глобальной переменной `$^W`, которая никак не связана с переменной, созданной нами).

Аналогия с пленкой

Для `local` существует одна полезная аналогия: вы как бы закрываете переменную пленкой, на которой можно временно записать изменения. Все, кто работает с переменной, например функция или обработчик сигнала, видят ее новое значение. Старое значение закрывается до выхода из блока. В этот момент пленка автоматически убирается, и вместе с ней исчезают все изменения, внесенные после вызова `local`.

Такая аналогия гораздо ближе к реальности, чем исходное описание с «созданием внутренней копии». При вызове `local` Perl не создает копию, а лишь ставит новую

[1] Perl не позволяет использовать `my` с именами специальных переменных, поэтому сравнение чисто теоретическое.

величину на более раннюю позицию в списке значений, проверяемых при обращении к переменной (т. е. «закрывает» оригинал). При выходе из блока удаляются все «закрывающие» значения, занесенные в список после входа в блок. При вызове `local` новая динамическая область видимости создается программистом, но вот главная причина, по которой мы рассматриваем локализацию: **служебные переменные, используемые при работе с регулярными выражениями, локализуются автоматически.**

Динамическая область видимости и побочные эффекты регулярных выражений

Какое отношение все эти разговоры о динамической области видимости имеют к регулярным выражениям? Самое прямое. В результате успешного совпадения некоторым переменным автоматически присваиваются значения — своего рода побочный эффект. К числу этих переменных, подробно описанных в следующем разделе, принадлежат, например, `$&` (текст совпадения) и `$1` (текст, совпавший с первым подвыражением в круглых скобках). Для этих переменных динамическая область видимости создается *автоматически* при входе в каждый блок.

Чтобы понять, для чего это нужно, примите во внимание, что каждый вызов функции определяет новый блок и, следовательно, для таких переменных создается новая динамическая область видимости. Поскольку значения, существовавшие перед входом в блок, восстанавливаются при выходе из него (т. е. при возврате из функции), функция не изменяет значения, видимые вызывающей стороне.

В качестве примера рассмотрим следующий фрагмент:

```
if (m/(…)/)
{
    DoSomeOtherStuff();
    print "the matched text was $1.\n";
}
```

Поскольку для переменной `$1` новая динамическая область действия создается автоматически при входе в каждый блок, данному фрагменту не важно, изменяет функция `DoSomeOtherStuff` значение `$1` или нет. Все изменения, вносимые в `$1` этой функцией, ограничиваются блоком, определяемым этой функцией, или, возможно, некоторым его вложенным блоком. Таким образом, они не могут повлиять на значение, видимое в команде `print` после возвращения из функции.

Автоматическое создание динамической области видимости приносит пользу и в менее очевидных ситуациях:

```
If ($result =~ m/ERROR=(.*)/) {
    warn "Hey, tell $Config{perladmin} about $1!\n";
}
```

(В стандартном библиотечном модуле `Config` определяется ассоциативный массив `%Config`, элемент которого `$Config{perladmin}` содержит адрес электронной почты локального Perl-мастера.) Если бы значение переменной `$1` не сохранялось, этот код преподнес бы вам сюрприз. Дело втом, что `%Config` в действительности является *связанной* переменной; это означает, что при любой ссылке на эту переменную происходит автоматический вызов функции. В данном случае функция, осуществляющая выборку нужного значения для `$Config{…}`, использует регулярное выражение. Поскольку эта операция поиска выполняется между вашей операцией поиска и выводом `$1`, при отсутствии динамической области видимости она испортила бы значение `$1`, которое вы собирались использовать. К счастью, любые изменения, внесенные в функции `$Config{…}`, надежно изолируются благодаря динамической видимости.

Динамическая и лексическая видимость

При продуманном использовании динамическая видимость приносит немалую пользу, но бессистемное применение `local` может превратить сопровождение кода в сущий кошмар. Читателю программы будет невероятно сложно разобраться во взаимодействиях между `local`, вызовами функций и ссылками на локализованные переменные.

Как упоминалось выше, объявление `my(…)` создает закрытую переменную с *лексической видимостью*. Лексическая видимость закрытой переменной является противоположностью глобальной видимости глобальных переменных, однако она не имеет отношения к динамической видимости (если не считать того, что к переменным `my` нельзя применять `local`). Помните: `local` — это *действие*, а `my` — это и действие *и*, что важно, объявление переменной.

Специальные переменные, изменяемые при поиске

Успешное выполнение поиска или замены задает значения набору глобальных, доступных только для чтения переменных, для которых автоматически создается новая динамическая область видимости. Значения этих переменных *никогда* не изменяются в том случае, если совпадение не найдено, и *всегда* изменяются в случае, если совпадение находится. В некоторых случаях переменным может быть присвоена пустая строка (т. е. строка, не содержащая ни одного символа) или неопределенное значение (похожее на пустую строку, но принципиально отличающееся от нее). Примеры приведены в табл. 7.5.

Таблица 7.5. Примеры использования специальных переменных, значения которых задаются после совпадения

После выполнения команды

```
                                  1 2                2  3    4     4  3 1
"Pi is 3.14159, roughly" =~ m/\b((tasty|fattening)|(\d+(\.\d*)?))\b/;
```

специальным переменным будут присвоены следующие значения:

Переменная	Описание	Значение
`$``	Текст перед совпадением	`Pi•is•`
`$&`	Текст совпадения	`3.14159`
`$'`	Текст после совпадения	`,•roughly`
`$1`	Текст, совпавший с первой парой круглых скобок	`3.14159`
`$2`	Текст, совпавший со второй парой круглых скобок	*undef*
`$3`	Текст, совпавший с третьей парой круглых скобок	`3.14159`
`$4`	Текст, совпавший с четвертой парой круглых скобок	`.14159`
`$+`	Содержимое переменной $1, $2 и т. д. с максимальным номером	`.14159`
`$^N`	Содержимое переменной $1, $2 и т. д., соответствующей последней закрытой паре круглых скобок	`3.14159`
`@-`	Массив начальных индексов совпадений в целевом тексте	`(6, 6, undef, 6, 7)`
`@+`	Массив конечных индексов совпадений в целевом тексте	`(13, 13, undef, 13, 13)`

Вот более подробное описание переменных, получивших значения после совпадения.

`$&` Копия текста, успешно совпавшего с регулярным выражением. Использовать эту переменную (так же, как и переменные `$`` и `$'`, описанные далее) не рекомендуется по соображениям эффективности (подробнее об этом на с. 445). В случае успешного совпадения переменной `$&` никогда не присваивается неопределенное значение, хотя может быть присвоена пустая строка.

**`$``** Копия целевого текста, предшествующего началу совпадения (т. е. расположенного слева от него). В сочетании с модификатором `/g` иногда бывает нужно, чтобы в переменной `$`` хранился текст от начальной *позиции поиска*, а не от начала строки. В случае успешного совпадения переменной `$`` никогда не присваивается неопределенное значение.

**$'** Копия целевого текста, следующего после совпадения (т. е. расположенного справа от него). После успешного совпадения строка `"$`$&$'"` всегда представляет собой копию исходного целевого текста[1]. В случае успешного совпадения переменной `$`` никогда не присваивается неопределенное значение.

$1,$2,$3,...

Текст, совпавший с первой, второй, третьей и т. д. парой сохраняющих круглых скобок (обратите внимание: переменная `$0` в список не входит — в ней хранится копия имени сценария, и эта переменная не имеет отношения к регулярным выражениям). Если переменная относится к паре скобок, не существующей в регулярном выражении или не задействованной в совпадении, ей гарантированно присваивается неопределенное значение.

Эти переменные используются после совпадения, в том числе и в строке замены оператора `s/.../.../`. Кроме того, они могут использоваться во встроенном коде или в конструкциях динамических регулярных выражений (☞ 411). В других случаях в самом регулярном выражении они не используются (для этого существует ⌜`\1`⌟ и другие метасимволы из того же семейства). (Раздел «Можно ли использовать `$1` в регулярном выражении?» на с. 381.)

Присваивание значений этим переменным наглядно демонстрируется различиями между ⌜`(\w+)`⌟ и ⌜`(w)+`⌟. Оба регулярных выражения совпадают с одним и тем же текстом, но текст, сохраненный в круглых скобках, будет разным. Допустим, выражения применяются к строке '`tubby`'. Для первого выражения переменной `$1` будет присвоена строка '`tubby`', а для второго — символ '`y`': квантификатор `+` находится вне круглых скобок, поэтому при каждой итерации текст сохраняется заново.

Кроме того, необходимо понимать различие между ⌜`(x)?`⌟ и ⌜`(x?)`⌟. В первом случае круглые скобки и заключенный в них текст являются необязательным элементом, поэтому переменная `$1` либо равна `x`, либо имеет неопределенное значение. Однако для выражения ⌜`(x?)`⌟ в скобки заключено обязательное совпадение — необязательным является его содержимое. Если все регулярное выражение совпадает, то и содержимое с чем-то совпадет, хотя это «что-то» может быть «ничем» — ⌜`x?`⌟ это разрешает. Таким образом, для ⌜`(x?)`⌟ допустимыми значениями `$1` являются `x` и пустая строка. Некоторые примеры приводятся в следующей таблице.

[1] В действительности, если исходный текст представляет собой переменную с неопределенным значением, но совпадение будет успешно найдено (маловероятная, но возможная ситуация), результат `"$`$&$'"` будет представлять собой пустую строку, а не неопределенную величину. Это единственное исключение из этого правила.

Команда	Значение $1	Команда	Значение $1
"::" =~ m/:(A?):/	пустая строка	"::" =~ m/:(\w*):/	пустая строка
"::" =~ m/:(A)?:/	undef	"::" =~ m/:(\w)*:/	undef
":A:" =~ m/:(A?):/	A	":Word:" =~ m/:(\w*):/	Word
":A:" =~ m/:(A)?:/	A	":Word:" =~ m/:(\w)*:/	d

Если скобки добавляются только для сохранения, как здесь, выбор определяется той семантикой, которая вам нужна. В рассмотренных примерах добавление круглых скобок не влияет на общее совпадение (все выражения совпадают с одним и тем же текстом), а различия между ними сводятся к побочному эффекту — значению, присвоенному `$1`.

`$+` Копия значения `$1`, `$2`, ... (с максимальным номером), присвоенного при поиске совпадения. Часто используется во фрагментах вида:

```
$url =~ m{
   href \s* = \s*  # Найти "href = ", затем...
   (?: "([^"]*)"    # значение в кавычках, или...
     | '([^']*)'    # значение в апострофах, или...
     | ([^'"<>]+) ) # свободное значение.
}ix;
```

Без переменной `$+` вам пришлось бы проверить каждую из переменных `$1`, `$2` и `$3` и определить, какая из них отлична от `undef`.

Если в выражении нет сохраняющих круглых скобок (или они не задействованы в совпадении), переменной присваивается неопределенное значение.

`$^N`

Копия значения `$1`, `$2`, ..., соответствующего последней закрытой паре круглых скобок, совпавшей в процессе поиска (т. е. переменной `$1`, `$2`, ..., ассоциированной с последней из закрывающих скобок). Если регулярное выражение не содержит круглых скобок (или ни одна пара скобок не была задействована при поиске), переменной присваивается неопределенное значение. Хороший пример использования этой переменной приведен на с. 431.

`@-` и **`@+`**

Массивы начальных и конечных смещений (индексов символов в строке) в целевом тексте. Странные имена этих массивов несколько усложняют работу с ними. Первый элемент массива относится ко всему совпадению; иначе говоря, первый элемент массива `@-` (доступный как `$-[0]`) определяет смещение от начала целевого текста, с которого начинается совпадение. Например, после выполнения фрагмента

```
$text = "Version 6 coming soon?";
   ⋮
$text =~ m/\d+/;
```

значение `$-[0]` будет равно 8, поскольку совпадение начинается с девятого символа целевой строки (в Perl индексация элементов в массивах начинается с нуля).

Первый элемент массива `@+` (доступный как `$+[0]`) определяет смещение конца всего совпадения. В нашем примере он будет равен 9, поскольку совпадение завершается на девятом символе от начала строки. Следовательно, выражение **`substr($text, $-[0], $+[0] -$-[0])`** эквивалентно `$&`, если переменная `$text` не модифицировалась, но при этом не приводит к затратам, связанным с применением `$&` (☞ 426). Простой пример использования `@-`:

```
1 while $line =~ s/\t/' ' x (8  $-[0] % 8)/e;
```

Эта команда заменяет символы табуляции в строке соответствующим количеством пробелов[1].

Остальные элементы массивов содержат начальное и конечное смещение для каждой из сохраненных подгрупп. Так, пара `$-[1]` и `$+[1]` определяет смещения для подвыражения `$1`, пара `$-[2]` и `$+[2]` — для подвыражения `$2` и т. д.

`$^R`

Переменная используется только во встроенном коде или в конструкциях динамических регулярных выражений и не имеет смысла за пределами регулярного выражения. В ней хранится результат последней исполняемой части встроенного кода с единственным исключением: часть *if* условных конструкций `(?`*if then*`|`*else*`)` (☞ 188) не изменяет состояние переменной `$^R`. Переменная автоматически локализуется для каждой части совпадения, поэтому значения, присвоенные в результате *выполнения* кода, который был позднее отменен из-за возврата, должным образом «забываются». Иначе говоря, в переменной хранится «самое свежее» значение для маршрута, по которому механизм пришел к текущему потенциальному совпадению.

В случае глобального применения регулярного выражения с модификатором `/g` значения этих переменных при каждой итерации присваиваются заново. В частности, это объясняет, почему переменная `$1` может использоваться в операнде

[1] У приведенного примера есть одно существенное ограничение: он работает только с «традиционным» текстом. При использовании многобайтовых кодировок результат окажется некорректным, поскольку один символ может занимать две позиции. Проблемы также возникают и с представлением в Юникоде таких символов, как à (☞ 148).

замены s/.../.../g и почему при каждой итерации она представляет новый фрагмент текста.

Можно ли использовать $1 в регулярном выражении?

В документации Perl неоднократно указывается, что ⌜\1⌟ не может использоваться в качестве обратной ссылки вне регулярного выражения (вместо этого следует использовать переменную $1). Переменная $1 ссылается на строку статического текста, совпавшего в результате уже завершенной операции поиска. С другой стороны, ⌜\1⌟ — это полноценный метасимвол регулярного выражения, который ссылается на текст, идентичный совпавшему с первым подвыражением в круглых скобках на тот момент, *когда управляемый регулярным выражением механизм НКА достигает* ⌜\1⌟. Текст, совпадающий с ⌜\1⌟, может измениться в процессе поиска совпадения с происходящими в НКА смещениями начальной позиции и возвратами.

С этим вопросом связан другой: можно ли использовать $1 в регулярном выражении-операнде? Эти переменные часто используются в *исполняемых* частях встроенного кода и динамических регулярных выражениях (☞ 410), но в других случаях их присутствие в регулярном выражении не имеет особого смысла. Переменная $1 в операнде-выражении обрабатывается точно так же, как и любая другая переменная: ее значение интерполируется перед началом операции поиска или замены. Таким образом, с позиций регулярного выражения значение $1 никак не связано с текущим совпадением, а наследуется от предыдущего совпадения.

Оператор qr/.../ и объекты регулярных выражений

В главах 2 (☞ 111) и 6 (☞ 350) уже упоминался унарный оператор qr/.../, который получает регулярное выражение-операнд и возвращает *объект регулярного выражения*. Полученный объект может использоваться в качестве операнда при последующих операциях поиска, замены или разбиения с помощью функции split, а также в качестве компонента регулярного выражения.

Объекты регулярных выражений чаще всего применяются для инкапсуляции выражений и их последующего использования при построении конструкций более высокого уровня, а также для повышения эффективности (тема управления компиляцией регулярных выражений рассматривается ниже).

Как упоминалось на с. 366, имеется возможность выбирать другие ограничители, например qr{...} или qr!...!. Кроме того, поддерживаются все базовые модификаторы /i, /x, /s, /m и /o.

Построение и использование объектов регулярных выражений

Рассмотрим следующий фрагмент, с небольшими изменениями позаимствованный из главы 2 (☞ 111):

```
my $HostnameRegex = qr/[-a-z0-9]+(?:\.[-a-z0-9]+)*\.(?:com|edu|info)/i;

my $HttpUrl = qr{
    http:// $HostnameRegex \b           # имя хоста
    (?:
        / [-a-z0-9_:\@&?=+,.!/~*'%\$]*  # Необязательный путь
            (?<![.,?!])             # Не может заканчиваться символами [.,?!]
    )?
}ix;
```

Первая строка инкапсулирует упрощенное выражение для имени хоста в объект регулярного выражения и сохраняет его в переменной `$HostnameRegex`. Далее полученный объект используется при построении объекта регулярного выражения для поиска HTTP URL, сохраняемого в переменной `$HttpUrl`. Существует много вариантов использования созданных объектов, например простое оповещение:

```
if ($text =~ $HttpUrl) {
   print "There is a URL\n";
}
```

или перечисление всех HTTP URL во фрагменте текста:

```
while ($text =~ m/($HttpUrl)/g) {
    print "Found URL: $1\n";
}
```

А теперь посмотрим, что произойдет, если заменить определение `$Host nameRegex` следующим фрагментом из главы 5 (☞ 266):

```
my $HostnameRegex = qr{
    # Один или более компонентов, разделенных точками...
    (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. )*
    # За которыми следует завершающий суффикс...
    (?: com|edu|gov|int|mil|net|org|biz|info|…|aero|[a-z][a-z] )
}xi;
```

По своей семантике новое определение не отличается от предыдущего (оно не содержит якорей ⌜^⌟ и ⌜$⌟, а также сохраняющих круглых скобок), поэтому переход на новое, более мощное выражение осуществляется простой заменой определения

компонента. В результате мы получаем новую переменную $HtppUrl с расширенными возможностями.

Фиксация режимов поиска

Оператор qr/.../ поддерживает базовые модификаторы поиска, перечисленные на с. 368. После построения объекта регулярного выражения вы не сможете изменить действующие в нем режимы поиска или назначить новые режимы, даже если объект используется в конструкции m/.../ с собственными модификаторами. Например, следующий фрагмент **не** работает:

```
my $WordRegex = qr/\b \w+ \b/; # Модификатор /x отсутствует!
   ⋮
if ($text =~ m/^($WordRegex)/x) {
    print "found word at start of text: $1\n";
}
```

Предполагается, что модификатор /x изменит режим поиска для выражения $WordRegex, но на самом деле режим не изменяется, поскольку модификаторы (или факт их отсутствия) фиксируются оператором qr/.../ при *создании* $WordRegex. Следовательно, все необходимые модификаторы должны быть указаны на этой стадии.

Правильная версия этого примера должна выглядеть так:

```
my $WordRegex = qr/\b \w+ \b/x; # Работает!
   ⋮
if ($text =~ m/^($WordRegex)/) {
    print "found word at start of text: $1\n";
}
```

Сравните с исходным вариантом:

```
my $WordRegex = '\b \w+ \b'; # Обычное присваивание строки
   ⋮
if ($text =~ m/^($WordRegex)/x) {
    print "found word at start of text: $1\n";
}
```

Исходный вариант работает, хотя при создании переменной $WordRegex с ней не ассоциировались никакие модификаторы. Дело в том, что $WordRegex — обычная строковая переменная, интерполируемая в литерал регулярного выражения m/.../. По ряду причин работать со строковыми представлениями менее удобно, чем с объектами регулярных выражений (например, в нашем примере нужно запомнить, что переменная $WordRegex должна применяться с модификатором /x).

Впрочем, даже при работе со строковым представлением некоторые проблемы решаются при помощи *интервальной модификации режима* при создании строки:

```
my $WordRegex = '(?x:\b \w+ \b)' ;    # Обычное присваивание строки
   ⋮
if ($text =~ m/^($WordRegex)/) {
    print "found word at start of text: $1\n";
}
```

После интерполяции строки в литерале `m/…/` механизму регулярных выражений будет передано выражение `⌜^((?x:\b•\w+•\b))⌟`, которое работает именно так, как требовалось.

Нечто похожее происходит при создании регулярного выражения, если не считать того, что объект регулярного выражения всегда явно задает состояние каждого модификатора (`/i`, `/x`, `/m` и `/s`) — так, оператор **`qr/\b•\w+•\b/x`** создает выражение `⌜(?x-ism:\b•\w+•\b)⌟`. Обратите внимание: в интервале `⌜(?x-ism:…)⌟` режим модификатора `/x` включен, а режимы модификаторов `/i`, `/s` и `/m` — отключены. Таким образом, оператор `qr/…/` фиксирует любое состояние модификатора (как установленное, так и сброшенное).

Просмотр содержимого объектов регулярных выражений

В предыдущем абзаце говорилось о том, что содержимое объекта регулярного выражения логически заключается в интервал модификации режима (вроде `⌜(?x-ism:…)⌟`). В этом нетрудно убедиться — если объект регулярного выражения находится там, где должна находиться строка, Perl использует текстовое представление объекта. Пример:

```
% perl -e 'print qr/\b \w+ \b/x, "\n"'
(?x-ism:\b \w+ \b)
```

А вот как выглядит содержимое переменной `$HttpUrl` со с. 382:

```
(?ix-sm:
    http:// (?ix-sm:
    # Один или более компонентов, разделенных точками...
    (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. )*
    # За которыми следует завершающий суффикс...
    (?: com|edu|gov|int|mil|net|org|biz|info|…|aero|[a-z][a-z] )
) \b         # хост
    (?:
        / [a-z0-9_:\@&?=+,.!/~*'%\$]* # Необязательный путь
```

```
        (?<![.,?!]) # Не может заканчиваться символами [.,?!]
    )?
)
```

Возможность преобразования объекта регулярного выражения в строку очень полезна для отладки.

Объекты регулярных выражений и повышение эффективности

Одна из главных причин использования регулярных выражений — возможность управления компиляцией регулярного выражения во внутреннее представление. Общие проблемы компиляции регулярных выражений кратко обсуждались в главе 6. Более подробно данная тема рассматривается в разделе «Компиляция регулярных выражений, модификатор `/o`, `qr/…/` и эффективность» (☞ 435).

Оператор поиска

Базовый оператор поиска

```
$text =~ m/выражение/
```

занимает центральное место в работе с регулярными выражениями в Perl. Поиск совпадений в Perl реализуется в виде *оператора*, который при вызове получает два *операнда* (целевую строку и регулярное выражение) и возвращает результат.

Способ проведения поиска и тип возвращаемого значения зависят от контекста, в котором производится поиск (☞ 370), и других факторов. Оператор поиска обладает достаточной гибкостью — он может использоваться для поиска совпадений регулярного выражения в строке, извлечения данных из текста и даже разбора строк на компоненты в сочетании с другими операторами поиска. Впрочем, широта возможностей несколько усложняет его изучение. Мы рассмотрим следующие темы.

- Определение операнда регулярного выражения.
- Определение модификаторов поиска и их смысл.
- Определение целевой строки, в которой осуществляется поиск.
- Побочные эффекты поиска.
- Значение, возвращаемое в результате поиска.
- Внешние факторы, влияющие на поиск.

Обобщенная форма оператора поиска выглядит так:

```
Строка =~ выражение
```

У оператора имеется несколько сокращенных форм, причем любая его часть в той или иной форме является необязательной. Примеры всех форм будут приведены в этом разделе.

Операнд регулярное выражение

Операнд регулярное выражение может задаваться литералом или объектом регулярного выражения (вообще говоря, он может быть строкой или произвольным выражением, но от этого не легче). Если операнд задается литералом регулярного выражения, в оператор также могут включаться модификаторы поиска.

Литерал регулярного выражения

Операнд регулярное выражение чаще всего задается литералом регулярного выражения в конструкции `m/…/` или только `/…/`. Начальный символ `m` необязателен, если ограничителями регулярного выражения являются символы `/` или `!` (вопросительные знаки имеют специальную интерпретацию, о которой будет рассказано ниже). Ради единства стиля я рекомендую всегда использовать символ `m`, даже если он необязателен. Как было сказано выше, при наличии символа `m` вы можете выбрать нужный тип ограничителя (☞ 366).

Если операнд задается в виде литерала, в команду могут включаться любые базовые модификаторы, перечисленные на с. 368. Оператор поиска также поддерживает два дополнительных модификатора, `/g` и `/c`, которые будут описаны ниже.

Объект регулярного выражения

Операнд может задаваться объектом регулярного выражения, созданным оператором `qr/…/`. Пример:

```
my $regex = qr/выражение/;
    ⋮
if ($text =~ $regex) {
    ⋮
```

Объект регулярного выражения может использоваться с конструкцией `m/…/`. Существует особый случай: если литерал регулярного выражения состоит *только* из интерполируемого объекта регулярного выражения, результат будет в точности

таким, как если бы этот объект использовался напрямую. Команду `if` из приведенного примера можно записать в виде:

```
if ($text =~ m/$regex/) {
    ⋮
```

Такая запись удобна, поскольку она выглядит более знакомо и позволяет использовать модификатор `/g` с объектом регулярного выражения (также возможно использование других модификаторов, поддерживаемых `m/…/`, но это бессмысленно, потому что они ни при каких условиях не отменяют режимы, зафиксированные в объекте регулярного выражения ☞ 383).

Регулярное выражение по умолчанию

Если регулярное выражение не задано, как в команде `m//` (или `m/$SomeVar/`, где переменная `$SomeVar` содержит пустую строку или имеет неопределенное значение), Perl заново использует *последнее успешно использованное регулярное выражение в объемлющей динамической области видимости*. Раньше регулярные выражения по умолчанию были полезны по соображениям эффективности, но с появлением объектов регулярных выражений они утратили актуальность (☞ 381).

Специальный поиск `?…?`

Ограничители `?…?` интерпретируются оператором поиска особым образом. Они активизируют довольно экзотический режим: после успешного совпадения `m?…?` дальнейшие совпадения `m?…?` находиться не будут до тех пор, пока в том же пакете не будет вызвана функция `reset`. В электронной документации Perl версии 1 говорилось, что эта возможность «является полезной оптимизацией в тех случаях, когда вы ищете совпадение в каждом файле из заданного набора», но я еще ни разу не видел, чтобы она использовалась в современном Perl.

При использовании ограничителей `?…?` (как и для `/…/`) символ `m` необязателен: `?…?` интерпретируется как `m?…?`.

Операнд целевой текст

Строка, в которой осуществляется поиск, обычно задается при помощи записи `=~`, например **`$text =~ m/…/`**. Не путайте последовательность `=~` с операторами присваивания или сравнения; это всего лишь экзотическое обозначение, связывающее оператор поиска с одним из его операндов, которое было позаимствовано из awk.

Поскольку вся конструкция «*выражение* **=~ m/…/**» сама является выражением, ее можно использовать везде, где разрешено использовать выражения. Приведу несколько примеров, разделенных пунктиром:

```
$text =~ m/…/ ;          # Просто выполнить - возможно, ради побочных эффектов.
-------------------------
if ($text =~ m /…/ ) {  # Выполнить некоторые действия в случае успеха
      ⋮
-------------------------
$result = ( $text =~ m /…/ ); # Присвоить Sresult результат поиска в $text
$result = $text =~ m/…/ ;     # То же; =~ обладает более высоким
                               # приоритетом, чем оператор =
-------------------------
  $copy = $text;               # Скопировать $text в $result...
  $copy =~ m /…/ ;             # ... и выполнить поиск в $result
( $copy = $text ) =~ m/…/ ;    # То же самое в одном выражении
```

Целевой текст по умолчанию

Если целевым операндом является переменная `$_`, то конструкцию **`$_ =~`** можно полностью исключить. Другими словами, `$_` является операндом целевого текста по умолчанию.

Итак, команда

```
$text =~ m/выражение/;
```

означает «Применить *выражение* к тексту `$text`; возвращаемое значение игнорируется, но побочные эффекты действуют». Если забыть о знаке '~', получится команда

```
$text = m/выражение/;
```

которая означает: «Применить выражение к тексту `$_` с побочными эффектами; вернуть логическую величину и присвоить ее `$text`». Другими словами, следующие команды эквивалентны:

```
$text =        m/выражение /;
$text = ($_ =~ m/выражение/);
```

Целевой текст по умолчанию удобен в сочетании с другими конструкциями, которые, если явно не указано действовать иначе, используют эту же переменную (такие конструкции встречаются очень часто). Например, довольно часто используется следующая идиома:

```
while (<>)
{
    if (m/…/){
        ⋮
    } elslf (m/…/){
        ⋮
```

Тем не менее на практике при злоупотреблении операндами по умолчанию программа становится менее понятной для неопытных программистов.

Инвертированный поиск

Вы также можете использовать вместо `=~ оператор !~`, чтобы логически инвертировать возвращаемое значение. Вскоре мы рассмотрим возвращаемые значения и побочные эффекты, а пока достаточно сказать, что в записи `!~` возвращаемое значение всегда относится к логическому типу (true или false). Следующие команды идентичны:

```
if ($text !~ m/…/)

if (not $text =~ m/…/)

unless ($text =~ m/…/)
```

Лично я предпочитаю средний вариант. Во всех трех вариантах действуют все стандартные побочные эффекты вроде присваивания переменной `$1`. Таким образом, `!~` всего лишь удобное обозначение, предназначенное для ситуаций типа «если совпадение отсутствует...».

Варианты использования оператора поиска

Оператор поиска не ограничивается возвратом простой логической величины и может возвращать дополнительную информацию об успешном поиске, а также использоваться в сочетании с другими операторами. Работа оператора зависит в основном от *контекста* (☞ 370) и наличия модификатора `/g`.

Простой поиск совпадения — скалярный контекст без модификатора /g

В скалярном контексте (например, при проверке результата в условии `if`) оператор возвращает логическую величину:

```
if ($target =~ m/…/) {
    # Действия для найденного совпадения
    ⋮
```

```
} else {
    # Действия для отсутствия совпадений
    ⋮
}
```

Результат также может присваиваться скалярной переменной для последующего анализа:

```
my $success = Starget =~ m/…/;
    ⋮
if ($success) {
    ⋮
}
```

Простое извлечение данных из строки — списковый контекст без модификатора /g

Списковый контекст без `/g` — распространенный способ извлечения информации из строки. Возвращаемое значение представляет собой список, каждый элемент которого соответствует паре сохраняющих круглых скобок в регулярных выражениях. Простейшим примером является обработка даты в формате 69/8/31:

```
my ($year, $month. $day) = $date =~ m{^ (\d+) / (\d+) / (\d+) $}x;
```

После выполнения этой команды три совпавших числа будут присвоены трем переменным (а также переменным `$1`, `$2` и `$3`.). Каждой паре сохраняющих круглых скобок в возвращаемом списке соответствует один элемент; в случае неудачи возвращается пустой список.

Конечно, некоторые пары могут не входить в совпадение. Например, в команде `m/(this)|(that)/` одна из пар скобок заведомо не войдет в совпадение. Для таких пар в список включается неопределенное значение `undef`. Если в выражении вообще отсутствуют сохраняющие круглые скобки, успешный поиск в списковом контексте без модификатора `/g` возвращает список `(1)`.

Применение спискового контекста может обеспечиваться разными средствами, в том числе и присваиванием результата массиву:

```
my @parts = $text =~ m/^(\d+)-(\d+)-(\d+)$/;
```

Если текст совпадения должен быть присвоен одной скалярной переменной, выполните преобразование к списковому контексту (иначе вместо совпадения переменной будет присвоен логический признак успеха). Сравните следующие команды:

```
my ($word)    = $text =~ m/(\w+)/;
my $success   = $text =~ m/(\w+)/;
```

В первом примере переменная заключена в круглые скобки, поэтому присваивание производится в списковом контексте (в данном случае присваивается текст совпадения). Во втором примере круглых скобок нет; присваивание производится в скалярном контексте, поэтому переменная `$success` будет содержать простой логический признак.

В следующем примере продемонстрирована удобная идиома:

```
if ( my ($year, $month. $day) = Sdate =~ m{^ (\d+) / (\d+) / (\d+) $}x ) {
    # Действия при найденном совпадении;
    # переменные $year и другие имеют определенные значения.
} else {
    # Действия при отсутствии совпадения...
}
```

Оператор поиска возвращает значение в списковом контексте (что обеспечивается конструкцией «`my(…) =`»), поэтому при успешном совпадении список переменных заполняется соответствующими значениями `$1`, `$2` и т. д. Тем не менее вся комбинация выполняется в скалярном контексте (условие `if`), поэтому после заполнения списка Perl преобразует его содержимое в скалярную величину — количество элементов в списке. Результат равен 0 в случае неудачи или отличен от 0 (т. е. соответствует логической истине) при наличии совпадений.

Извлечение всех совпадений — списковый контекст с модификатором /g

Эта полезная конструкция возвращает список всего текста, совпавшего с сохраняющими круглыми скобками (при отсутствии круглых скобок — текста, совпавшего со всем выражением), причем не только для одного совпадения, как в списковом контексте без модификатора `/g`, но и для всех совпадений в строке.

Ниже приведен простой пример извлечения всех целых чисел из строки:

```
my @nums = $text =~ m/\d+/g;
```

Если переменная `$text` содержит IP-адрес '`64.156.215.240`', список `@nums` будет содержать четыре элемента: '`64`', '`156`', '`215`' и '`240`'. В сочетании с другими конструкциями мы получаем простой способ преобразования IP-адреса в шестнадцатеричное число из восьми цифр ('`409cd7f0`'), подходящее для построения компактных журнальных файлов:

```
my $hex_ip = join '', map { sprintf("%02x", $_) } $ip =~ m/\d+/g;
```

Обратное преобразование выполняется аналогично:

```
my $ip = join '.', map { hex($_) } $hex_ip =~ m/../g
```

Или другой пример: для поиска всех вещественных чисел в строке можно воспользоваться выражением

```
my @nums = $text =~ m/\d+(?:\.\d+)?|\.\d+/g;
```

Обратите внимание на несохраняющие круглые скобки, это важно: применение сохраняющих скобок изменит возвращаемую информацию. Впрочем, как показывает следующий пример, команды с одной парой круглых скобок тоже бывают полезны:

```
my @Tags = $Html =~ m/<(\w+)/g;
```

Список `@Tags` заполняется всеми тегами HTML, найденными в тексте `$Html` (предполагается, что целевой текст не содержит посторонних символов '<').

Рассмотрим пример команды с несколькими парами сохраняющих скобок. Предположим, весь текст почтового ящика Unix хранится в одной переменной, содержимое которой разбито на логические строки вида:

```
alias Jeff       jfriedl@regex.info
alias Perlbug    perl5porters@perl.org
alias Prez       president@whitehouse.gov
```

Для извлечения псевдонимов и адресов из одной логической строки можно воспользоваться командой **`m/^alias\s+(\S+)\s+(.+)/m`** (без модификатора `/g`). В списковом контексте команда вернет список из двух элементов ('`Jeff`', '`jfriedl@regex.info`'). Чтобы найти все подобные пары, мы добавим модификатор `/g`. Команда вернет список вида:

```
( 'Jeff', 'jfriedl@regex.info', 'Perlbug',
  'perl-5porters@perl.org', 'Prez', 'president@whitehouse.gov' )
```

Если возвращаемые элементы образуют пары «ключ/значение», как в приведенном примере, результат можно присвоить ассоциативному массиву (хешу). После выполнения команды

```
my $alias = $text =~ m/^alias\s+(\S+)\s+(.+)/mg;
```

полный адрес `Jeff` доступен через элемент хеша `$alias{Jeff}`.

Интерактивный поиск — скалярный контекст с модификатором /g

Скалярный контекст `m/…/g` представляет собой специальную конструкцию, заметно отличающуюся от трех других ситуаций. Как и обычный оператор `m/…/`, он находит только одно совпадение, но по аналогии со списковым оператором `m/…/g`

запоминает позицию предыдущего совпадения. При каждом выполнении m/…/g в скалярном контексте находится «следующее» совпадение. Когда очередной поиск завершится неудачей, следующая проверка снова начинается с начала строки.

Простой пример:

```
$text = "WOW! This is a SILLY test.";

$text =~ m/\b([a-z]+\b)/g;
print "The first all-lowercase word: $1\n";

$text =~ m/\b([A-Z]+\b)/g;
print "The subsequent all-uppercase word: $1\n";
```

В обоих случаях поиска используется скалярный контекст с модификатором /g, поэтому результат выглядит так:

```
The first all-lowercase word: is
The subsequent all-uppercase word: SILLY
```

Операции поиска в скалярном контексте с модификатором /g связаны друг с другом: первая операция устанавливает «текущую позицию» за совпавшим словом, записанным строчными буквами, а вторая продолжает поиск и находит первое слово, записанное прописными буквами, *начиная с этой позиции*. Модификатор /g указывается при каждой операции, чтобы в процессе поиска учитывалась «текущая позиция», поэтому если *хотя бы в одной из команд* отсутствует модификатор /g, вторая команда найдет слово 'WOW'.

Эту конструкцию удобно использовать в условии цикла while. Рассмотрим следующий фрагмент:

```
while ($ConfigData =~ m/^(\w+)=(.*)/mg) {
    my($key, $value) = ($1, $2);
    ⋮
}
```

В цикле будут найдены все совпадения, но между совпадениями (вернее, *после* каждого совпадения) будет выполняться тело цикла. После того как очередная попытка поиска завершится неудачей, результат окажется ложным и цикл while завершится. После неудачи также сбрасывается состояние /g, поэтому следующий поиск с модификатором /g начнется с начала строки.

Сравните фрагменты:

```
while ($text =~ m/(\d+)/) { # Опасно!
   print "found: $1\n";
}
```

и

```
while ($text =~ m/(\d+)/g) {
   print "found: $1\n";
}
```

Они отличаются только присутствием модификатора /g, но это очень существенное различие. Скажем, если переменная $text содержит IPадрес из предыдущего примера, второй фрагмент выведет именно то, что нужно:

```
found: 64
found: 156
found: 215
found: 240
```

С другой стороны, первый фрагмент будет снова и снова выводить строку «found: 64». Без модификатора /g команда просто находит «первое вхождение ⌜(\d+)⌟ в $text», а это всегда '64', независимо от количества проверок. Использование модификатора /g в конструкции со скалярным контекстом изменяет ее смысл — в новом варианте команда находит «следующее вхождение ⌜(\d+)⌟ в $text» и поэтому последовательно выводит все числа.

Начальная позиция поиска и функция pos()

С каждой строкой в Perl ассоциируется «начальная позиция поиска», откуда начинается поиск. Начальная позиция является атрибутом строки и не ассоциируется с каким-либо конкретным регулярным выражением. После создания или модификации строки поиск начинается с самого начала, но после того как будет найдено успешное совпадение, начальная позиция перемещается в конец найденного совпадения. При следующем поиске с модификатором /g механизм приступает к анализу строки от текущей начальной позиции.

Для работы с начальной позицией поиска в Perl используется функция pos(…). Рассмотрим следующий пример:

```
my $ip = "64.156.215.240";
while ($ip =~ m/(\d+)/g) {
    printf "found '$1' ending at location %d\n", pos($ip);
}
```

Результат:

```
found: '64' ending at location 2
found: '156' ending at location 6
found: '215' ending at location 10
found: '240' ending at location 14
```

Вспомните: индексация в строках начинается с нуля, поэтому «позиция 2» находится перед третьим символом. После успешного поиска с модификатором /g

значение `$+[0]` (первый элемент массива `@+` ☞ 379) совпадает со значением, возвращаемым `pos` для целевой строки.

Функция `pos()` использует тот же аргумент по умолчанию, что и оператор поиска, — переменную `$_`.

Предварительная настройка начальной позиции поиска

Настоящая сила функции `pos()` заключается в том, что ей можно присвоить значение. Тем самым вы указываете механизму регулярных выражений позицию, с которой должен начинаться поиск (разумеется, если в нем используется модификатор `/g`). Например, журналы веб-сервера, с которым я работаю на Yahoo!, хранятся в специализированном формате; каждая запись содержит 32-байтовый заголовок, за которым следует запрашиваемая страница и прочая информация. Чтобы извлечь из записи данные о странице, можно воспользоваться конструкцией `⌜^.{32}⌟` и пропустить заголовок фиксированной длины:

```
if ($logline =~ m/^.{32}(\S+)/) {
    $RequestedPage = $1;
}
```

Метод «грубой силы» неэлегантен. Кроме того, механизму регулярных выражений приходится выполнять работу по пропуску первых 32 байтов. Такое решение и менее эффективно, и менее наглядно, чем ручной перевод начальной позиции:

```
pos($logline) = 32; # Страница начинается после 32 символа
                    # начать поиск с этой позиции...
if ($logline =~ m/(\S+)/g) {
    $RequestedPage = $1;
}
```

Такое решение лучше, но оно не эквивалентно прежнему. Поиск *начинается* с нужной позиции, но в отличие от оригинала второе решение не требует обязательного совпадения *в этой позиции*. Если по какой-то причине 33-й символ не совпадет с `⌜\S⌟`, в первом варианте поиск завершится неудачей, а во втором варианте, не привязанном к определенной позиции в строке, механизм продолжит поиск после смещения. Таким образом, он может вернуть совпадение `⌜\S+⌟`, начиная с более поздней позиции в строке. К счастью, у этой проблемы имеется простое решение, описанное в следующем разделе.

Якорный метасимвол `⌜\G⌟`

Якорный метасимвол `⌜\G⌟` обозначает «позицию, в которой завершилось предыдущее совпадение». Именно это нам и требуется для решения проблемы, описанной в предыдущем разделе:

```
pos($logline) = 32; # Страница следует после 32 символа,
                    # поэтому поиск следует начинать с этой позиции…
if (Slogline =~ m/\G(\S+)/g) {
    $RequestedPage = $1;
}
```

Наличие метасимвола ⌜\G⌟ фактически означает: «не смещать текущую позицию в этом регулярном выражении — если совпадение не находится от начальной позиции, немедленно сообщить о неудаче».

Метасимвол ⌜\G⌟ рассматривается в предыдущих главах — общее описание было приведено в главе 3 (☞ 177), а подробный пример — в главе 5 (☞ 274).

В Perl метасимвол ⌜\G⌟ надежно работает лишь в том случае, если он находится в самом начале регулярного выражения, а само выражение не содержит высокоуровневой конструкции выбора. Например, в главе 6 при оптимизации примера с разбором данных CSV (☞ 342) регулярное выражение начиналось с конструкции ⌜\G(?:^|,)…⌟. Поскольку при совпадении более жесткого ограничения ⌜^ ⌟ дополнительная проверка ⌜\G⌟ не нужна, возникает искушение перейти на выражение ⌜(?:|\G,)…⌟. К сожалению, в Perl такой вариант не работает, а его результаты непредсказуемы[1].

Поиск с модификаторами /gc

Обычно в результате неудачной попытки поиска m/…/g позиция pos возвращается в начало целевого текста. Но если к модификатору /g добавить модификатор /c, неудача не будет приводить к сбросу начальной позиции поиска. Модификатор /c никогда не используется без /g, поэтому я буду использовать общую запись /gc.

Конструкция m/…/gc чаще всего используется в сочетании с ⌜\G⌟ для создания «лексеров», разбирающих строку на компоненты. Ниже приведен простой пример разбора кода HTML в переменной $html:

```
while (not $html =~ m/\G\z/gc) # Продолжать до конца текста...
{
    if    ($html =~ m/\G( <[^>]+>  ) /xgc) { print "TAG: $1\n" }
    elsif ($html =~ m/\G( &\w+;     )/xgc) { print "NAMED ENTITY: $1\n" }
    elsif ($html =~ m/\G( &\#\d+;   )/xgc) { print "NUMERIC ENTITY: $1\n" }
    elsif ($html =~ m/\G( [^<>&\n]+ )/xgc) { print "TEXT: $1\n"  }
    elsif ($html =~ m/\G  \n         /xgc) { print "NEWLINE\n" }
    elsif ($html =~ m/\G( .        ) /xgc) { print "ILLEGAL CHAR: $1\n" }
```

[1] Он должен работать в большинстве других диалектов, поддерживающих ⌜\G⌟, но даже в этом случае я не рекомендую им пользоваться, поскольку выигрыш от оптимизации за счет включения ⌜\G⌟ в начало выражения обычно перевешивает затраты на дополнительную проверку ⌜\G⌟ (с. 314).

```
    else {
        die "$0: oops, this shouldn't happen!";
    }
}
```

В каждом регулярном выражении имеется часть, совпадающая с одним из типов конструкций HTML (выделена жирным шрифтом). Все проверки осуществляются последовательно, начиная с текущей позиции (из-за `/gc`), однако совпадение может начинаться *только* с этой позиции (из-за ⌜`\G`⌟). Регулярные выражения перебираются до тех пор, пока цикл не опознает текущую лексему и не выведет информацию о ней. В результате позиция `pos` для переменной `$html` перемещается в начало следующей лексемы, которая обрабатывается на следующей итерации цикла.

Цикл завершается тогда, когда находится совпадение для `m/\G\z/gc`, т. е. когда текущая позиция ⌜`\G`⌟ перейдет в конец строки (⌜`\z`⌟).

Важная особенность приведенного решения заключается в том, что при каждой итерации один из вариантов *должен* совпадать. Если совпадение не будет найдено (и цикл не будет прерван), произойдет зацикливание, поскольку ничто не приведет к продвижению или сбросу позиции `pos` для строки `$html`. В наш пример включена завершающая секция *else*; в представленной версии управление никогда не передается в эту секцию, но в процессе редактирования (а это произойдет совсем скоро) в программу могут быть внесены ошибки, поэтому присутствие секции *else* вполне оправданно. Если данные содержат незапланированные последовательности (например, '<>'), представленная версия выдает одно предупреждение для каждого непредвиденного символа.

У такого решения имеется еще один важный аспект — порядок проверок. Например, выражение ⌜`\G(.)`⌟ проверяется в последнюю очередь. Предположим, мы хотим расширить приложение так, чтобы оно опознавало блоки `<script>`:

```
$html =~ m/\G ( <script[^>]*>.*?</script> )/xgcsi
```

(Ого, целых пять модификаторов!) Чтобы программа работала правильно, новую проверку необходимо вставить *перед* проверкой ⌜`<[^>]+>`⌟, которая сейчас находится на первом месте, или ⌜`<[^>]+>`⌟ совпадет с открывающим тегом `<script>` и перехватит совпадение.

Более сложный пример использования конструкции `/gc` приведен в главе 3 (☞ 179).

Позиция pos: краткая сводка

Ниже приведена краткая сводка взаимодействия оператора поиска с позицией `pos` целевой строки.

Тип поиска	Начало	Позиция pos при успехе	Позиция pos при неудаче
`m/…/`	Начало целевого текста (позиция `pos` игнорируется)	`undef`	`undef`
`m/…/g`	Позиция `pos` целевого текста	Конец совпадения	`undef`
`m/…/gc`	Позиция `pos` целевого текста	Конец совпадения	Не изменяется

Любая модификация строки приводит к сбросу позиции `pos` в состояние `undef` (исходное состояние, обозначающее конец строки).

Внешние связи оператора поиска

В этом разделе обобщается все, что говорилось ранее о взаимодействии оператора поиска со средой Perl.

Побочные эффекты

Побочные эффекты успешного совпадения нередко оказываются более важными, чем значение, возвращаемое оператором. Более того, оператор поиска нередко используется в неопределенном контексте (т. е. возвращаемое значение вообще не используется программой) просто для достижения некоторых побочных эффектов. В таких случаях оператор работает так, как он работал бы в скалярном контексте. Ниже перечислены побочные эффекты *успешной* попытки поиска:

- В текущей области видимости устанавливаются переменные, описывающие результаты поиска, например `$1` и `@+` (☞ 377).
- В текущей области видимости устанавливается регулярное выражение *по умолчанию* (☞ 387).
- Если совпадение было найдено для конструкции `m?…?`, для нее (специфика оператора `m?…?`) устанавливается запрет дальнейших совпадений (по крайней мере, до следующего вызова `reset` в том же пакете ☞ 387).

Напомню, что эти побочные эффекты возникают только при успешном совпадении, — к неудачным попыткам поиска они не относятся. Тем не менее некоторые побочные эффекты возникают при *любых* попытках поиска:

- Сброс или установка позиции `pos` для целевого текста (☞ 393).

- При использовании модификатора `/o` регулярное выражение «привязывается» к оператору, что предотвращает возможность его повторной компиляции (☞ 438).

Внешние факторы, влияющие на оператор поиска

Работа оператора поиска зависит не только от операндов и модификаторов. Ниже перечислены факторы, влияющие на работу оператора поиска.

Контекст

Контекст, в котором применяется оператор поиска (скалярный, списковый или неопределенный), в значительной степени влияет на проведение поиска, а также на его возвращаемое значение и побочные эффекты.

Позиция `pos(…)`

Позиция `pos` целевого текста, установленная явно или косвенно вследствие предыдущего совпадения, указывает, с какой позиции целевого текста должен начинаться следующий поиск с модификатором **`/g`**. Кроме того, она определяет совпадение для метасимвола `\G`.

Регулярное выражение по умолчанию

Если при вызове оператора поиска регулярное выражение не указано, используется регулярное выражение по умолчанию (☞ 387).

`study`

Вызов `study` для целевого текста не влияет на результат поиска, но может повысить (или понизить) его эффективность. Дополнительная информация приведена в разделе «Функция study» (☞ 449).

Оператор `m?…?` и `reset`

Удачный поиск и вызовы `reset` изменяют состояние флага «были совпадения/не было совпадений» для оператора `m?…?` (☞ 387).

Помните о контексте!

Перед тем как завершить описание оператора поиска, я хочу задать вам вопрос. При использовании регулярных выражений в управляющих конструкциях команд `while`, `if` и `foreach` необходимо действовать очень внимательно. Как вы думаете, что выведет следующий фрагмент?

```
while ("Larry Curly Moe" =~ m/\w+/g) {
    print "WHILE stooge is $&.\n";
```

```
}
print "\n";
if ("Larry Curly Moe" =~ m/\w+/g) {
    print "IF stooge is $&.\n";
}
print "\n";
foreach ("Larry Curly Moe" =~ m/\w+/g) {
    print "FOREACH stooge is $&.\n";
}
```

Задача не из простых. ❖ Проверьте свой ответ на с. 402.

Оператор подстановки

Оператор подстановки Perl, **s/…/…/**, расширяет концепцию поиска текста до поиска с заменой. В обобщенном виде он выглядит так:

```
$text =~ s/выражение/замена/модификаторы
```

Текст, совпадающий с выражением-операндом, заменяется указанным текстом. С модификатором `/g` регулярное выражение многократно применяется к тексту, следующему за первым найденным совпадением, и заменяет все совпадения.

Как и в случае с оператором поиска, операнд целевого текста и соединительная конструкция **=~** не являются обязательными, если целевой текст хранится в переменной `$_`. Но в отличие от символа `m` в операторе поиска, знак `s` должен обязательно присутствовать в команде замены.

Вы уже убедились в том, что оператор поиска весьма сложен, — процесс его работы и возвращаемый результат зависят от контекста вызова, позиции `pos` целевой строки и использованных модификаторов. Оператор подстановки работает гораздо проще: он всегда возвращает одну и ту же информацию (количество выполненных подстановок), а модификаторы, влияющие на его работу, не так сложны.

Оператор подстановки поддерживает все базовые модификаторы, перечисленные на с. 368, но у него также есть два дополнительных модификатора: `/g` и `/e`.

Операнд-замена

В обычной конструкции `s/…/…/` операнд-замена указывается сразу же после регулярного выражения-операнда, поэтому в общей сложности используется три экземпляра ограничителя вместо двух в конструкции `m/…/`. Если регулярное выражение заключается в парные ограничители (например, <…>), операнд-замена также заклю-

чается в собственную пару ограничителей (а итоговый оператор содержит четыре ограничителя вместо трех). Например, конструкции `s{…}{…}`, `s[…]/…/` и `s<…>'…'` являются синтаксически правильными. Две пары ограничителей могут разделяться пропусками, а при наличии пропуска возможно наличие комментариев. Парные ограничители обычно используются в сочетании с `/x` и `/e`:

```
$test =~ s{
    ...большое регулярное выражение с обширными комментариями...
} {
    ...фрагмент кода Perl, вычисление которого дает текст замены...
}ex;
```

Вы должны четко понять различия между двумя операндами — регулярным выражением и заменой. Первый обрабатывается с учетом специфики регулярных выражений, со своим набором ограничителей (☞ 366), а второй — по правилам обычных строк в кавычках. Обработка производится после найденного совпадения (а при использовании модификатора `/g` — после каждого совпадения), поэтому переменные `$1` и т. д. относятся к нужному совпадению.

Существуют две ситуации, в которых операнд-замена не интерпретируется по правилам строк в кавычках:

- если операнд-замена заключен в апострофы, он обрабатывается по правилам строк в апострофах (т. е. без интерполяции переменных);
- с модификатором `/e`, о котором будет рассказано в следующем разделе, операнд-замена интерпретируется как мини-сценарий Perl, а не как строка в кавычках. Мини-сценарий выполняется после каждого найденного совпадения, а его результат используется в качестве замены.

Модификатор /e

Модификатор `/e` означает, что операнд-замена должен обрабатываться как код сценария Perl (по аналогии с **`eval{…}`**). При загрузке мини-сценария Perl проверяет его и убеждается в синтаксической правильности кода, но после каждого сценария код выполняется заново. После каждого найденного совпадения операнд-замена обрабатывается в скалярном контексте, а полученный результат подставляется на место найденного совпадения. Рассмотрим простой пример:

```
$text =~ s/-time-/localtime/ge;
```

Все вхождения строки `-time-` заменяются результатом вызова функции Perl `localtime` в скалярном контексте (т. е. представлением текущего времени в строковом формате вида «`Mon Sep 25 18:36:51 2006`»).

ОТВЕТ

❖ *Ответ на вопрос со с. 400.*

Приведенный фрагмент выведет следующий результат:

```
WHILE stooge is Larry.
WHILE stooge is Curly.
WHILE stooge is Moe.

IF stooge is Larry.

FOREACH stooge is Moe.
FOREACH stooge is Moe.
FOREACH stooge is Moe.
```

Обратите внимание: если бы в команде `print` в цикле `foreach` вместо `$&` использовалась переменная `$_`, то результат совпадал бы с результатом цикла `while`. Однако в этом случае результат, возвращаемый командой `m/.../g`, (‘`Larry`’, ‘`Curly`’, ‘`Moe`’), оставался бы неиспользованным. Вместо этого используется побочный эффект `$&`, почти всегда свидетельствующий об ошибке программирования, поскольку побочные эффекты `m/.../g` в списковом контексте редко приносят пользу.

Поскольку после каждого совпадения операнд обрабатывается заново, для работы с текстом последнего совпадения можно использовать переменные `$1` и т. д. Например, в URL допускается кодирование специальных символов при помощи префикса `%`, за которым следует шестнадцатеричный код из двух цифр. Следующая команда кодирует в строке все символы, кроме алфавитно-цифровых:

```
$url =~ s/(["a-zA-ZO-9])/sprintf('%%%02x', ord($1))/ge;
```

Команда декодирования выглядит так:

```
$url =~ s/%([0-9a-f][0-9a-f])/pack("C", hex($1))/ige;
```

Функция **`sprintf('%%%02x', ord(`***символ***`))`** преобразует символы в их числовые коды, а функция **`pack("C",`***число***`)`** решает противоположную задачу; за дополнительной информацией обращайтесь к документации Perl.

Многократное использование `/e`

Обычно повторение модификаторов не влияет на работу программы (хотя может слегка запутать читателя), но модификатор `/e` является исключением. Если он входит в команду несколько раз, операнд-замена также будет вычислен многократно. Вероятно, эта возможность пригодится только в конкурсе на Самую Загадочную Программу на Perl, и все же о ней стоит упомянуть.

Впрочем, подобные конструкции иногда приносят практическую пользу. Допустим, вы хотите провести ручную интерполяцию переменных в строке (так, словно строка была прочитана из конфигурационного файла). Другими словами, имеется строка '… $var …', и вы хотите заменить подстроку '$var' значением переменной $var.

Простейшее решение может выглядеть так:

```
$data =~ s/(\$[a-zA-Z_]\w*)/$1/eeg;
```

Без модификаторов /e команда лишь произведет тождественную замену совпадения '$var', что не принесет особой пользы. С одним модификатором /e команда просто получит подстроку $1 из операнда-замены, которая будет расширена до '$var', в результате чего совпавший текст опять же заменится самим собой (что тоже бесполезно). Но с двумя модификаторами /e результат будет вычислен повторно, вследствие чего вместо '$var' будет подставлено текущее значение переменной. В результате фактически будет выполнена интерполяция переменной.

Контекст и возвращаемое значение

Я уже говорил о том, что оператор поиска возвращает различные значения для разных сочетаний контекста с модификатором /g. С оператором подстановки дело обстоит проще — он всегда возвращает либо количество выполненных подстановок, либо (если ни одной подстановки не сделано) пустую строку.

При логической интерпретации (например, в условии команды if) возвращаемое значение удобно интерпретируется как истина, если была выполнена хотя бы одна подстановка, и как ложь в противном случае.

Оператор разбиения

Многогранный оператор split (в просторечии часто называемый *функцией*) обычно используется как некая противоположность конструкции m/…/g в списковом контексте (☞ 391). Последняя возвращает текст, совпавший с регулярным выражением, тогда как split с тем же регулярным выражением возвращает текст, *разделяемый* совпадениями. Так, применение команды **$text =~ m/:/g** к переменной $text, содержащей строку '**IO.SYS:225558:95-10-03:-a-sh:optional**', возвращает список из четырех элементов:

```
(':', ':', ':', ':' )
```

Вряд ли этот список принесет какую-нибудь пользу. С другой стороны, команда **split(/:/, $text)** возвращает список из пяти элементов:

```
('IO.SYS', '225558', '95-10-03', '-a-sh', 'optional')
```

В обоих примерах `/:/` совпадает четыре раза. При использовании `split` эти четыре совпадения разделяют копию целевого текста на пять частей, возвращаемых в виде списка из пяти строк.

В этом примере целевая строка разбивалась по одному символу, однако разбиение может производиться по любому регулярному выражению. Например, команда

```
@Paragraphs = split(m/\s*<p>\s*/I, $html);
```

разбивает код HTML в переменной `$html` на фрагменты, разделенные тегами `<p>` и `<P>`, по соседству с которыми могут находиться необязательные пропуски. В качестве разделителей могут использоваться не только символы и их комбинации, но и позиции строки; например:

```
@Lines = split(m/^/m, $lines);
```

разбивает текст на логические строки.

В простейшей форме и с простыми данными, как в приведенном примере, оператор `split` вполне понятен, а польза от него очевидна. Тем не менее существует множество факторов, особых случаев и исключений, из-за которых все становится гораздо сложнее. Прежде чем углубляться в детали, я хотел бы представить два особенно полезных случая.

- Специальный операнд **//** разбивает целевой текст на символы, из которых он состоит. Например, команда **split(//, "short test")** возвращает список из десяти элементов: **("s", "h", "o", …,"s","t")**.
- Специальный операнд **" "** (строка, состоящая из одного пробела) разбивает целевой текст по пропускам, как с операндом `m/\s+/`, но начальные и конечные пропуски при этом игнорируются. Например, команда **split(" ", " a short test ")** возвращает строки ‘a’, ‘short’ и ‘test’.

Эти и другие особые случаи будут рассмотрены ниже, а мы начнем с базовых принципов использования `split`.

Простейшее разбиение

Оператор `split` выглядит как функция и получает до трех операндов:

```
split (совпадение, целевая_строка, ограничение)
```

Круглые скобки необязательны. Для пропущенных операндов используются значения по умолчанию (подробнее — ниже в этом разделе).

Оператор `split` всегда используется в списковом контексте. Ниже приведены типичные способы его применения:

```
($var1, $var2, $var3, …) = split(…);
-----------------------------
@array = split(…);
-----------------------------
for my $item (split(…) ) {
   ⋮
}
```

Первый операнд (совпадение)

У первого операнда существует несколько особых случаев, но обычно он задается по тем же правилам, что и операнд регулярного выражения в операторе поиска. Это означает, что вы можете использовать `/…/`, `m{…}` и аналогичные конструкции, объект регулярного выражения или любое выражение, интерпретируемое как строка. Поддерживаются только базовые модификаторы, перечисленные на с. 368.

Если вам потребуются круглые скобки для группировки, обязательно применяйте несохраняющий синтаксис `(?:…)`. Как показано ниже, сохраняющие скобки в `split` активизируют весьма специфический режим.

Второй операнд (целевая строка)

Оператор `split` только анализирует целевую строку и никогда не модифицирует ее. Если целевая строка не задана, по умолчанию используется содержимое `$_`.

Третий операнд (ограничение)

Главная функция третьего операнда — ограничение количества фрагментов, на которые `split` разбивает строку. Например, для приведенного выше примера команда **`split(/:/, $text, 3)`** возвращает список:

```
('IO.SYS', '225558', '95-10-03:-a-sh:optional')
```

Как видно из примера, `split` прекращает дальнейшие поиски после двух совпадений `/:/`, в результате чего строка делится на три фрагмента. В принципе, целевой текст содержит и другие потенциальные совпадения, но в данном случае это несущественно из-за установленного ограничения на количество фрагментов. Операнд лишь устанавливает верхнюю границу и гарантирует, что большее количество элементов

не будет возвращено ни при каких условиях, однако он не гарантирует возврата заданного количества фрагментов; если разбиение не обеспечивает заданного количества фрагментов, дополнительные фрагменты не генерируются. Например, команда **split(/:/,$text, 99)** все равно вернет список из пяти элементов. Впрочем, между командами **split(/:/, $text)** и **split(/:/, $text, 99)** существует важное отличие, которое не проявляется в этом примере; подробности будут приведены ниже.

Также следует помнить, что операнд *ограничивает* количество *фрагментов*, а не количество совпадений. Если бы ограничение относилось к самим совпадениям, то предыдущий пример для трех совпадений возвращал бы следующий список:

```
('IO.SYS', '225558', '95-10-03', '-a-sh:optional')
```

На практике происходит совсем иное. И еще одно замечание из области эффективности. Допустим, вы хотите ограничиться выборкой нескольких начальных полей:

```
($filename, $size, $date) = split(/:/, $text);
```

Для повышения эффективности Perl прекращает разбиение после заполнения заданного количества полей. Для этого автоматически задается количество фрагментов, на единицу большее количества элементов в списке.

Нетривиальное разбиение

В примерах, рассмотренных выше, оператор `split` казался весьма простым, однако существуют три ситуации, в которых он значительно усложняется:

- возвращение пустых элементов;
- специальные операнды регулярных выражений;
- регулярные выражения с сохраняющими круглыми скобками.

Подробнее об этих особых ситуациях рассказывается в следующих разделах.

Возвращение пустых элементов

Оператор `split` прежде всего предназначен для возвращения текста между совпадениями, но иногда возвращаемый текст представляет собой пустую строку (строку нулевой длины, т. е. `""`). Рассмотрим следующий пример:

```
@nums = split(m/:/, "12:34::78");
```

Команда возвращает следующий список:

```
("12", "34", "", "78")
```

Регулярное выражение `/:/` совпадает три раза, поэтому список состоит из четырех элементов. Пустой третий элемент указывает на то, что выражение совпало два раза подряд без разделения текстом.

Завершающие пустые элементы

Обычно завершающие пустые элементы не возвращаются. Например, команда

```
@nums = split(m/:/, "12:34::78:::");
```

заполняет `@nums` четырьмя элементами:

```
("12", "34", "", "78")
```

Мы получили тот же список, что и в предыдущем примере, хотя регулярное выражение совпало еще несколько раз в конце строки. По умолчанию оператор `split` не возвращает пустые элементы в конце списка. Впрочем, вы можете запретить Perl удалять завершающие пустые строки, но для этого придется специальным образом использовать третий операнд.

Еще одна функция третьего операнда

Помимо возможного ограничения количества фрагментов, ненулевое значение третьего операнда также запрещает удаление всех пустых элементов в конце списка (при нулевом значении третьего операнда `split` ведет себя в точности так же, как если бы операнд вообще не был задан). Если вы не хотите ограничивать количество возвращаемых фрагментов, а лишь хотите оставить в списке пустые элементы, достаточно передать в третьем операнде очень большое число, а еще лучше использовать число `-1`, поскольку отрицательное число трактуется как очень большое значение ограничения: команда **`split(/:/, $text, -1)`** возвращает весь список, включая пустые элементы в конце.

С другой стороны, если вы хотите удалить из списка *все* пустые элементы, поставьте **`grep{length}`** перед вызовом `split`. Функция `grep` оставит в списке только элементы, имеющие ненулевую длину (т. е. непустые):

```
my @NonEmpty = grep { length } split(/:/, $text);
```

Специальные совпадения в конце строки

Совпадение в самом начале строки обычно порождает пустой элемент списка:

```
@nums = split(m/:/, ":12::34::78");
```

Содержимое `@nums` выглядит так:

```
("", "12", "34", "", "78")
```

Первый пустой элемент означает, что выражение совпало в начале строки. Существует особый случай: если при совпадении в начале или в конце строки регулярное выражение не совпало с текстом (т. е. совпадение было чисто позиционным), начальные и/или конечные пустые элементы *не* создаются. Рассмотрим простой пример: команда **`split(/\b/, "a simple test")`** может совпасть в шести отмеченных позициях строки '␣a␣•␣simple␣•␣test␣'. Хотя совпадение находится шесть раз, оператор возвращает не семь элементов, а всего пять: **`("a", "•", "simple", "•", "test")`**. Кстати, мы уже встречались с этим особым случаем в примере **`@Lines= split(m/^/m, $lines)`** на с. 403.

Специальные значения первого операнда split

Первый операнд `split` обычно задает литерал или объект регулярного выражения, как и соответствующий операнд оператора поиска, но для него определены некоторые специальные значения:

- Пустое регулярное выражение, передаваемое при вызове `split`, означает не «использовать текущее регулярное выражение по умолчанию», а «разбивать строку после каждого символа». Пример уже приводился в начале обсуждения `split`, когда я упоминал о том, что **`split(//,"short test")`** возвращает список из десяти элементов: **`("s", "h", "o", …, "s", "t")`**.

- Операнд, который представляет собой *строку* (**не** регулярное выражение!), состоящую ровно из одного пробела, — особый случай. Он почти эквивалентен `/\s+/`, если не считать игнорирования начальных пропусков. Эта конструкция предназначалась для имитации стандартного разбиения по разделителям входных записей в awk, но она, несомненно, находит немало применений и в более общих случаях.

- Чтобы сохранить начальные пропуски, достаточно использовать `m/ \s+/` напрямую. Чтобы сохранить завершающие пропуски, передайте `-1` в третьем операнде.

- Если первый операнд вообще не задан, по умолчанию используется строка, состоящая из одного пробела (специальное значение из предыдущего пункта). Таким образом, вызов `split` без операндов эквивалентен **`split(' ', $_, 0)`**.

- При использовании регулярного выражения `/^/` автоматически используется модификатор `/m` (расширенный режим привязки к границам строк). Почему-то для `/$/` этого не происходит. В конструкции **`m/^/m`** нет ничего сложного, поэтому

для наглядности я рекомендую использовать именно ее. Вызов `split` с операндом `m/^/m` позволяет легко разделить текст на отдельные логические строки.

Разбиение не имеет побочных эффектов

Первый операнд `split` внешне *похож* на оператор поиска, но он не имеет ни одного побочного эффекта, присущего этому оператору. Использование регулярного выражения в `split` не изменяет регулярное выражение по умолчанию для последующих операторов поиска и замены. Вызов `split` не изменяет состояние переменных `$&`, `$'`, `$1` и т. д. Короче говоря, в отношении побочных эффектов оператор `split` полностью изолирован от остальной программы[1].

Сохраняющие круглые скобки в первом операнде split

Использование сохраняющих круглых скобок в первом операнде `split` изменяет принцип работы `split`. В этом случае возвращаемый массив содержит дополнительные, независимые элементы, чередующиеся с элементами, совпавшими с подвыражениями в круглых скобках. Это означает, что текст, обычно полностью *исключавшийся* `split` при разбиении, теперь включается в возвращаемый список.

Например, в процессе обработки HTML-кода команда `split(/(<[^>]*>)/)` для текста

```
…•and•<B>very•<FONT•color=red>very</FONT>•much</B>•effort…
```

возвращает список

```
( '…•and•', '<B>', 'very•', '<FONT•color=red>',
  'very', '</FONT>', '•much', '</B>', '•effort…' )
```

Если убрать сохраняющие круглые скобки, команда `split(/<[^>]*>/)` вернет список:

```
( '…•and•', 'very•', 'very', '•much', '•effort…' )
```

[1] Вообще говоря, существует один побочный эффект, связанный с возможностью, которая давно считается устаревшей, но еще не была исключена из языка. При использовании в скалярном или пустом (void) контексте `split` записывает результаты в переменную `@_` (которая также используется при передаче аргументов функций, поэтому будьте внимательны и избегайте случайного применения `split` в этих контекстах). Директива `use warnings` и ключ командной строки `-w` выдают предупреждение при использовании `split` в обоих названных контекстах.

Дополнительные элементы не учитываются в общем числе фрагментов (третий операнд определяет количество фрагментов, на которые разбивается исходная строка, а не количество возвращаемых элементов).

При наличии дополнительных пар сохраняющих круглых скобок в список для каждого совпадения включаются несколько элементов. Если пара скобок не участвует в совпадении, для нее в список вставляется значение `undef`.

Специфические возможности Perl

Многие концепции регулярных выражений, поддерживаемые в других языках, когда-то были доступны только в Perl. Среди примеров стоит упомянуть несохраняющие скобки, опережающую (а позднее — и ретроспективную) проверку, режим свободного форматирования, многие режимы поиска, а также `\A`, `\Z` и `\z`), атомарную группировку, `\G` и условную конструкцию. Все эти элементы уже существуют не только в Perl и рассматриваются в основной части книги.

Но разработчики Perl не стоят на месте, поэтому в настоящее время также существуют конструкции, поддерживаемые только в Perl. Особенно интересно выглядит возможность выполнения произвольного кода *во время поиска совпадения*. Высокий уровень интеграции регулярных выражений с программным кодом всегда был присущ Perl, но теперь эта интеграция выходит на принципиально новый уровень.

Далее следует краткий обзор этих и других новшеств, которые в настоящее время доступны только в Perl.

Динамические регулярные выражения `(??{` *код perl* `})`

Если эта конструкция встречается в процессе применения регулярного выражения, выполняется *код Perl*. Результат (объект регулярного выражения или строка, интерпретируемая как регулярное выражение) используется как часть текущего совпадения.

В простом примере `^(\d+)(??{ "X{$1}" })$` конструкция динамического регулярного выражения выделена подчеркиванием. Приведенное выражение совпадает с числом в начале строки, за которым следует заданное количество символов 'X' до конца строки.

В частности, оно совпадает с '3XXX' и '12XXXXXXXXXXXX', но не с '3X' или '7XXXX'.

Если проанализировать пример '3XXX', вы увидите, что начальное подвыражение `(\d+)` совпадает с '3XXX', в результате чего переменной `$1` присваивается значе-

ние '3'. Затем механизм регулярных выражений достигает конструкции динамического регулярного выражения, выполняет код «X{$1}» и получает строку 'X{3}'. Строка интерпретируется как выражение ⌜X{$1}⌟, применяется как часть текущего регулярного выражения и совпадает с '3XXX'. Далее завершающее подвыражение ⌜$⌟ применяется в позиции '3XXX▲' с получением общего совпадения.

Как будет показано в следующих примерах, динамические регулярные выражения особенно полезны при поиске вложенных конструкций произвольного уровня.

Встроенный код ⌜(?{ *произвольный код perl* })⌟

Данная конструкция, как и динамические регулярные выражения, тоже выполняет код Perl в процессе поиска, однако она является более общей, поскольку не обязана возвращать какой-то определенный результат. Обычно возвращаемое значение вообще не используется (хотя в том же регулярном выражении к нему можно обратиться при помощи переменной $^R ☞ 380).

Впрочем, в одной ситуации значение, сгенерированное этим кодом, все же используется: речь идет о применении встроенного кода в части *if* условной конструкции ⌜(? *if then* | *else*)⌟ (☞ 188). В этом случае результат интерпретируется как логическая величина, в зависимости от значения которой применяется либо часть *then*, либо *else*.

Встроенный код находит много практических применений, но он особенно полезен при отладке. Ниже приведен простой пример, который выводит сообщение при каждом применении регулярного выражения; встроенный код выделен подчеркиванием:

```
"have a nice day" =~ m{
    (?{ print "Starting match.\n" })
    \b(?: the | an | a )\b
}x;
```

В данном случае выражение совпадает полностью всего один раз, но сообщение выводится шестикратно. По выходным данным можно узнать, что регулярное выражение по крайней мере частично применялось в пяти позициях до обнаружения полного совпадения в шестой.

Перегрузка литералов регулярных выражений

Под термином «перегрузка литералов регулярных выражений» понимается нестандартная предварительная обработка литералов регулярных выражений перед их передачей механизму для дальнейшей обработки. На практике это позволяет расширить диалект регулярных выражений Perl. Например, в Perl нет отдельных метасимволов для начала и конца слова (есть лишь универсальный метасимвол

границы слова \b), но вы можете распознать последовательности \< и \> и самостоятельно преобразовать их в конструкции, поддерживаемые Perl.

Для перегрузки регулярных выражений установлены существенные ограничения, заметно снижающие ее практическую ценность. Примеры таких ограничений (а также примеры использования перегрузки) приводятся ниже в этом разделе.

При работе с кодом Perl в регулярных выражениях (динамическими регулярными выражениями или встроенным кодом) желательно ограничиваться глобальными переменными, во всяком случае, пока вы не разберетесь в специфике использования переменных my (☞ 424).

Применение динамических регулярных выражений для поиска вложенных конструкций

Динамические регулярные выражения чаще всего применяются для поиска вложенных конструкций произвольного уровня (когда-то считалось, что эта задача в принципе не решается при помощи регулярных выражений). Наиболее характерным примером является поиск текста в круглых скобках произвольного уровня вложенности. Чтобы понять, как динамические регулярные выражения помогают в решении этой задачи, давайте сначала посмотрим, почему она не решается на базе традиционных конструкций.

Простое регулярное выражение для текста, заключенного в круглые скобки, выглядит так: `\(([^()])*\)`. Оно не допускает присутствия внутренних круглых скобок, поэтому вложенные конструкции в принципе невозможны (иначе говоря, поддерживается нулевой уровень вложенности). Регулярное выражение можно преобразовать в объект и использовать в программе:

```
my $Level0 = qr/ \( ( [^()] )* \) /x; # Текст в круглых скобках
    ⋮
if ($text =~ m/\b( \w+$Level0 )/x) {
    print "found function call: $1\n";
}
```

Приведенный фрагмент найдет совпадение «substr($str, 0, 3)», но не сможет найти «substr($str, 0, (3+2))» из-за вложенных круглых скобок. Попробуем усовершенствовать регулярное выражение так, чтобы оно справилось с этой ситуацией, т. е. реализуем поддержку первого уровня вложенности.

Наша задача — сделать так, чтобы во внешних круглых скобках распознавалась другая конструкция, заключенная в круглые скобки. Для этого к выражению, описывающему символы между внешними скобками (в предыдущей версии — `[^()]`),

необходимо добавить подвыражение, совпадающее с текстом в круглых скобках. Что ж, эта задача уже решена: нужное выражение хранится в переменной `$Level0`. Используя эту переменную, мы создаем следующий уровень вложенности:

```
my $Level0 = qr/ \( ( [^()]          )* \) /x; # Текст в круглых скобках
my $Level1 = qr/ \( ( [^()]| $Level0 )* \) /x: # Первый уровень
```

Переменная `$Level0` сохранила прежнее значение, но теперь она используется при построении переменной `$Level1`, которая совпадает с собственной парой круглых скобок *и* скобками переменной `$Level0`. В результате мы получаем первый уровень вложенности.

Чтобы добавить еще один уровень, можно действовать аналогичным образом и создать переменную `$Level2`, использующую переменную `$Level1` (которая, в свою очередь, использует `$Level0`):

```
my $Level0 = qr/ \( ( [^()]          )* \) /x; # Текст в круглых скобках
my $Level1 = qr/ \( ( [^()]| $Level0 )* \) /x: # Первый уровень
my $Level2 = qr/ \( ( [^()]| $Level1 )* \) /x: # Второй уровень
```

Так может продолжаться до бесконечности:

```
my $Level3 = qr/ \( ( [^()]| $Level2 )* \) /x; # Третий уровень
my $Level4 = qr/ \( ( [^()]| $Level3 )* \) /x: # Четвертый уровень
my $Level5 = qr/ \( ( [^()]| $Level4 )* \) /x: # Пятый уровень
```

На рис. 7.1 начальные уровни вложенности представлены в графическом виде.

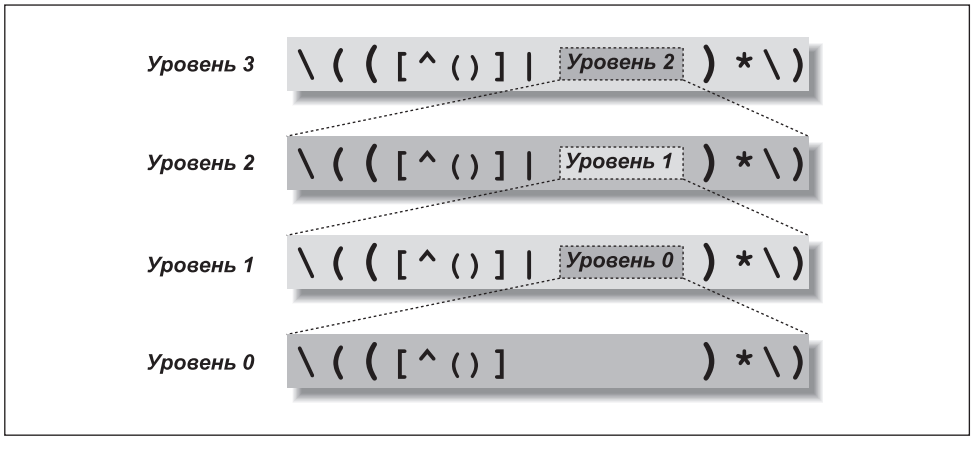


Рис. 7.1. Поиск текста в круглых скобках

Интересно посмотреть, какое выражение будет сгенерировано в результате. Вот как выглядит переменная `$Level3`:

```
\(([^()]|\(([^()]|\(([^()]|\(([^()])*\))*\))*\))*\)
```

Выглядит довольно жутко.

К счастью, нам не придется интерпретировать эту строку (пусть этим занимается механизм регулярных выражений). Работать с переменными `Level` достаточно просто, но у такого подхода есть существенный недостаток: вложение ограничивается тем количеством уровней, для которого были построены переменные `$Level`. Поиск вложенных конструкций *произвольного уровня* невозможен (следствие из закона Мэрфи: если мы обеспечим поддержку *X* уровней вложенности, немедленно появятся данные с *X+1* уровнем).

Проблема поиска вложенных конструкций произвольного уровня решается при помощи динамических регулярных выражений. Прежде всего необходимо понять, что все переменные `$Level`, кроме первой, конструируются одинаково: выражение для каждого следующего уровня вложенности создается на базе выражения для предыдущего уровня. Но если все переменные строятся по одному принципу, любая из этих переменных фактически включает копию *самой себя*. Если бы включение происходило в тот момент, когда возникает необходимость в очередном уровне вложенности, это позволило бы рекурсивно обработать вложенные конструкции *любого* уровня.

Именно это и делается при помощи динамических регулярных выражений. Если создать объект регулярного выражения для одной из переменных `$Level`, на него можно будет сослаться из динамического регулярного выражения (динамические выражения могут содержать произвольный код Perl при условии, что его результат может интерпретироваться как регулярное выражение; разумеется, код, который просто возвращает существующий объект регулярного выражения, удовлетворяет этому условию). Если сохранить наш аналог `$Level` в переменной `$LevelN`, на него можно будет ссылаться в конструкциях вида `(??{$LevelN})`:

```
my $LevelN; # Переменная должна быть объявлена заранее, потому что она
            # используется при присваивании значения самой себе.
$LevelN = qr/ \ (( [^()] | (??{ $LevelN }))* \) /x;
```

Выражение совпадает с круглыми скобками произвольной вложенности и используется в программе так же, как переменная `$Level0` в приведенном выше фрагменте:

```
if ($text =~ m/\b( \w+$LevelN )/x) {
    print "found function call: $1\n";
}
```

Впечатляет, не правда ли? Возможно, разобраться в этой теме непросто, но когда все встанет на свои места, динамические регулярные выражения станут полезным инструментом в вашей работе.

Остается лишь внести небольшие исправления для повышения эффективности. Заменим сохраняющие круглые скобки конструкцией атомарной группировки (ни сохранение, ни возврат в программе не используются), а затем преобразуем ⌜`[^()]`⌟ в ⌜`[^()]+`⌟ (без атомарной группировки этого делать не следует, поскольку это приведет к бесконечному перебору вариантов ☞ 289).

Наконец, ⌜`\(`⌟ и ⌜`\)`⌟ следует переместить так, чтобы они непосредственно прилегали к динамическому регулярному выражению. В этом случае динамическое регулярное выражение будет обрабатываться механизмом, только если он уверен в наличии искомого текста. Обновленная версия выглядит так:

```
$LevelN = qr/ (?> [^()]+ | \( (??{ $LevelN }) \) )* /x;
```

Поскольку в этом варианте внешние ограничители ⌜`\(…\)`⌟ отсутствуют, мы должны включить их при обращении к `$LevelN`.

При этом возникает полезный побочный эффект: выражение становится более гибким. Оно может применяться не только там, где *обязательно* присутствуют парные круглые скобки, но и там, где они *могут* встречаться:

```
if ($text =~ m/\b( \w+ \( $LevelN \) )/x) {
    print "found function call: $1\n";
}
```

```
if (not $text =~ m/^ $LevelN $/x) {
    print "mismatched parentheses !\n";
}
```

Другой пример практического применения `$LevelN` приведен на с. 430.

Встроенный код

Встроенный код особенно полезен при отладке регулярных выражений и сборе информации о совпадении в процессе поиска. В этом разделе будет рассмотрена серия примеров, которая постепенно приведет нас к имитации семантики поиска по стандарту POSIX. Возможно, сам процесс интереснее конечного результата (если, конечно, вам не потребуется воспроизвести семантику POSIX), потому что попутно будут представлены некоторые полезные приемы и интересные факты.

Начнем с рассмотрения простейших средств отладки регулярных выражений.

Вывод информации в процессе поиска

Фрагмент:

```
"abcdefgh" =~ m{
    (?{ print "starting match at [$`|$']\n" })
    (?:d|e|f)
}x;
```

выводит следующий результат:

```
starting match at [|abcdefgh]
starting match at [a|bcdefgh]
starting match at [ab|cdefgh]
starting match at [abc|defgh]
```

Конструкция встроенного кода стоит в выражении на первом месте и выполняет команду

```
print "starting match at [$`|$']\n"
```

каждый раз, когда механизм начинает новую попытку поиска совпадения. Выводимая строка состоит из переменных `$`` и `$'` (☞ 377)[1], представляющих текст до и после совпадения, между которыми вставляется символ '|', отмечающий начальную позицию, с которой применяется выражение. По выходным данным можно сказать, что совпадение было найдено после четырех смещений позиции (☞ 198).

Если добавить в конце регулярного выражения фрагмент

```
(?{ print "matched at [$`<$&>$']\n" })
```

будет выведено следующее совпадение:

```
matched at [abc<d>efgh]
```

Теперь сравните этот пример со следующим, который очень похож на него, но в качестве «основного» регулярного выражения вместо `(?:d|e|f)` использует символьный класс `[def]`:

```
"abcdefgh" =~ m{
    (?{ print "starting match at [$`;$']\n" })
    [def]
}x;
```

[1] Обычно я не рекомендую использовать специальные переменные `$``, `$&` и `$'`, так как они заметно снижают быстродействие программы (☞ 444), но для отладки они подойдут.

PANIC: TOP_ENV

Если программа, использующая встроенный код или динамическое регулярное выражение, вдруг выдает сообщение

```
panic: top_env
```

скорее всего, это свидетельствует об ошибке в кодовой части. В настоящее время Perl не справляется с некоторыми синтаксическими ошибками и «впадает в панику». Конечно, ошибку следует найти и исправить.

Теоретически результаты должны быть идентичными, но в новом варианте выводится всего одна строка:

```
starting match at [abc|defgh]
```

Чем объясняются эти различия? Perl достаточно умен, поэтому он применяет к регулярному выражению с `[def]` оптимизацию исключения по начальному символьному классу (☞ 313) и обходит попытки, заведомо обреченные на неудачу. Таким образом обходятся все попытки, кроме той, которая привела к совпадению. Мы узнали об этом благодаря встроенному коду.

Использование встроенного кода для вывода всех совпадений

В Perl используется традиционный механизм НКА, поэтому с обнаружением совпадения поиск немедленно прерывается, даже если существуют другие потенциальные совпадения. Умное использование встроенного кода позволяет «обмануть» Perl и вывести *все* возможные совпадения. Чтобы понять, как это делается, достаточно вернуться к глупому примеру ‘`oneself`’ на с. 233:

```
"oneselfsufficient" =~ m{
    one(self)?(selfsufficient)?
    (?{ print “matched at [$`<$&>$']\n” })
}x;
```

Как и ожидалось, будет выведена строка

```
matched at [<oneself>sufficient]
```

Она указывает на то, что регулярное выражение совпало с текстом ‘`oneself`sufficient’.

Обратите внимание на одну тонкость: команда `print` выводит не «совпадение», а «совпадение *до данной позиции*». В нашем примере различия чисто академические, поскольку конструкция со встроенным кодом завершает регулярное выражение.

Мы знаем, что вместе со встроенным кодом завершается и само регулярное выражение, а выведенный результат соответствует общему совпадению.

А что, если добавить ⌜(?!)⌟ перед встроенным кодом? Негативная опережающая проверка ⌜(?!)⌟ всегда завершается неудачей. Когда это происходит сразу же после обработки встроенного кода (после вывода сообщения «matched»), неудача заставляет механизм продолжить поиск других совпадений. Неудача автоматически происходит после каждого вывода совпадения, поэтому в результате будут исследованы все возможные варианты, а мы увидим все возможные совпадения:

```
matched at [<oneself>sufficient]
matched at [<oneselfsufficient>]
matched at [<one>selfsufficient]
```

В итоге поиск всегда завершается неудачей, но механизм регулярных выражений выводит все совпадения. Без ⌜(?!)⌟ Perl просто вернет первое найденное совпадение, а с этой конструкцией мы увидим все остальные совпадения.

Как вы думаете, что выведет следующий фрагмент?

```
"123" =~ m{
    \d+
    (?{ print "matched at [$`<$&>$']\n" })
    (?!)
}x;
```

Результат выглядит так:

```
matched at [<123>]
matched at [<12>3]
matched at [<1>23]
matched at [1<23>]
matched at [1<2>3]
matched at [12<3>]
```

Присутствие первых трех строк вполне понятно, но остальные данные могут показаться неожиданными. Впрочем, после некоторых размышлений все встает на свои места. Подвыражение (?!) вызывает возврат и последующее обнаружение совпадений второй и третьей строки. Когда все попытки поиска от начала строки завершаются неудачей, происходит смещение текущей позиции, и регулярное выражение применяется от позиции перед вторым символом (смещение очень подробно описано в главе 4). Четвертая и пятая строки относятся к этой позиции, а шестая строка — к позиции, предшествующей третьему символу.

Итак, после добавления (?!) выводятся *все* возможные совпадения, а не только начинающиеся с конкретной начальной позиции. Впрочем, и этот вариант тоже возможен; вскоре мы вернемся к этой задаче.

Поиск самого длинного совпадения

От вывода всех совпадений перейдем к задаче поиска совпадения максимальной длины. В программе определяется переменная для хранения наибольшей длины, встречавшейся до настоящего момента. Содержимое этой переменной сравнивается с длиной каждого нового потенциального совпадения. Ниже приведено решение этой задачи для примера 'oneself':

```
$longest_match = undef; # Переменная для хранения текущей длины

"oneselfsufficient" =~ m{
    one(self)?(selfsufficient)?
    (?{
        # Проверить, не является ли текущее совпадение ($&)
        # самым длинным из найденных
        if (not defined($longest_match)
            or
            length($&) > length($longest_match))
        {
            $longest_match = $&;
        }
    })
    (?!) # Форсировать неудачу при поиске, чтобы продолжить поиск
         # других потенциальных совпадений
}x;

# Вывести накопленный результат, если были найдены совпадения
if (defined($longest_match)) {
    print "longest match=[$longest_match]\n";
} else {
    print "no match\n";
}
```

Как и следовало ожидать, программа выводит строку 'longest match=[oneselfsufficient]'. Встроенный код получился довольно длинным, вдобавок он еще пригодится нам в будущем, поэтому мы инкапсулируем его и `(?!)` в объекты регулярных выражений:

```
my $RecordPossibleMatch = qr{
    (?{
        # Проверить, не является ли текущее совпадение ($&)
        # самым длинным из найденных
        if (not defined($longest_match)
            or
            length($&) > length($longest_match))
        {
            $longest_match = $&;
```

```
        }
    })
    (?!) # Форсировать неудачу при поиске, чтобы продолжить поиск
         # других потенциальных совпадений
}x;
```

В следующем простом примере находится подстрока '9938', самое длинное из *всех* совпадений:

```
$longest_match = undef; # Переменная для хранения текущей длины

"800-998-9938" =~ m{ \d+ $RecordPossibleMatch }x;

# Вывести накопленный результат, если были найдены совпадения
if (defined($longest_match)) {
    print "longest match=[$longest_match]\n";
} else {
    print "no match\n";
}
```

Поиск самого длинного совпадения, ближнего к левому краю

Разобравшись с поиском самого длинного совпадения во всей строке, мы переходим к следующей задаче — поиску самого длинного совпадения, *ближнего к левому краю*. Именно это совпадение будет найдено механизмом POSIX НКА (☞ 233). Для решения этой задачи необходимо запретить смещение текущей позиции *после* обнаружения совпадения. После обнаружения первого совпадения стандартный процесс возврата найдет все остальные совпадения, начинающиеся с той же позиции (среди которых можно будет выбрать самое длинное), а запрет на смещение предотвратит дальнейший поиск совпадений, начинающихся с последующих позиций.

Perl не предоставляет программисту средств для управления подсистемой смещения, поэтому запретить смещение напрямую не удастся, но желаемого эффекта можно добиться обходным путем — заблокировать поиск в том случае, если переменная `$longest_match` имеет определенное значение. Проверка осуществляется конструкцией `(?{ defined $longest_match})`, но этого недостаточно — в конце концов, это всего лишь проверка. Использование результатов проверки в выражении основано на применении *условной конструкции*.

Использование встроенного кода в условных конструкциях

Чтобы механизм регулярных выражений отреагировал на результат проверки, мы включим проверку в часть *if* условной конструкции `(? if then|else)` (☞ 188). В случае истинности условия регулярное выражение должно прекратить дальнейшие поиски, поэтому в части *then* используется заведомо несовпадающее подвы-

ражение ⌜(?!)⌟ (часть *else* не нужна, поэтому она просто опускается). Следующий объект регулярного выражения инкапсулирует условную конструкцию:

```
my $BailIfAnyMatch = qr/(?(?{ defined $longest_match})(?!))/;
```

Часть *if* подчеркнута, а часть *then* выделена жирным шрифтом. Ниже приведен пример практического применения этого объекта в сочетании с `$RecordPossibleMatch`:

```
"800-998-9938" =~ m{ $BailIfAnyMatch \d+ $RecordPossibleMatch }x;
```

Команда находит '`800`' — самое длинное совпадение, ближнее к левому краю (по стандарту POSIX).

Ключевое слово local во встроенном коде

Во встроенном коде ключевое слово `local` интерпретируется особым образом. Но чтобы усвоить этот материал, необходимо хорошо понять суть *динамической области видимости* (☞ 371) и разобраться в «крошечной аналогии» из главы 4, приведенной при обсуждении принципов работы механизма НКА, управляемого регулярными выражениями (☞ 209). Следующий хитроумный (но как вы вскоре увидите, ущербный) пример поможет вам в этом убедиться. Приведенный фрагмент сначала проверяет, состоит ли строка только из ⌜\w+⌟ и ⌜\s+⌟, а затем подсчитывает, сколько из ⌜\w+⌟ в действительности являются ⌜\d+\b⌟:

```
my $Count = 0;

$text =~ m{
    ^ (?> \d+ (?{ $Count++ }) \b | \w+ | \s+ )* $
}x;
```

Если применить это выражение к строке вида '`123•abc•73•9271•xyz`', в переменную `$Count` заносится значение 3. Однако для строки '`123•abc•73xyz`' значение переменной равно `2`, хотя должно быть равно только `1`. Дело в том, что переменная `$Count` обновляется после обнаружения совпадения в тексте '`73`' (совпадает с подвыражением ⌜\d+⌟), которое позднее отменяется из-за возврата, связанного с отсутствием совпадения для следующего подвыражения ⌜\b⌟. Проблема возникает потому, что встроенный код некоторым образом «выполняется обратно», когда совпадение соответствующей части регулярного выражения отменяется из-за возврата.

Если вы еще не полностью разобрались с атомарной группировкой ⌜(?>…)⌟ (☞ 187) и происходящими возвратами, я поясню, что атомарная группировка используется для предотвращения бесконечного поиска (☞ 341) и не влияет на возвраты в границах конструкции — только на *возврат внутрь* конструкции после выхода из нее.

ИНТЕРПОЛЯЦИЯ ВСТРОЕННОГО КОДА В PER

В качестве меры безопасности Perl обычно не разрешает интерполяцию встроенного кода ⌜(?{…})⌟ и динамических подвыражений ⌜(??{…})⌟ в строковых переменных (хотя это разрешено для объектов регулярных выражений, как в примере $RecordPossib leMatch на с. 419). Иначе говоря, команда

```
m{ (?{ print "starting\n" }) выражение… }x;
```

допустима, а следующий фрагмент недопустим:

```
my $ShowStart = '(?{ print "starting\n" })';
   ⋮
m{ $ShowStart выражение… }x;
```

Данное ограничение связано с тем, что пользовательский ввод издавна включается в регулярные выражения, но появление новых конструкций, позволяющих выполнить произвольный код Perl, создает огромную брешь в системе безопасности. Поэтому по умолчанию подобная интерполяция запрещена.

Если вы все же хотите ее разрешить, ограничение снимается директивой

```
use re 'eval';
```

(Директива use re с различными параметрами часто применяется при отладке; ☞ 451.)

Проверка пользовательского ввода перед интерполяцией

Если вы используете эту конструкцию и хотите разрешить интерполяцию пользовательского ввода, сначала убедитесь в том, что он не содержит встроенного кода Perl и динамических регулярных выражений. Для проверки можно воспользоваться регулярным выражением ⌜\(\s*\?+[p{]⌟. Если для него находится совпадение во входных данных, использовать такое выражение небезопасно. Присутствие ⌜\s+⌟ объясняется тем, что модификатор /x допускает наличие пробелов после открывающих круглых скобок (мне кажется, что этого делать не следовало, но факт остается фактом). Плюс квантифицирует ⌜\?⌟, чтобы распознавались обе опасные конструкции. Наконец, символ p учитывает возможное использование устаревшей конструкции ⌜(?p{…}⌟, предшествовавшей появлению конструкции ⌜(??{…})⌟.

Пожалуй, в Perl следовало бы создать какой-нибудь модификатор, позволяющий разрешить или запретить применение встроенного кода на уровне регулярного выражения или подвыражений, но до появления такого модификатора вам придется проверять пользовательский ввод самостоятельно.

Следовательно, при отсутствии совпадения для ⌜\b⌟ совпадение ⌜\d+⌟ вполне может быть отменено вследствие возврата.

У задачи имеется простое решение — ⌜\b⌟ ставится перед увеличением $Count, чтобы переменная изменялась лишь тогда, когда ее модификация не будет отменена. И все же я приведу другое решение с переменной local, чтобы продемонстрировать ее эффект при выполнении кода Perl в процессе поиска совпадений. Рассмотрим новую версию:

```
our $Count = 0;

$text =~ m{
    ^ (?> \d+ (?{ local ($Count) = $Count + 1 }) \b | \w+ | \s+ )* $
}x;
```

Прежде всего обратите внимание на то, что переменная $Count из переменной my превратилась в глобальную переменную (если использовать директиву **use strict**, как я всегда рекомендовал, вы не сможете использовать неуточненную глобальную переменную без ее «объявления» с ключевым словом our).

Другое изменение — локализация приращения $Count. Это очень принципиальный момент: *если переменная локализована в регулярном выражении, при «отмене» кода, содержащего* local, *вследствие возврата восстанавливается исходное значение переменной (а новое значение теряется)*. Следовательно, хотя команда local($Count) = $Count+1 выполняется после совпадения '73' с подвыражением ⌜\d+⌟, а переменная $Count становится равной 2, это изменение локализуется «на пути успешного совпадения» того регулярного выражения, в котором находится local. Когда поиск совпадения для ⌜\b⌟ завершается неудачей, происходит логический возврат к состоянию перед local, а переменная $Count возвращается к исходному значению 1. Именно это значение будет сохранено при завершении поиска.

Итак, ключевое слово local нужно для того, чтобы переменная $Count правильно отражала текущее состояние поиска до конца регулярного выражения. Если включить в конец выражения конструкцию ⌜(?{ print "Final count is $Count.\n" })⌟, она выведет правильное значение счетчика. Значение $Count должно использоваться после поиска, поэтому перед «официальным» завершением поиска его необходимо сохранить в нелокализованной переменной (после завершения все значения, локализованные в процессе поиска, теряются).

Пример:

```
my $Count = undef;
our $TmpCount = 0;
$text =~ m{
    ^ (?> \d+ (?{ local($TmpCount) = $TmpCount + 1 }) \b | \w+ | \s+ )* $
```

```
     (?{ $Count = $TmpCount }) # Сохранить итоговое значение $Count
                               # в нелокализованной переменной
}x;
if (defined $Count) {
    print "Count is $Count.\n";
} else {
    print "no match\n";
}
```

На первый взгляд решение кажется слишком сложным для такой простой задачи, но я напомню, что этот искусственный пример всего лишь демонстрирует механику использования локализованных переменных в регулярном выражении. Примеры ее практического применения приведены в разделе «Имитация именованного сохранения» на с. 433.

Встроенный код и переменные my

Если переменная `my` объявляется *за пределами* регулярного выражения, но ссылки на нее присутствуют во встроенном коде *внутри* регулярного выражения, необходимо помнить об одной тонкости, которая приводит к весьма серьезным последствиям. Прежде чем останавливаться на этом вопросе, стоит заметить, что если во встроенном коде используются только глобальные переменные, этой проблемы не возникает, поэтому чтение этого раздела можно смело пропустить. И еще одно предупреждение: материал этого раздела отнюдь не является «легким чтивом».

Суть проблемы поясняет следующий пример:

```
sub CheckOptimizer
{
    my $text = shift;  # Первый аргумент - проверяемый текст
    my $start = undef; # Переменная для хранения начальной позиции

    my $match = $text =~ m{
        (?{ $start = $-[0] if not defined $start}) # Сохранение исходной
                                                   # начальной позиции
        \d # Проверяемое регулярное выражение
    }x;

    if (not defined $start) {
        print "The whole match was optimized away.\n";
        if ($match) {
            # Такого быть не должно!
            print "Whoa, but it matched! How can this happen!?\n";
        }
    } elsif ($start == 0) {
```

```
        print "The match start was not optimized.\n";
    } else {
        print "The optimizer started the match at character $start.\n"
    }
}
```

В программе объявлены три переменные `my`, но лишь одна переменная — `$start` — относится к рассматриваемой теме (остальные переменные не вызываются из встроенного кода). Сначала переменной `$start` присваивается неопределенное значение, а затем производится поиск; выражение начинается с конструкции встроенного кода, которая присваивает `$start` начальную позицию совпадения, но лишь в том случае, если значение переменной не было задано ранее. «Начальная позиция» определяется `$-[0]` (первым элементом массива `@` ☞ 377).

Итак, при вызове функции вида

```
CheckOptimizer("test 123");
```

будет получен следующий результат:

```
The optimizer started the match at character 5.
```

Вроде бы все хорошо, но если повторить тот же вызов, результат окажется совсем другим:

```
The whole match was optimized away.
Whoa, but it matched! How can this happen!?
```

Хотя проверяемый текст остался прежним (как и само регулярное выражение), результат сильно изменился. Похоже, выражение стало работать неправильно. Почему? Потому что при втором проходе встроенный код обновляет ту же переменную `$start`, которая существовала на первом проходе, при компиляции регулярного выражения. С другой стороны, переменная `$start`, используемая далее в функции, в действительности является новой переменной, созданной при обработке ключевого слова `my` в начале вызова функции.

Дело в том, что переменные `my` во встроенном коде ассоциируются с конкретным экземпляром («связываются», как говорят программисты) переменной `my`, активной *на момент* компиляции регулярного выражения (компиляция регулярных выражений подробно описана на с. 435). При каждом вызове `CheckOptimizer` создается *новый экземпляр* переменной `$start`, но переменная `$start` во встроенном коде загадочным образом продолжает относиться к первому экземпляру, который давно перестал существовать. Таким образом, экземпляр `$start`, используемый функцией, не получает значения, присвоенного ему в регулярном выражении.

Подобная привязка к экземпляру называется *замыканием* (*closure*). В «*Programming Perl*» и «*Object Oriented Perl*» подробно объясняется, для чего была предусмотрена эта

возможность. Впрочем, в сообществе Perl до сих пор идут дискуссии относительно полезности замыканий. Многие программисты считают их крайне нелогичными.

Как решить эту проблему? Не ссылайтесь на переменные `my` в регулярном выражении, если вы не уверены в том, что литерал регулярного выражения будет компилироваться по крайней мере с той же периодичностью, с какой обновляются экземпляры. Например, переменная `my $NestedStuffRegex` используется в функции `SimpleConvert`, в листинге на с. 432, но мы точно знаем, что это не вызовет проблем, поскольку в программе существует только один экземпляр `$NestedStuffRegex`. Ключевое слово `my` не находится ни в функции, ни в цикле, поэтому переменная создается всего один раз при загрузке сценария и продолжает существовать до завершения программы.

Поиск вложенных конструкций

В примере на с. 412 показано, как организовать поиск вложенных парных конструкций произвольного уровня с применением динамических регулярных выражений. Вообще говоря, это самое простое решение, но в учебных целях весьма поучительно рассмотреть другой метод, использующий только встроенный код.

Решение строится на простом принципе: программа подсчитывает открывающие круглые скобки, не закрытые до текущего момента, и разрешает закрывающие скобки лишь при наличии незакрытых пар. Состояние счетчика изменяется во встроенном коде в процессе поиска. Прежде чем анализировать конкретное выражение, рассмотрим заготовку:

```
my $NestedGuts = qr{
    (?>
        (?:
            # Любые символы, кроме круглых скобок
            [^()]+
            # Открывающая круглая скобка
            | \(
            # Закрывающая круглая скобка
            | \)
        )*
    )
)x;
```

Атомарная группировка повышает эффективность поиска и предотвращает бесконечный перебор совпадений для `「([…]+|…)*」` (☞ 289), если `$NestedGuts` используется как часть большего выражения, в котором может произойти возврат. Например, если использовать `$NestedGuts` в конструкции `m/^\( $NestedGuts \)$/x` и применить ее к тексту '`(this•is•missing•the•close`', дело кончится длительным

перебором вариантов, если не отсечь лишние состояния при помощи атомарной группировки.

Подсчет круглых скобок выполняется в четыре этапа:

❶ Перед началом подсчета счетчик инициализируется нулевым значением:

```
(?{ local $OpenParens = 0 })
```

❷ При обнаружении открывающей скобки счетчик открытых пар скобок увеличивается:

```
(?{ $OpenParens++ })
```

❸ При обнаружении закрывающей скобки мы проверяем состояние счетчика, и если оно положительно, уменьшаем его на единицу (незакрытых пар стало на единицу меньше). Если счетчик равен нулю, продолжение поиска невозможно, так как закрывающие круглые скобки не согласуются с открывающими, поэтому мы применяем подвыражение `(?!)`, обеспечивающее неудачу при поиске:

```
(?(?{ $OpenParens }) (?{ $OpenParens--}) | (?!) )
```

В этом фрагменте используется условная конструкция **(?** *if then* **|** *else* **)** (☞ 188), а проверка счетчика в условии *if* производится встроенным кодом.

❹ После завершения поиска следует проверить, равен ли счетчик нулю. Если счетчик отличен от нуля, количество открывающих и закрывающих скобок в программе не совпадает, поэтому поиск завершается неудачей:

```
(?(?{ $OpenParens != 0 })(?!))
```

Включив все упомянутые элементы в выражение, мы получаем:

```
my $NestedGuts = qr{
    (?{ local $OpenParens = 0 }) # ❶ Счетчик незакрытых круглых скобок
    (?>   # атомарная группировка для повышения эффективности
        (?:
            # Все, что не является круглыми скобками
            [^()]+
            # ❷ Открывающая круглая скобка
            | \( (?{ $OpenParens++ })
            # ❸ Разрешается закрывающая круглая скобка
            # если имеются незакрытые пары
            | \) (?(?{ $OpenParens != 0 }) (?{ $OpenParens--}) | (?!) )
        )*
     )
    (?(?{ $OpenParens != 0 })(?!)) # ❹ Если остались незакрытые пары скобок,
                                   # поиск завершается неудачей
}x;
```

Полученный объект используется аналогично переменной `$LevelN` (☞ 414).

Ключевое слово `local` используется из соображений предосторожности, чтобы отделить использование `$OpenParens` от остальных обращений к глобальной переменной в программе. В отличие от предыдущего раздела здесь `local` не используется для защиты от возвратов, поскольку атомарная группировка в регулярном выражении предотвращает «отмену» совпавших альтернатив. В данном случае атомарная группировка используется в целях эффективности, а также дает абсолютную гарантию того, что совпадения рядом со встроенным кодом не будут отменены, что привело бы к рассогласованию значения `$OpenParens` и количества фактически совпавших скобок.

Перегрузка литералов регулярных выражений

Перегрузка (*overloading*) позволяет выполнить предварительную обработку литеральных частей литерала регулярного выражения. Далее рассматриваются некоторые примеры практического применения перегрузки.

Метасимволы начала и конца слов

Perl не поддерживает метасимволы ⌜`\<`⌟ и ⌜`\>`⌟, обозначающие начало и конец слова. Наверное, это объясняется тем, что в большинстве случаев ⌜`\b`⌟ оказывается вполне достаточно. Тем не менее поддержку этих метасимволов можно реализовать самостоятельно. Воспользуйтесь перегрузкой и организуйте замену конструкций '`\<`' и '`\>`' в регулярном выражении конструкциями ⌜`(?<!\w)(?=\w)`⌟ и ⌜`(?<=\w)(?!\w)`⌟ соответственно.

Сначала определяется функция (скажем, `MungeRegexLiteral`), которая выполняет нужную предварительную обработку:

```
sub MungeRegexLiteral($)
{
    my ($RegexLiteral) = @_; # Строковый аргумент
    $RegexLiteral =~ s/\\</(?<!\\w)(?=\\w)/g; # \< имитирует начало слова
    $RegexLiteral =~ s/\\>/(?<=\\w)(?!\\w)/g; # \> имитирует конец слова
    return $RegexLiteral; # Вернуть строку (возможно, модифицированную)
}
```

Когда этой функции передается строка типа '...`\<`...', она преобразует ее и возвращает строку '...`(?<!\w)(?=\w)`...'. Напомню, что строка замены воператоре `s/.../.../` воспринимается как обычная строка в кавычках, поэтому, чтобы получить значение '`\w`', необходимо указывать '`\\w`'.

Чтобы эта функция автоматически вызывалась для каждой литеральной части литерала регулярного выражения, сохраните ее в файле (например, *MyRegexStuff.pm*) с использованием средств перегрузки Perl:

```
package MyRegexStuff; # Пакету следует присвоить уникальное имя
use strict;   # Рекомендуется использовать всегда
use warnings; # Рекомендуется использовать всегда
use overload; # Разрешает использовать механизм перегрузки в Perl
# Назначение обработчика регулярных выражений
sub import { overload;:constant qr => \&MungeRegexLiteral }

sub MungeRegexLiteral($)
{
    my ($RegexLiteral) = @_; # Строковый аргумент
    $RegexLiteral =~ s/\\</(?<!\\w)(?=\\w)/g; # \< имитирует начало слова
    $RegexLiteral =~ s/\\>/(?<=\\w)(?!\\w)/g; # \> имитирует конец слова
    return $RegexLiteral; # Вернуть строку (возможно, модифицированную)
}

1; # Стандартная идиома. Включается для того, чтобы вызов use
   # для данного файла возвращал true
```

Разместив файл *MyRegexStuff.pm* в библиотечном каталоге Perl (см. раздел PERLLIB в документации Perl), вы сможете вызывать его из сценариев Perl, в которых задействованы новые возможности. Впрочем, в процессе тестирования его можно оставить в одном каталоге с тестируемым сценарием и включить в программу фрагмент вида

```
use lib '.';      # Библиотечные файлы находятся в текущем каталоге
use MyRegexStuff; # Новая возможность становится доступной!
   ⋮
$text =~ s/\s+\</ /g; # Любые виды пропусков перед словом нормализуются
                      # до одного пробела.
   ⋮
```

Директива `use MyRegexStuff` должна присутствовать во всех файлах, в которых задействована поддержка дополнительных метасимволов в литералах регулярных выражений, но файл *MyRegexStuff.pm* достаточно создать всего один раз (кстати, в самом файле *MyRegexStuff.pm* новые возможности недоступны, поскольку в нем отсутствует директива **`use MyRegexStuff`**, — поверьте, делать этого не стоит).

Поддержка захватывающих квантификаторов

Попробуем дополнить файл *MyRegexStuff.pm* и включить в него поддержку захватывающих квантификаторов типа `x++` (☞ 190). Захватывающие квантификаторы работают как обычные максимальные квантификаторы, если не считать

того, что они никогда не уступают (т. е. не возвращают) текст из найденного совпадения. Они легко имитируются при помощи атомарной группировки, для чего достаточно убрать завершающий знак '+' и заключить всю конструкцию в атомарные ограничители. Например, ⌜выражение*+⌟ превращается в ⌜(?>*выражение**)⌟ (☞ 229).

Выражение может быть конструкцией в круглых скобках, метапоследовательностью наподобие ⌜\w⌟ или ⌜\x{1234}⌟, и даже простым символом. Обработка всех возможных случаев получается громоздкой, поэтому для простоты ограничимся применением квантификаторов ?+, *+ и ++ к выражению в круглых скобках. Воспользуемся объектом $LevelN (с. 415) и включим в функцию MungeRegexLiteral команду:

```
$RegexLiteral =~ s/( \( $LevelN \)[*+?] )\+/(?>$1)/gx;
```

Вот и все! После включения этой команды в пакет перегрузки в литералах регулярных выражений можно использовать захватывающие квантификаторы:

```
$text =~ s/"(\\.|[^"])*+"//; # Удаление строк в кавычках
```

С другими квантифицированными выражениями дело обстоит сложнее, поскольку синтаксис регулярного выражения может быть весьма разнообразным. Одна из попыток выглядит так:

```
$RegexLiteral =~ s{
 (
    # Поиск квантифицируемой конструкции...
    (?: \\[\\abCdDefnrsStwWX] # \n, \w и др.
      | \\c. # \cA
      | \\x[\da-fA-F]{1,2}     # \xFF
      | \\x\{[\da-fA-F]*\}     # \x{1234}
      | \\[pP]\{[^{}]+\}       # \p{Letter}
      | \[\]?[^]]+\]           # Упрощенный класс
      | \\\W                   # \*
      | \( $LevelN \)          # (…)
      | [^()*+?\\]             # Почти все остальное
    )
    # ...Квантифицируется...
    (?: [*+?] | \{\d+(?:,\d*)?\} )
)
\+  # ...и содержит дополнительный '+' после квантификатора.
}{(?>$1)}gx;
```

Общий принцип работы регулярного выражения не изменился: найти квантифицируемый элемент, удалить '+' и заключить результат в конструкцию ⌜(?>…)⌟. Впрочем, это всего лишь упрощенное подобие сложного синтаксиса регулярных выражений

Perl. Особенно нуждается в усовершенствовании подвыражение для символьного класса, в котором не распознаются экранированные символы. Более того, недостатки заложены в самом подходе, поскольку он не учитывает некоторых аспектов регулярных выражений Perl. Например, столкнувшись с конструкцией '`\(blah\)++`', приведенный фрагмент не проигнорирует открывающую скобку и решит, что ⌜`++`⌟ относится к чему-то большему, чем ⌜`\)`⌟.

Решение этих проблем потребует основательного труда. Возможно, стоит воспользоваться методикой, основанной на последовательном переборе компонентов регулярного выражения от начала к концу (по аналогии с решением, приведенным на врезке на с. 178). Подвыражение для символьного класса будет доработано, но тратить время на остальные компоненты не стоит по двум причинам. Во-первых, недостатки остальных компонентов проявляются лишь в нетривиальных ситуациях, поэтому исправление только символьного класса уже сделает это выражение пригодным для практического применения. Во-вторых, у перегрузки регулярных выражений Perl в настоящее время имеется фатальный недостаток (об этом далее), из-за которого она приносит значительно меньше пользы, чем могла бы.

Ограничения перегрузки литералов регулярных выражений

Перегрузка регулярных выражений могла бы стать чрезвычайно мощным средством (по крайней мере, теоретически), но, к сожалению, на практике она особой пользы не приносит. Дело в том, что перегружаются только *литеральные*, но не интерполированные части литералов регулярных выражений. Например, при выполнении команды `m/($MyStuff)*+/` функция `MungeRegexLiteral` вызывается дважды — для литеральной части выражения перед интерполяцией («`(`») и после нее («`)*+`»). Содержимое `$MyStuff` ей никогда не передается. Нашей функции обе составляющие нужны одновременно, поэтому интерполяция переменной фактически нарушает ее работу.

Для поддержки метасимволов `\<` и `\>`, о которых говорилось выше, эта проблема не столь существенна, поскольку эти последовательности вряд ли будут разбиваться интерполируемыми переменными. Но поскольку перегрузка не относится к содержимому интерполированных переменных, внутренние вхождения '`\<`' и '`\>`' в интерполированной строке или в объекте регулярного выражения в процессе перегрузки не обрабатываются. Кроме того, как упоминалось в предыдущем разделе, при обработке литерала регулярного выражения довольно трудно обеспечить абсолютную точность. Проблемы возникают даже с такими простыми конструкциями, как наша поддержка `\>`, когда она вмешивается в обработку строки '`\\>`', обозначающей «символ '\', за которым следует '>'». У перегрузки есть и другой

ИМИТАЦИЯ ИМЕНОВАННОГО СОХРАНЕНИЯ

```
package MyRegexStuff;
use strict;
use warnings;
use overload;
sub import { overload::constant('qr' => \&MungeRegexLiteral) }

my $NestedStuffRegex; # Переменная используется в собственном определении.
                      # Поэтому она должна определяться заранее.
$NestedStuffRegex = qr{
(?>
  (?: # Не круглые скобки, не '#' и не '\'...
      [^()\#\\]+
      # Экранирование...
    | (?s: \\. )
      # Комментарии в регулярном выражении...
    | \#.*\n
      # Круглые скобки, внутри которых могут находиться
      # другие вложенные конструкции...
    | \( (??{ $NestedStuffRegex }) \)
  )*
)
}x;

sub SimpleConvert($); # Функция вызывается рекурсивно, поэтому
                      # ее необходимо объявить заранее
sub SimpleConvert($)
{
  my $re = shift; # Регулярное выражение для обработки
  $re =~ s{
    \(\? # "(?"
      < ( (?>\w+) ) >        # <$1 > $1  идентификатор
      ( $NestedStuffRegex )  # $2 – вложенные конструкции
    \)                       # ")"
}{
    my $id    = $1;
    my $guts = SimpleConvert($2);
    # Мы заменяем
    #     (?<id>guts)
    # на
    #     (?: (guts) # идентификация содержимого
    #       (?{
    #         local($^N{$id}) = $guts # Сохранить в локализованном
                                      # элементе %^T
    #       })
    #     )
```

```
    "(?:($guts)(?{ local(\$^T{'$id'}) = \$^N }))"
  }xeog;
  return $re; # Вернуть обработанное регулярное выражение
}

sub MungeRegexLiteral($)
{
  my ($RegexLiteral) = @_; # Аргумент - строка
  # print "BEFORE: $RegexLiteral\n"; # Снять комментарий при отладке
  my $new = SimpleConvert($RegexLiteral);
  if ($new ne $RegexLiteral)
  {
    my $before = q/(?{ local(%^T) = () })/; # Локализация временного хеша
    my $after = q/(?{ %^N = %^T })/; # Копирование временного хеша
                                     # в "настоящий"
   $RegexLiteral = "$before(?:$new)$after";
  }
  # print "AFTER: $RegexLiteral\n"; # Снять комментарий при отладке
  return $RegexLiteral;
}

1;
```

недостаток — в процессе перегрузки ничего не известно о модификаторах, с которыми применяется регулярное выражение. Например, обработка выражения в значительной степени зависит от наличия модификатора `/x`, но пока эта информация остается недоступной.

Также следует заметить, что применение перегрузки блокирует возможность включения символов по именам Юникода (конструкция `「\N{`*имя*`}」` ☞ 365).

Имитация именованного сохранения

Невзирая на все недостатки перегрузки, будет полезно рассмотреть сложный пример, в котором объединяются многие специальные конструкции. Именованное сохранение (☞ 186) в Perl не поддерживается, однако оно легко имитируется при помощи сохраняющих круглых скобок и переменной `$^N` (☞ 377), содержащей текст совпадения для последней закрытой пары круглых скобок (кстати, я настоял на включении в Perl переменной `$^N` именно для того, чтобы имитировать именованное сохранение).

Рассмотрим простой пример:

```
「href\s*=\s*($HttpUrl)(?{ $url = $^N })」
```

В этом выражении используется объект регулярного выражения $HttpUrl, созданный на с. 382. Подчеркнутая конструкция встроенного кода сохраняет текст, совпавший с $HttpUrl, в переменной $url. В этой простой ситуации применение $^N вместо $1 (и вообще применение встроенного кода) выглядит чрезмерным; на первый взгляд, было бы проще использовать $1 после поиска. Но подумайте, что произойдет, если инкапсулировать часть этого выражения в объекте и затем несколько раз применить его:

```
my $SaveUrl = qr{
    ($HttpUrl)        # Найти HTTP URL...
    (?{ $url = $^N }) # ...и сохранить в переменной $url
}x;
$text =~ m{
    http \s*=\s*   ($SaveUrl)
  | src  \s*=\s* ($SaveUrl)
}xi;
```

Переменной $url присваивается найденный URL. В этом конкретном случае также можно было прибегнуть к другим средствам (например, переменной $+ ☞ 379), однако объект $SaveUrl может использоваться в более сложных ситуациях, затрудняющих сопровождение решений со служебными переменными, поэтому сохранить URL в именованной переменной может оказаться значительно удобнее.

Недостаток этого примера заключается в том, что данные, записанные в $url, не восстанавливаются при отмене конструкции, осуществившей запись, в результате возврата. Следовательно, в процессе поиска следует использовать локализованную временную переменную и осуществить запись в «настоящую» переменную только после подтверждения общего совпадения, как в примере на с. 424.

Приведенный выше листинг демонстрирует один из возможных способов. С точки зрения пользователя, после применения ⌜(?<Num>\d+)⌟ число, совпавшее с ⌜\d+⌟, доступно через глобальный хеш %^N в виде $^N{Num}. Хотя в будущих версиях Perl может появиться специальная системная переменная %^N, в настоящее время это имя свободно, а его использование ничем не ограничивается.

Я мог бы выбрать имя вида %NamedCapture, но выбрал %^N, что объясняется несколькими причинами. Во-первых, это имя похоже на $^N. Во-вторых, его не нужно заранее объявлять с ключевым словом our при использовании директивы use strict. Наконец, я надеюсь, что со временем в Perl появится встроенная поддержка именованного сохранения, и как мне кажется, ее реализация через %^N выглядела бы вполне логично. Если это произойдет, имя %^N будет автоматически приведено к динамической области видимости, как и остальные специальные переменные, связанные с регулярными выражениями (☞ 376). Но на данный момент это обычная глобальная переменная, для которой автоматическое приведение к динамической области видимости не поддерживается.

Новое решение получилось более сложным, но и оно обладает всеми недостатками, присущими решениям на базе перегрузки литералов регулярных выражений (несовместимостью с интерполяцией переменных и т. д.).

Проблемы эффективности в Perl

Проблемы эффективности в Perl обычно решаются так же, как и во всех остальных программах с традиционным механизмом НКА — за счет использования приемов, описанных в главе 6: внутренних оптимизаций, раскрутки и др. Все эти приемы относятся и к Perl.

Конечно, существуют приемы, специфические для Perl. В этом разделе будут рассмотрены перечисленные ниже темы.

- **У каждой задачи есть несколько решений.** Perl — набор инструментов, позволяющий решить одну и ту же задачу несколькими способами. Умение грамотно идентифицировать задачу основано на осознании «пути Perl», а умение выбрать правильный инструмент для ее решения является большим шагом на пути к построению более эффективных и понятных программ. Иногда эффективность и наглядность программы кажутся взаимоисключающими требованиями; тем не менее лучшее понимание вопроса поможет вам с правильным выбором средств.

- **Компиляция регулярных выражений, `qr/…/`, модификатор `/o` и эффективность.** Интерполяция и компиляция выражений-операндов открывают широкие возможности для экономии времени. Модификатор /o, который практически не рассматривался, в сочетании с объектами регулярных выражений (`qr/…/`) позволяет до определенной степени управлять выбором момента повторной компиляции, обычно связанной с большими затратами.

- **Затраты `$&`.** Значения трех переменных `$`` , `$&` и `$'` присваиваются в качестве побочного эффекта поиска. Эти переменные удобны, но любое использование их в сценарии *отрицательно* влияет на его эффективность. Более того, эти переменные даже не обязательно использовать — весь сценарий страдает даже в том случае, если одна из этих переменных просто *присутствует* в нем.

- **Study.** Функция `study(…)` существует в Perl с незапамятных времен. Как правило, многие что-то слышали о том, что `study` ускоряет обработку регулярных выражений, но лишь немногие понимают, что же именно она делает. Посмотрим, удастся ли нам в этом разобраться.

- **Хронометраж.** В конечном счете самая быстрая программа — та, которая первой заканчивает свою работу. Хронометраж всегда позволяет наиболее объективно

оценить скорость работы программного кода, будь то маленькая функция, большой фрагмент или целая программа, работающая с реальными данными. В Perl существуют простые и удобные средства хронометража, хотя у этой задачи, как у большинства остальных, тоже существует несколько решений. Я покажу вам тот способ, которым пользуюсь сам, — простой прием, при помощи которого я провел несколько сотен тестов во время работы над книгой.

- **Отладка регулярных выражений.** При помощи флага отладки регулярных выражений Perl можно получить информацию о том, какие виды оптимизаций выполняются или не выполняются механизмом регулярных выражений. Вскоре вы узнаете, как это делается, и заставите Perl поделиться некоторыми секретами.

У каждой задачи есть несколько решений

Довольно часто к решению конкретной задачи можно подойти разными способами, поэтому при балансировании между эффективностью и наглядностью программ ничто не заменит четкого понимания того, чем руководствуется в работе Perl. Рассмотрим простой пример — дополнение IP-адресов (`18.181.0.24`) нулями, чтобы каждый из четырех компонентов состоял ровно из трех цифр (`018.181.000.024`). Одно простое и наглядное решение выглядит так:

```
$ip = sprintf ("%03d.%03d.%03d.%03d", split(/\./, $ip)):
```

Это хорошее, но не единственное решение. Для сравнения в табл. 7.6 приведены различные способы решения той же задачи с указанием относительной эффективности (по ее убыванию). Конечно, наш пример прост и не особенно интересен, однако подобные ситуации часто возникают при обработке текстов, поэтому я рекомендую потратить немного времени на изучение различных решений. Возможно, вы даже найдете что-нибудь новое.

Все эти решения для правильного IP-адреса выдают те же результаты, что и исходный вариант, но при ошибке в данных каждое из них ведет себя по-своему. Если существует вероятность получения искаженных данных, любого из этих решений окажется недостаточно. Кроме того, между приведенными решениями существуют практические отличия, связанные с эффективностью и наглядностью. Впрочем, что касается наглядности, варианты 1 и 13 выглядят наиболее тривиально (тем более интересно, что они так существенно различаются по эффективности). Также достаточно просты варианты 3, 4 (аналоги 1) и 8 (аналог 13). Остальные решения выглядят в лучшем случае загадочно.

А как же эффективность? Почему некоторые решения уступают другим по эффективности? Это объясняется различиями во взаимодействии с механизмом НКА

(глава 4), применением многих оптимизацией Perl (глава 6) и скоростью обработки других конструкций Perl (например, `sprintf` и оператора подстановки). Модификатор `/e` иногда оказывает неоценимую помощь, но в данном случае он оказывается в нижней части списка.

Интересно сравнить две пары: 3–4 и 8–14. Регулярные выражения каждой пары отличаются только использованием круглых скобок — выражения без скобок работают чуть быстрее. Но переменная `$&` в варианте 8, позволяющая обойтись без круглых скобок, обходится слишком дорого. Мы вернемся к этой теме на с. 444.

Таблица 7.6. Варианты дополнения IP-адреса нулями

Ранг	Время	Решение
1	1,0×	`$ip = sprintf("%03d.%03d.%03d.%03d", split(m/\./, $ip));`

2	1,3×	`substr($ip, 0, 0)  = '0' if substr($ip, 1,  1) eq '.';`
		`substr($ip, 0, 0)  = '0' if substr($ip, 2,  1) eq '.';`
		`substr($ip, 4, 0)  = '0' if substr($ip, 5,  1) eq '.';`
		`substr($ip, 4, 0)  = '0' if substr($ip, 6,  1) eq '.';`
		`substr($ip, 8, 0)  = '0' if substr($ip, 9,  1) eq '.';`
		`substr($ip, 8, 0)  = '0' if substr($ip, 10, 1) eq '.';`
		`substr($ip, 12, 0) = '0' while length($ip) < 15;`

3	1,6×	`$ip = sprintf("%03d.%03d.%03d.%03d", $ip =~ m/\d+/g);`
4	1,8×	`$ip = sprintf("%03d.%03d.%03d.%03d", $ip =~ m/(\d+)/g);`
5	1,8×	`$ip = sprintf("%03d.%03d.%03d.%03d",` `$ip =~ m/^(\d+)\.(\d+)\.(\d+)\.(\d+)$/);`
6	2,3×	`$ip =~ s/\b(?=\d\b)/00/g; $ip =~ s/\b(?=\d\d\b)/0/g;`
7	3,0×	`$ip =~ s/\b(\d(\d?)\b)/$2 eq '' ? "00$1" : "0$1"/eg;`
8	3,3×	`$ip =~ s/\d+/sprintf("%03d", $&)/eg;`
9	3,4×	`$ip =~ s/(?:(?<=\.)\|^)(?=\d\b)/00/g;` `$ip =~ s/(?:(?<=\.)|^)(?=\d\d\b)/0/g;`
10	3,4×	`$ip =~ s/\b(\d\d?\b)/'0' x (3-length($1)) . $1/eg;`
11	3,4×	`$ip =~ s/\b(\d\b)/00$1/g; $ip =~ s/\b(\d\d\b)/0$1/g;`
12	3,4×	`$ip =~ s/\b(\d\d?\b)/sprintf("%03d", $1)/eg;`
13	3,5×	`$ip =~ s/\b(\d{1,2}\b)/sprintf("%03d", $1)/eg;`
14	3,5×	`$ip =~ s/(\d+)/sprintf("%03d", $1)/eg;`
15	3,6×	`$ip =~ s/\b(\d\d?(?!\d))/sprintf("%03d", $1)/eg;`
16	4,0×	`$ip =~ s/(?:(?<=\.)|^)(\d\d?(?!\d))/sprintf("%03d", $1)/eg;`

Компиляция регулярных выражений, модификатор /o, qr/.../ и эффективность

Среди факторов, влияющих на эффективность работы с регулярными выражениями в Perl, стоит особо выделить объем подготовительной работы, выполняемой Perl при передаче управления оператору, перед непосредственным применением регулярного выражения. Конкретные действия зависят от типа операнда регулярного выражения. В самом распространенном случае операнд представляет собой литерал, как в конструкциях `m/…/`, `s/…/` или `qr/…/`. В таких ситуациях Perl приходится выполнять ряд действий, требующих определенного времени, которое нам хотелось бы по возможности сократить. Сначала посмотрим, какая работа при этом выполняется, а затем — как ее избежать.

Внутренняя механика подготовки регулярного выражения

Вспомогательные операции по подготовке регулярных выражений операндов в общих чертах рассматриваются в главе 6 (☞ 308), но в Perl есть своя специфика.

Предварительная обработка операндов в Perl делится на две фазы.

1. **Обработка литералов регулярных выражений.** Если операнд представляет собой литерал регулярного выражения, он обрабатывается так, как описано в разделе «Порядок обработки литералов регулярных выражений» (☞ 367). В частности, на этой стадии осуществляется интерполяция переменных.

2. **Компиляция регулярного выражения.** Регулярное выражение анализируется, и если оно имеет правильный синтаксис, компилируется во внутреннюю форму, непосредственно используемую механизмом регулярных выражений (при наличии ошибок пользователь получает соответствующее сообщение).

После получения откомпилированного регулярного выражения Perl применяет его к целевому тексту. Этот процесс подробно описан в главах 4–6.

Однако предварительную обработку не обязательно выполнять при каждом использовании каждого выражения-операнда. Она всегда выполняется при *первом* использовании литерала регулярного выражения в программе, но если литерал многократно используется в программе (например, в цикле или при повторном вызове функции), Perl иногда может использовать готовые результаты. В нескольких ближайших подразделах будет показано, когда и как это происходит, а также описаны некоторые приемы, позволяющие программисту повысить эффективность поиска.

Сокращение затрат на компиляцию регулярных выражений в Perl

Мы рассмотрим два способа сокращения объема предварительной обработки литералов регулярных выражений: безусловное кэширование и перекомпиляция при необходимости.

Безусловное кэширование

Если литерал регулярного выражения не содержит интерполируемых переменных, Perl знает, что регулярное выражение не изменяется между применениями, поэтому после однократной компиляции выражения откомпилированная форма сохраняется (кэшируется) на случай, если управление будет повторно передано в эту же точку программы. Регулярное выражение анализируется и компилируется всего один раз, независимо от частоты его использования при выполнении программы. Большинство регулярных выражений, приведенных в книге, не содержит интерполируемых переменных, поэтому кэширование обеспечивает максимальную эффективность в этом отношении.

Переменные во встроенном коде и динамические регулярные выражения на кэширование не влияют, поскольку они не *интерполируются* в содержимое регулярного выражения, а лишь являются частью неизменяемого кода, исполняемого регулярным выражением. При использовании переменных `my` во встроенном коде иногда даже требуется, чтобы выражение каждый раз интерпретировалось заново, о чем говорится в предупреждении на с. 424.

Кэширование действует только во время работы программы — между двумя запусками кэшированные данные не сохраняются.

Перекомпиляция при необходимости

Не все выражения-операнды могут кэшироваться. Рассмотрим следующий фрагмент:

```
my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];
# $today содержит обозначение дня недели ("Mon", "Tue" и т.д.)

while (<LOGFILE>) {
    if (m/^$today:/i) {
        ⋮
```

Регулярное выражение в `m/^$today:/` использует интерполяцию, но в цикле оно используется таким образом, что результат интерполяции все время остается одним и тем же. Было бы крайне неэффективно снова и снова компилировать одно и то

же выражение в цикле, поэтому Perl автоматически выполняет простую строковую проверку и сравнивает предыдущий результат интерполяции с текущим. При совпадении результатов кэшированное выражение, использованное в прошлый раз, используется повторно, а необходимость в повторной компиляции отпадает. Но если результаты интерполяции различаются, регулярное выражение компилируется заново. Итак, дополнительные затраты на повторную интерполяцию и сравнение результатов позволяют по возможности избежать относительно «дорогой» повторной компиляции.

Какой же выигрыш обеспечивает такой подход? Весьма значительный. Для примера я измерил время предварительной обработки трех разновидностей `$HttpUrl` на с. 382 (с усовершенствованной версией `$HostnameRegex`). Хронометраж был организован так, чтобы тесты выводили затраты на предварительную обработку регулярного выражения (интерполяцию, проверку строк, компиляцию и другие вспомогательные операции), а не на фактическое применение регулярного выражения, которое оставалось постоянной величиной.

Результаты получились довольно интересными. Сначала я протестировал версию без интерполяции (где все регулярное выражение вводится вручную внутри `m/…/`) и взял полученное время за основу при последующих сравнениях. Подготовка с интерполяцией и проверкой (при неизменяемом исходном выражении) занимает в 25 раз больше времени. Полная предварительная обработка (с перекомпиляцией регулярного выражения при каждом применении) выполняется медленнее почти в 1000 раз!

Чтобы вы лучше понимали реальный смысл этих цифр, необходимо сказать, что даже полная предварительная обработка, выполняемая в 1000 раз медленнее обработки статических литералов регулярных выражений, на моем компьютере занимает всего 0,00026 секунды (в моих тестах за секунду выполнялось 3846 таких операций, а при предварительной обработке статических литералов — 3,7 миллиона операций). И все же отказ от интерполяции дает очень существенную экономию, а без перекомпиляции экономия становится прямо-таки фантастической. В нескольких ближайших разделах вы узнаете, как добиться аналогичных результатов в других ситуациях.

Модификатор однократной компиляции /o

Если применить модификатор `/o` к литералу регулярного выражения, используемому в качестве операнда, этот литерал анализируется и компилируется всего один раз независимо от того, используется в нем интерполяция или нет. При отсутствии интерполяции модификатор `/o` ничего не дает, поскольку выражения без интерполяции всегда кэшируются автоматически. Но при наличии интерполяции при первой передаче управления оператору с литералом регулярного выражения

происходит полная предварительная обработка, а полученная внутренняя форма кэшируется. Если в будущем управление снова будет передано этому оператору, кэшированная форма используется напрямую.

Ниже приведен предыдущий пример с добавлением модификатора `/o`:

```
my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

while (<LOGFILE>) {
    if (m/^$today:/io) {
        ⋮
```

Новый вариант работает гораздо эффективнее, потому что интерполяция `$today` игнорируется во всех итерациях цикла, кроме первой. Отказ от интерполяции или иной предварительной обработки и компиляции регулярного выражения обеспечивает реальную экономию, которую Perl не может реализовать автоматически из-за интерполяции переменных: переменная `$today` *может* измениться, поэтому Perl вынужден заботиться о безопасности и каждый раз проверять ее заново. Используя модификатор `/o`, мы «фиксируем» регулярное выражение после первоначальной обработки и компиляции литерала Подобная фиксация вполне безопасна, если переменные, интерполированные в литерал, остаются неизменными или если в случае их изменения новые значения не должны использоваться.

Потенциальные проблемы при использовании модификатора /o

При использовании модификатора `/o` необходимо помнить об одном важном обстоятельстве. Допустим, наш пример оформляется в виде функции:

```
sub CheckLogfileForToday()
{
    my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

    while (<LOGFILE>) {
        if (m/^$today:/io) { # Опасно!
            ⋮
        }
    }
}
```

Вспомните: модификатор `/o` означает, что выражение-операнд компилируется *однократно*. При первом вызове `CheckLogfileForToday()` операнд, представляющий текущий день недели, фиксируется. Если в будущем функция будет вызвана повторно, а переменная `$today` к тому моменту изменится, ее значение не будет анализироваться заново; исходное зафиксированное значение будет использоваться в течение всей работы программы.

Это серьезный недостаток, но, как будет показано в следующем разделе, данная проблема решается при помощи объектов регулярных выражений.

Эффективность и объекты регулярных выражений

Все, что говорилось выше о предварительной обработке, относилось к *литералам* регулярных выражений, а нашей главной целью было получение откомпилированного регулярного выражения с наименьшими затратами. Другое решение этой задачи основано на использовании *объектов* регулярных выражений, которые фактически представляют собой откомпилированные регулярные выражения, хранящиеся в переменных и готовые к использованию. Объекты регулярных выражений создаются оператором `qr/…/` (☞ 381).

Ниже приведена новая версия нашего примера с использованием объекта регулярного выражения:

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];
  my $RegexObj = qr/^$today:/i; # компилируется при каждом вызове функции
  while (<LOGFILE>) {
      if ($_ =~ $RegexObj) {
          ⋮
      }
  }
}
```

Новый объект регулярного выражения создается при каждом вызове функции, но после этого он напрямую используется в каждой строке файла журнала. Когда объект регулярного выражения используется в качестве операнда, он не проходит предварительную обработку, о которой говорилось выше. Предварительная обработка выполняется *при создании* объекта регулярного выражения, а не при его последующем *использовании*. Объект регулярного выражения можно рассматривать как своего рода «автономный кэш» — это откомпилированное и готовое к использованию регулярное выражение, которое можно применить везде, где вы сочтете нужным.

Данное решение является оптимальным; оно эффективно, поскольку при каждом вызове функции компиляция выполняется только один раз (а не для каждой строки в файле журнала), но в отличие от предыдущего примера, где применение модификатора `/o` приводило к нежелательным последствиям, он корректно работает при повторных вызовах `CheckLogfileForToday()`.

Обратите внимание на то, что в приведенном примере задействованы два операнда. Операндом `qr/…/` является *не* объект регулярного выражения, а литерал, на основа-

нии которого *создается* объект. Затем полученный объект используется в качестве операнда оператора поиска **=~**, вызываемого в цикле.

Использование m/…/ с объектами регулярных выражений

Следующую конструкцию с объектом регулярного выражения

```
if ($_ =~ $RegexObj) {
```

можно записать в виде

```
if (m/$RegexObj/) {
```

Несмотря на внешнее сходство, это не обычный литерал регулярного выражения. Если литерал не содержит ничего, кроме объекта регулярного выражения, его применение эквивалентно применению соответствующего объекта. Такая запись удобна по нескольким причинам. Во-первых, многие считают, что запись `m/…/` выглядит знакомо и с ней удобнее работать. Во-вторых, вам не приходится явно задавать целевую строку `$_`, что в сочетании с другими операторами, использующими этот операнд по умолчанию, улучшает внешний вид программы. Наконец, эта запись позволяет использовать модификатор `/g` с объектами регулярных выражений.

Использование модификатора /o с оператором qr/…/

Модификатор `/o` может использоваться в операторе `qr/…/` (хотя в рассмотренном примере этого, конечно, делать не стоит). По аналогии с другими операторами регулярных выражений, конструкция `qr/…/o` фиксирует регулярное выражение при первом использовании, поэтому в нашем примере это привело бы к тому, что `$RegexObj` будет получать один и тот же объект при каждом вызове функции независимо от значения `$today`. Подобная ошибка была допущена при использовании `m/…/o` на с. 441.

Регулярное выражение по умолчанию

Регулярное выражение по умолчанию (☞ 387) также может использоваться для повышения эффективности, хотя с появлением объектов регулярных выражений необходимость в нем в основном отпала. И все же я кратко опишу этот вариант. Рассмотрим фрагмент:

```
sub CheckLogfileForToday()
{
    my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

    # Перебирать варианты, пока не будет обнаружено совпадение
```

```
    # с заполнением регулярного выражения по умолчанию.
    "Sun:" =~ m/^$today:/i or
    "Mon:" =~ m/^$today:/i or
    "Tue:" =~ m/^$today:/i or
    "Wed:" =~ m/^$today:/i or
    "Thu:" =~ m/^$today:/i or
    "Fri:" =~ m/^$today:/i or
    "Sat:" =~ m/^$today:/i;

    while (<LOGFILE>) {
        if (m//) { # Использовать регулярное выражение по умолчанию
        ⋮
        }
    }
}
```

Чтобы использовать регулярное выражение по умолчанию, необходимо сначала присвоить ему значение при успешном совпадении (именно этим и объясняются все хлопоты с поиском совпадения в нашем примере после задания `$today`). Как видно из приведенного примера, решение получается громоздким и неестественным, поэтому я не рекомендую его использовать.

Предварительное копирование

При выполнении операций поиска и замены Perl иногда тратит дополнительное время и память на копирование целевого текста. Как показано ниже, в некоторых случаях эта копия реально используется, иногда — нет. Если копия целевого текста создается, но не используется, то затраты ресурсов на копирование оказываются лишними и их хотелось бы избежать (особенно если текст имеет очень большую длину или если критична скорость поиска).

В нескольких ближайших разделах будет показано, когда и почему Perl выполняет предварительное копирование целевого текста, когда используется эта копия и как предотвратить копирование в ситуациях, критичных по эффективности.

Поддержка переменных $1, $&, $', $+, ...

Perl создает предварительную копию исходного целевого текста, к которому применяется поиск или подстановка, для поддержки `$1`, `$&` и других служебных переменных, хранящих текст (☞ 377). Perl не создает эти переменные после каждого совпадения, поскольку многие из них (а то и все) могут не использоваться программой. Вместо этого Perl просто сохраняет копию исходного текста, запоминает, *в какой* позиции исходного текста было найдено совпадение, а затем, встретив

обращение к $1 или другой переменной, обращается к нужному тексту. Тем самым уменьшается объем текущей работы, и это хорошо, потому что довольно часто некоторые из служебных переменных вообще не используются.

Отказ от создания переменных $1 и их аналогов до момента фактического применения экономит время, но Perl все равно создает лишнюю копию целевого текста. Но почему это необходимо? Почему Perl не может просто сослаться на исходный текст? Рассмотрим команду:

```
$Subject =~ s/^(?:Re:\s*)+//;
```

После ее выполнения переменная $& будет ссылаться на текст, удаленный из $Subject. Но раз этот текст был *удален*, искать его в $Subject при обращении к $& было бы довольно странно. Аналогичная логика действует во фрагментах вида:

```
if ($Subject =~ m/^SPAM:(.+)/i) {
    $Subject = "-- spam subject removed --";
    $SpamCount{$1}++;
}
```

К моменту ссылки на $1 исходное содержимое $Subject уже потеряно. Следовательно, Perl приходится создавать внутреннюю дополнительную копию целевого текста.

Предварительное копирование не всегда необходимо

На практике предварительное копирование чаще всего используется при обращении к переменным $1, $2, $3 и т. д. А если выражение не содержит сохраняющих круглых скобок? В этом случае беспокоиться о переменной $1 вообще не нужно, поэтому любые действия по ее поддержке становятся лишними. Значит ли это, что для регулярных выражений без сохраняющих скобок можно обойтись без копирования? Не всегда...

«Вредные» переменные $`, $& и $'

Переменные $`, $& и $' не связаны с сохраняющими скобками. Они представляют соответственно текст перед совпадением, текст совпадения и текст после совпадения и поэтому теоретически могут использоваться при *любой* операции поиска и замены. Perl не может заранее определить, к какой операции относится обращение той или иной переменной, и поэтому вынужден выполнять предварительное копирование *каждый раз*.

На первый взгляд кажется, что избежать копирования невозможно, однако Perl понимает, что если эти переменные *вообще* не встречаются в программе (в том числе и в используемых ею библиотеках), копирование для их поддержки становится

лишним. **Таким образом, полное отсутствие обращений к $`, $& и $' в программе позволит избежать предварительного копирования во всех операциях, в которых не задействованы сохраняющие круглые скобки!** Но если хотя бы одна из переменных $`, $& и $' встречается в программе, эта оптимизация полностью теряется. Из-за этого я и назвал эти три переменные «вредными».

Затраты на предварительное копирование

Я провел простой тест, в котором конструкция m/c/ применялась к каждой из 130 000 строк программного кода C в исходных текстах Perl. В процессе хронометража я просто проверял, присутствует ли в строке буква 'c', но полученная информация никак не обрабатывалась, поскольку конечной задачей было определение последствий от копирования. Хронометраж проводился дважды: без предварительного копирования и с ним. Таким образом, по разности между двумя результатами можно было судить о затратах, связанных с копированием.

Проведенная серия тестов показала, что дополнительное копирование стабильно увеличивало время выполнения программы более чем на 40%. Конечно, эти данные следует рассматривать как своего рода «случай средней тяжести». Чем больше реальной работы выполняется в программе, тем меньшие (в процентном отношении) последствия будет иметь копирование. В моих тестах никакой реальной работы не выполнялось, поэтому эффект особенно хорошо заметен.

С другой стороны, в действительно тяжелых случаях большая часть времени работы программы может быть потрачена на дополнительное копирование. Я провел тот же тест для тех же данных, но на этот раз вместо 130 000 строк средней длины данные были организованы в виде *одной большой строки* объемом более 3,5 Мбайт. На этом примере можно было оценить относительное быстродействие одной операции поиска. Без копирования операция выполнялась практически мгновенно, поскольку буква 'c' встречалась где-то неподалеку от начала строки. Как только буква была найдена, поиск завершался. Проверка *с* копированием работала так же, но с одним исключением — сначала создавалась копия строки, объем которой превышал мегабайт. Программа стала работать почти в 7000 раз медленнее! Зная последствия применения некоторых конструкций, вы сможете добиться от своего кода максимальной эффективности.

Предотвращение предварительного копирования

Конечно, было бы замечательно, если бы Perl знал о намерениях программиста и создавал копии лишь в случае необходимости. Но следует учитывать, что копирование — это далеко не всегда плохо. Именно благодаря тому, что Perl берет на выполнение подобные второстепенные задачи, мы выбираем этот язык вместо C или ассемблера. В самом деле, Perl был разработан как раз для того, чтобы поль-

зователь мог избавиться от механических манипуляций с битами и сосредоточить все внимание на создании творческих решений.

Не используйте «вредные» переменные. И все же лишней работы хотелось бы по возможности избежать. Конечно, прежде всего следует *полностью* исключить использование переменных `$``, `$&` и `$'` в вашей программе. Переменная `$&` легко имитируется заключением всего выражения в сохраняющие круглые скобки и использованием ссылки на `$1`. Например, для преобразования некоторых тегов HTML к нижнему регистру вместо выражения `s/<\w+>/\L$&\E/g` следует использовать `s/(<\w+>)/\L$1\E/g`.

Переменные `$``и `$'` легко имитируются при наличии неизмененной копии исходного целевого текста. В следующей таблице перечислены заменители этих переменных после применения выражения к заданному тексту:

Переменная	Имитация
`$``	`substr(`*целевая строка*`, 0, $-[0])`
`$&`	`substr(`*целевая строка*`, $-[0], $+[0] - $-[0])`
`$'`	`substr(`*целевая строка*`, $+[0])`

Поскольку массивы `@-` и `@+` (☞ 379) содержат *позиции* исходной целевой строки, а не *текст*, их использование не отражается на эффективности.

В таблицу также включена замена для `$&`. Иногда этот вариант предпочтительнее варианта с сохраняющими круглыми скобками и `$1`, поскольку он позволяет полностью отказаться от сохраняющих круглых скобок. Вспомните: мы стараемся обойтись без `$&` и других переменных этого семейства именно для того, чтобы избежать копирования целевого текста для выражений, не содержащих сохраняющих круглых скобок. Если после исключения из программы `$&` в каждом выражении появятся сохраняющие круглые скобки, вы абсолютно ничего не добьетесь.

Не используйте «вредные» модули. Естественно, чтобы в программе не использовались переменные `$``, `$&` и `$'`, она также не должна включать модули, в которых встречаются эти переменные. Базовые модули, входящие в поставку Perl, не используют эти переменные. Единственным исключением является модуль `English`; если вам понадобится этот модуль, запретите использование этих трех переменных директивой:

```
use English '-no_match_vars';
```

Далее модуль может свободно использоваться в программе. Если новый модуль загружается из архива CPAN или из другого источника, проверьте его и выясните, используются ли в нем эти переменные. Методика проверки «заражения» программы «вредными» переменными описана во врезке на с. 448.

ПРОВЕРКА «ЗАГРЯЗНЕНИЯ» ПРОГРАММ ИСПОЛЬЗОВАНИЕМ $&

Далеко не всегда можно уверенно определить, встречаются ли в вашей программе ссылки на `$``, `$&` и `$'`, особенно при использовании библиотек. Вероятно, проще всего воспользоваться аргументами командной строки **`-c`** и **`-Mre=debug`** (☞ 451) и найти в выходных данных одну из строк '**`Enabling $` $& $' support`**' или '**`Omitting $` $& $' support`**'. Первое сообщение означает, что программа «загрязнена».

Существует небольшая вероятность того, что программа «загрязнена» вследствие использования переменных в конструкции `eval`, о чем Perl неизвестно до момента выполнения. Одно из возможных решений основано на установке пакета `Devel::SawAmpersand` из архива CPAN (*http://www.cpan.org*):

```
END {
    require Devel::SawAmpersand;
    if (Devel::SawAmpersand::sawampersand) {
        print "Naughty variable was used!\n";
    }
}
```

Пакет `Devel::SawAmpersand` дополняется пакетом `Devel::FindAmpersand`, позволяющим узнать, где находится «вредная» переменная. К сожалению, в последних версиях Perl этот пакет работает ненадежно. Кроме того, установка обоих пакетов сопряжена с определенными трудностями, поэтому все зависит от конкретной ситуации (за возможными обновлениями обращайтесь по адресу *http://regex.info/*).

Также интересно посмотреть, как реализовать проверку «на вредность» за счет измерения быстродействия:

```
use Time::HiRes;
sub CheckNaughtiness()
{
  my $text = 'x' x 10_000; # Сгенерировать большой объем данных.

  # Измерить время выполнения пустого цикла.
  my $start = Time::HiRes::time();
  for (my $i = 0; $i < 5_000; $i++) { }
  my $overhead = Time::HiRes::time() - $start;

  # Измерить время выполнения того же количества поисков.
  $start = Time::HiRes::time();
  for (my $i = 0; $i < 5_000; $i++) { $text =~ m/^/ }
  my $delta = Time::HiRes::time()  $start;

  # Если $delta превышает $overhead в 5 раз или более, значит, программа
  # загрязнена (оценка получена эвристическим путем).
  printf "It seems your code is %s (overhead=%.2f, delta=%.2f)\n",
    ($delta > $overhead+5) ? "naughty" : "clean", $overhead, $delta;
}
```

Функция study

Функция `study(…)` оптимизирует не регулярное выражение, а доступ кинформации о *строке*. В случае применения регулярного выражения (или нескольких регулярных выражений) можно достигнуть существенного выигрыша за счет наличия кэшированных сведений о строке. Общий синтаксис вызова `study` выглядит так:

```
while (<>)
{
    study($_); # Обработка целевого текста по умолчанию $_
               # перед выполнением большого количества операций поиска
  if (m/выражение 1/) { … }
  if (m/выражение 2/) { … }
  if (m/выражение 3/) { … }
  if (m/выражение 4/) { … }
```

Понять принцип работы `study` несложно; значительно сложнее разобраться в том, обеспечивает она в данном конкретном случае какой-нибудь выигрыш или нет. Функция абсолютно не влияет на значения, обрабатываемые или возвращаемые программой. В результате ее применения Perl расходует больше памяти, а общее время работы программы может увеличиться, остаться прежним или уменьшиться (для чего, собственно, и предназначена эта функция).

При обработке строки функцией `study` Perl расходует некоторое количество времени и памяти на построение списка позиций, в которых каждый символ встречается в строке. В большинстве систем затраты памяти в 4 раза превышают размер строки. Выигрыш от вызова `study` увеличивается при каждом последующем поиске регулярного выражения в строке, но лишь до момента модификации строки. При любой модификации строки построенный список становится недействительным, как и при вызове `study` для другой строки.

Польза от вызова study для целевого текста сильно зависит от специфики регулярного выражения и от тех оптимизаций, которые Perl может применить. Например, вызов `study` заметно ускоряет поиск литерального текста (`m/foo/`); при больших размерах возможен выигрыш в 10 000 раз! Однако с модификатором `/i` это преимущество исчезает, поскольку `/i` практически ликвидирует преимущества `study` (а также ряда других оптимизаций).

Когда не следует использовать study

- Не используйте `study` для строк, поиск в которых будет осуществляться с модификатором `/i`, а также если весь литеральный текст в строке находится под управлением конструкций ⌜`(?i)`⌟ или ⌜`(?i:…)`⌟, компенсирующих преимущества от вызова `study`.

- Не используйте `study` для коротких целевых строк, в таких случаях вполне достаточно обычной оптимизации с проверкой фиксированных строк (☞ 314). Какие строки считать «короткими»? Сложно сказать. Длина строки — всего лишь один фактор из большого комплекса, плохо поддающегося анализу, поэтому только хронометраж *ваших* выражений для *ваших* данных покажет, дает ли выигрыш от `study`. Впрочем, я с трудом представляю себе смысл вызова `study` для строк объемом менее нескольких килобайтов.
- Не используйте `study` при поиске небольшого количества совпадений в целевой строке (по крайней мере, перед модификацией строки или вызовом `study` для другой строки). Общее ускорение более вероятно, если время, затраченное на анализ строки функцией `study`, распределяется по многим попыткам поиска. При малом количестве операций время, затраченное на построение списка `study`, может перевесить всю экономию.
- Используйте `study` только для поиска в строках, в которых регулярное выражение содержит «выделенный» литеральный текст (☞ 323). Без знания символов, которые должны присутствовать в любом совпадении, вызов `study` бесполезен. Рассуждая по аналогии, можно решить, что вызов `study` ускорит выполнение функции `index`, но это не так.

Когда study может помочь

Наибольший эффект при вызове `study` достигается в ситуации, когда у вас имеется большая строка, в которой перед модификацией будет производиться многократный поиск. Хорошим примером является фильтр, написанный мной при подготовке этой книги. При написании книги я использую свой собственный стиль разметки, который преобразуется фильтром в SGML (который затем преобразуется в формат *troff*, который, в свою очередь, преобразуется в PostScript). Во время работы фильтра вся глава в конечном счете превращается в одну громадную строку (в этой главе ее объем превышал 475 Кбайт). Перед завершением я выполняю ряд проверок, предназначенных для поиска возможных ошибок в разметке. Проверки не модифицируют строку и в них часто встречаются фиксированные строки, — это как раз та ситуация, для которой создавалась функция `study`.

Хронометраж

Лучший способ оценки эффективности вашей программы — хронометраж. В Perl имеется модуль `Benchmark`, снабженный хорошей документацией (`«perldoc Benchmark»`). Скорее просто по привычке, чем из каких-то других соображений, я обычно пишу собственные хронометражные тесты. После выполнения директивы

```
use Time::HiRes 'time';
```

проверяемый код просто заключается в конструкцию вида:

```
my $start = time;
    ⋮
my $delta = time  $start;
printf "took %.1f seconds\n", $delta;
```

При планировании хронометража необходимо проследить за тем, чтобы объем данных был достаточен для получения осмысленных результатов, в тесты включалось как можно больше «содержательных» операций, а «побочные» операции занимали как можно меньше времени. Эта тема подробно рассматривается в главе 6 (☞ 295). Возможно, умение грамотно организовать хронометраж приходит с опытом, но его результаты оказываются весьма поучительными и полезными.

Отладочная информация регулярных выражений

Стремясь как можно быстрее найти совпадение для регулярного выражения, Perl выполняет феноменальное количество оптимизаций. Наименее таинственные разновидности оптимизаций перечислены в разделе «Стандартные оптимизации» главы 6 (☞ 305), но это далеко не все. Многие оптимизации применяются только в очень специфических ситуациях, поэтому для любого регулярного выражения используется только часть этих оптимизаций (а иногда оптимизации вообще не применяются).

В Perl существует отладочный режим, в котором можно получить информацию о некоторых оптимизациях. При первой компиляции регулярного выражения Perl выбирает оптимизации, а отладочный режим выводит информацию о некоторых из них. В отладочном режиме также можно узнать много интересного о том, как механизм применяет выражение. Подробный анализ отладочных данных выходит за рамки книги, но я приведу краткую сводку.

Отладочный режим включается директивой **`use re 'debug';`** и отключается директивой **`no re'debug';`**. Автоматическое отключение отладочного режима происходит в конце блока или файла, в котором этот режим был включен (директива `use re` уже встречалась раньше с другими аргументами для разрешения интерполяции встроенного кода ☞ 422).

Если вы хотите включить отладочный режим для всего сценария, воспользуйтесь аргументом командной строки `-Mre=debug`. Это особенно часто делается в процессе анализа компиляции регулярного выражения. Пример приводится ниже (некоторые строки, не представляющие интереса, удалены):

```
❶ % perl -cw -Mre=debug -e 'm/^Subject: (.*)/'
❷ Compiling REx `^Subject: (.*)'
❸ rarest char j at 3
```

```
❹       1: BOL(2)
❺       2: EXACT <Subject: >(6)
                ⋮
❻      12: END(0)
❼ anchored `Subject: ' at 0 (checking anchored) anchored(BOL) minlen 9
❽ Omitting $` $& $' support.
```

В точке ❶ *perl* запускается в командной строке с ключами **-c** (проверка сценария без фактического выполнения), **-w** (выдача предупреждений о конструкциях, сомнительных с точки зрения Perl; рекомендуется использовать всегда) и **-Mre=debug** (включение отладочного режима). Ключ **-e** означает, что следующий аргумент, '`m/^Subject:•(.*)/`', представляет собой мини-сценарий Perl, который должен быть выполнен или проверен.

В строке ❸ указывается «самый редкий» (по крайней мере, с точки зрения Perl) символ из самой длинной фиксированной подстроки регулярного выражения. Perl использует эту информацию для некоторых видов оптимизации (например, предварительной проверки обязательных символов/подстрок ☞ 311).

В строках ❹–❻ приведена откомпилированная форма регулярного выражения, непосредственно используемая Perl. Как правило, эти сведения интереса не представляют. Впрочем, смысл строки ❺ более или менее понятен.

Большая часть полезной информации выводится в строке ❼. Ниже перечислены некоторые сведения, которые могут здесь появиться:

`anchored` *'строка'* `at` *смещение*

Означает, что любое совпадение должно содержать заданную *строку* с указанным *смещением* от начала поиска. Если сразу же после *'строки'* следует символ '`$`', *строка* завершает совпадение.

`floating` *'строка'* `at` *начало..конец*

Означает, что любое совпадение должно содержать заданную *строку*, начало которой принадлежит заданному интервалу. Если сразу же после *'строки'* следует символ '`$`', *строка* завершает совпадение.

`strclass` *'список'*

Список символов, с которых может начинаться совпадение.

`anchored(MBOL)`, `anchored(BOL)`, `anchored(SBOL)`

Регулярное выражение начинается с `^`. `MBOL` выводится при использовании модификатора /m, а при его отсутствии — `BOL` и `SBOL` (различия между `BOL`

и SBOL несущественны для современного Perl; SBOL относится к переменной $*, которая давно считается устаревшей).

`anchored(GPOS)`

Регулярное выражение начинается с ⌜\G⌟.

`Implicit`

Perl автоматически добавляет anchored(MBOL), поскольку регулярное выражение начинается с ⌜.*⌟.

`minlen` *длина*

Любое совпадение должно иметь длину не меньше указанной.

`with eval`

Регулярное выражение содержит конструкцию ⌜(?{…})⌟ или ⌜(??{…})⌟.

Строка ❽ не связана с конкретным регулярным выражением. Она появляется только в том случае, если сам интерпретатор Perl был скомпилирован с флагом -DDEBUGGING. В этом случае после загрузки всей программы Perl сообщает о том, включена ли поддержка $& и других переменных того же семейства (☞ 444).

Отладочная информация на стадии выполнения

Выше уже приводился пример использования встроенного кода для получения информации о ходе поиска (☞ 416), однако средства отладки Perl способны на большее. Если при вызове не указывался ключ c, Perl выводит достаточно подробную информацию о каждой попытке.

Если будет выведена строка «Match rejected by optimizer», это означает, что вследствие некоторой оптимизации механизм регулярных выражений понял, что регулярное выражение никогда не совпадет с целевым текстом, поэтому поиск вообще не производится. Пример:

```
% perl -w -Mre=debug -e '"this is a test" =~ m/^Subject:/;'
    ⋮
Did not find anchored substr “Subject:'…
Match rejected by optimizer
```

При включении отладочного режима выводится информация обо всех используемых регулярных выражениях, не только о ваших. Пример:

```
% perl -w -Mre=debug -e 'use warnings'
   ...подробная отладочная информация...
          ⋮
```

Команда всего лишь загружает модуль `warnings`, но из-за большого количества регулярных выражений в этом модуле выводится большой объем отладочной информации.

Другие возможности вывода отладочной информации

Я уже упоминал о возможности вывода отладочной информации директивой «`use re 'debug';`» или ключом `Mre=debug`. Если заменить `de bug` на `debugcolor`, а терминал поддерживает управляющие последовательности ANSI, информация будет выводиться с цветовым выделением, упрощающим чтение данных.

Если сам интерпретатор файл Perl был откомпилирован с расширенной отладочной поддержкой, вместо ключа `Mre=debug` может использоваться ключ командной строки `-Dr`.

Последний комментарий

Наверное, вы уже поняли, что я в полном восторге от регулярных выражений Perl, и, как было сказано в самом начале главы, на то есть веские причины. Несомненно, Ларри Уолл, создатель Perl, руководствовался здравым смыслом и вдохновением. Пусть у его творения есть свои недостатки, и все же я не перестаю наслаждаться изысканным богатством возможностей диалекта регулярных выражений Perl.

Впрочем, не считайте меня бездумным фанатиком — Perl не обладает некоторыми возможностями, которые я бы хотел в нем видеть. Многие элементы, о которых я упоминал в первом издании книги, были включены в язык, поэтому я продолжу список пожеланий. Вероятно, самым значительным упущением является отсутствие именованного сохранения (☞ 186), поддерживаемого другими программами. В этой главе описана методика имитации, но она имеет серьезные ограничения; было бы лучше иметь встроенную поддержку именованного сохранения. Также в Perl хотелось бы видеть операции множеств с символьными классами (☞ 170), хотя ценой определенных усилий они имитируются на базе опережающей проверки (☞ 172).

Далее идут захватывающие квантификаторы (☞ 190). В Perl поддерживается атомарная группировка, которая в общем случае обладает несколько большими возможностями, но, несмотря на это, захватывающие квантификаторы в некоторых ситуациях обеспечивают более наглядное и элегантное решение, поэтому я бы предпочел видеть оба варианта. Кроме того, мне хотелось бы видеть две конструкции, которые в настоящее время не поддерживаются ни одним диалектом. Первая — простой оператор «отсечения» (скажем, `⌜\v⌟`), который немедленно уничтожает все сохраненные состояния, существующие на данный момент (в этом случае запись

⌜x+\v⌟ эквивалентна ⌜x++⌟ и атомарной группировке ⌜(?>x+)⌟). Другая конструкция запретила бы любые дальнейшие смещения при поиске. Она должна означать: «либо совпадение находится по текущему пути, либо совпадение вообще невозможно». Например, ее можно было бы обозначить ⌜\v⌟.

Другое пожелание имеет некоторое отношение к ⌜\v⌟. На мой взгляд, было бы полезно реализовать общие средства, позволяющие управлять смещением текущей позиции. Такое новшество упростило бы решение задачи на с. 420.

Наконец, как упоминалось на с. 422, было бы неплохо предусмотреть дополнительный контроль над интерполяцией встроенного кода в регулярные выражения.

Я не считаю Perl идеальным языком для работы с регулярными выражениями, но он очень близок к идеалу и постоянно совершенствуется.

8 Java

Встроенная поддержка регулярных выражений в Java в виде пакета `java.util.regex` появилась лишь в версии 1.4, вышедшей в начале 2002 года. Пакет реализует мощный и современный прикладной интерфейс для работы с регулярными выражениями, обладает отличной поддержкой Юникода, сопровождается качественной документацией и имеет высокую эффективность, а также может обеспечивать высочайшую гибкость благодаря возможности использовать объекты, реализующие интерфейс `CharSequence`.

Уже первоначальная версия пакета `java.util.regex` имела весьма впечатляющие характеристики. Набор возможностей, скорость выполнения и относительно небольшое число ошибок вызывали удивление, особенно если учесть, что это была первая версия. Последней версией Java 1.4 была Java 1.4.2, а к моменту написания этих строк свет увидела уже версия Java 1.5.0 (также называемая Java 5.0), а Java 1.6.0 (Java 6.0, или «Mustang») находится в стадии второй бета-версии. Формально в этой книге рассматривается Java 1.5.0, но я буду останавливаться и на особо значимых отличиях Java 1.4.2 и 1.6 там, где это будет необходимо. (Кроме того, краткий перечень отличий приводится в конце этой главы ☞ 502)[1].

Прежде чем рассказывать, о чем эта глава, стоит упомянуть, что в ней не пересказывается материал глав 1–6. Читатели, интересующиеся только языком Java, могут начать чтение именно с этой главы, хотя я рекомендую им ознакомиться с предисловием и предыдущими главами. В главах 1, 2 и 3 представлены основные концепции, возможности и приемы, используемые при работе с регулярными выражениями, тогда как главы 4, 5 и 6 содержат информацию, очень важную для понимания регулярных выражений и непосредственно относящуюся к пакету регу-

[1] В этой книге рассматривается Java 1.5.0 в подверсии Update 7. Первоначальный выпуск Java 1.5 содержал некоторые ошибки, связанные с поиском без учета регистра символов. Эти ошибки были исправлены в подверсии Update 2, поэтому я рекомендую вам обновить версию Java 1.5, если вы пользуетесь ею. К моменту написания этих строк Java 1.6 находилась в состоянии второй бета-версии, сборка 59g.

лярных выражений `java.util.regex`. Из наиболее важных тем, рассматривавшихся в предыдущих главах, можно упомянуть основы работы механизма регулярных выражений НКА, связанные с поиском совпадений, максимализмом, возвратами и вопросами эффективности.

Таблица 8.1. Список методов в алфавитном порядке с указанием номеров страниц

Таблица 8.1 помещена здесь для простоты поиска. Пояснение API регулярных выражений начнется со с. 463.

Здесь мне хочется заметить, что, несмотря на наличие удобных таблиц, таких как на с. 459 в этой главе или на с. 156 и 166 в главе 3, данная книга не претендует на роль справочного руководства. Главная ее цель — научить вас *искусству* составления регулярных выражений.

Примеры с использованием пакета `java.util.regex` уже приводились в предыдущих главах (☞ 116, 132, 136, 280, 299), но еще большее число примеров будет приведено в этой главе. Позднее будет рассмотрена объектная модель пакета и особенности ее фактического применения, но сначала мы познакомимся с диалектом регулярных выражений, поддерживаемым `java.util.regex`, и модификаторами, характерными для этого диалекта.

Диалект регулярных выражений

Пакет `java.util.regex` использует традиционный механизм НКА, поэтому к нему в полной мере применимы сведения, изложенные в главах 4, 5 и 6. В табл. 8.2 приведена сводка метасимволов. Некоторые особенности диалекта зависят от режимов

поиска, активизируемых при помощи флагов функций или модификаторов ⌜(?*мод-мод*)⌟ и ⌜(?*мод-мод*:...)⌟. Режимы перечислены в табл. 8.3, на с. 460.

Естественно, диалект регулярных выражений невозможно полностью описать в одной маленькой таблице, поэтому ниже приводятся некоторые комментарии по поводу табл. 8.2.

① Символ `\b`, является сокращенным обозначением символа «забой» (backspace) только внутри символьных классов. За пределами символьных классов он представляет границу слова (☞ 180).

В таблице приводятся «низкоуровневые» символы \ вместо последовательностей `\\`, которые должны использоваться при передаче регулярных выражений в строковых литералах Java. Например, последовательность ⌜`\n`⌟ из таблицы должна записываться в виде «`\\n`» в строке Java. См. раздел «Строки как регулярные выражения» (☞ 140).

Конструкция **`\x##`** может содержать ровно две шестнадцатеричные цифры (например, выражение ⌜**`\xFC`**`ber`⌟ совпадает с '**`über`**').

Конструкция **`\u####`** может содержать ровно четыре шестнадцатеричные цифры (например, выражение ⌜**`\u00FC`**`ber`⌟ совпадает с '**`über`**', а ⌜`\u20AC`⌟ совпадает с '€').

Последовательность **`\0`***восьм* обязательно начинается с нуля и содержит от одной до трех восьмеричных цифр.

Конструкция **`\c`***символ* является *чувствительной к регистру символов* — код указанного символа слепо объединяется с числом 0x40 операцией «исключающего ИЛИ». Это странное поведение означает, что в отличие от всех остальных диалектов, встречавшихся мне, конструкции `\ca` и `\cA` считаются различными. Чтобы обеспечить традиционную интерпретацию `\x01`, следует использовать символы верхнего регистра. Последовательность `\ca` эквивалентна `\x21` и совпадает с '`!`'.

② Метасимволы **`\w`**, **`\d`** и **`\s`** (и их антиподы в верхнем регистре) совпадают только с символами ASCII и не включают другие алфавитно-цифровые символы или пропуски Юникода. Таким образом, запись `\d` в точности эквивалентна `[0-9]`, запись `\w` эквивалентна `[0-9a-zA-Z_]`, а `\s` — `[ \t\n\f\r\x0B]` (`\x0B` — практически не используемый ASCII символ вертикальной табуляции).

Полноценная поддержка Юникода обеспечивается при помощи свойств Юникода (☞ 164): вместо `\w` используется `\p{L}`, вместо `\d` — `\p{Nd}`, а вместо `\s` — `\p{Z}` (вместо `\W`, `\D` и `\S` следует использовать версии `\P{…}` этих конструкций).

③ Конструкции **`\p{…}`** и **`\P{…}`** поддерживают большинство стандартных свойств и блоков Юникода, а также некоторые «свойства Java». Алфавиты не поддерживаются. Подробнее об этом будет сказано ниже.

Таблица 8.2. Общие сведения о диалекте регулярных выражений пакета java.util.regex

Сокращенные обозначения символов ①	
☞ 156 (C)	`\a [\b] \e \f \n \r \t \0`*восьм* `\x## \u#### \c`*символ*
Символьные классы и аналогичные конструкции	
☞ 161 (C)	Обычные классы: `[…]` и `[^…]` (поддерживаются операции множеств ☞ 171)
☞ 162	Почти любой символ: *точка* (значение изменяется в зависимости от режима поиска)
☞ 164 (C)	Сокращенные обозначения классов:② `\w \d \s \W \D \S`
☞ 164 (C)	Свойства и блоки Юникода:③ `\p{`*свойство*`} \P{`*свойство*`}`
Якорные метасимволы и другие проверки с нулевой длиной совпадения	
☞ 462	Начало строки/логической строки: `^ \A`
☞ 462	Конец строки/логической строки: `$ \z \Z`
☞ 177	Начало текущего совпадения: `\G`
☞ 180	Границы слов:④ `\b \B`
☞ 181	Позиционная проверка:⑤ `(?=…) (?!…) (?<=…) (?<!…)`
Комментарии и модификаторы режимов	
☞ 182	Модификаторы режимов: `(?`*мод*`-`*мод*`)`. Допустимые модификаторы: `x d s m i u`
☞ 183	Интервальное изменение режима: `(?`*мод*`-`*мод*`:…)`
☞ 183	Комментарии: от символа # до новой строки (если разрешено)⑥
☞ 183	Литеральный текст:⑦ `\Q…\E`
Группировка и сохранение	
☞ 184	Сохраняющие круглые скобки: `(…) \1 \2 ...`
☞ 185	Группирующие круглые скобки: `(?:…)`
☞ 187	Атомарная группировка: `(?>…)`
☞ 187	Конструкция выбора: `|`
☞ 189	Максимальные квантификаторы: `* + ? {n} {n,} {x, y}`
☞ 190	Минимальные квантификаторы: `*? +? ?? {n}? {n,}? {x, y}?`
☞ 190	Захватывающие квантификаторы: `*+ ++ ?+ {n}+ {n,}+ {x, y}+`
(C) — может использоваться в символьных классах.	①…⑦ — см. текст

④ Метасимволы `\b` и `\B` представляют себе «символ слова» совсем не так, как `\w` и `\W`. Они распознают свойства символов Юникода, тогда как `\w` и `\W` совпадают исключительно с ASCII-символами.

⑤ *Опережающая* проверка может производиться по любым регулярным выражениям, тогда как *ретроспективная* проверка ограничивается подвыражениями, допустимые совпадения которых имеют конечную длину. В частности, это означает, что квантификатор ⌜?⌟ может использоваться при ретроспективной проверке, а ⌜*⌟ и ⌜+⌟ — не могут. Дополнительная информация приведена в главе 3 на с. 181.

⑥ Последовательности #…[NL] воспринимаются как комментарии только при использовании параметра Pattern.COMMENTS (☞ 460) или с модификатором **x**. (Не забывайте включать символы новой строки в многострочные литералы, как показано в примере на с. 501.) Неэкранированные пропуски ANSI игнорируются. **Примечание**: в отличие от большинства реализаций, поддерживающих этот режим, комментарии и пропуски *распознаются* в символьных классах.

⑦ Конструкция \Q…\E поддерживается, но ее использование *внутри* символьных классов вызывало ошибку в версиях Java до версии 1.6.

Таблица 8.3. Режимы поиска и форматирования в пакете java.regex.util

Ключ компиляции	(?режим)	Описание
Pattern.UNIX_LINES	d	Изменяет режим поиска совпадений для *точки* и ⌜^⌟ (☞ 462)
Pattern.DOTALL	s	Разрешает совпадение *точки* с любым символом (☞ 152)
Pattern.MULTILINE	m	Расширяет возможности совпадения для ⌜^⌟ и ⌜$⌟ (☞ 462)
Pattern.COMMENTS	x	Режим свободного форматирования (☞ 106). (Действует даже внутри символьных классов)
Pattern.CASE_INSENSITIVE	i	Поиск без учета регистра для ASCII-символов
Pattern.UNICODE_CASE	u	Поиск без учета регистра для символов, не входящих в ASCII
Pattern.CANON_EQ		Режим «канонической эквивалентности» символов Юникода (разные кодировки одного символа считаются идентичными ☞ 149)
Pattern.LITERAL		Аргументы регулярного выражения интерпретируются как обычный текст, а не как регулярное выражение

Поддержка конструкций \p{...} и \P{...} в Java

Конструкции ⌜\p{…}⌟ и ⌜\P{…}⌟ поддерживают стандартные свойства и блоки Юникода, а также некоторые специальные «свойства Java» символов. Пакет поддерживает Юникод версии 4.0.0. (Java 1.4.2 поддерживает Юникод версии 3.0.0.)

Свойства Юникода

Используются только короткие имена свойств, такие как `\p{Lu}`. (Список свойств приводится на с. 165.) Длинные конструкции вида `\p{Lowercase_Letter}` не поддерживаются. В однобуквенных именах свойств допускается отсутствие фигурных скобок — запись `\pL` эквивалентна `\p{L}`.

В Java 1.5 и более ранних версиях свойства **`Pf`** и **`Pi`** не поддерживаются и потому символы, соответствующие этим свойствам, не совпадают с конструкцией `\p{P}`. (Эти свойства поддерживаются в Java 1.6.)

Комбинация `\p{C}` не совпадает с символами, которые совпадают с `\p{Cn}`.

Композитное свойство `\p{L&}` не поддерживается.

Имеется поддержка псевдосвойства `\p{all}`, эквивалентного `(?s:.)`. Псевдосвойства `\p{assigned}` и `\p{unassigned}` не поддерживаются: вместо них следует использовать `\P{Cn}` и `\p{Cn}`.

Блоки Юникода

Пакет поддерживает блоки Юникода с обязательным префиксом '`In`'. Сведения об именах поддерживаемых блоков, которые могут появляться в конструкциях `\p{…}` и `\P{…}` в зависимости от версии, приводятся на с. 503.

Для сохранения обратной совместимости в Java 1.5 два блока Юникода, имена которых различаются в Юникоде версии 3.0.0 и 4.0.0, доступны как под старыми, так и под новыми именами: блоки из Юникода 4.0.0 с именами **Combining Diacritical Marks for Symbols и Greek and Coptic** ныне доступны и под прежними именами **Combining Marks for Symbols и Greek**.

В Java 1.5 была исправлена ошибка, присутствовавшая в Java 1.4.2 и связанная с именами блоков **Arabic Presentation Forms-B и Latin Extended-B** (☞ 503).

Специальные свойства символов Java

Начиная с Java версии 1.5.0, конструкции `\p{…}` и `\P{…}` включают поддержку современных методов `is`*`Something`* из модуля `java.lang.Character`. Для доступа к требуемому методу из регулярного выражения внутри конструкций `\p{…}` и `\P{…}` следует изменить префикс '`is`' в имени метода на '`java`'. Например, символы, соответствующие `java.lang.Character.isJavaIdentifierStart`, будут совпадать с регулярным выражением `\p{javaJavaIdentifierStart}`. (Полный перечень доступных методов приводится в документации к классу `java.lang.Character`.)

Завершители строк Юникода

До появлений Юникода диалекты регулярных выражений традиционно обеспечивали специальную интерпретацию символа новой строки (ASCII-символ LF) по отношению к метасимволам `.`, `^`, `$` и `\Z`. В Java большинство *завершителей строки* Юникода (☞ 150) также интерпретируется особым образом.

В Java, как правило, следующие символы рассматриваются как завершители строк.

Коды символов	Обозначения		Описание
U+000A	LF	`\n`	Перевод строки (ASCII)
U+000D	CR	`\r`	Возврат каретки (ASCII)
U+000D U+000A	CR/LF	`\r\n`	Возврат каретки/перевод строки (ASCII)
U+0085	NEL		Следующая строка (Юникод)
U+2028	LS		Разделитель строк (Юникод)
U+2029	PS		Разделитель абзацев (Юникод)

Интерпретация символов и ситуаций выполняется особым образом для `.`, `^`, `$` и `\Z` и изменяется в зависимости от выбранного режима поиска (☞ 460).

Режим поиска	Символы, на которые воздействует	Описание
`UNIX_LINES`	`^.$ \Z`	Производится возврат к традиционной интерпретации завершителя «перевод строки»
`MULTILINE`	`^ $`	Добавляет в список позиций внутренние завершители строк, перед которыми могут быть найдены совпадения с `^` и после которых могут быть найдены совпадения с `$`
`DOTALL`	`.`	Для *точки* отменяется совпадение с завершителями строк — *точка* соответствует любому символу

Двухсимвольный завершитель `CR/LF` заслуживает специального упоминания. По умолчанию (т. е. когда не используется режим `UNIX_LINES`) последовательность `CR/LF` воспринимается метасимволами границы строки как единый элемент, и они не будут обнаруживать соответствие *между* этими двумя символами.

Например, обычно `$` и `\Z` обнаруживают соответствие непосредственно перед завершителем строки. ASCII-символ `LF` является завершителем строки, но он распознается метасимволами `$` и `\Z` только в том случае, если он не является ча-

стью последовательности символов CR/LF (т. е. когда символу LF *не* предшествует символ CR).

Для метасимволов $ и ^ в режиме MULTILINE также выполняется расширенная интерпретация: ^ может совпадать с символом CR только в том случае, когда вслед за символом CR не следует символ LF, а $ может обнаруживать соответствие перед внутренним символом LF, только если ему не предшествует символ CR.

Режим DOTALL вообще не влияет на порядок интерпретации последовательности CR/LF (режим DOTALL влияет только на состав допустимых совпадений метасимвола ⌜.⌟, который всегда рассматривает символы по отдельности), а режим UNIX_LINES ликвидирует проблему в целом (в этом режиме LF и другие завершители строк, не являющиеся символом новой строки, интерпретируются обычным образом).

Использование пакета java.util.regex

Механика применения регулярных выражений в пакете java.util.regex весьма проста. Все основные функциональные возможности обеспечиваются всего двумя классами, одним интерфейсом и одним исключением:

```
java.util.regex.Pattern
java.util.regex.Matcher
java.util.regex.MatchResult
java.util.regex.PatternSyntaxException
```

Первые два класса я буду называть просто «Pattern» и «Matcher». В большинстве случаев используются только эти два класса. Объект Pattern представляет откомпилированное регулярное выражение, которое может быть применено к любому количеству строк, а объект Matcher — конкретный экземпляр этого регулярного выражения, примененный к заданному целевому тексту.

Появившийся в Java 1.5 интерфейс MatchResult предоставляет возможность доступа к результатам успешного поиска. Результаты сохраняются непосредственно в объекте Matcher, пока не будет предпринята очередная попытка поиска, однако их можно сохранить в отдельном объекте типа MatchResult.

Исключение PatternSyntaxException инициируется при попытке компиляции регулярного выражения с неправильным синтаксисом (например, такого как ⌜[oops)⌟). Он наследует класс java.lang.IllegalArgumentException.

Ниже приведен простой пример поиска:

```
public class SimpleRegexTest {
    public static void main(String[] args)
    {
```

```
        String myText = "this is my 1st test string";
        String myRegex = "\\d+\\w+"; // регулярное выражение 「\d+\w+」
        java.util.regex.Pattern p = java.util.regex.Pattern.compile(myRegex);
        java.util.regex.Matcher m = p.matcher(myText);

        if (m.find()) {
            String matchedText = m.group();
            int matchedFrom = m.start();
            int matchedTo = m.end();
            System.out.println("matched [" + matchedText + "] " +
                               "from " + matchedFrom +
                               " to" + matchedTo + ".");
         } else {
            System.out.println("didn't match");
         }
    }
}
```

Программа выводит строку '**matched [1st] from 12 to 15.**'. Как и во всех примерах настоящей главы, имена переменных выделены курсивом. Части, выделенные жирным шрифтом, могут отсутствовать, если в начало программы (как в примерах главы 3 ☞ 132) включена директива

```
import java.util.regex.*;
```

что соответствует принятым стандартам и, кроме того, облегчает сопровождение программного кода. В остальных примерах этой главы предполагается наличие директивы `import`.

Объектная модель пакета `java.util.regex` несколько отличается от объектной модели, которой следуют большинство других пакетов. В предыдущем примере объект `m`, принадлежащий классу `Matcher`, созданный в результате связывания объекта `Pattern` с целевой строкой, используется как для проведения поиска (методом `find`), так и для получения результатов (методами `group`, `start` и `end`).

На первый взгляд такой подход выглядит немного странным, но вы быстро привыкнете к нему.

Метод Pattern.compile()

Объект регулярного выражения `Pattern` создается методом `Pattern.compile`. Первый аргумент метода — строка, интерпретируемая как регулярное выражение (☞ 140). Во втором аргументе могут передаваться необязательные ключи, перечисленные в табл. 8.3 на с. 460. Пример создания объекта `Pattern` на базе строки, хранящейся в переменной *myRegex*, для проведения поиска без учета регистра символов:

```
Pattern pat = Pattern.compile(myRegex,
                    Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
```

Предопределенные константы, задающие параметры компиляции (например, `Pattern.CASE_INSENSITIVE`), имеют достаточно громоздкие имена[1], поэтому я предпочитаю пользоваться встроенными модификаторами (☞ 150). Примеры на с. 472 содержат модификатор `(?x)` и модификаторы `(?s)` и `(?i)` на с. 498.

Однако имейте в виду, что громоздкие имена констант существенно упрощают понимание листингов программ для начинающих программистов. Если бы страницы книги имели неограниченную ширину, в листинге на с. 479 во втором аргументе метода `Pattern.compile` я предпочел бы использовать константы

```
Pattern.UNIX_LINES | Pattern.CASE_INSENSITIVE
```

вместо малопонятного `(?id)` в начале регулярного выражения.

Как следует из названия метода, он анализирует регулярное выражение и компилирует его во внутреннее представление. Вопросы компиляции рассматриваются во всех подробностях в главе 6 (☞ 307), однако в двух словах напомню, что этап компиляции выражения может оказаться самым длительным по времени в процессе поиска. По этой причине компиляция выделена в отдельный этап, который выполняется в первую очередь; это позволяет скомпилировать регулярное выражение один раз, а затем использовать его многократно.

Безусловно, когда скомпилированное регулярное выражение применяется всего один раз, совершенно неважно, когда оно будет скомпилировано, но при многократном его использовании (например, при выполнении поиска в каждой строке из файла) есть смысл выполнить предварительную компиляцию выражения в объект `Pattern`.

При вызове `Pattern.compile` могут инициироваться исключения двух типов: при недопустимом синтаксисе регулярного выражения инициируется исключение `PatternSyntaxException`, а при ошибке в аргументе — исключение `IllegalArgumentException`.

Метод Pattern.matcher()

Объект `Pattern` поддерживает ряд вспомогательных методов, перечисленных ниже (☞ 492), но вся основная работа выполняется всего одним методом `matcher`. При вызове методу передается единственный аргумент — строка, в которой осущест-

[1] Особенно для представления листингов в книге, где страницы имеют ограниченную ширину!

вляется поиск[1]. В действительности метод не применяет регулярное выражение, но готовит обобщенный объект `Pattern` для его применения к конкретной строке. Метод `matcher` возвращает объект класса `Matcher`.

Объект Matcher

После создания объекта `Matcher`, связывающего регулярное выражение с целевым текстом, вы можете применять регулярное выражение в различных режимах и запрашивать информацию о результатах. Например, если объект *m* принадлежит к классу `Matcher`, вызов *m*`.find()` применяет регулярное выражение к тексту и возвращает логический признак наличия совпадения. Если совпадение найдено, вызов *m*`.group()` возвращает строку, представляющую совпавший текст.

Прежде чем перейти к рассмотрению различных методов объекта `Matcher`, будет далеко не лишним коротко остановиться на том, какую информацию он содержит. Нижеприведенные списки содержат ссылки на страницы с подробным описанием каждого пункта и могут служить справочником. В первом списке перечислены элементы, которые доступны программисту для записи, во втором — доступные только для чтения.

Элементы, доступные программисту для записи:

- После создания объекта `Matcher` программист получает в свое распоряжение объект `Pattern`. Он может быть изменен с помощью метода `usePattern()` (☞ 491). Доступ к текущему объекту `Pattern` осуществляется с помощью метода `pattern()`.
- После создания объекта `Matcher` программист получает в свое распоряжение объект `целевая текстовая строка` (или другой объект, реализующий интерфейс `CharSequence`). Эта строка доступна для изменения с помощью метода `reset(`*текст*`)`(☞ 490).
- *Область* целевого текста (☞ 482). По умолчанию область включает в себя весь целевой текст, но она может быть изменена программистом путем выделения части целевой строки с помощью метода `region`. Некоторые методы поиска (хотя и не все) ограничиваются рассмотрением только этой области.
- Текущие смещения начального и конечного символов области доступны через методы `regionStart` и `regionEnd` (☞ 482). Метод `reset` (☞ 490) по умолчанию

[1] Вообще говоря, благодаря чрезвычайной гибкости `java.util.regex` в качестве аргумента может использоваться любой объект, реализующий интерфейс `CharSequence` (например, объекты `String`, `StringBuffer` и `CharBuffer`).

устанавливает границы области в начало и в конец всего текста, равно как и любой другой метод, вызывающий метод `reset` (☞ 490).

- Флаг *закрепления границ*. Если область включает в себя только часть целевого текста, появляется возможность находить совпадения с границами области как с «началом текста», так и с «концом текста» при помощью метасимволов границ строки (`\A ^ $ \z \Z`).

- По умолчанию флаг имеет значение `true`, однако он может изменяться и проверяться с помощью методов `useAnchoringBounds`(☞ 485) и `hasAnchoringBounds` соответственно. Метод `reset` не изменяет значение этого флага.

- Флаг *прозрачности границ*. Если область включает в себя только часть всего текста, включение режима «прозрачности границ» позволяет метасимволам в конструкциях поиска (опережающая и ретроспективная проверки, а также метасимволы границ слова) выполнять проверку за пределами обозначенной области.

  По умолчанию этот флаг имеет значение `false`, однако он может изменяться и проверяться с помощью методов `useTransparentBounds` (☞ 484) и `hasTransparentBounds` соответственно. Метод `reset` не изменяет значение этого флага.

Элементы, доступные только для чтения:

- Количество сохраняющих круглых скобок в текущем шаблоне. Получить это значение можно с помощью метода `groupCount`(☞ 470).

- *Позиция совпадения*, или *текущая позиция*, в целевом тексте. Используется для поддержки операции «поиск следующего совпадения» (с помощью метода `find` ☞ 468).

- *Позиция дополнения* в целевом тексте используется для копирования областей текста, для которых не было обнаружено совпадения, в процессе выполнения операции копирования с заменой (☞ 475).

- Флаг, свидетельствующий о достижении конца целевой строки при предыдущей попытке найти совпадение, когда решение об успехе или неудаче еще не принято. Значение можно получить с помощью метода `hitEnd` (☞ 487).

- *Результаты поиска*. Все данные, которые были получены в результате последней успешной попытки найти совпадение, называются результатами поиска (☞ 471). К этим данным относятся: текст найденного совпадения (метод `group()`), смещения начала и конца совпадения в целевом тексте (методы `start()` и `end()`) и информация о совпадениях, обнаруженных каждой группой сохраняющих круглых скобок (с помощью методов `group(`*номер*`)`, `start(`*номер*`)` и `end(`*номер*`)`).

Инкапсулированные результаты поиска доступны также через отдельный объект `MatchResult`, который может быть получен с помощью метода `toMatchResult`. Объект `MatchResult` обладает своими собственными методами `group`, `start` и `end`, которые равноценны одноименным методам объекта `Matcher` (☞ 471).

- Флаг, свидетельствующий о том, завершится ли неудачей попытка поиска совпадения, если целевой текст будет дополнен новыми данными (имеет смысл только после успешного поиска совпадения). Флаг всегда принимает значение `true`, когда метасимволы границ оказывают влияние на решение об успешном совпадении. Значение флага можно получить с помощью метода `requireEnd` (☞ 488).

Эти списки слишком объемны, чтобы сразу запомнить их, гораздо проще будет воспринимать эту информацию в процессе обсуждения методов, сгруппированных по функциональным возможностям, которое приводится в следующих разделах. Помимо этого, дополнительным справочником при работе с данной главой может послужить список методов, который приводится в начале главы (☞ 457).

Применение регулярного выражения

Ниже описаны основные методы `Matcher` для применения регулярного выражения к строке:

`boolean` **`find()`**

Применяет регулярное выражение, представленное объектом, к текущей области (☞ 479) и возвращает логический признак наличия совпадения. При многократном вызове каждый раз возвращается новое совпадение.

Ниже приводится простой пример:

```
String regex = "\\w+"; // 「\w+」
String text = "Mastering Regular Expressions";
Matcher m = Pattern.compile(regex).matcher(text);
if (m.find ())
    System.out.println("match [" + m.group() + "]");
```

Он выведет:

```
match [Mastering]
```

Однако если условный оператор if заменить оператором цикла `while`, как показано ниже

```
while (m.find())
    System.out.println("match [" + m.group() + "]");
```

будут найдены все совпадения в строке:

```
match [Mastering]
match [Regular]
match [Expressions]
```

`boolean` **`find(`**`int` *`смещение`*`)`

Если метод `find` вызывается с целочисленным аргументом, попытка поиска начинается с заданным *смещением* (в символах) от начала строки. Если *смещение* отрицательное или превышает длину текста, инициируется исключение `IndexOutOfBoundsException`.

Данная форма метода `find` не учитывает границы текущей области, поскольку сразу же после вызова (обращением к методу `reset` ☞ 490) переустанавливает границы области так, чтобы она включала в себя всю текстовую строку.

Прекрасный пример использования этой формы метода `find` можно найти на врезке на с. 500 (где приводится ответ на вопрос, заданный на с. 499).

`boolean` **`matches()`**

Метод возвращает логический признак *точного* совпадения регулярного выражения с текущей областью целевого текста (☞ 479). То есть совпадение должно начинаться с позиции начала области и заканчиваться в позиции конца области (которая по умолчанию охватывает всю целевую строку). Когда текущая область охватывает всю строку, метод `matches` отличается от применения регулярного выражения, заключенного в метасимволы `\A(?:…)\z`, только относительной простотой и удобством.

Когда текущая область включает в себя не всю строку (☞ 480), метод `matches` позволяет выполнить поиск совпадений по данной области, независимо от состояния флага закрепления границ (☞ 485).

Например, допустим, что с помощью вашего приложения пользователь выполняет редактирование текста, который хранится в объекте `CharBuffer`, где текущая область соответствует части строки, выделенной пользователем с помощью мыши. Теперь, если пользователь щелкнет мышью по выделенной области, можно с помощью *`m`*`.usePattern(`*`urlPattern`*`).matches()` проверить, не является ли выделенный текст адресом URL (и если это действительно URL, можно будет выполнить некоторые специфические действия).

Объекты класса `String` также поддерживают метод `matches`:

```
"1234".matches("\\d+"); // истинно
"123!".matches("\\d+"); // ложно
```

`boolean` **`lookingAt()`**

Возвращает логический признак, указывающий, находится ли найденное совпадение внутри текущей области целевого текста, начинаясь от начала этой области. Этим метод `lookingAt` напоминает метод `matches`, за исключением того, что здесь не требуется совпадения всей области — совпадение должно начинаться с позиции начала области.

Получение информации о результатах

Следующая категория методов объекта `Matcher` возвращает информацию об успешном поиске. Если регулярное выражение еще не применялось к строке или при последнем поиске не было найдено совпадение, инициируется исключение `IllegalStateException`. Методы, вызываемые с числовым аргументом (определяющим номер пары сохраняющих круглых скобок), инициируют исключение `IndexOutOfBoundsException` при неверном значении аргумента.

Обратите внимание: методы `start` и `end` возвращают значения смещений без учета границ текущей области — возвращаемые значения определяют смещения относительно начала *текста*, которое необязательно должно совпадать с началом *текущей области*.

Вслед за перечнем методов приводится пример, который иллюстрирует использование большинства из них.

`String` **`group()`**

Возвращает текст, совпавший при предыдущем применении регулярного выражения.

`int` **`groupCount()`**

Возвращает количество пар сохраняющих круглых скобок в регулярном выражении. Числа, не превышающие полученной величины, могут использоваться при вызове методов `group`, `start` и `end`, описываемых ниже[1].

`String` **`group(`**`int` *число*`)`

Возвращает текст, совпавший с заданной парой сохраняющих круглых скобок, или `null`, если эта пара не участвует в совпадении. Нулевой аргумент обозначает все совпадение, поэтому вызов `group(0)` эквивалентен вызову `group()`.

[1] Метод `groupCount` может вызываться в любой момент, в отличие от других методов, описываемых в этом разделе, которые можно вызывать только после того, как совпадение будет найдено.

`int start(int число)`

Возвращает абсолютное смещение (в символах) начала совпадения заданной пары сохраняющих круглых скобок от начала строки. Если пара не участвует в совпадении, возвращается `-1`.

`int start()`

Возвращает абсолютное смещение начала совпадения; вызов метода эквивалентен `start(0)`.

`int end(int число)`

Возвращает абсолютное смещение (в символах) конца совпадения заданной пары сохраняющих круглых скобок от начала строки. Если пара не участвует в совпадении, возвращается `-1`.

`int end()`

Возвращает абсолютное смещение конца совпадения; вызов метода эквивалентен `end(0)`.

`MatchResult toMatchResult()`

Был добавлен в Java 1.5.0. Возвращает объект `MatchResult`, содержащий полную информацию о последнем найденном совпадении. Он содержит те же методы `group`, `start`, `end` и `groupCount`, что и класс `Matcher`.

Если попытка найти совпадение еще не предпринималась или поиск закончился неудачей, обращение к методу `toMatchResult` инициирует исключение `IllegalStateException`.

Пример получения результатов

Ниже приводится пример, демонстрирующий использование большинства методов, применяемых для получения результатов поиска. В этом примере анализируется строка URL и определяются: тип протокола (`http` или `https`), имя хоста и номер порта (если таковой указан):

```
String url = "http://regex.info/blog";
String regex = "(?x) ^(https?):// ([^/:]+) (?:(\\d+))?";
Matcher m = Pattern.compile(regex).matcher(url);

if (m.find())
{
    System.out.print(
        "Overall [" + m.group() + "]" +
```

```
        " (from " + m.start() + " to " + m.end() + ")\n" +
        "Protocol [" + m.group(1) + "]" +
        " (from " + m.start(1) + " to " + m.end(1) + ")\n" +
        "Hostname [" + m.group(2) + "]" +
        " (from " + m.start(2) + " to " + m.end(2) + ")\n"
    );
    // Третья пара скобок может не участвовать в совпадении,
    // поэтому сначала нужно выполнить  проверку
    if (m.group(3) == null)
        System.out.println("No port; default of '80' is assumed");
    else {
        System.out.print("Port is [" + m.group(3) + "] " +
                         "(from "  + m.start(3) + " to " + m.end(3) + ")\n";
    }
}
```

По завершении будет выведено:

```
Overall [http://regex.info] (from 0 to 17)
Protocol [http] (from 0 to 4)
Hostname [regex.info] (from 7 to 17)
No port; default of '80' is assumed
```

Простой поиск с заменой

Операции поиска с заменой можно реализовать средствами, описанными выше, но в объекте `Matcher` предусмотрены удобные методы для выполнения простых подстановок.

`String` **`replaceAll(`**`String` *замена*`)`

Возвращаемое значение представляет собой копию исходной строки, в которой все совпадения заменяются строкой *замена*, которая интерпретируется специальным образом, о чем подробно говорится на с. 474.

Данный метод не учитывает границы текущей области (так как внутри вызывает метод `reset`). Однако на с. 476 приводится самодельная версия метода, которая выполняет замену только в пределах текущей области.

Замена также может осуществляться методом `replaceAll` класса `String`. Вызов метода

```
string.replaceAll(выражение, замена);
```

эквивалентен

```
Pattern.compile(выражение).matcher(строка).replaceAll(замена)
```

`String` **`replaceFirst(`**`String` *замена*`)`

Напоминает метод `replaceAll`, но замене подвергается только первое совпадение (если таковое будет найдено).

Замена также может осуществляться методом `replaceFirst` класса `String`.

`static String` **`quoteReplacement(`**`String` *текст*`)`

Этот статический метод, появившийся в Java 1.5, возвращает строку, которая представляет собой литеральное значение *текста* и может использоваться в качестве аргумента *замена*. Для этого метод экранирует в копии *текста* все символы, которые предполагают специальную интерпретацию, о чем будет говориться в следующем разделе. (В этом разделе приводится пример использования метода `Matcher.quoteReplacement`.)

Примеры простого поиска с заменой

Этот простой пример заменяет все вхождения подстроки «Java 1.5» на «Java 5.0»:

```
String text = "Before Java 1.5 was Java 1.4.2. After Java 1.5 is Java 1.6";
String regex = "\\bJava\\s*1\\.5\\b";
Matcher m = Pattern.compile(regex).matcher(text);
String result = m.replaceAll("Java 5.0");
System.out.println(result);
```

В результате будет получено:

```
Before Java 5.0 was Java 1.4.2. After Java 5.0 is Java 1.6
```

Если скомпилированная версия регулярного выражения и объект Matcher далее не потребуются, то все действия можно объединить в цепочку, присвоив результат переменной *result*:

```
Pattern.compile("\\bJava\\s*1\\.5\\b").matcher(text).replaceAll("Java 5.0")
```

(Если одно и то же регулярное выражение будет многократно использоваться программой, тогда гораздо эффективнее будет выполнить предварительную компиляцию объекта `Pattern` ☞ 464.)

Добавив в регулярное выражение (и в строку замены, о чем подробнее будет говориться ниже) незначительные изменения, можно аналогичным образом выполнить замену подстроки «Java 1.6» на «Java 6.0».

```
Pattern.compile("\\bJava\\s*1\\.([56])\\b").matcher(text).
                                          replaceAll("Java $1.0")
```

В результате для той же строки будет получено:

```
Before Java 5.0 was Java 1.4.2. After Java 5.0 is Java 6.0
```

Для замены только первого совпадения в любом из этих примеров вместо метода `replaceAll` можно использовать метод `replaceFirst`. Разумеется, метод `replaceFirst` следует использовать только в случае необходимости ограничиться единственной заменой. Однако с точки зрения эффективности имеет смысл использовать `replaceFirst` и тогда, когда заранее точно известно, что в проверяемом тексте возможно только одно совпадение. (Такое вполне возможно, поскольку в ваших руках имеется полная информация об исходных данных и о регулярном выражении.)

Аргумент замена

В методах `replaceAll` и `replaceFirst` (а также в методе `appendReplacement`, который будет обсуждаться ниже) аргумент *замена*, прежде чем будет вставлен на место найденного совпадения, интерпретируется особым образом:

- Все вхождения '`$1`', '`$2`' и т. д. заменяются текстом, совпавшим с соответствующей парой сохраняющих круглых скобок (`$0` заменяется текстом всего совпадения).
- Если символ, следующий за '`$`', не является ASCII-цифрой, инициируется исключение `IllegalArgumentException`.
- После '`$`' используется столько цифр, сколько «оправданно» контекстом выражения. Например, если выражение содержит три пары сохраняющих скобок, '`$25`' в выражении интерпретируется как последовательность `$2`, за которой следует '`5`'. Тем не менее в этой ситуации '`$6`' в строке замены инициирует исключение `IndexOutOfBoundsException`.
- Символ `\` экранирует следующий символ, поэтому для включения знака `$` в строку замены вставляется последовательность '`\$`'. Аналогично '`\\`' обозначает в строке замены знак `\`. (А если текст замены передаетсяв виде строкового литерала Java, символ `\` в строке замены представляется комбинацией «`\\\\`».) Если строка содержит, скажем, 12 пар сохраняющих круглых скобок и вы хотите включить текст, совпавший с первой парой, за которым следует символ '`2`', используйте последовательность '`$1\2`'.

Если содержимое строки замены заранее неизвестно, следует использовать метод `Matcher.quoteReplacement`, чтобы выполнить экранирование всех служебных символов, которые могут находиться в ней. Допустим, что регулярное выражение находится в переменной *uRegex*, а строка замены — в переменной *uRepl*, тогда операцию поиска с заменой можно будет выполнить следующим образом:

```
Pattern.compile(uRegex).matcher(text).replaceAll(Matcher.
                                    quoteReplacement(uRepl))
```

Расширенный поиск с заменой

Два других метода класса `Matcher` предоставляют непосредственный доступ к механике поиска с заменой. В совокупности эти два метода используются для пошагового построения результата в переданном строковом буфере `StringBuffer`. Первый метод, вызываемый после каждого успешного совпадения, включает в выходной буфер промежуточный текст и результат замены. Второй метод вызывается после обнаружения всех совпадений и включает в выходной буфер текст, оставшийся после последнего совпадения.

`Matcher` **`appendReplacement(`**`StringBuffer` *результат*, `String` *замена***`)`**

Вызывается немедленно после успешного применения регулярного выражения (например, вызовом `find`). Метод присоединяет к заданному *буферу результата* две строки: фрагмент исходного целевого текста, непосредственно предшествующий совпадению, а затем строку *замены*, которая обрабатывается по специальным правилам, о чем говорилось в предыдущем разделе.

Предположим, имеется объект *m* класса Matcher, ассоциирующий регулярное выражение `\w+` со строкой '`-->one+test<--`'. При первой итерации цикла `while`

```
while (m.find())
m.appendReplacement(sb, "XXX")
```

вызов `find` совпадает с подчеркнутым фрагментом '`-->one+test<--`'. Первый вызов метода `appendReplacement` заполняет буфер *`sb`* текстом, предшествующим совпадению, т. е. '`-->`', обходит совпадение и заносит вместо него в *`sb`* текст замены — строку '`XXX`'.

При второй итерации цикла `find` находит совпадение '`-->one+test<--`'. Вызов `appendReplacement` записывает в буфер текст, предшествующий совпадению '`+`', после чего снова присоединяет строку замены '`XXX`'.

В итоге в *`sb`* сохраняется строка '`-->XXX+XXX`', а текущая позиция в целевой строке объекта *`m`* остается в позиции '`-->one+tes▲t<--`'.

Теперь можно воспользоваться методом `appendTail`, который описывается ниже.

`StringBuffer` **`appendTail(`**`StringBuffer` *результат*`)`

Вызывается после того, как будут найдены все совпадения (или после получения нужного количества совпадений — при желании поиск можно прервать раньше). Метод записывает в буфер весь оставшийся текст.

В продолжение предыдущего примера команда

```
m.appendTail(sb)
```

запишет в *sb* строку '<--'. В итоге буфер заполняется окончательным результатом поиска с заменой — текстом '-->XXX+XXX<--'.

Примеры поиска с заменой

Следующий пример демонстрирует имитацию метода `replaceAll` с использованием этих двух методов (делать этого не следует, но пример весьма поучителен).

```
public static String replaceAll(Matcher m, String replacement)
{
    m.reset(); // Инициализировать объект Matcher
    StringBuffer result = new StringBuffer(); // Буфер для построения
                                              // обновленной копии

    while (m.find())
        m.appendReplacement(result, replacement);

    m.appendTail(result);
    return result.toString(); // Преобразовать объект в строку и вернуть
}
```

Как уже говорилось ранее, настоящий метод `replaceAll` никак не учитывает границы текущей области (☞ 482), так как сбрасывает ее в состояние по умолчанию перед выполнением операции поиска с заменой.

Этот недостаток исправлен в следующей ниже версии `replaceAll`, где *учитывается* возможность несовпадения границ области с границами целевого текста. Измененные и добавленные участки программного кода выделены жирным шрифтом:

```
public static String replaceAllRegion(Matcher m, String replacement)
{
    Integer start = m.regionStart();
    Integer end = m.regionEnd();
    m.reset().region(start, end); // Инициализировать Matcher, после чего
                                  // восстановить границы области
     StringBuffer result = new StringBuffer(); // Буфер для построения
                                               // обновленной копии

    while (m.find())
        m.appendReplacement(result, replacement);
```

```
    m.appendTail(result);
    return result.toString(); // Преобразовать объект в строку и вернуть
}
```

Объединение методов `reset` и `region` в одно выражение — пример *конвейерной обработки*, которая будет обсуждаться, начиная со с. 486.

В другом, более сложном примере содержимое переменной `metric`, содержащей температуру по Цельсию, преобразуется в шкалу Фаренгейта:

```
// Построение объекта Matcher для поиска чисел с суффиксом "C"
// внутри переменной "Metric"
// Используется регулярное выражение: ⌜(\d+(?:\.\d*)? )C\b⌟
Matcher m = Pattern.compile("(\\d+(?:\\.\\d*)?)C\\b").matcher(metric);
StringBuffer result = new StringBuffer(); // Буфер для построения
                                          // обновленной копии
while (m.find())
{
    float celsius = Float.parseFloat(m.group(1));  // Получить число
    int fahrenheit = (int)(celsius + 9/5 + 32);    // Преобразовать по шкале
                                                   // Фаренгейта
    m.appendReplacement(result, fahrenheit + "F"); // Вставка
}
m.appendTail(result);
System.out.println(result.toString());             // Вывести результат
```

Например, если переменная `metric` содержит текст '**`from 36.3C to 40.1C.`**', будет получен результат '**`from 97F to 104F.`**'.

Поиск с заменой по месту

До сих пор мы использовали пакет `java.util.regex` только для работы с объектами класса `String`, однако объект `Matcher` в состоянии работать с любыми объектами, реализующими интерфейс `CharSequence`, поэтому имеется возможность выполнять все операции непосредственно над самим текстом в процессе работы программы.

Из классов, реализующих интерфейс `CharSequence`, наиболее часто используются `StringBuffer` и `StringBuilder`, первый из которых может применяться в многопоточных приложениях, но при этом обладает более низкой эффективностью. Оба класса могут использоваться аналогично классу `String`, но в отличие от класса `String` они допускают возможность модификации своего содержимого. В примерах из этой книги используется класс `StringBuilder`, однако в многопоточной среде с равным успехом можно было бы применять и `StringBuffer`.

Ниже приводится простой пример, демонстрирующий возможность поиска и замены всех слов, состоящих только из заглавных символов, на их аналоги, состоящие только из строчных символов, в объекте `StringBuilder`:[1]

```
StringBuilder text = new StringBuilder("It's SO very RUDE to shout!");
Matcher m = Pattern.compile("\\b[\\p{Lu}\\p{Lt}]+\\b").matcher(text);
while (m.find())
    text.replace(m.start(), m.end(), m.group().toLowerCase()); System.out.
println(text);
```

Результат работы этого примера:

```
It's so very rude to shout!
```

Два совпадения были найдены за два вызова `text.replace`. Первые два аргумента определяют последовательность символов, которая должна быть заменена (здесь передаются границы совпадения с регулярным выражением), далее следует текст замены (текст найденного совпадения, символы которого преобразованы в нижний регистр).

Пока размер текста замены совпадает с размером замещаемого фрагмента, как в данном примере, операция поиска с заменой по месту выполняется достаточно просто. Простота подхода сохраняется и при однократном выполнении поиска с заменой, но дело осложняется, когда приходится выполнять поиск с заменой в цикле.

Использование строки замены другого размера

Обработка данных существенно осложняется, когда размер строки замены не совпадает с размером замещаемого фрагмента. Изменения, которые выполняются в целевом тексте, происходят «за спиной» объекта `Matcher`, поэтому его значение *позиции совпадения* (которое используется для определения позиции, откуда метод `find` начнет поиск) может оказаться ошибочным.

Обойти эту проблему можно за счет непосредственного управления позицией совпадения, явно передавая методу `find` смещение, откуда следует начать поиск. Именно такой подход выбран при попытке модифицировать предыдущий пример так, чтобы он добавлял теги `<b>…</b>` вокруг текста замены:

```
StringBuilder text = new StringBuilder("It's SO very RUDE to shout!");
Matcher m = Pattern.compile("\\b[\\p{Lu}\\p{Lt}]+\\b").matcher(text);
int matchPointer = 0;// Поиск начинать с начала строки
```

[1] В этом примере используется регулярное выражение `⌜\b[\p{Lu} \p{Lt}]+\b⌟`. В главе 3 говорилось, что конструкция `\p{Lu}` соответствует всему диапазону символов Юникода в верхнем регистре, а конструкция `\p{Lt}` соответствует титульным символам. ASCII-версия этого регулярного выражения: `⌜\b[AZ]+\b⌟`.

```
while (m.find(matchPointer)) {
    matchPointer = m.end(); // Следующий поиск начинается с позиции, где
                            // завершилась предыдущая  попытка
    text.replace(m.start(), m.end(), "<b>"+ m.group().toLowerCase() +"</b>");
    matchPointer += 7; // Учесть добавленные теги '<b>' и '</b>'
}
System.out.println(text);
```

Результат работы этого примера:

```
It's <b>so</b> very <b>rude</b> to shout!
```

Область в объекте Matcher

Начиная с версии Java 1.5, объект Matcher реализует концепцию изменяемой *области* текста, посредством которой можно ограничить область поиска. Обычно область охватывает весь целевой текст, но с помощью метода `region` имеется возможность изменять положение и размеры области.

В следующем примере выполняется обход строки в формате HTML, в которой отыскиваются теги `<img>`, где отсутствует атрибут `ALT`. Здесь используются два объекта `Matcher`, которые выполняют поиск в одном и том же тексте (в формате HTML), но с помощью разных регулярных выражений: один отыскивает теги `<img>`, а второй — атрибут `ALT`.

Хотя оба объекта `Matcher` работают с одним и тем же текстом, они являются независимыми объектами, связывает их лишь то, что результаты поиска каждого тега `<img>` используются для ограничения области поиска атрибута `ALT`. С этой целью данные, полученные от методов `start` и `end` после успешного поиска тега `<img>`, используются для определения области в объекте `Matcher`, выполняющем поиск атрибута `ALT`, перед вызовом метода `find`.

Определив границы требуемого тега, результат поиска атрибута `ALT` будет свидетельствовать о наличии или отсутствии этого атрибута именно в данном теге, а не во всей строке.

```
// Объект поиска тегов <img>. Переменная 'html' содержит текст HTML
Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);

// Объект поиска атрибута ALT
// (отыскивает атрибут ALT в теге IMG в той же самой переменной 'html')
Matcher mAlt = Pattern.compile("(?ix)\\b ALT \\s* =").matcher(html);

// Для каждого найденного тега <img>...
while (mImg.find()) {
    // Ограничить область поиска атрибута ALT
```

```
    mAlt.region( mImg.start(1), mImg.end(1) );

    // Сообщить об ошибке, если атрибут ALT отсутствует,
    // вывести полное определение тега, найденное выше
    if (! mAlt.find())
        System.out.println("Missing ALT attribute in: " + mImg.group());
}
```

Может показаться странным, что целевой текст назначается объекту в одном месте программы (там, где создается объект *mAlt*), а границы области устанавливаются в другом (при вызове метода *mAlt*.region). Можно поступить и иначе: создавать объект *mAlt* с пустой целевой строкой, а границы области изменять вызовом *mAlt*.**reset(***html***)**.region(…). Такое дополнительное обращение к методу reset несколько снижает эффективность программы, но то, что целевой текст определяется в том же месте программы, что и область, делает программу более понятной.

В любом случае хочу повторить, что если бы сфера поиска для объекта *mAlt* не была ограничена определенной областью, метод find выполнил бы поиск совпадения по всей строке, сообщив о факте наличия или отсутствия подстроки 'ALT=' в не интересующем нас месте строки HTML.

Давайте добавим в наш пример возможность вывода номера строки в тексте HTML, где встречен тег <img> без атрибута ALT. Для этого ограничим видимую область HTML участком, предшествующим найденному тегу <img>, и подсчитаем в нем количество символов новой строки.

Добавленный программный код выделен жирным шрифтом:

```
// Объект поиска тегов <img>. Переменная 'html' содержит текст HTML
Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);
// Объект поиска атрибута ALT
// (отыскивает атрибут ALT в теге IMG в той же самой переменной 'html')
Matcher mAlt = Pattern.compile("(?ix)\\b ALT \\s* =").matcher(html);

// Объект Matcher для поиска символов новой строки
Matcher mLine = Pattern.compile("\\n").matcher(html);

// Для каждого найденного тега <img>...
while (mImg.find())
{
    // Ограничить область поиска атрибута ALT
    mAlt.region( mImg.start(1), mImg.end(1) );
    // Сообщить об ошибке, если атрибут ALT отсутствует,
    // вывести полное определение тега, найденное выше
    if (! mAlt.find()) {
```

```
            // Подсчитать количество символов новой строки перед найденным тегом
            mLine.region(0, mImg.start());
            int lineNum = 1; // Нумерация строк начинается с 1
            while (mLine.find())
                lineNum++;   // При встрече каждого символа новой строки
                             // следует увеличить счетчик строк на 1
            System.out.println("Missing ALT attribute on line " + lineNum);
        }
}
```

Как и прежде, определение области поиска для объекта *mAlt* производится с использованием метода `start(1)`, чтобы определить, где в строке HTML начинается *тело* тега `<img>`. Однако при определении области поиска символов новой строки мы используем метод `start()` без аргументов, потому что он позволяет определить позицию начала самого тега `<img>` (именно в этой позиции нам нужно завершить подсчет символов новой строки).

Что следует помнить

Очень важно помнить, что некоторые методы, связанные с поиском совпадений, не только игнорируют установленные границы области, но они фактически вызывают метод `reset` и тем самым переопределяют границы области так, что она начинает охватывать «весь текст».

- Следующие методы принимают во внимание установленные границы области:

  `matches`

  `lookingAt`

  `find()` (при вызове без аргументов)

- Следующие методы переинициализируют объект Matcher и сбрасывают установки области:

  `find(`*текст*`)` (при вызове с единственным аргументом)

  `replaceAll`

  `replaceFirst`

  `reset` (само собой разумеется)

Не менее важно помнить, что позиции начала и конца найденного совпадения (т. е. значения, которые возвращаются методами `start` и `end`) всегда подсчитываются от начала целевого текста и не зависят от того, как были установлены границы области.

Установка и инспектирование границ области

Операции установки и проверки границ области выполняются следующими тремя методами.

`Matcher` **`region(`**`int` *начало*, `int` *конец*`)`

Этот метод определяет область в пределах целевого текста между *началом* и *концом*, которые представляют собой смещения в символах от начала целевого текста. Кроме того, он переинициализирует объект Matcher и переносит *указатель текущей позиции* в начало области, поэтому при следующем вызове метода `find` поиск начнется с этой позиции.

Границы области остаются неизменными, пока не будут переустановлены или пока не будет вызван метод `reset` (явно или одним из методов, вызывающих `reset` ☞ 490).

Этот метод возвращает объект `Matcher`, который может использоваться в конвейерной обработке (☞ 486).

Если аргумент *начало* или *конец* выходит за рамки целевого текста или *начало* больше, чем *конец*, инициируется исключение `IndexOutOfBoundsException`.

`int` **`regionStart()`**

Возвращает смещение в символах от начала текущей области в объекте Matcher. Значение по умолчанию 0.

`int` **`regionEnd()`**

Возвращает смещение в символах от конца текущей области в объекте `Matcher`. Значение по умолчанию равно длине целевого текста.

Метод `region` требует явного указания значений аргументов *начало* и *конец*, поэтому его не очень удобно использовать, когда требуется переопределить только одну из границ. В табл. 8.4 приводятся варианты решения этой проблемы.

Таблица 8.4. Настройки одной границы области

Начало области	Конец области	Программный код Java
устанавливается явно	остается неизменной	`m.region(`*начало*`, m.regionEnd());`
остается неизменной	устанавливается явно	`m.region(m.regionStart(), `*конец*`);`
устанавливается явно	сбрасывается в значение по умолчанию	`m.reset().region(`*начало*`, m.regionEnd());`
сбрасывается в значение по умолчанию	устанавливается явно	`m.region(0, `*конец*`);`

Поиск за пределами текущей области

Переустановка границ текущей области в значения, отличные от значений по умолчанию, обычно приводит к сокрытию некоторых участков текста от механизма регулярных выражений. Это означает, например, что начало области будет соответствовать метасимволу ⌜^⌟ даже в том случае, когда оно не совпадает с началом целевого текста.

Однако существуют некоторые ограниченные возможности проверять данные за пределами области в процессе поиска. Установка флага *прозрачности границ* дает возможность конструкциям «просмотра» (опережающая и ретроспективная проверки, а также метасимволы границ слова) выходить за пределы области, а установка флага *закрепления границ* позволит настроить границы области так, что они *не* будут рассматриваться как начало и/или конец ввода (если они действительно таковыми не являются).

Причина, по которой может потребоваться изменять эти флаги, тесно связана с причиной, по которой возникает необходимость изменять положение границ области. В предыдущих примерах не было никакой необходимости изменять значения флагов, потому что в них, при поиске в пределах области, не использовались ни якорные метасимволы, ни конструкции проверки.

Но представьте себе еще раз приложение, где пользователь выполняет редактирование текста, который хранится в объекте `CharBuffer`. Так, если пользователь выполняет операцию поиска или поиска с заменой, вполне естественно будет ограничиться текстом, расположенным после курсора. Следовательно, появляется необходимость установить начало текущей области в позицию курсора. Предположим, что курсор находится в отмеченной позиции в следующем тексте:

```
Madagas▲car is much too large to see on foot, so you'll need a car.
```

и выполняется поиск совпадения с выражением ⌜`\bcar\b`⌟ с целью заменить его словом «automobile». После соответствующей настройки области (чтобы ограничиться при поиске только текстом справа от курсора) запускается процедура поиска и... возможно, к вашему удивлению, совпадение будет найдено в самом начале области, в слове '`Madagascar`'. Это совпадение будет найдено по той простой причине, что по умолчанию флаг прозрачности границ сброшен (имеет значение `false`) и конструкция ⌜`\b`⌟ будет считать, что начало области совпадает с началом текста. Она не сможет «увидеть», что именно находится перед началом области. Если бы флаг прозрачности границ был установлен, конструкция ⌜`\b`⌟ обнаружила бы символ '`s`' перед первым символом '`c`' в начале области и поняла бы, что в данном месте нет совпадения с ⌜`\b`⌟.

Прозрачность границ

Следующие методы тесно связаны с флагом прозрачности границ:

`Matcher` **`useTransparentBounds(`**`boolean` *b*`)`

Устанавливает флаг прозрачности границ в значение *true* или *false*, в соответствии со значением аргумента. Значение по умолчанию — *false*.

Возвращает сам объект `Matcher`, благодаря чему этот метод можно использовать для организации конвейерной обработки данных (☞ 486).

`boolean` **`hasTransparentBounds()`**

Возвращает значение *true*, если флаг прозрачности границ установлен, в противном случае — *false*.

По умолчанию флаг прозрачности границ в объекте `Matcher` имеет значение *false*, поэтому границы области непрозрачны для таких конструкций «просмотра», как опережающая и ретроспективная проверка, а также метасимволы границ слова. Кроме того, любые символы, находящиеся за пределами области, будут недоступны для механизма регулярных выражений[1]. Это означает, что даже если начало области находится в середине слова, метасимвол ⌜`\b`⌟ обнаружит соответствие в начале области, поскольку ему будут недоступны для просмотра символы, находящиеся непосредственно перед началом области.

Следующий пример демонстрирует действие флага прозрачности границ со значением *false* (по умолчанию):

```
String regex = "\\bcar\\b"; // ⌜\bcar\b⌟
String text = "Madagascar is best seen by car or bike.";
Matcher m = Pattern.compile(regex).matcher(text);
m.region(7, text.length());
m.find();
System.out.println("Matches starting at character " + m.start());
```

Результат работы этого фрагмента:

```
Matches starting at character 7
```

[1] В версии Java 1.5 Update 7 существует одно исключение из этого правила, обусловленное трудноуловимой ошибкой, сообщение о которой я отправил в Sun. `Pattern.MULTILINE` версия метасимвола ⌜`^`⌟ (которая в контексте использования области со значениями границ, отличными от значений по умолчанию, может рассматриваться как одна из конструкций просмотра) может обнаруживать соответствие в начале области, если непосредственно перед ней находится символ-завершитель строки, даже если флаг закрепления границ сброшен, а флаг прозрачности установлен.

наглядно демонстрирует, что метасимвол границы слова действительно находит соответствие в начале области, прямо посреди слова `Madagascar`, несмотря на то что фактически эта позиция не является границей слова. Непрозрачная граница области «имитирует» границу слова.

Однако стоит добавить строку:

```
m.useTransparentBounds(true);
```

перед вызовом метода `find`, и результат примера изменится:

```
Matches starting at character 27
```

На этот раз границы области стали прозрачными, механизм регулярных выражений обнаружил символ 's' непосредственно перед началом области и смог предотвратить совпадение `\b` с началом области. Благодаря этому было обнаружено соответствие '…by•car•or•bike.'

Еще раз напомню: изменять флаг прозрачности границ имеет смысл только тогда, когда границы области не совпадают с границами «всего текста». Обратите также внимание на тот факт, что метод `reset` *не* изменяет значение данного флага.

Закрепление границ

Следующие методы тесно связаны с флагом закрепления границ:

`Matcher` **`useAnchoringBounds(`**`boolean` ***b*****`)`**

Устанавливает флаг закрепления границ в значение *true* или *false* в соответствии со значением аргумента. Значение по умолчанию — *true*.

Возвращает сам объект `Matcher`, благодаря чему этот метод можно использовать для организации конвейерной обработки данных (☞ 486).

`boolean` **`hasAnchoringBounds()`**

Возвращает значение *true*, если флаг закрепления границ установлен, и *false* — в противном случае.

По умолчанию флаг закрепления границ в объекте `Matcher` имеет значение *true*, поэтому метасимволы границ строки (`^ \A $ \z \Z`) соответствуют границам области, даже если эти границы не совпадают с началом и концом целевой строки. После установки флага в значение *false* метасимволы границ строки будут обнаруживать соответствие только в истинных границах целевой строки при условии, что область будет включать их.

Сбрасывать флаг закрепления границ может потребоваться по тем же самым причинам, по которым может потребоваться устанавливать флаг прозрачности

границ — для обеспечения соответствия семантики области в строке с ожиданиями пользователя, когда «курсор ввода находится не в начале текста».

Как и в случае с флагом прозрачности границ, изменять флаг закрепления границ имеет смысл только тогда, когда границы области не совпадают с границами «всего текста». Обратите также внимание на тот факт, что метод `reset` *не* изменяет значение данного флага.

Объединение методов в конвейер

Рассмотрим следующую последовательность действий, которая выполняет подготовку объекта `Matcher` к работе и делает настройку некоторых его параметров:

```
Pattern p = Pattern.compile(regex);  // Компиляция регулярного выражения.
Matcher m = p.matcher(text);         // Связывание текста с регулярным
                                     // выражением за счет создания
                                     // объекта Matcher.
m.region(5, text.length());          // Переносит начало области на пять
                                     // символов вперед.
m.useAnchoringBounds(false);         // Не позволять ^ совпадать с началом
                                     // области.
m.useTransparentBounds(true);        // Позволить конструкциям просмотра
                                     // заглядывать за границы области.
```

В предыдущих примерах было показано, что если шаблон регулярного выражения не требуется нигде, кроме как в создаваемом объекте `Matcher` (что случается достаточно часто), можно объединить первые две строки в одну:

```
Matcher m = Pattern.compile(regex).matcher(text);
m.region(5, text.length());      // Переносит начало области
                                 // на пять символов вперед.
m.useAnchoringBounds(false);     // Не позволять ^ совпадать с началом области.
m.useTransparentBounds(true);    // Позволить конструкциям просмотра
                                 // заглядывать за границы области.
```

Учитывая, что следующие ниже методы тоже возвращают объект `Matcher`, их также можно объединить в одной строке (хотя для того, чтобы эта строка уместилась на странице книги, она была разбита на две строки):

```
Matcher m = Pattern.compile(regex).matcher(text).region(5,text.length())
          .useAnchoringBounds(false).useTransparentBounds(true);
```

Это не дает никакой дополнительной функциональности, просто такая форма записи более удобна. Такой способ объединения методов в конвейер может осложнить составление и форматирование документации к программному коду, но в этом

случае описание будет стремиться отвечать на вопросы *почему*, а не *что*, поэтому, возможно, это не так уж и плохо. Прием объединения методов в конвейер позволил сохранить ясность и краткость примера на с. 498.

Методы для построения сканеров

В Java 1.5 объект `Matcher` обрел два новых метода — `hitEnd` и `requireEnd`, которые в основном используются для построения сканеров. Сканер выполняет анализ потока символов и преобразует его в поток лексем. Например, сканер, который является частью компилятора, может принимать на входе строку '`var•<•34`' и возвращать последовательность из трех лексем `IDENTIFIER LESS_THAN INTEGER`.

Эти методы помогают сканеру решить, следует ли использовать результаты только что завершенного поиска для оценки текущего ввода. Вообще говоря, значение *true*, возвращаемое обоими методами, говорит о том, что для принятия определенного решения требуются дополнительные данные. Например, текущий ввод (скажем, символы, вводимые пользователем с клавиатуры в интерактивном отладчике) содержит единственный символ '`<`'. В такой ситуации благоразумнее будет узнать, не является ли следующий символ знаком '`=`', чтобы точно определить тип очередной лексемы — `LESS_THAN` или `LESS_THEN_OR_EQUAL`.

В большинстве проектов, связанных с обработкой регулярных выражений, эти методы не используются, но когда они востребованы, их значимость сложно переоценить. Ценность целевого применения метода `hitEnd` делает тем более обидной ошибку в его реализации в Java 1.5, не позволяющую полагаться на этот метод. К счастью, в Java 1.6 эта ошибка, похоже, была исправлена, а для Java 1.5 существует обходной маневр, который описывается в конце этого раздела.

Тема построения сканеров выходит далеко за рамки этой книги, поэтому я ограничу рассмотрение этих узкоспециализированных методов их описанием, дополненным несколькими иллюстративными примерами. (Если у вас появится необходимость создать свой сканер, обратите внимание на пакет `java.util.Scanner`.)

`boolean` **`hitEnd()`**

(Этот метод не гарантирует полной надежности в Java 1.5; способ решения этой проблемы представлен на с. 489.)

Этот метод свидетельствует о попытке механизма регулярных выражений заглянуть за текущий конец входной строки в процессе поиска совпадения (независимо от того, увенчалась эта попытка успехом или нет). Сюда относятся попытки обнаружить такие границы, как ⌜`\b`⌟ и ⌜`$`⌟.

Если `hitEnd` возвращает значение *true*, то появление дополнительных данных может привести к изменению результата (неудача может превратиться в успех,

успех — в неудачу или изменится текст совпадения). С другой стороны, возвращаемое значение *false* означает, что результат предыдущей попытки поиска совпадения был получен исключительно из существующей входной строки, с которой работал механизм регулярных выражений, и что появление новых данных, скорее всего, не повлияет на этот результат.

Как правило, возвращаемое значение *true*, полученное от метода `hitEnd` после успешной попытки поиска, свидетельствует о том, что следует дождаться появления дополнительных данных, прежде чем принимать решение. Если поиск совпадения потерпел неудачу и при этом `hitEnd` возвращает *true*, то, прежде чем прерывать анализ входных данных с признаком синтаксической ошибки, следует дождаться появления дополнительной информации на входе.

`boolean requireEnd()`

Возвращаемое значение этого метода, которое имеет смысл только в случае успешной попытки поиска совпадения, свидетельствует о том, повлиял ли текущий конец входной строки на достижение успеха. Проще говоря, если `requireEnd` возвращает *true*, это говорит о том, что появление дополнительных входных данных может привести к изменению результата поиска, но может и не привести.

Как правило, возвращаемое значение *true*, полученное от метода `requireEnd`, свидетельствует о том, что следует дождаться появления дополнительных данных, прежде чем принимать решение.

Оба метода, `hitEnd` и `requireEnd`, принимают во внимание границы текущей области.

Примеры, иллюстрирующие использование методов hitEnd и requireEnd

В табл. 8.5 приводятся значения, возвращаемые методами `hitEnd` и `requireEnd` после выполнения операции поиска с помощью метода `lookingAt`. Хотя два использованных регулярных выражения отличаются далекой от реальности простотой, тем не менее они достаточно наглядно иллюстрируют эти методы.

Регулярное выражение в верхней половине табл. 8.5 отыскивает неотрицательные целые числа и четыре оператора сравнения: больше, меньше, больше или равно и меньше или равно. Регулярное выражение в нижней половине еще проще, оно отыскивает слова `set` и `setup`. Еще раз напомню, что, несмотря на свою простоту, эти примеры являются весьма поучительными.

Например, обратите внимание на строку 5 в табл. 8.5: здесь, несмотря на полное совпадение целевого текста с регулярным выражением, метод `hitEnd` возвращает значение `false`. Причина состоит в том, что в совпадение попал последний символ текста, а механизм регулярных выражений не смог проверить следующий символ (чтобы узнать, будет ли это граница слова или какой-то другой символ).

Таблица 8.5. Значения, возвращаемые методами hitEnd и requireEnd после поиска методом lookingAt

	Регулярное выражение	Текст	Совпадение	hitEnd()	requireEnd()
1	`\d+\b|[><]=?`	'1234'	'1234'	true	true
2	`\d+\b|[><]=?`	'1234•>•567'	'1234•>•567'	false	false
3	`\d+\b|[><]=?`	'>'	'>'	true	false
4	`\d+\b|[><]=?`	'>•567'	'>•567'	false	false
5	`\d+\b|[><]=?`	'>='	'>='	false	false
6	`\d+\b|[><]=?`	'>=•567'	'>=•567'	false	false
7	`\d+\b|[><]=?`	'oops'	*нет совпадения*	false	
8	`(set|setup)\b`	'se'	*нет совпадения*	true	
9	`(set|setup)\b`	'set'	'set'	true	true
10	`(set|setup)\b`	'setu'	*нет совпадения*	true	
11	`(set|setup)\b`	'setup'	'setup'	true	true
12	`(set|setup)\b`	'set•x=3'	'set•x=3'	false	false
13	`(set|setup)\b`	'setup•x'	'setup•x'	false	false
14	`(set|setup)\b`	'self'	*нет совпадения*	false	
15	`(set|setup)\b`	'oops'	*нет совпадения*	false	

Ошибка в реализации метода hitEnd и способ ее обхода

«Ошибка `hitEnd`» в Java 1.5 (исправлена в Java 1.6)[1] приводит к ненадежности результатов, получаемых от метода `hitEnd` в одной весьма специфичной ситуации: когда необязательный односимвольный компонент регулярного выражения используется в режиме поиска без учета регистра символов (точнее, когда такая попытка поиска совпадения терпит неудачу).

Например, попытка поиска с помощью регулярного выражения ⌜`>=?`⌟ в режиме без учета регистра символов (самого по себе или как части более крупного регулярного выражения) приводит к ошибочному результату, потому что символ '=' не является обязательным. Еще один пример, ⌜`a|an|the`⌟ в режиме без учета регистра символов (также само по себе или в составе более крупного регулярного выражения) приводит к ошибочному результату, потому что альтернатива ⌜`a`⌟ соответствует

[1] Пока книга готовилась к печати, Sun сообщила мне, что эта ошибка будет исправлена в версии «5.0u9», т. е. в Java 1.5 Update 9. (Если вы еще помните, в сноске на с. 456 упоминалось, что в данной книге рассматривается версия Java 1.5 Update 7.) Кроме того, в Java 1.6 beta эта ошибка уже исправлена.

единичному символу и, будучи одним из вариантов конструкции выбора, является необязательной.

В качестве примеров можно также привести выражения: ⌜`values?`⌟ и ⌜`\r?\n\r?\n`⌟.

Обходной маневр

Решение проблемы заключается в ликвидации условий, порождающих ошибку: либо в отказе от режима поиска без учета регистра символов (по крайней мере, для спорного подвыражения), либо в замене односимвольных конструкций чем-то другим, например классами символов.

Следуя первому способу, выражение ⌜`>=?`⌟ можно заменить на ⌜`(?-i:>=?)`⌟, где с помощью конструкции локального изменения режима (☞ 150) отменяется поиск с учетом регистра символов на данное подвыражение (для которого регистр символов не имеет никакого значения).

Следуя второму способу, выражение ⌜`a|an|the`⌟ можно заменить на ⌜`[aA]|an|the`⌟, сохранив режим нечувствительности к регистру символов, установленный с помощью флага `Pattern.CASE_INSENSITIVE`.

Другие методы Matcher

Следующие методы объекта `Matcher` не попали ни в одну из категорий, обсуждавшихся ранее.

`Matcher` **`reset()`**

Этот метод выполняет повторную инициализацию большинства параметров объекта `Matcher`: полностью уничтожает информацию о предыдущем успешном поиске совпадения, переустанавливает указатель текущей позиции в начало текста и переустанавливает границы *области* так, чтобы она полностью охватывала весь текст (☞ 479). Без изменения остаются только флаги прозрачности и закрепления границ (☞ 485).

Метод `reset` вызывается из следующих трех методов объекта Matcher: `replaceAll`, `replaceFirst` и `find` (с одним аргументом), что в качестве побочного эффекта влечет за собой изменение границ области.

Этот метод возвращает сам объект `Matcher`, что позволяет использовать его в конвейерной обработке (☞ 486).

`Matcher` **`reset`**`(CharSequence` *текст*`)`

Этот метод также производит повторную инициализацию объекта `Matcher`, как и метод `reset()`, но, кроме того, замещает целевой текст новым объектом класса `String` (или любого другого, реализующего интерфейс `CharSequence`).

При необходимости многократного использования одного и того же регулярного выражения (например, при поиске по отдельным строкам в процессе чтения из файла), более эффективно будет вызывать метод `reset` с новой текстовой строкой в качестве аргумента, чем каждый раз создавать новый объект `Matcher`.

Этот метод возвращает сам объект `Matcher`, что позволяет использовать его в конвейерной обработке (☞ 486).

`Pattern` **`pattern()`**

Метод `pattern` возвращает объект `Pattern`, ассоциированный с объектом `Matcher`. Чтобы получить текст регулярного выражения, можно воспользоваться конструкцией *m*`.pattern().pattern()`, где производится вызов метода `pattern` объекта `Pattern` (несмотря на идентичность имен, это совершенно разные методы).

`Matcher` **`usePattern(`**`Pattern` *p*`)`

Этот метод, появившийся в Java 1.5, замещает объект `Pattern`, ассоциированный с объектом `Matcher`, новым, из аргумента *p*. Этот метод не вызывает `reset` объекта `Matcher`, что позволяет в цикле использовать различные шаблоны для организации поиска совпадений с «текущей позиции» внутри текста объекта `Matcher`. Пример использования метода `usePattern` приводится на с. 498.

Этот метод возвращает сам объект `Matcher`, что позволяет использовать его в конвейерной обработке (☞ 486).

`String` **`toString()`**

Этот метод, также появившийся в Java 1.5, возвращает строку, содержащую некоторую базовую информацию об объекте `Matcher`, которая может быть полезна при отладке. Информационное наполнение и формат строки, возможно, еще претерпят некоторые изменения, но в версии Java 1.6 следующий фрагмент:

```
Matcher m = Pattern.compile("(\\w+)").matcher("ABC 123");
System.out.println(m.toString());
m.find();
System.out.println(m.toString());
```

выводит следующее:

```
java.util.regex.Matcher[pattern=(\w+) region=0,7 lastmatch=]
java.util.regex.Matcher[pattern=(\w+) region=0,7 lastmatch=ABC]
```

В Java 1.4.2 класс `Matcher` обладает универсальным методом `toString`, унаследованным от `java.lang.Object`, но он возвращает менее полезную информацию: '`java.util.regex.Matcher@480457`'.

Извлечение целевого текста из объекта Matcher

В классе `Matcher` отсутствует метод, с помощью которого можно было бы извлечь текущий целевой текст, поэтому ниже приводится пример функции, которая восполняет этот недостаток:

```
// Следующий шаблон используется в функции, поэтому для повышения
// эффективности он компилируется и сохраняется здесь.
static final Pattern pNeverFail = Pattern.compile("^");

// Возвращает целевой текст, ассоциированный с объектом Matcher.
public static String text(Matcher m)
{
    // Запомнить следующие параметры, чтобы потом можно было восстановить их.
    Integer regionStart = m.regionStart();
    Integer regionEnd = m.regionEnd();
    Pattern pattern = m.pattern();

    // Извлечь строку можно единственным способом.
    String text = m.usePattern(pNeverFail).replaceFirst("");

    // Восстановить значения параметров, которые были изменены
    // (или могли измениться).
    m.usePattern(pattern).region(regionStart, regionEnd);

    // Вернуть полученный текст
    return text;
}
```

Чтобы получить неизмененную копию целевого текста в виде объекта класса `String`, функция использует метод `replaceFirst`, фиктивный шаблон и пустую строку замены. В процессе работы происходит повторная инициализация объекта `Matcher`, но сохраняется положение границ области. Такое решение не отличается изяществом (и эффективностью); кроме того, функция всегда возвращает объект класса `String`, хотя целевой текст мог быть представлен объектом другого класса; тем не менее придется довольствоваться им, пока Sun не предложит более эффективную реализацию.

Другие методы Pattern

Помимо основного метода `compile`, класс `Pattern` содержит ряд вспомогательных функций и методов.

`split`

Начиная со следующей страницы, во всех подробностях будут рассмотрены две формы обращения к этому методу.

`String` **`pattern()`**

Возвращает строку регулярного выражения, которая использовалась при создании объекта `Pattern`.

`String` **`toString()`**

Этот метод был добавлен в Java 1.5. Является синонимом метода `pattern`.

`int` **`flags()`**

Возвращает значения флагов (в виде целого числа), которые передавались методу `compile` при создании объекта `Pattern`.

`static String` **`quote(`**`String` *текст*`)`

Этот статический метод был добавлен в Java 1.5. Возвращает строку регулярного выражения, после чего она может использоваться в качестве аргумента метода `Pattern.compile`. Например, вызов `Pattern.quote("main()")` вернет строку '`\Qmain()\E`', которая затем может использоваться в качестве регулярного выражения `「\Qmain()\E」`, соответствующего исходному аргументу '`main()`'.

`static boolean` **`matches(`**`String` *выражение*, `CharSequence` *текст*`)`

Статический метод возвращает логический признак, указывающий, совпадает ли *точно* заданное *выражение* в *тексте* (который, как объект метода `matcher`, представлен объектом, реализующим интерфейс *CharSequence*, например *String*). Вызов метода фактически эквивалентен:

```
Pattern.compile(regex).matcher(text).matches();
```

Если вам потребуется передать параметры компиляции или получить дополнительную информацию о совпадении (помимо простого логического признака успеха), используйте методы, описанные выше.

Если этот метод будет вызываться несколько раз (например, в цикле или из другого участка программы, к которому выполняются частые обращения), вы без труда заметите, что гораздо эффективнее сначала создать скомпилированную версию регулярного выражения в виде объекта `Pattern`, а затем использовать его там, где это необходимо.

Метод split класса Pattern с одним аргументом

`String[]` **`split(`**`CharSequence` *текст*`)`

Этот метод получает *текст* (`CharSequence`) и возвращает массив строк, разделенных совпадениями регулярного выражения. Операция также может быть выполнена методом `split` класса `String`.

Тривиальный пример:

```
String[] result = Pattern.compile("\\.").split("209.204.146.22");
```

Команда возвращает массив из четырех строк (‘209’, ‘204’, ‘146’ и ‘22’), разделенных тремя вхождениями `\.` в текст. В этом простом примере разбиение производится по одному литеральному символу, однако для разбивки может использоваться произвольное регулярное выражение. Например, для приблизительного разделения текста на «слова» можно использовать разбиение по последовательностям символов, не являющихся алфавитно-цифровыми:

```
String[] result = Pattern.compile("\\W+").split(Text);
```

Для строки вида ‘What's up, Doc’ возвращаются четыре строки (‘What’, ‘s’, ‘up’, ‘Doc’), разделенные тремя совпадениями регулярного выражения. При работе с текстом в кодировке Юникод вместо `\W+` следует использовать регулярное выражение `\P{L}+` или `[^\p{L}\p{N}_]` (☞ 458).

Пустые элементы со смежными совпадениями

Если регулярное выражение совпадает в начале *текста*, первая строка, возвращаемая `split`, является пустой (т. е. возвращается допустимая строка, не содержащая ни одного символа). Аналогично, если регулярное выражение может совпасть два и более раза подряд, для текста нулевой длины, «разделяемого» соседними совпадениями, тоже возвращаются пустые строки. Пример:

```
String[] result = Pattern.compile("\\s*,\\s*").split(", one, two , ,, 3");
```

Приведенная команда осуществляет разбиение по запятым и окружающим их пробелам. В результате возвращается массив из пяти элементов: пустая строка, ‘one’, ‘two’, две пустые строки и ‘3’.

Все пустые строки, которые могут находиться в *конце* списка, подавляются:

```
String[] result = Pattern.compile(":").split(":xx:");
```

Команда создает список из двух строк: пустой строки и ‘xx’. Для сохранения завершающих пустых элементов используйте версию `split` с двумя аргументами, описанную ниже.

Метод split класса Pattern с двумя аргументами

`String[]` **`split(`**`CharSequence` *текст*, `int` *предел*`)`

Эта версия `split` позволяет в некоторой степени контролировать количество применений регулярного выражения `Pattern`, а также включение пустых элементов в конец списка.

Интерпретация аргумента *предел* зависит от его знака.

Вызов split с отрицательным пределом

Любые отрицательные значения аргумента *предел* означают, что завершающие пустые элементы должны включаться в массив. Таким образом, команда

```
String[] result = Pattern.compile(":").split(":xx:", -1);
```

возвращает массив из трех строк (пустая строка, 'xx' и еще одна пустая строка).

Вызов split с нулевым пределом

Нулевой *предел* обрабатывается по тем же правилам, как при вызове `split` без аргументов (т. е. завершающие пустые элементы подавляются).

Вызов split с положительным пределом

С положительным *пределом* `split` возвращает массив, содержащий не более заданного количества элементов. Это означает, что регулярное выражение применяется максимум *предел* — 1 раз (например, с пределом 3 создаются три строки, разделенные двумя совпадениями).

После того как будет найдено *предел* — 1 совпадение, дальнейший поиск прекращается, а весь остаток строки за последним совпадением возвращается в последнем элементе массива.

Например, если вы хотите выделить из строки

```
Friedl,Jeffrey,Eric Francis,America,Ohio,Rootstown
```

только три начальных поля, строку следует разбить на четыре части (три поля и остаток строки).

```
Stnng[] Namelnfo = Pattern compile(",") split(Text, 4);
// NameInfo[0] - фамилия
// NameInfo[l] - имя
// NameInfo[2] - отчество (в моем случае состоит из двух имен)
// NameInfo[3] - остальная информация. Нас она не интересует, поэтому этот
// элемент просто игнорируется
```

Ограничение количества элементов при разбиении производится для повышения эффективности: зачем тратить время на поиски остальных совпадений, создание новых строк, увеличения размеров массива и т. д., если вы все равно не собираетесь использовать эти результаты? При установленном пределе будет выполнена только необходимая работа.

Дополнительные примеры

Добавление атрибутов WIDTH и HEIGHT в теги <img>

В этом разделе описывается расширенный пример поиска с заменой, который отыскивает все теги `<img>` в HTML-документе и вставляет в них недостающие атрибуты WIDTH и HEIGHT. (HTML-документ должен быть представлен объектом класса `StringBuilder`, `StringBuffer` или любого другого класса, реализующего интерфейс `CharSequence`.)

Загрузка веб-страницы, в которой имеется хотя бы один элемент `<img>` с отсутствующими атрибутами размера, будет выглядеть медленной. Связано это с тем, что браузер должен полностью загрузить такие изображения, прежде чем он сможет правильно определить положение элементов страницы. При наличии атрибутов размера текст и другие компоненты могут быть сразу правильно скомпонованы на странице, что вызывает у пользователя ощущение более быстрой загрузки[1].

При обнаружении тега `<img>` программа отыскивает в нем атрибуты SRC, WIDTH и HEIGHT и извлекает их значения. Если какой-либо атрибут (WIDTH или HEIGHT) отсутствует, загружается само изображение и по нему определяются его размеры, которые затем используются для конструирования недостающих атрибутов.

Если в оригинальном теге отсутствуют сразу оба атрибута — WIDTH и HEIGHT, при создании обоих атрибутов используются истинные размеры изображения. Однако если в теге присутствует хотя бы один из атрибутов размера, то добавляться будет только другой атрибут, со значением, вычисленным с учетом коэффициента масштабирования, полученного при анализе присутствующего атрибута. (Например, если в атрибуте WIDTH указано значение в 2 раза меньшее, чем истинная ширина изображения, то при добавлении атрибута HEIGHT он получит значение также в 2 раза меньшее истинной высоты изображения; такое решение имитирует поведение современных браузеров в подобных ситуациях.)

В этом примере используется указатель позиции совпадения, как описывалось в разделе «Использование строки замены другого размера».

В нем создаются области (☞ 478) и выполняется объединение методов в конвейеры (☞ 479). Ниже приводится программный код примера:

---

[1] С самых первых дней в Yahoo! существовало непреложное правило: «Все элементы `<img>` *обязательно должны* иметь атрибуты размеров!». Что самое удивительное, и сейчас существует великое множество веб-сайтов, отправляющих клиентам объемные веб-страницы, в которых теги `<img>` не имеют атрибутов размеров.

```
// Объект Matcher для работы с отдельными тегами <img>
Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);
// Объекты Matcher для работы с атрибутами SRC, WIDTH и HEIGHT
// (с очень простыми регулярными выражениями)
Matcher mSrc    = Pattern.compile("(?ix)\\bSRC   =(\\S+)").matcher(html);
Matcher mWidth  = Pattern.compile("(?ix)\\bWIDTH =(\\S+)").matcher(html);
Matcher mHeight = Pattern.compile("(?ix)\\bHEIGHT=(\\S+)").matcher(html);

int imgMatchPointer = 0; // Поиск начинается с начала строки
while (mImg.find(imgMatchPointer))
{
    imgMatchPointer = mImg.end(); // Поиск следующего тега <img> начинается
                          // с той точки, где был закончен предыдущий поиск
    // Отыскать атрибуты в теле только что найденного тега
    Boolean hasSrc    = mSrc.region( mImg.start(1), mImg.end(1) ).find();
    Boolean hasHeight = mHeight.region( mImg.start(1), mImg.end(1) ).find();
    Boolean hasWidth  = mWidth.region( mImg.start(1), mImg.end(1) ).find();

    // Если имеется атрибут SRC и отсутствуют атрибуты WIDTH и/или HEIGHT...
    if (hasSrc && (! hasWidth || ! hasHeight))
    {
        java.awt.image.BufferedImage i = // загрузить изображение
            javax.imageio.ImageIO.read(new java.net.URL(mSrc.group(1)));
        String size; // Хранит отсутствующие атрибуты WIDTH и/или HEIGHT
        if (hasWidth)
            // Ширина известна, необходимо найти высоту
            // с учетом коэффициента масштабирования
            size = "height='" + (int)(Integer.parseInt(mWidth.group(1)) *
                                      i.getHeight() / i.getWidth()) + "' ";
        else if (hasHeight)
            // Высота известна, необходимо найти ширину,
            // с учетом коэффициента масштабирования
            size = "width='" + (int)(Integer.parseInt(mHeight.group(1)) *
                                     i.getWidth() / i.getHeight()) + "' ";
        else
            // Ни один из размеров не известен
            // вставит фактические значения ширины и высоты
               size = "width='"  + i.getWidth() + "' " +
                      "height='" + i.getHeight() + "' ";
        html.insert(mImg.start(1), size); // Дополнить HTML-документ
        imgMatchPointer += size.length(); // Учесть изменение длины тега
    }
}
```

Несмотря на наглядность этого примера, хочется сделать несколько замечаний. Так как основное внимание в примере уделено операции поиска с заменой непосредственно в документе, я постарался упростить некоторые аспекты, которые не

имеют прямого отношения к этой задаче. Например, регулярные выражения не допускают возможности появления пропусков вокруг знаков равенства или кавычек, окружающих значения атрибутов. (На с. 265 приводится более реалистичный вариант регулярных выражений языка Perl для поиска атрибутов HTML-тегов.) Программа не обрабатывает относительные или некорректно оформленные URL и не перехватывает исключения, которые могут быть инициированы программным кодом, выполняющим загрузку изображений.

Тем не менее это достаточно поучительный пример, демонстрирующий ряд важных концепций.

Проверка корректности HTML-кода с использованием нескольких регулярных выражений на один объект Matcher

Ниже приводится Java-версия Perl-программы, выполняющей проверку подмножества тегов HTML (☞ 178). В этом фрагменте с помощью метода `usePattern` производится смена регулярных выражений на лету. Это позволяет производить поиск по строке с использованием нескольких регулярных выражений, каждое из которых начинается с `\G`. Дополнительные пояснения к используемому подходу вы найдете на с. 178.

```
Pattern pAtEnd    = Pattern.compile("\\G\\z");
Pattern pWord     = Pattern.compile("\\G\\w+");
Pattern pNonHtml  = Pattern.compile("\\G[^\\w<>&]+");
Pattern pImgTag   = Pattern.compile("\\G(?i)<img\\s+([^>]+)>");
Pattern pLink     = Pattern.compile("\\G(?i)<A\\s+([^>]+)>");
Pattern pLinkX    = Pattern.compile("\\G(?i)</A>");
Pattern pEntity   = Pattern.compile("\\G&(#\\d+;\\w+);");

Boolean needClose = false;
Matcher m = pAtEnd.matcher(html); // Наш объект Matcher может быть создан
                                  // любым из существующих объектов Pattern
while (! m.usePattern(pAtEnd).find())
{
    if (m.usePattern(pWord).find()) {
        ...получив слово или число в m.group(), его можно
           подвергнуть проверке...
    } else if (m.usePattern(pImgTag).find()) {
        ...получили тег <img> — можно проверить его корректность...
    } else if (! needClose && m.usePattern(pLink).find()) {
        ...получили ссылку — можно проверить ее корректность...
        needClose = true;
    } else if (needClose && m.usePattern(pLinkX).find()) {
```

```
            System.out.println("/LINK [" + m.group() + "]");
            needClose = false;
        } else if (m.usePattern(pEntity).find()) {
            // Допустимые сущности, такие как &gt; и &#123;
        } else if (m.usePattern(pNonHtml).find()) {
            // Прочие элементы (не слова), не имеющие отношения к HTML --
            // просто допускаем возможность их появления
        } else {
            // Мы получили текст, который не совпал ни с одним регулярным
            // выражением. Скорее всего, это ошибка. Извлечь с дюжину символов
            // из текущей позиции, чтобы сделать сообщение
            // об ошибке более информативным
            m.usePattern(Pattern.compile("\\G(?s).{1,12}")).find();
            System.out.println("Bad char before '" + m.group() + "'");
            System.exit(1);
        }
    }
    if (needClose) {
        System.out.println("Missing Final </A>");
        System.exit(1);
    }
```

В пакете `java.util.regex` присутствует ошибка, приводящая к тому, что при попытке поиска по регулярному выражению *pNonHtml* оно старается «поглотить» символ целевого текста, который с ним не совпадает, поэтому я поместил проверку с этим регулярным выражением в самый конец. От этого ошибка никуда не делась, а обнаружить ее можно по тексту сообщения об ошибке, которое построено с учетом невозможности получить первый символ фрагмента целевого текста, добавляемого в сообщение. Об этой ошибке я уже сообщил в Sun.

Существует ли возможность решить эту задачу с использованием метода `find`, вызываемого с одним аргументом? ❖ Переверните страницу, чтобы получить ответ на этот вопрос.

Разбор данных CSV

Ниже приведена версия примера с разбором данных CSV из главы 6 (☞ 342) для пакета `java.util.regex`. Регулярное выражение было модифицировано: для большей наглядности использованы захватывающие квантификаторы (☞ 190) вместо атомарных круглых скобок.

```
String regex = // Поля в кавычках поместить в группу(1),
               // остальные в группу(2)
    " \\G(?:^|,)                                          \n"+
    " (?:                                                 \n"+
```

МНОЖЕСТВО РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ И МЕТОД FIND, ВЫЗЫВАЕМЫЙ С ОДНИМ АРГУМЕНТОМ

❖ *Ответ на вопрос со с. 499.*

Ошибка в пакете `java.util.regex`, которая была описана на с. 499, вызывает некорректное перемещение указателя «текущей позиции», вследствие чего при следующем вызове метод `find` начинает поиск не с того места. Обойти эту ошибку можно за счет отслеживания «текущей позиции» и использования метода `find` с одним аргументом, чтобы явно указывать позицию, с которой следует начинать поиск совпадения.

Изменения, внесенные в пример со с. 498, выделены жирным шрифтом:

```
Pattern pWord     = Pattern.compile("\\G\\w+");
Pattern pNonHtml  = Pattern.compile("\\G[^\\w<>&]+");
Pattern pImgTag   = Pattern.compile("\\G(?i)<img\\s+([^>]+)>");
Pattern pLink     = Pattern.compile("\\G(?i)<A\\s+([^>]+)>");
Pattern pLinkX    = Pattern.compile("\\G(?i)</A>");
Pattern pEntity   = Pattern.compile("\\G&(#\\d+;\\w+);");
Boolean needClose = false;
Matcher m = pWord.matcher(html); // Наш объект Matcher может быть создан
                                 // любым из существующих объектов Pattern
Integer currentLoc = 0;          // Начинать поиск с начала строки
while (currentLoc < html.length())
{
    if (m.usePattern(pWord).find(currentLoc)) {
        ...получив слово или число в m.group(), его можно
           подвергнуть проверке...
    } else if (m.usePattern(pImgTag).find(currentLoc)) {
        ...получили тег <img> – можно проверить его корректность...
  } else if (! needClose && m.usePattern(pLink).find(currentLoc)) {
        ...получили ссылку – можно проверить ее корректность...
        needClose = true;
    } else if (needClose && m.usePattern(pLinkX).find(currentLoc)) {
        System.out.println("/LINK [" + m.group() + "]");
        needClose = false;
    } else if (m.usePattern(pEntity).find(currentLoc)) {
        // Допустимые сущности, такие как &gt; и &#123;
    } else {
        // Мы получили текст, который не совпал ни с одним
        // регулярным выражением. Скорее всего, это ошибка.
        // Извлечь с дюжину символов из текущей позиции, чтобы
        // сделать сообщение об ошибке более информативным.
        m.usePattern(Pattern.compile("\\G(?s).{1,12}")).find(currentLoc);
        System.out.println("Bad char at '" + m.group() + "'");
```

```
            System.exit(1);
        }
        currentLoc = m.end(); // "Текущая позиция" теперь находится там,
                              // где завершилась предыдущая попытка поиска
    }
    if (needClose) {
        System.out.println("Missing Final </A>");
        System.exit(1);
    }
```

В отличие от предыдущего примера здесь используется версия метода `find`, которая выполняет повторную инициализацию объекта `Matcher`, поэтому данная версия не подходит для случаев, когда необходимо учитывать положение границ области. Однако вы можете самостоятельно добавить поддержку области поиска, вставив соответствующие вызовы метода `region` перед каждым вызовом метода `find`, например:

```
m.usePattern(pWord).region(start,end).find(currentLoc)
```

```
    "         # Любое поле в кавычках...                    \n"+
    "         \" # открывающая кавычка                      \n"+
    "         (     [^\"]*+ (?: \"\" [^\"]*+ )*+ )          \n"+
    "         \" # закрывающая кавычка                      \n"+
    "     | # ...или...                                     \n"+
    "         # любой текст без кавычек и не запятая...    \n"+
    "         ([^\",]*+)                                    \n"+
    "     )                                                 \n";
// Создать объект Matcher для разбора CSV строки текста с помощью
// регулярного выражения, приведенного выше.
Matcher mMain = Pattern.compile(regex, Pattern.COMMENTS).matcher(line);

// Создать объект Matcher для выражения 「""」 с фиктивным текстом.
Matcher mQuote = Pattern.compile("\"\"").matcher("");

while (mMain.find())
{
    String field;
    if (mMain.start(2) >= 0)
        field = mMain.group(2); // Поле не в кавычках можно использовать
                                // как есть
    else
        // Поле в кавычках, необходимо заменить каждую пару кавычек
        // на одну кавычку.
        field = mQuote.reset(mMain.group(1)).replaceAll("\"");

    // Теперь можно обработать поле...
    System.out.println("Field [" + field + "]");
}
```

Такое решение работает эффективнее, чем приведенное на с. 280, по двум причинам; во-первых, само регулярное выражение более эффективно (за подробностями обращайтесь к обсуждению главе 6, с. 271), а во-вторых, в нем многократно используется один объект `Matcher` вместо создания и уничтожения объектов при каждом поиске.

Различия между версиями Java

Как уже говорилось в начале главы, в данной книге рассматривается версия Java 1.5.0. К моменту написания этих строк все еще широко использовалась версия Java 1.4.2 и уже появилась бета-версия Java 1.6 (скорее всего, официальный выпуск состоится в ближайшее время). Принимая это во внимание, я приведу основные отличия, касающиеся работы с регулярными выражениями, между версиями 1.4.2 и 1.5.0 (Update 7) и между 1.5.0 и текущей «сборкой 59g» второй бета-версии Java 1.6.

Различия между 1.4.2 и 1.5.0

Версия Java 1.5.0 отличается от версии Java 1.4.2 в основном реализацией новых методов. Большая часть новых методов была добавлена для поддержки новой концепции области в объекте `Matcher`. Кроме того, была немного усовершенствована поддержка Юникода. Все изменения подробно описываются в следующих двух разделах[1].

Новые методы в Java 1.5.0

Методы поддержки области в объекте `Matcher`, которые отсутствуют в Java 1.4.2:

- `region`
- `regionStart`
- `regionEnd`
- `useAnchoringBounds`
- `hasAnchoringBounds`
- `useTransparentBounds`
- `hasTransparentBounds`

[1] Недокументированная рекурсивная конструкция `(?1)`, имевшаяся в Java 1.4.2, более недоступна в Java 1.5.0. По своему поведению она напоминает аналогичную конструкцию из PCRE (☞ 595), но не была описана в документации к Java.

Прочие методы объекта `Matcher`, которые отсутствуют в Java 1.4.2:

- `toMatchResult`
- `hitEnd`
- `requireEnd`
- `usePattern`
- `toString`

Следующий ниже статический метод также отсутствует в Java 1.4.2:

- `Pattern.quote`

Различия в поддержке Юникода между 1.4.2 и 1.5.0

Ниже приводятся различия между Java 1.4.2 и Java 1.5.0, которые касаются поддержки Юникода:

- В Java 1.5.0 поддерживается Юникод версии 4.0.0, тогда как в Java 1.4.2 — Юникод версии 3.0.0. Это изменение затрагивает многие вопросы, такие как наборы символов (например, в Юникоде 3.0.0 отсутствовали символы с кодовыми пунктами выше `\uFFFF`), их свойства и определения блоков Юникода.
- Был расширен диапазон имен блоков, которые можно использовать в конструкциях `\p{…}` и `\P{…}`. (Перечень блоков и их официальные названия вы найдете в документации к Java, в разделе `Character.UnicodeBlock`.)
- В Java 1.4.2 существовало правило: «не использовать пробелы в формальных именах блоков и префикс `In`». Таким образом, ссылки на блоки выглядели следующим образом: `\p{InHangulJamo}` и `\p{InArabicPresentationForms-A}`.
- В Java 1.5.0 появились дополнительные формы записи имен блоков. Во-первых, префикс '`In`' стал частью официальных имен блоков; таким образом, теперь можно использовать имена `\p{InHangul•Jamo}` и `\p{InArabic•Presentation•Forms-A}`. Во-вторых, включена поддержка префикса '`In`' в Java-идентификаторах блоков (которые представляют собой разновидность официальных названий, где пробелы и дефисы заменяются символами подчеркивания): `\p{InHangul_Jamo}` и `\p{InArabic_Presentation_Forms_A}`.
- В Java 1.5.0 была исправлена неприятная ошибка, имевшаяся в версии 1.4.2, из-за которой приходилось ссылаться на блоки **Arabic Presentation Forms-B** и **Latin Extended-B**, как если бы завершающий символ «`B`» на самом деле был бы словом «`Bound`», т. е. как `\p{InArabicPresentationForms-Bound}` и `\p{InLatinExtended-Bound}`.

- В Java 1.5.0 была добавлена поддержка методов `Character` *`isSomething`* (☞ 461).

Различия между 1.5.0 и 1.6.0

В версии Java 1.6, вышедшей к моменту написания этих строк (вторая бета-версия), существовало всего два незначительных отличия от версии 1.5.0, которые имеют отношение к регулярным выражениям:

- В Java 1.6 появилась поддержка категорий Юникода **Pi** и **Pf**, которая отсутствовала ранее.
- В Java 1.6 была исправлена ошибка в конструкции `\Q…\E`, поэтому теперь ее можно использовать даже внутри символьных классов.

9 .NET

Платформа Microsoft .NET Framework, используемая в Visual Basic, C# и C++ (а также в других языках), содержит общую библиотеку регулярных выражений, обеспечивающую единую семантику операций с регулярными выражениями в разных языках. В этой библиотеке реализован полноценный, мощный механизм, обеспечивающий максимум свободы в выборе баланса между скоростью и удобством работы[1].

Каждый язык обладает собственным синтаксисом операций с объектами и методами, но основные объекты и методы не зависят от языка, поэтому даже сложные конструкции напрямую переводятся на другие языки платформы .NET. В примерах этой главы используется Visual Basic.

О предыдущих главах

Прежде чем представлять читателю содержимое этой главы, я должен подчеркнуть, что она в значительной степени основана на материале глав 1–6. Я понимаю, что некоторые читатели, программирующие только на языках .NET, могут начать сразу с этой главы, но я рекомендую ознакомиться с предисловием (особенно с системой условных обозначений) и предыдущими главами. В главах 1, 2 и 3 представлены многие концепции, возможности и приемы, используемые при работе с регулярными выражениями, а материал глав 4, 5 и 6 представляет ключевую информацию для понимания регулярных выражений, которая напрямую относится к механизму регулярных выражений .NET. Из наиболее важных тем, рассматривавшихся в предыдущих главах, можно упомянуть основы работы механизма регулярных выражений НКА, связанные с поиском совпадений, максимализмом, возвратами и вопросами эффективности.

---

[1] В книге рассматривается .NET версии 2.0 (которая распространяется в составе «Visual Studio 2005»).

Здесь мне хочется заметить, что, несмотря на наличие удобных таблиц, таких как на с. 507 в этой главе или на с. 165 и 169 в главе 3, данная книга не претендует на роль справочного руководства. Главная ее цель — научить вас *искусству* составления регулярных выражений.

Глава начинается с описания диалекта регулярных выражений .NET. Вы узнаете, какие метасимволы в нем поддерживаются и как именно, а также познакомитесь с некоторыми специальными аспектами программирования .NET. Далее приводится краткий обзор объектной модели регулярных выражений .NET, рассматриваются основные принципы работы с регулярными выражениями, после чего подробно описываются все классы, связанные с регулярными выражениями. Глава завершается примером построения личной библиотеки регулярных выражений посредством объединения готовых регулярных выражений в общую сборку (shared assembly).

Диалект регулярных выражений .NET

Поддержка регулярных выражений в .NET использует традиционный механизм НКА, поэтому к ней в полной мере применимы сведения, изложенные в главах 4, 5 и 6. В табл. 9.1 приведена краткая сводка диалекта регулярных выражений .NET. Большинство перечисленных конструкций рассматривается в главе 3.

Некоторые особенности диалекта зависят от *режимов поиска* (☞ 150), активизируемых при помощи флагов функций и конструкторов, получающих регулярные выражения, а в некоторых случаях — включающихся и выключающихся на уровне регулярного выражения при помощи конструкций ⌜`(?`*мод-мод*`)`⌟ и ⌜`(?`*мод-мод*`:…)`⌟. Режимы перечислены в табл. 9.2 на с. 509.

В таблице приводятся «низкоуровневые» экранированные последовательности вида ⌜`\w`⌟. Они могут использоваться непосредственно в строковых литералах VB.NET `("\w")` и в буквально интерпретируемых (verbatim) строках C# `(@"\w")`. В языках без строковых литералов, учитывающих контекст регулярных выражений (например, C++), каждый символ `\` в регулярном выражении представляется двумя символами `\` в строковом литерале `("\\w")`. См. раздел «Строки как регулярные выражения» (☞ 140).

Далее приводятся дополнительные замечания к табл. 9.1:

① `\b` представляет символ Backspace только в символьных классах; за их пределами он представляет границу слова (☞ 180).

Конструкция **`\x##`** может содержать ровно две шестнадцатеричные цифры (например, выражение ⌜**`\xFC`**`ber`⌟ совпадает с '**`über`**').

Конструкция **\u####** может содержать ровно четыре шестнадцатеричные цифры (например, выражение ⌜**\u00FC**ber⌟ совпадает с '**über**', а ⌜\u20AC⌟ совпадает с '**€**').

② В .NET Framework 2.0 поддерживается операция *вычитания* для символьных классов, например ⌜**[a-z-[aeiou]]**⌟ для ASCII-символов нижнего регистра (☞ 170). В символьных классах дефис, за которым следует определение символьного класса, вычитает из класса, слева от дефиса, символы класса, расположенного справа от дефиса.

③ Метасимволы **\w**, **\d** и **\s** (и их антиподы в верхнем регистре) обычно совпадают с полным интервалом соответствующих символов Юникода, но с параметром `RegexOptions.ECMAScript` (☞ 514) совпадение происходит только в кодировке ASCII.

По умолчанию **\w** совпадает со свойствами Юникода \p{Ll}, \p{Lu}, \p{Lt}, \p{Lo}, \p{Nd} и \p{Pc}. Обратите внимание: свойство \p{Lm} в список не входит (список свойств приведен на с. 160).

По умолчанию **\s** совпадает с ⌜[\f\n\r\t\v\x85\p{Z}]⌟. U+0085 — управляющий символ Юникода «следующая строка», а \p{Z} совпадает с символами-«разделителями» в Юникоде (☞ 169).

Таблица 9.1. Общие сведения о диалекте регулярных выражений .NET

Сокращенные обозначения символов ①	
☞ 156 (C)	**\a [\b] \e \f \n\ r \t \v ***восьм* **\x**## **\u**#### **\c***символ*
Символьные классы и аналогичные конструкции	
☞ 161	Обычные классы:② **[…]** и **[^…]**
☞ 162	Почти любой символ: *точка* (значение изменяется в зависимости от режима поиска)
☞ 164 (C)	Сокращенные обозначения классов:③ **\w \d \s \W \D \S**
☞ 164 (C)	Свойства и блоки Юникода:④ **\p{***свойство***} \P{***свойство***}**
Якорные метасимволы и другие проверки с нулевой длиной совпадения	
☞ 175	Начало строки/логической строки: **^ \A**
☞ 175	Конец строки/логической строки: **$ \Z \z**
☞ 177	Конец предыдущего совпадения:⑤ **\G**
☞ 180	Границы слов: **\b \B**
☞ 181	Позиционная проверка: ⑥ **(?=…) (?!…) (?<=…) (?<!…)**

Таблица 9.1 (*окончание*)

Комментарии и модификаторы режимов	
☞ 182	Модификаторы режимов: **(?***мод-мод***)**. Допустимые модификаторы: **x s m i n** (☞ 509)
☞ 183	Интервальное изменение режима: **(?***мод-мод***:…)**
☞ 183	Комментарии: **(?#…)**
Группировка, сохранение, условные и управляющие конструкции	
☞ 184	Сохраняющие круглые скобки:⑦ **(…) \1 \2** …
☞ 523	Сбалансированная группировка: **(?<***имя-имя***>…)**
☞ 510	Именованное сохранение и последующая ссылка: **(?<***имя***>…) \?k<***имя***>**
☞ 185	Группирующие круглые скобки: **(?:…)**
☞ 187	Атомарная группировка: **(?>…)**
☞ 187	Конструкция выбора: **|**
☞ 189	Максимальные квантификаторы: *** + ? {n} {n,} {x,y}**
☞ 190	Минимальные квантификаторы: ***? +? ?? {n}? {n,}? {x,y}?**
☞ 511	Условная конструкция: **(?***if then***|***else***)** — в части *if* может находиться встроенный фрагмент кода, позиционная проверка или (*число*)
(C) — может использоваться в символьных классах. ①...⑦ — см. текст	

④ Конструкции **\p{…}** и **\P{…}** поддерживают большинство стандартных свойств и блоков Юникода версии 4.0.1. Алфавиты не поддерживаются.

При использовании имен блоков требуется префикс 'Is' (подробности в таблице на с. 170) и могут использоваться только низкоуровневые имена без пробелов и символов подчеркивания. Например, имена \p{Is_Greek_Extended} и \p{Is Greek Extended} считаются недопустимыми; следует использовать \p{IsGreekExtended}.

Используются только короткие имена свойств, такие как \p{Lu}; длинные конструкции вида \p{Lowercase_Letter} не поддерживаются. Однобуквенные свойства *обязательно* должны записываться в фигурных скобках (т. е. сокращенная запись \pL для \p{L} не поддерживается) (подробности в таблицах на с. 165 и 166).

Специальная композитная комбинация \p{L&}, а также комбинации \p{All}, \p{Assigned} и \p{Unassigned} не поддерживают специальные свойства, вместо них следует использовать ⌜(?s:…)⌟, ⌜\P{Cn}⌟ и ⌜\p{Cn}⌟ соответственно.

⑤ Метасимвол **\G** совпадает в конце *предыдущего* совпадения, хотя в документации утверждается, что он совпадает в *начале текущей* попытки поиска (☞ 177).

⑥ При опережающей *и* ретроспективной проверке могут использоваться любые регулярные выражения. На момент написания книги механизм регулярных выражений .NET был единственным из известных мне механизмов, в которых условие ретроспективной проверки могло совпадать с текстом произвольной длины (☞ 181).

⑦ Параметр `RegexOptions.ExplicitCapture`, также доступный при помощи модификатора режима **(?n)**, отключает сохранение для обычных круглых скобок ⌜`(…)`⌟. Сохранение с явно заданным именем (типа ⌜**`(?<num>\d+)`**⌟) при этом продолжает работать (☞ 180). При использовании именованных сохранений этот параметр позволяет применять при группировке более простые конструкции ⌜`(…)`⌟ вместо ⌜`(?:…)`⌟.

Таблица 9.2. Модификаторы режимов в .NET

Параметр RegexOptions	(? режим)	Описание
`.Singleline`	`S`	Разрешает совпадение *точки* с любым символом (☞ 152)
`.Multiline`	`m`	Расширяет возможности для совпадения ⌜`^`⌟ и ⌜`$`⌟ (☞ 153)
`.IgnorePatternWhitespace`	`x`	Режим свободного форматирования (☞ 106).
`.IgnoreCase`	`i`	Поиск без учета регистра для ASCII-символов
`.ExplicitCapture`	`n`	Запрет сохранения текста для ⌜`(…)`⌟, чтобы сохранение производилось только конструкцией ⌜`(?<имя>)`⌟
`.ECMAScript`		Метасимволы ⌜`\w`⌟, ⌜`\d`⌟ и ⌜`\s`⌟ совпадают только с ASCII-символами; кроме того, есть и другие эффекты (☞ 514).
`.RightToLeft`		Регулярное выражение применяется нормальным образом, но в противоположном направлении (т. е. смещение происходит от конца строки к началу). К сожалению, режим реализован с ошибками (☞ 513)
`.Compiled`		Выражение проходит предварительную оптимизацию, что ускоряет поиск (☞ 512)

Замечания по поводу диалекта .NET

Некоторые проблемы не помещаются в одном абзаце и заслуживают более подробного обсуждения.

Именованное сохранение

.NET поддерживает возможность именованного сохранения (☞ 186) в конструкциях ⌜(?<*имя*>…)⌟ и ⌜(?'*имя*'…)⌟. Два варианта синтаксиса эквивалентны; вы можете использовать любой из них, однако я предпочитаю вариант с <…> — мне кажется, он получит более широкое распространение.

Для ссылок на сохраненный текст в регулярном выражении используются конструкции ⌜\k<*имя*>⌟ и ⌜\k'*имя*'⌟.

После совпадения (а точнее, после создания объекта `Match` — обзор объектной модели .NET приведен на с. 519) для обращения к результату именованного сохранения может использоваться свойство `Groups(`*имя*`)` объекта `Match` (в C# используется синтаксис `Groups[`*имя*`]`).

В строке замены (☞ 528) для обращения к результату именованного сохранения используется синтаксис `${`*имя*`}`.

Чтобы ко всем группам можно было обращаться по порядковым номерам, что иногда бывает удобно, конструкциям именованного сохранения также присваиваются номера. Нумерация производится *после* того, как будут пронумерованы обычные круглые скобки:

```
1  1  3        3  2   2
⌜(\w) (?<Num>\d+) (\s+)⌟
```

Для обращения к тексту, совпавшему с подвыражением ⌜\d+⌟, могут использоваться конструкции `Groups("Num")` или `Groups(3)`. Это два варианта ссылки на одну и ту же группу (по имени и по номеру).

Нежелательные последствия

Смешивать в выражении обычные сохраняющие круглые скобки с именованными сохранениями не рекомендуется. Если вы все же поступите подобным образом, необходимо учитывать некоторые важные последствия способа нумерации сохраняющих конструкций. Порядок нумерации играет важную роль при вызове `Split` с сохраняющими круглыми скобками (☞ 530) и при интерпретации '`$+`'в строке замены (☞ 528).

Условные проверки

Часть *if* условной конструкции ⌜(?*if then*|*else*)⌟ (☞ 188) может содержать любые разновидности позиционных проверок, а также ссылки на сохраняющие группы (по имени или по порядковому номеру). Простой текст (или простое регулярное выражение) в этой части автоматически интерпретируется как позитивная опережающая проверка (т. е. неявно заключается в ⌜(?=…)⌟). В результате могут возникать неоднозначные ситуации: например, если в регулярном выражении не встречается именованное сохранение ⌜(?<Num>…)⌟, то подвыражение ⌜(Num)⌟ в ⌜…(?(Num) *then* | *else*)…⌟ будет преобразовано в ⌜(?=Num)⌟, т. е. опережающую проверку 'Num'). Если такое именованное сохранение существует, то результат *if* определяется ее успешностью.

Я не советую полагаться на «автоматическую опережающую проверку». Явные конструкции ⌜(?=…)⌟ делают ваши намерения более понятными для читателя программы, а также предотвратят неприятные сюрпризы, если в будущих версиях механизма регулярных выражений синтаксис *if* изменится.

Псевдокомпиляция

В предыдущих главах под термином «компиляция» понималась операция, выполняемая перед применением регулярного выражения, в ходе которой система проверяет синтаксис выражения и преобразует его во внутреннюю форму, пригодную для применения к тексту. В .NET этот процесс называется «разбором» (parsing), а две разновидности «компиляции» относятся к разным оптимизациям, выполняемым на стадии разбора.

Ниже приводится подробное описание порядка оптимизации.

- **Разбор.** Когда регулярное выражение впервые встречается в процессе выполнения программы, оно проверяется и преобразуется во внутреннюю форму, используемую механизмом регулярных выражений. В остальных главах книги этот процесс называется «компиляцией» (☞ 307).

- **Оперативная компиляция.** Параметр `RegexOptions.Compiled` входит в число параметров, используемых при построении регулярных выражений. Он означает, что механизм регулярных выражений не ограничивается простым преобразованием выражения в стандартную внутреннюю форму, но *компилирует* его в низкоуровневый код MSIL (Microsoft Intermediate Language), который, в свою очередь, преобразуется в еще более быстрый машинный код JIT-компилятором при фактическом применении регулярного выражения.

Такой подход требует дополнительных затрат памяти и времени, но полученное регулярное выражение работает быстрее. Преимущества и недостатки оперативной компиляции рассматриваются ниже в этом разделе.

- **Предварительная компиляция регулярных выражений.** Объект (или объекты) `Regex` может быть преобразован в сборку, записанную на диск в виде DLL-библиотеки. В дальнейшем содержимое библиотеки может использоваться другими программами; это называется «компиляцией сборки». Дополнительная информация приведена в разделе «Сборки регулярных выражений» (☞ 540).

При оценке оперативной компиляции с параметром `RegexOptions.Compiled` следует учитывать такие важные факторы, как время начальной инициализации, затраты памяти и ускорение поиска:

Характеристика	Без параметра RegexOptions.Compiled	С параметром RegexOptions.Compiled
Инициализация	Быстрее	Медленнее (примерно в 60 раз)
Затраты памяти	Низкие	Высокие (5–15 Кбайт на каждое выражение)
Скорость поиска	Невысокая	До 10 раз быстрее

Разбор регулярного выражения (выполняемый по умолчанию, без параметра `RegexOptions.Compiled`) происходит относительно быстро. Даже мой убогий старый «ящик» с 550-мегагерцовым процессором за секунду компилирует свыше 1500 сложных выражений. С параметром `RegexOptions.Compiled` производительность падает до 25 выражений в секунду, а затраты памяти увеличиваются примерно на 10 Кбайт для каждого регулярного выражения. Еще важнее то, что эта память не освобождается и продолжает использоваться на протяжении всего жизненного цикла программы.

Можно с уверенностью заявить, что применение параметра `RegexOptions.Compiled` обоснованно в критических ситуациях, когда крайне важна скорость выполнения, особенно если выражение работает с большим объемом текста. С другой стороны, нет смысла применять этот параметр к простым выражениями, которые работают с небольшими объемами текста. Для промежуточных ситуаций однозначных рекомендаций не существует — вам придется самостоятельно взвесить «за» и «против» и принять решение для каждого конкретного случая.

Иногда откомпилированные выражения разумно сохранить в отдельной DLL-библиотеке в виде предварительно откомпилированных объектов `Regex`. Тем самым снижаются затраты памяти, поскольку программе не приходится загружать весь

пакет компиляции регулярных выражений, а также ускоряется загрузка (выражения компилируются при построении DLL, и вам не придется ожидать их компиляции при использовании). Другое побочное преимущество заключается в том, что откомпилированные выражения могут использоваться другими программами и перед вами открывается отличная возможность для построения личной библиотеки регулярных выражений, о чем говорится в разделе «Построение библиотеки регулярных выражений» на с. 542.

Поиск справа налево

Концепция «обратного» поиска (т. е. поиска справа налево, а не слева направо) давно будоражила умы разработчиков регулярных выражений. Возможно, самые большие трудности связаны с определением того, что же понимается под поиском «справа налево». Изменяется ли регулярное выражение? Или символы целевого текста переставляются в обратном порядке? А может, регулярное выражение применяется обычным образом от каждой позиции целевого текста, но в начале поиска текущая позиция устанавливается в конец текста вместо его начала и отходит назад при каждом смещении?

Для определенности рассмотрим применение выражения ⌜`\d+`⌟ к строке '`123•and•456`'. Мы знаем, что в обычном случае выражение совпадает с подстрокой '`123`', а интуиция подсказывает, что при обратном поиске оно должно совпасть с '`456`'. Тем не менее при использовании семантик, перечисленных в предыдущем абзаце, с отличиями только в выборе начальной позиции и направлении смещения результаты оказываются довольно странными. В этих семантиках механизм работает обычным образом (просто он «смотрит» на текст справа налево), поэтому при первом применении ⌜`\d+`⌟ в строке '`…456`▲' совпадение не находится. Но при второй попытке, '`…45`▲`6`', совпадение успешно находится — механизм «видит» цифру '`6`', которая несомненно совпадает с ⌜`\d+`⌟. Итак, итоговое совпадение состоит из единственной цифры '`6`'.

Выше упоминался параметр `RegexOptions.RightToLeft` библиотеки регулярных выражений .NET. Какую семантику он использует? Хороший вопрос... Его семантика не документирована, а в результате проведения тестов мне так и не удалось ее вычислить. Во многих случаях (как в примере '`123•and•456`') параметр работает на удивление предсказуемо и находит совпадение '`456`'. Тем не менее в одних случаях он *вообще* не находит совпадения, а в других — находит совпадение, которое на фоне других результатов выглядит совершенно нелогично.

Не исключено, что в вашей ситуации `RegexOptions.RightToLeft` будет работать именно так, как нужно, но в конечном счете вы используете этот режим на собственный страх и риск.

Неоднозначная интерпретация комбинаций «\+цифра»

Символ \, за которым следует цифра, может интерпретироваться либо как число в восьмеричной записи, либо как обратная ссылка. Выбор и правила интерпретации зависят от того, был ли задан параметр `Regex Options.ECMAScript`. Если вы не хотите разбираться в столь тонких различиях, всегда используйте для обратных ссылок обозначение ⌜`\k<`*число*`>`⌟, а восьмеричные коды начинайте с нуля (например, ⌜`\08`⌟). Такая схема всегда работает одинаково, независимо от наличия или отсутствия параметра `RegexOptions.ECMAScript`.

Если параметр `RegexOptions.ECMAScript` не используется, экранированные последовательности из одной цифры ⌜`\1`⌟ — ⌜`\9`⌟ всегда интерпретируются как обратные ссылки, а экранированная последовательность цифр, начинающаяся с нуля, всегда интерпретируется как восьмеричный код (например, ⌜`\012`⌟ всегда совпадает с ASCII-символом перевода строки). В остальных случаях число интерпретируется как обратная ссылка, если подобная интерпретация «осмысленна» (т. е. если число не больше количества сохраняющих круглых скобок в регулярном выражении). В других случаях, если число находится в интервале от `\000` до `\377`, оно интерпретируется как восьмеричный код. Например, ⌜`\12`⌟ воспринимается как обратная ссылка, если выражение содержит минимум 12 пар сохраняющих круглых скобок, и как восьмеричный код в противном случае.

Определение и описание семантики параметра `RegexOptions.ECMAScript` даются в следующем разделе.

Режим ECMAScript

ECMAScript — стандартизированная версия JavaScript[1], обладающая собственной семантикой разбора и применения регулярных выражений. Регулярные выражения .NET, созданные с параметром `RegexOptions.ECMAScript`, пытаются имитировать эту семантику. Если вы не знаете, что такое ECMAScript, или проблемы совместимости вас не интересуют, этот раздел можно смело пропустить.

Что происходит при использовании параметра `RegexOptions`?

- С `RegexOptions.ECMAScript` могут комбинироваться только следующие параметры:

```
RegexOptlons.IgnoreCase;
RegexOptions.Multiline;
RegexOptions.Compiled.
```

[1] Сокращение ECMA означает «European Computer Manufacturers Association», т. е. «Ассоциация европейских производителей компьютеров». Группа ECMA была создана в 60-е годы для стандартизации развивающейся компьютерной отрасли.

- Метасимволы \w, \d и \s (а также \W, \D и \S) переключаются в режим совпадения только с ASCII-символами.
- Когда в выражении встречается последовательность «\+цифра», при выборе между обратной ссылкой и восьмеричным кодом предпочтение всегда отдается обратной ссылке, даже если из-за этого приходится игнорировать некоторые из следующих цифр. Например, в конструкции ⌜(…)\10⌟ подвыражение ⌜\10⌟ воспринимается как ссылка на первую группу, за которой следует литерал '0'.

Использование регулярных выражений в .NET

Регулярные выражения .NET обладают богатыми возможностями, они логичны, а для работы с ними предусмотрен полный и удобный интерфейс. Программисты Microsoft отлично справились с созданием пакета, но с документацией все наоборот — она просто ужасна. Документация недостаточно полна, плохо написана, хаотично организована, а иногда попросту неверна. Мне пришлось довольно долго разбираться, как же на самом деле работает пакет. Надеюсь, что материал этой главы поможет вам начать работу с регулярными выражениями в .NET.

Основные принципы работы с регулярными выражениями

В простейших случаях вы можете сразу приступить к работе с пакетом регулярных выражений .NET, не разбираясь в его объектной модели. Технические подробности помогут вам быстрее получить нужную информацию, но приведенные ниже примеры доказывают, что простейшие операции могут выполняться без явного создания экземпляров каких-либо классов. Помните, что это лишь примеры; дополнительная информация будет приведена ниже.

В начале любой программы, использующей библиотеку регулярных выражений (☞ 518), должна находиться директива

```
Imports System.Text.RegularExpressions
```

Предполагается, что в следующих примерах эта директива присутствует.

Все приводимые далее примеры работают с текстом, хранящимся в переменной TestStr класса String. Как и прежде, выбранные мной имена выделяются курсивом.

Проверка совпадения

Первый пример просто проверяет наличие совпадения в строке:

```
If Regex.IsMatch (TestStr, "^\s*$")
    Console.WriteLine ("line is empty")
Else
    Console.WriteLine ("line is not empty")
End If
```

Аналогичный пример с использованием параметра режима:

```
If Regex.IsMatch (TestStr, "^subject:", RegexOptions.IgnoreCase)
    Console.WriteLine ("line is a subject line")
Else
    Console.WriteLine ("line is not a subject line")
End If
```

Поиск и получение совпавшего текста

В следующем примере идентифицируется текст, совпавший с регулярным выражением. Если совпадение отсутствует, переменной `TheNum` присваивается пустая строка.

```
Dim TheNum as String = Regex.Match (TestStr, "\d+").Value
If TheNum <> ""
    Console.WriteLine("Number is: " & TheNum}
End If
```

Аналогичный пример с использованием параметра режима:

```
Dim ImgTag as String = Regex.Match(TestStr, "<img\b[^>]*>", _
                                   RegexOptions.IgnoreCase).Value
If ImgTag <> ""
    Console.WriteLine("Image tag:" & ImgTag)
End If
```

Поиск и получение сохраненного текста

В следующем примере текст первой сохраняющей группы (т. е. **$1**) присваивается строковой переменной:

```
Dim Subject as String = _
    Regex.Match(TestStr, "^Subject: (.*)").Groups(1).Value
If Subject <> ""
    Console.WriteLine("Subject is:" & Subject)
End If
```

В C# вместо `Groups(1)` используется запись `Groups[1]`.

То же самое с использованием параметра режима:

```
Dim Subject as String = _
    Regex.Match(TestStr, "^subject: (.*)", _
                RegexOptions.IgnoreCase).Groups(1).Value
If Subject <> ""
    Console.WriteLine("Subject is: " & Subject)
End If
```

То же самое, но с использованием именованного сохранения:

```
Dim Subject as String = _
    Regex.Match(TestStr, "^subject: (?<Subj>.*)", _
                RegexOptions.IgnoreCase).Groups("Subj").Value
If Subject <> ""
    Console.WriteLine("Subject is: " & Subject)
End If
```

Поиск с заменой

Следующий пример обеспечивает «безопасность» включения тестовой строки в код HTML, для чего специальные символы HTML заменяются своими подстановочными эквивалентами:

```
TestStr = Regex.Replace(TestStr, "&", "&amp;")
TestStr = Regex.Replace(TestStr, "<", "&lt;")
TestStr = Regex.Replace(TestStr, ">". "&gt;")
Console.WriteLine("Now safe in HTML: " & TestStr)
```

Строка замены (третий аргумент) интерпретируется особым образом, как описано на с. 528. Например, последовательность '$&' заменяется текстом, фактически совпавшим с регулярным выражением. Следующая команда заключает слова, начинающиеся с прописной буквы, в теги <B>…</B>:

```
TestStr = Regex.Replace(TestStr, "\b[A-Z]\w*", "<B>$&</B>")
Console.WriteLine("Modified string: " & TestStr)
```

В другом примере пары тегов <B>…</B> (без учета регистра символов) заменяются на <I>…</I>:

```
TestStr = Regex.Replace(TestStr, "<b>(.*?)</b>", "<I>$1</I>", _
                        RegexOptlons.IgnoreCase)
Console.WriteLine("Modified string: " & TestStr)
```

Общие сведения о пакете

В полной мере все возможности пакета регулярных выражений .NET раскрываются лишь при использовании его богатой и удобной иерархии классов. Чтобы вы получили некоторое представление об этих классах, ниже приводится пример консольного приложения, в котором простой поиск проводится при помощи специально создаваемых объектов:

```
Option Explicit On ' Директивы не обязательны для работы с регулярными
Option Strict On   ' выражениями, но их рекомендуется использовать всегда.
' Получение доступа к классам, используемым при работе
' с регулярными выражениями.
Imports System.Text.RegularExpressions

Module SimpleTest
Sub Main()
    Dim SampleText as String = "this is the 1st test string"
    Dim R as Regex = New Regex("\d+\w+") ' Компиляция выражения.
    Dim M as Match = R.match(SampleText) ' Поиск совпадения в строке.
    If not M.Success
        Console.WriteLine("no match")
    Else
        Dim MatchedText as String = M.Value ' Получение результатов...
        Dim MatchedFrom as Integer = M.Index
        Dim MatchedLen as Integer = M.Length
        Console.WriteLine("matched [" & MatchedText & "]" & _
                          " from char#" & MatchedFrom.ToString() & _
                          " for " & MatchedLen.ToString() & " chars.")
    End If
End Sub
End Module
```

При запуске в командной строке приложение ищет в тексте совпадение для выражения `\d+\w+` и выводит следующий результат:

```
matched [1st] from char#12 for 3 chars.
```

Импортирование пространства имен

Вы заметили директиву `Imports.System.Text.RegularExpressions` в начале программы? Она должна присутствовать в любой программе VB, работающей с регулярными выражениями, чтобы сделать их доступными для компилятора.

Аналогичная команда C# выглядит так:

```
using System.Text.RegularExpressions; // Для языка C#
```

Приведенный пример демонстрирует применение двух объектов. Две главные строки:

```
Dim R as Regex = New Regex("\d+\w+") ' Компиляция выражения.
Dim M as Match = R.match(SampleText) ' Поиск совпадения в строке.
```

можно объединить в одну строку:

```
Dim M as Match = Regex.Match(SampleText, "\d+\w+") ' Поиск совпадения
                                                   ' по заданному шаблону.
```

С объединенной версией проще работать, поскольку при этом сокращается объем вводимого текста, а программисту приходится следить за меньшим количеством объектов. Тем не менее это приводит к некоторому снижению эффективности (☞ 538). На нескольких ближайших страницах вы сначала познакомитесь с основными классами, после чего мы рассмотрим «вспомогательные» функции вроде статической функции `Regex.Match` и выясним, когда их следует использовать.

Для экономии места в те примеры, которые не являются полными программами, не включаются строки:

```
Option Explicit On
Option Strict On
Imports System.Text.RegularExpressions
```

Также будет полезно рассмотреть примеры VB, приводившиеся ранее на с. 134, 137, 264, 282 и 301.

Краткая сводка основных объектов

Прежде чем углубляться в детали, рассмотрим общую структуру объектной модели регулярных выражений .NET. Объектная модель представляет собой совокупность классов, через которую пользователь получает доступ к функциональности регулярных выражений. В .NET эту функциональность обеспечивают семь тесно взаимодействующих классов, но на практике обычно бывает достаточно трех классов, представленных на рис. 9.1. На рисунке схематично изображен процесс многократного поиска совпадений `⌜\s+(\d+)⌟` в строке '`May•16, •1998`'.

Объект Regex

Прежде всего необходимо создать объект `Regex`:

```
Dim R as Regex = New Regex("\s+(\d+)")
```

Приведенная команда создает объект, представляющий регулярное выражение `\s+(\d+)`, и сохраняет его в переменной `R`. После создания объекта `Regex` его можно применить к тексту методом `Match(`*текст*`)`, возвращающим информацию о первом найденном совпадении:

```
Dim M as Match = R.Match("May 16, 1998")
```

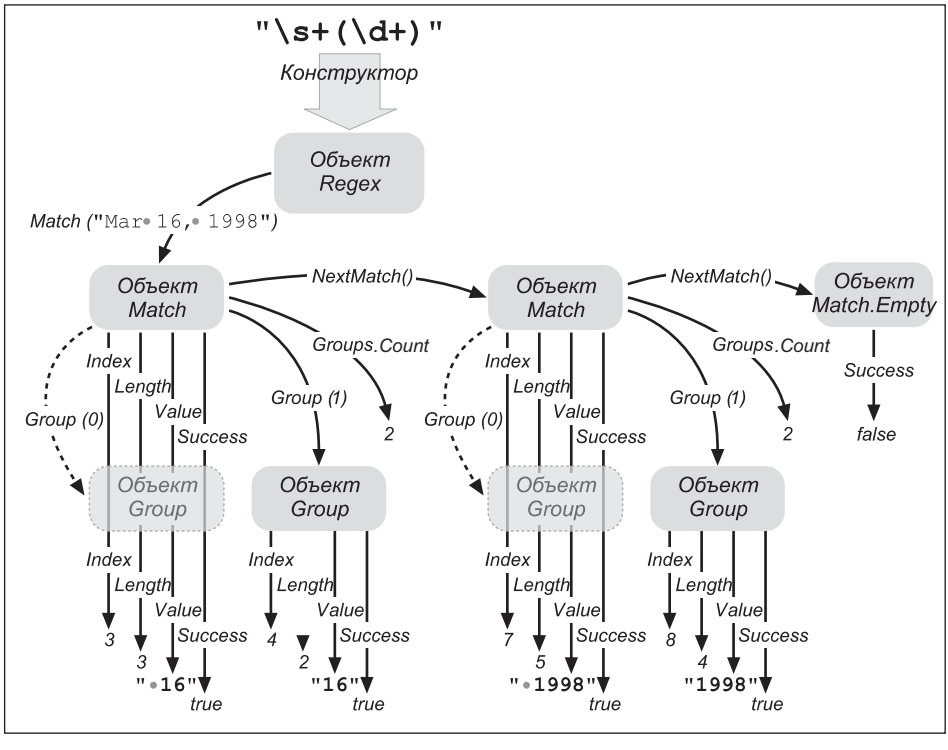


Рис. 9.1. Объектная модель регулярных выражений .NET

Объект Match

Метод `Match(...)` объекта `Regex` предоставляет информацию о результате поиска и возвращает объект `Match`. Объект `Match` обладает рядом полезных свойств, в том числе `Success` (логический признак успешного совпадения) и `Value` (копия совпавшего текста при успешном поиске). Полный список свойств объекта `Match` приводится ниже.

Объект `Match` также позволяет получить информацию о тексте, совпавшем с сохраняющими круглыми скобками. В приводимых ранее примерах на языке Perl текст, совпавший с первой парой сохраняющих круглых скобок, идентифицировался переменной `$1`. В .NET существует два способа получения этих данных. Первый

способ основан на индексации свойства Groups объекта Match: например, Groups(1).Value является аналогом переменной Perl $1 (обратите внимание: в C# используется другой синтаксис, Groups[1].Value). Другое решение основано на использовании метода Result (который будет рассматриваться на с. 538).

Объект Group

Приведенное выше выражение Groups(1) в действительности ссылается на объект Group, а уточнение .Value *относится* к свойству Value этого объекта (т. е. тексту, ассоциированному с группой). Каждая пара сохраняющих круглых скобок представляется своим объектом Group; также существует «виртуальная группа» с нулевым номером, содержащая информацию обо всем совпадении.

Таким образом, *MatchObj*.Value и *MatchObj*.Groups(0) эквивалентны — оба выражения представляют текст всего совпадения. Первый, более короткий вариант отличается большей наглядностью и удобством, но о нулевой группе также нельзя забывать, потому что она учитывается в значении *MatchObj*.Groups.Count (количестве групп, информация о которых хранится в объекте Match). При успешном совпадении выражения ⌜\s+(\d+)⌟ значение *MatchObj*.Groups.Count равно двум («нулевая» группа для всего совпадения и $1).

Объект Capture

Объект Capture на практике почти не используется. Дополнительная информация о нем приведена на с. 545.

Все результаты вычисляются во время поиска

Все результаты поиска совпадений регулярного выражения в строке (общее совпадение, совпадения всех сохраняющих групп и т. д.) инкапсулируются в объекте Match в момент применения выражения. Свойства и методы объекта Match, в том числе его объекты Group (а также их свойства и методы), попросту извлекают уже вычисленные данные.

Основные объекты

От общих сведений о классах мы переходим к их детальному рассмотрению. Вы узнаете, как создать объект Regex, как применить его к строке для получения объекта Match и как работать с этим объектом и его объектами Group.

На практике часто удается обойтись без специального создания объекта Regex, но полезно представлять себе, как эти объекты работают, поэтому при описании

основных объектов регулярных выражений .NET я всегда буду явно создавать объекты Regex. Позднее мы рассмотрим более короткий и более удобный способ, который предоставляет .NET.

В последующие списки не включены редко используемые методы, унаследованные от класса Object.

Создание объектов Regex

Объекты Regex создаются простым конструктором, который вызывается либо с одним аргументом (регулярное выражение в виде строки), либо с двумя аргументами (регулярное выражение и набор параметров). Пример вызова с одним аргументом:

```
Dim StripTrailWS = new Regex("\s+$") ' Удаление завершающих пропусков
```

Команда создает объект Regex и готовит его к дальнейшему использованию; поиск на этой стадии еще не производится.

Пример с двумя аргументами:

```
Dim GetSubject = new Regex("^subject: (.*)", RegexOptions.IgnoreCase)
```

В данном примере передается один флаг RegexOptions, но существует возможность передавать сразу несколько флагов, объединяя их операцией OR:

```
Dim GetSubject = new Regex("^subject: (.*)", _
                           RegexOptions.IgnoreCase OR RegexOptions.Multiline)
```

Обработка исключений

При недопустимой комбинации метасимволов в регулярном выражении инициируется исключение ArgumentException. Если вы твердо уверены в том, что регулярное выражение работает, перехватывать исключение не нужно, но очень важно организовать его обработку при использовании «внешних» выражений (например, введенных пользователем или прочитанных из конфигурационного файла). Пример:

```
Dim R As Regex
Try
    R = New Regex(SearchRegex)
Catch e As ArgumentException
    Console.WriteLine("*ERROR* bad regex: " & e.ToStrlng)
    Exit Sub
End Try
```

Конечно, в реальном приложении вместо вывода информации на консоль можно выполнить другие действия.

Параметры конструктора Regex

При создании объекта `Regex` могут указываться параметры:

`RegexOptions.IgnoreCase`

Параметр означает, что регулярное выражение должно применяться без учета регистра символов (☞ 151).

`RegexOptions.IgnorePatternWhitespace`

Регулярное выражение должно обрабатываться в режиме свободного форматирования (☞ 151). Если вы используете комментарии `#...`, каждая логическая строка должна завершаться символом новой строки, в противном случае первый комментарий распространяется до конца регулярного выражения.

В VB.NET для включения символа новой строки может использоваться символ `chr(10)`, как показано ниже:

```
Dim R as Regex = New Regex( _
    "# Match a floating-point number ...             " & chr(10) & _
    "  \d+(?:\.\d*)? # with a leading digit...       " & chr(10) & _
    "  |             # or ...                         " & chr(10) & _
    "  \.\d+         # with a leading decimal point  ", _
    RegexOptions.IgnorePatternWhitespace)
```

Запись получается слишком громоздкой; в VB.NET удобнее использовать комментарии `(?#...)`.

```
Dim R as Regex = New Regex( _
    "(?# Поиск вещественного числа...              )" & _
    " \d+(?:\.\d*)? (?# начинается с цифры...      )" & _
    " |             (?# или...                       )" & _
    " \.\d+         (?# с десятичной точки         ) ", _
RegexOptions.IgnorePatternWhitespace)
```

`RegexOptions.Multiline`

Параметр означает, что регулярное выражение должно применяться в расширенном режиме привязки к границам строки. В этом режиме метасимволы `^` и `$` могут совпадать в начале и в конце логических строк, а не только на границах целевого текста.

`RegexOptions.Singleline`

Параметр означает, что регулярное выражение должно применяться в режиме совпадения точки со всеми символами (☞ 152). По умолчанию *точка* совпадает с любым символом, кроме символа новой строки.

`RegexOptions.ExplicitCapture`

В этом режиме круглые скобки `(…)`, которые обычно сохраняют совпавший текст, ведут себя как группирующие (несохраняющие) круглые скобки `(?:…)`. Допускается только именованное сохранение с использованием конструкции `(?<имя>…)`. Если вы используете именованное сохранение и хотите группировать текст при помощи несохраняющих круглых скобок, параметр `RegexOptions.ExplicitCapture` сделает регулярное выражение менее громоздким.

`RegexOptions.RightToLeft`

Регулярное выражение применяется в режиме поиска справа налево (☞ 513).

`RegexOptions.Compiled`

Регулярное выражение компилируется в специальный высокооптимизированный формат, что обычно приводит к ускорению поиска. С другой стороны, компиляция замедляет первоначальную обработку регулярного выражения и увеличивает затраты памяти в течение его применения.

Если регулярное выражение используется однократно (или небольшое количество раз), использование параметра `RegexOptions.Compiled` неоправданно, поскольку выделенная память не освобождается после уничтожения объекта `Regex`. Но если регулярное выражение применяется в ситуации, критичной по времени, компиляция выражения может принести пользу.

В примере, приведенном на с. 301, компиляция выражения почти вдвое ускорила выполнение теста. Возможность компиляции в сборку рассматривается на с. 540.

`RegexOptions.ECMAScript`

Лексический разбор регулярного выражения производится по правилам совместимости с ECMAScript (☞ 514). Если вы не знаете, что такое ECMAScript или совместимость вас не интересует, не используйте этот параметр.

`RegexOptions.None`

Обозначение «пустого набора параметров»; в случае необходимости может использоваться для инициализации переменной `RegexOptions`. Если позднее потребуется добавить другие параметры, объедините их с существующим значением операцией `OR`.

Использование объектов Regex

Объект регулярного выражения приносит пользу лишь после того, как он будет применен для поиска в тексте. Ниже перечислены методы объекта `Regex`, предназначенные для выполнения различных операций поиска.

`RegexObj``.IsMatch(`*`текст`*`)`
`RegexObj``.IsMatch(`*`текст, смещение`*`)`

Тип возвращаемого значения: **Boolean**

Метод `IsMatch` применяет регулярное выражение, представленное объектом, к заданному *тексту* и возвращает логический признак успешного совпадения. Пример:

```
Dim R as RegexObj = New Regex("^\s*$")
    ⋮
If R.IsMatch(Line) Then
    ' Пустая строка ...
    ⋮
Endif
```

Если при вызове `IsMatch` указано целочисленное *смещение*, то регулярное выражение применяется после заданного количества символов от от начала строки.

`RegexObj``.Match(`*`текст`*`)`
`RegexObj``.Match(`*`текст, смещение`*`)`
`RegexObj``.Match(`*`текст, смещение, макс_длина`*`)`

Тип возвращаемого значения: объект **Match**

Метод `Match` применяет регулярное выражение, представленное объектом, к заданному *тексту* и возвращает объект `Match`. Методы объекта `Match` позволяют запросить информацию о результатах поиска (было ли найдено совпадение, какой текст совпал и т. д.) и инициировать «следующее» применение того же регулярного выражения в строке. Дополнительная информация об объекте `Match` приведена на с. 534.

Если при вызове `Match` указано целочисленное *смещение*, то регулярное выражение применяется после заданного количества символов от начала строки.

Если задан аргумент *макс_длина*, поиск производится в особом режиме — с точки зрения механизма регулярных выражений целевой *текст* состоит из интервала заданной длины, начинающегося с указанного *смещения*. Считается, что символы за пределами этого интервала вообще не существуют, поэтому метасимвол ⌜`^`⌟ совпадает в начале интервала, а ⌜`$`⌟ — в его конце. Кроме того, ретроспективная проверка «не видит» символы за пределами интервала. Такое поведение принципиально отличается от вызова `Match` с заданием только *смещения*, поскольку в этом случае изменяется лишь начальная позиция применения регулярного выражения — механизм по-прежнему «видит» весь целевой текст.

В следующей таблице приведены примеры, поясняющие смысл аргументов *смещение* и *макс_длина*.

Вызов метода	**Результаты для RegexObj, созданного с выражением**		
	⌜d\d⌟	⌜^\d\d⌟	⌜^\d\d$⌟
RegexObj.**Matches(**«May 16, 1998»**)**	соответствует ‘16'	нет соответствия	нет соответствия
RegexObj.**Matches(**«May 16, 1998», 9**)**	соответствует ‘99'	нет соответствия	нет соответствия
RegexObj.**Matches(**«May 16, 1998», 9, 2**)**	соответствует ‘99'	соответствует ‘99'	соответствует ‘99'

`RegexObj.Matches(текст)`
`RegexObj.Matches(текст, смещение)`

Тип возвращаемого значения: объект **MatchCollection**

Метод `Matches` аналогичен методу `Match`, но он возвращает коллекцию объектов `Match`, представляющих *все* совпадения в целевом *тексте*, вместо одного объекта `Match`, представляющего *первое* совпадение. Возвращаемый объект относится к классу `MatchCollection`.

Предположим, была выполнена следующая инициализация:

```
Dim R as New Regex("\w+")
Dim Target as String = "a few words"
```

В этом случае фрагмент

```
Dim BunchOfMatches as MatchCollection = R.Matches (Target)
Dim I as Integer
For I = 0 to BunchOfMatches.Count - 1
    Dim MatchObj as Match = BunchOfMatches.Item(I)
    Console.WriteLine("Match: " & MatchObj.Value)
Next
```

выведет следующий результат:

```
Match: a
Match: few
Match: words
```

Как показывает следующий пример, выводящий тот же результат, вы можете обойтись без отдельной переменной `MatchCollection`:

```
Dim MatchObj as Match
For Each MatchObj in R.Matches(Target)
    Console.WriteLine("Match: " & MatchObj.Value)
Next
```

Наконец, для сравнения будет показано, как добиться той же цели с использованием метода `Match` (вместо метода `Matches`):

```
Dim MatchObj as Match = R.Match(Target)
While MatchObj.Success
    Console.WriteLine("Match: " & MatchObj.Value)
    MatchObj = MatchObj.NextMatch()
End While
```

```
RegexObj.Replace(текст, замена)
RegexObj.Replace(текст, замена, количество)
RegexObj.Replace(текст, замена, количество, смещение)
```

Тип возвращаемого значения: **String**

Метод `Replace` выполняет поиск с заменой в целевом *тексте* и возвращает его копию (вероятно, измененную). Он применяет регулярное выражение, представленное объектом `Regex`, но вместо возвращения объекта `Match` производится замена совпавшего текста. Новый текст определяется аргументом *замена*. Этот аргумент перегружен, в нем может передаваться строка или делегат `MatchEvaluator`. В первом случае строка *замена* интерпретируется по правилам, приведенным во врезке на с. 528. Пример:

```
Dim R_CapWord as New Regex("\b[A-Z]\w*")
    ⋮
Text = R_CapWord.Replace(Text, "<B>$0</B>")
```

Этот фрагмент заключает все слова, начинающиеся с прописных букв, в теги `<B>…</B>`. Если при вызове `Replace` задан аргумент *количество*, количество выполняемых замен ограничивается указанной величиной (по умолчанию выполняются все замены). Например, чтобы заменить только первое из найденных совпадений, задайте аргумент *количество* равным `1`. Если вас интересует только первое совпадение, ограничение количества замен повышает эффективность, поскольку методу `Replace` не приходится перебирать дополнительные совпадения. Значение `-1` означает «заменить все совпадения» (режим используется по умолчанию, если аргумент *количество* не задан).

СПЕЦИАЛЬНЫЕ ЗАМЕНЯЮЩИЕ ПОСЛЕДОВАТЕЛЬНОСТИ

Методы `Regex.Replace` и `Match.Result` получают строку замены, которая интерпретируется по особым правилам. Следующие последовательности внутри этой строки заменяются текстом совпадения:

Последовательность	Заменяется
`$&`	Текст всего совпадения (также доступен в виде `$0`)
`$1, $2, …`	Текст, совпавший с заданной парой сохраняющих круглых скобок
`$ (`*имя*`)`	Текст, совпавший с конструкцией именованного сохранения
`$``	Часть целевого текста, *предшествующая* совпадению
`$'`	Часть целевого текста, *следующая* за совпадением
`$$`	Одиночный символ '`$`'
`$_`	Копия исходного целевого текста
`$+`	(см. ниже)

Последовательность `$+` в ее текущей реализации практически бесполезна. Ее происхождение связано с весьма полезной переменной Perl `$+`, которая ссылается на пару сохраняющих круглых скобок с наибольшим номером, *участвующую* в совпадении (пример приведен на с. 264). Однако в строках замены .NET эта переменная просто ссылается на пару сохраняющих круглых скобок с наибольшим номером, присутствующую в выражении. Ее бесполезность становится особенно очевидной с учетом автоматической перенумерации сохраняющих круглых скобок при использовании именованных сохранений (☞ 510).

Все вхождения '`$`' в строку замены во всех остальных ситуациях, кроме перечисленных в таблице, интерпретируются обычным образом.

Если при вызове `Replace` указано целочисленное *смещение*, то регулярное выражение применяется после заданного количества символов от начала строки. Пропущенные символы копируются в результирующую строку без изменений.

Например, канонизация всех пропусков (т. е. замена последовательностей пропусков одним пробелом) может производиться следующим образом:

```
Dim AnyWS as New Regex("\s+")
    ⋮
Target = AnyWS.Replace (Target, " ")
```

В результате канонизации строка ‘some•••••random•••••spacing’ преобразуется в ‘some•random•spacing’. Следующий пример делает то же самое, но все *начальные* пропуски остаются без изменений:

```
Dim AnyWS as New Regex("\s+")
Dim LeadingWS as New Regex("^\s+")
    ⋮
Target = AnyWS.Replace(Target, " ", -1. LeadingWS.Match (Target).Length)
```

Строка ‘••••some•••random•••••spacing’ преобразуется в строку ‘••••so me•random•spacing’. Длина текста, совпавшего с LeadingWS, определяет смещение (количество символов, пропускаемых от начала строки) при поиске с заменой. В этом примере задействована одна удобная особенность объекта Match, представленного переменной LeadingWS.Match(Target): свойство Length этого объекта может использоваться даже в том случае, если поиск завершился неудачей. При неудачном поиске свойство Length равно нулю; именно это и нужно, чтобы выражение AnyWS применялось ко всему целевому тексту.

Использование делегатов при замене

Аргумент *замена* не ограничивается обычными строковыми значениями. В нем также может передаваться *делегат* (в сущности, указатель на функцию). Делегированная функция вызывается после каждого совпадения и генерирует текст, используемый в качестве замены. В ней можно выполнить любые действия по вашему усмотрению, поэтому механизм замены с использованием делегатов обладает практически неограниченными возможностями.

Делегат относится к типу MatchEvaluator и вызывается один раз для каждого совпадения. Делегированная функция получает объект Match, представляющий совпадение, выполняет любые действия по усмотрению программиста и возвращает текст, который должен использоваться в качестве замены.

Например, следующие два фрагмента генерируют одинаковые результаты:

```
Target = R.Replace(Target, "<<$&>>"))
```

```
Function MatchFunc(ByVal M as Match) as String
    return M.Result("<<$&>>")
End Function
Dim Evaluator as MatchEvaluator = New MatchEvaluator(AddressOf MatchFunc)
    ⋮
Target = R.Replace(Target, Evaluator)
```

В обоих фрагментах текст совпадения выделяется при помощи угловых скобок <<…>>. Одно из преимуществ использования делегатов заключается в том, что при

вычислении замены может быть задействован сколь угодно сложный код. В следующем примере функция `MatchFunc` преобразует температуру по Цельсию в шкалу Фаренгейта:

```
Function MatchFunc(ByVal M as Match) as String
    ' Получить числовое значение температуры из переменной $1
    ' и преобразовать его в шкалу Фаренгейта
    Dim Celsius as Double = Double.Parse(M.Groups(1).Value)
    Dim Fahrenheit as Double = Celsius * 9/5 + 32
    Return Fahrenheit & "F" ' Присоединить суффикс "F" и вернуть управление
End Function

Dim Evaluator as MatchEvaluator = New MatchEvaluator(AddressOf MatchFunc)
    ⋮
Dim R_Temp as Regex = New Regex("(\d+)C\b", RegexOptions.IgnoreCase)
Target = R_Temp.Replace(Target, Evaluator)
```

Если переменная `Target` содержит текст '`Temp is 37C.`', он будет заменен текстом '`Temp is 98.6F.`'.

```
RegexObj.Split(текст)
RegexObj.Split(текст, количество)
RegexObj.Split(текст, количество, смещение)
Тип возвращаемого значения: массив элементов типа String
```

Метод `Split` применяет регулярное выражение, представленное объектом, к заданному целевому *тексту* и возвращает массив строк, *разделенных* совпадениями. Тривиальный пример:

```
Dim R as New Regex("\.")
Dim Parts as String() = R.Split("209.204.146.22")
```

В результате вызова `R.Split` будет получен массив из четырех строк ('`209`', '`204`', '`146`' и '`22`'), разделенных тремя совпадениями `「\.」` в тексте.

Если при вызове `Replace` задан аргумент *количество*, возвращаемый массив содержит не более указанного *количества* элементов (если в выражении не используются сохраняющие круглые скобки, но об этом ниже). Если аргумент *количество* отсутствует, `Split` возвращает все строки, разделенные совпадениями. Если при переданном аргументе *количество* поиск будет прекращен до последнего совпадения, в последнем элементе массива возвращается остаток строки:

```
Dim R as New Regex("\.")
Dim Parts as String() = R.Split("209.204.146.22", 2)
```

В этом случае массив `Parts` заполняется двумя строками: '`209`' и '`204. 146.22`'.

Если при вызове `Split` указано целочисленное *смещение*, то регулярное выражение применяется после заданного количества символов от начала строки. Пропущенный текст включается в первую из возвращаемых строк (с параметром `RegexOptions:RightToLeft` пропущенный текст включается в *последнюю* возвращаемую строку).

Split и сохраняющие круглые скобки

Если регулярное выражение содержит сохраняющие круглые скобки любого типа, в массив *обычно* включаются дополнительные элементы, представляющие сохраненный текст (о том, когда это не происходит, рассказано ниже). Рассмотрим простой пример: разбиение строк вида '`2006-12-31`' или '`04/12/2007`' на компоненты может производиться по выражению `「[-/]」`:

```
Dim R as New Regex("[-/]")
Dim Parts as String() = R.Split(MyDate)
```

Фрагмент возвращает список из трех чисел (в виде строк). Тем не менее при включении в выражение сохраняющих круглых скобок и применении выражения `「([-/,])」` метод `Split` вернет пять строк: если `MyDate` содержит строку '`2006-12-31`', будут получены строки '`2006`', '`-`', '`12`', '`-`' и '`31`'. Дополнительные элементы '`-`' сгенерированы сохраняющими круглыми скобками `($1)`, входящими в каждое совпадение.

Если выражение содержит несколько пар сохраняющих круглых скобок, совпадения включаются в числовом порядке их следования (а это означает, что все именованные сохранения включаются после всех неименованных сохранений ☞ 510).

Функция `Split` устойчиво работает с сохраняющими круглыми скобками при условии, что все пары скобок фактически участвуют в совпадении. Тем не менее в текущей версии .NET существует ошибка: если какая-либо пара сохраняющих круглых скобок в совпадении не участвует, то она и все последующие пары не включаются в возвращаемый список.

Рассмотрим другой, несколько искусственный пример. Предположим, требуется разбить строку по запятым с возможными прилегающими пропусками, но пропуски должны быть включены в список возвращаемых элементов. Теоретически задача должна решаться выражением `「(\s+)?,(\s+)?」`. При применении `Split` к строке '`this•,••that`' возвращается список из четырех элементов: '`this`', '`•`', '`••`' и '`that`'. Однако при применении к строке '`this,•that`' отсутствие совпадения для первой пары круглых скобок приводит к тому, что эта пара (и все последующие)

не включается в результат, поэтому метод вернет всего две строки, 'this' и 'that'. Вы не можете заранее определить, сколько строк будет возвращено для каждого совпадения, и это является серьезным недостатком текущей реализации.

В нашем конкретном примере проблема обходится простым выражением `「(\s*),(\s*)」` (в этом случае обе группы заведомо участвуют во всех общих совпадениях). Однако переписать подобным образом более сложное выражение удается не всегда.

```
RegexObj.GetGroupNames()
RegexObj.GetGroupNumbers()
RegexObj.GroupNameFromNumber(число)
RegexObj.GroupNumberFromName(имя)
```

Методы возвращают информацию о номерах и именах (при использовании именованных сохранений) сохраняющих групп в регулярном выражении. Они не связаны с конкретными совпадениями и относятся только к именам и номерам групп, существующих в регулярном выражении. Пример использования этих методов приведен в следующей врезке.

```
RegexObj.ToString()
RegexObj.RightToLeft
RegexObj.Options
```

Методы предназначены для получения информации об объекте `Regex` и не связаны с применением регулярного выражения к строке. Метод `ToString()` возвращает строку, переданную при вызове конструктора. Свойство `RightToLeft` возвращает логическую величину, которая сообщает, был ли задан для данного регулярного выражения параметр `RegexOptions.RightToLeft.` Свойство `Options` возвращает параметры `Regex Options`, ассоциированные с выражением. В следующей таблице приведены значения отдельных параметров, суммируемые в возвращаемой величине.

0	None	16	Singleline
1	IgnoreCase	32	IgnorePatternWhitespace
2	Multiline	64	RightToLeft
4	ExplisitCapture	256	ECMAScript
8	Compiled		

Пропущенное значение 128 зарезервировано Microsoft для целей отладки и не поддерживается в окончательной версии продукта.

Пример использования этих методов приведен в следующей врезке.

ВЫВОД ИНФОРМАЦИИ ОБ ОБЪЕКТЕ РЕГУЛЯРНОГО ВЫРАЖЕНИЯ

Следующая программа выводит информацию об объекте Regex, хранящемся в переменной R:

```
' Вывод информации об объекте Regex, хранящемся в переменной R
Console.WriteLine("Regex is: " & R.ToString()) Console.WriteLine("Options
are: " & R.Options)
If R.RightToLeft
    Console.WriteLine("Is Right-To-Left: True")
Else
    Console.WriteLine("Is Right-To-Left: False")
End If

Dim S as String
For Each S in R.GetGroupNames()
    Console.WriteLine("Name """ & S & """ is Num #" & _
                      R.GroupNumberFromName(S))
Next
Console.WriteLine("---")
Dim I as Integer
For Each I in R.GetGroupNumbers()
    Console.WriteLine("Num #" & I & " is Name """ & _
                      R.GroupNameFromNumber(I) & """")
Next
```

Выполните программу для каждого из двух объектов Regex, созданных командами

```
New Regex("^(\w+)://[^/]+)(/\S*)")
New Regex("^(?<proto>\w+)://(?<host>[^/]+)(?<page>/\S*)",
          RegexOptions.Compiled)
```

Результаты будут выглядеть так:

```
Regex is: ^(\w+)://([^/]+)(/\S*)
Option are: 0
Is Right-To-Left: False
Name "0" is Num #0
Name "1" is Num #1
Name "2" is Num #2
Name "3" is Num #3
---
Num #0 is Name "0"
Num #1 is Name "1"
Num #2 is Name "2"
Num #3 is Name "3"
```

```
Regex is: ^(?<proto>\w+)://(?<host>…
Option are: 8
Is Right-To-Left: False
Name "0" is Num #0
Name "proto" is Num #1
Name "host" is Num #2
Name "page" is Num #3
---
Num #0 is Name "0"
Num #1 is Name "proto"
Num #2 is Name "host"
Num #3 is Name "page"
```

Использование объектов Match

Объекты `Match` создаются методом `Match` объекта `Regex`, статической функцией `Regex.Match` (см. ниже) и методом `NextMatch` самого объекта `Match`. В них инкапсулируется вся информация, относящаяся к одному применению регулярного выражения. Ниже перечислены свойства и методы объектов `Match`:

***MatchObj*.Success**

Логический признак успешного совпадения. Если поиск завершился неудачей, объект является копией статического объекта `Match.Empty` (☞ 539).

***MatchObj*.Value**
***MatchObj*.ToString()**

Копии фактически совпавшего текста.

***MatchObj*.Length**

Длина фактически совпавшего текста.

***MatchObj*.Index**

Целое число, представляющее позицию целевого текста, в которой было обнаружено совпадение. Индексация начинается с нуля, поэтому значение совпадает с количеством символов от начала (левого края) строки до начала (левого края) совпавшего текста. Интерпретация не изменяется даже в том случае, если при создании регулярного выражения, породившего объект `Match`, был указан параметр `RegexOptions.RightToLeft`.

***MatchObj*.Groups**

Свойство относится к типу `GroupCollection` и представляет собой коллекцию объектов `Group`. Это обычный объект коллекции, обладающий свойствами `Count` и `Item`, но при обращениях к нему обычно используется индексация с выборкой отдельного объекта `Group`. Например, запись `M.Groups(3)` обозначает объект `Group`, относящийся к третьей паре сохраняющих круглых скобок, а запись `M.Groups("HostName")` — объект группы для именованного сохранения «`HostName`» (например, после использования `(?<HostName>…)` в регулярном выражении).

В C# используется синтаксис `M.Groups[3]` и `M.Groups["HostName"]`.

Нулевая группа представляет все совпадение, а запись *MatchObj*`.Groups(0).Value` идентична *MatchObj*`.Value`.

***MatchObj*.NextMatch()**

Метод `NextMatch()` ищет в исходной строке следующее совпадение регулярного выражения и возвращает новый объект `Match`.

MatchObj.**Result(***строка***)**

Метод возвращает заданную строку, в которой специальные последовательности заменяются по правилам, приведенным на с. 528. Рассмотрим простой пример:

```
Dim M as Match = Regex.Match(SomeString, "\w+")
Console.WriteLine(M.Result("The first word is '$&'"))
```

Таким способом можно получить копию текста слева и справа от совпадения:

```
M.Result("$`") ' Текст слева от совпадения
M.Result("$'") ' Текст справа от совпадения
```

В процессе отладки иногда бывает полезно выводить выражения типа

```
M.Result("[$`<$&>$']"))
```

Для объекта `Match`, созданного применением выражения ⌜`\d+`⌟ к строке '`May 16, 1998`', будет выведен текст '`May <16>, 1998`', наглядно показывающий границы совпадения.

MatchObj.**Synchronized()**

Метод возвращает новый объект `Match`, идентичный текущему, но безопасный для использования в многопоточных приложениях.

MatchObj.**Captures**

Свойство `Captures` используется относительно редко, тем не менее о нем вкратце говорится на с. 544.

Использование объектов Group

Объект `Group` содержит информацию о совпадении одной пары сохраняющих круглых скобок (для группы с нулевым номером — обо всем совпадении). Ниже перечислены свойства и методы объекта `Group`.

GroupObj.**Success**

Логический признак участия группы в совпадении. Не все группы «участвуют» в успешном общем совпадении; например, при успешном совпадении ⌜`(this)|(that)`⌟ в совпадении заведомо участвует только одна пара скобок из двух. Другой пример приведен в сноске на с. 188.

GroupObj.**Value**
GroupObj.**ToString()**

Копия текста, сохраненного группой. В случае неудачного поиска возвращается пустая строка.

GroupObj.**Length**

Длина текста, сохраненного группой. В случае неудачного поиска возвращается ноль.

GroupObj.**Index**

Целое число, представляющее позицию целевого текста, в которой было обнаружено совпадение. Индексация начинается с нуля, поэтому значение совпадает с количеством символов от начала (левого края) строки до начала (левого края) сохраненного текста. Интерпретация не изменяется даже в том случае, если при создании регулярного выражения, породившего объект `Match`, был указан параметр `RegexOptions.RightToLeft`.

GroupObj.**Captures**

У объекта `Group` также имеется свойство `Captures`, описанное на с. 544.

Статические вспомогательные функции

Как упоминалось в разделе «Основные принципы работы с регулярными выражениями» на с. 515, иногда можно обойтись без явного создания объектов `Regex`. Следующие статические функции позволяют напрямую применять регулярные выражения к тексту:

```
Regex.IsMatch(текст, шаблон)
Regex.IsMatch(текст, шаблон, параметры)
Regex.Match(текст, шаблон)
Regex.Match(текст, шаблон, параметры)
Regex.Matches(текст, шаблон)
Regex.Matches(текст, шаблон, параметры)
Regex.Replace(текст, шаблон, замена)
Regex.Replace(текст, шаблон, замена, параметры)
Regex.Split(текст, шаблон)
Regex.Split(текст, шаблон, параметры)
```

Все эти функции представляют собой «обертки» для базового конструктора `Regex` и методов, описанных выше. Они автоматически конструируют временный объект `Regex`, используют его для вызова запрашиваемого метода и затем уничтожают объект (вообще говоря, объект уничтожается не сразу, но об этом ниже).

Например, команда:

```
If Regex.IsMatch(Line, "^\s*$")
    ⋮
```

эквивалентна

```
Dim TemporaryRegex = New Regex("^\s*$")
If TemporaryRegex.IsMatch(Line)
    ⋮
```

А еще точнее, эквивалентная запись выглядит так:

```
If New Regex("^\s*$").IsMatch(Line)
    ⋮
```

Вспомогательные функции хороши тем, что в общем случае они упрощают выполнение базовых операций и делают запись более компактной. В результате объектно-ориентированный пакет начинает выглядеть как пакет с процедурным интерфейсом (☞ 132). Главный недостаток вспомогательных функций — необходимость многократной обработки регулярных выражений (аргумент *шаблон*).

Если регулярное выражение используется всего один раз за все время выполнения программы, применение вспомогательной функции не отражается на эффективности программы. Но в случае многократного применения регулярного выражения (например, в цикле или в часто вызываемой функции) лишние операции подготовки регулярного выражения заметно увеличивают непроизводительные затраты (☞ 308). Именно для предотвращения этих затрат была разработана схема с однократным построением объекта `Regex` и его многократным применением при проверке текста. Тем не менее, как показано в следующем разделе, в .NET существуют средства, объединяющие лучшие стороны обоих решений: удобство процедурного подхода и эффективность объектно-ориентированного подхода.

Кэширование регулярных выражений

Необходимость строить и сохранять отдельный объект `Regex` для каждого регулярного выражения, встречающегося в программе, привела бы к чрезмерному загромождению программы. Поэтому просто замечательно, что пакет регулярных выражений, входящий в состав .NET, предоставляет статические методы. Однако в теории статические методы при каждом вызове создают временный объект `Regex`, применяют его и затем уничтожают, что отрицательно сказывается на производительности. Такое положение вещей может привести к выполнению огромного числа избыточных операций, если одно и то же регулярное выражение используется в цикле.

К счастью, во избежание подобных ситуаций в .NET применяется кэширование временных объектов регулярных выражений, создаваемых статическими методами. Общие принципы кэширования уже обсуждались в главе 6 (☞ 310), однако в двух словах замечу, что когда в программе «наиболее часто» используется одно и то же регулярное выражение, построенный ранее временный объект `Regex` используется повторно, а вы избавляетесь от хлопот с сохранением и поддержанием объектов `Regex`.

Под словами «наиболее часто» понимаются 15 последних регулярных выражений. Если в цикле используется более 15 выражений, преимущества, которые дает механизм кэширования, будут утрачены, потому что шестнадцатое выражение вытеснит первое и к моменту начала новой итерации первое выражение уже будет отсутствовать в кэше и в результате временный объект `Regex` придется создавать заново.

Если размер кэша по умолчанию (15 регулярных выражений) окажется слишком мал для вашего приложения, его можно будет увеличить с помощью команды:

```
Regex.CacheSize = 123
```

Если вам потребуется вообще запретить кэширование, установите размер кэша равным нулю.

Дополнительные функции

Кроме функций, описанных в предыдущем разделе, существуют и другие статические функции:

`Regex.Escape (`*строка*`)`

Функция `Regex.Escape(…)` возвращает копию строки с экранированными метасимволами регулярных выражений. В результате исходная строка становится пригодной для включения в регулярное выражение в виде литерала.

Предположим, строковая переменная `SearchTerm` содержит данные, введенные пользователем. Значение переменной может использоваться при построении регулярного выражения:

```
Dim UserRegex as Regex = New Regex("^" & Regex.Escape(SearchTerm) & "$", _
                                   RegexOptions.IgnoreCase)
```

В результате критерий поиска может содержать символы, особым образом интерпретируемые в регулярных выражениях, без их экранирования. Без предварительного вызова `Escape` при выполнении фрагмента для переменной `SearchTerm`, равной '`:-)`', будет инициировано исключение `ArgumentException` (☞ 522).

`Regex.Unescape (`*строка*`)`

Эта странная функция получает строку и возвращает ее копию, в которой интерпретируются метасимволы регулярных выражений, а лишние символы `\` удаляются. Например, для строки '`\:\-\)`' эта функция вернет '`:-)`'.

При вызове `Unescape` также происходит декодирование экранированных обозначений символов. Если исходная строка содержит '`\n`', в возвращаемом тексте эта последовательность замещается символом новой строки. А если исходная строка

содержит '\u1234', в строку вставляется соответствующий символ Юникода. Полный перечень обозначений символов, которые интерпретируются функцией Unescape, приводится на с. 507.

Я не могу представить себе хорошее использование функции Regex.Unescape, связанное с регулярным выражением, но она может быть полезна в качестве общего инструмента для создания строк VB с учетом знаний об управляющих символах.

Match.Empty

Функция возвращает объект Match, представляющий неудачную попытку поиска. Вероятно, она предназначена для инициализации объектов Match, от которых в будущем вы собираетесь получать информацию (если заранее неизвестно, будет ли объект заполнен в результате поиска). Простой пример:

```
Dim SubMatch as Match = Match.Empty ' Инициализировать на тот случай,
                                    ' если значение не будет задано в цикле
    ⋮
Dim Line as String
For Each Line in EmailHeaderLines
    ' Если строка содержит тему сообщения, сохранить совпадение на будущее...
    Dim ThisMatch as Match = Regex.Match(Line, "^Subject:\s*(.*)", _
                                         RegexOptions.IgnoreCase)
    If ThisMatch.Success
        SubMatch = ThisMatch
    End If
        ⋮
Next
    ⋮
If SubMatch.Success
    Console.WriteLine(SubMatch.Result("The subject is: $1"))
Else
    Console.WriteLine("No subject!")
End If
```

Если строковый массив EmailHeaderLines в действительности не содержит ни одной строки (или не содержит строк Subject), значение переменной SubMatch не будет задано в процессе перебора, поэтому если переменная не была инициализирована заранее, то обращение к SubMatch после цикла вызовет исключение по нулевой ссылке. В подобных случаях возвращаемое значение Match.Empty удобно использовать в качестве инициализатора.

Regex.CompileToAssembly(…)

Создание сборки, инкапсулирующей объект Regex, — за дополнительной информацией обращайтесь к следующему разделу.

Нетривиальные возможности .NET

В оставшейся части этой главы рассматриваются некоторые специфические возможности .NET: построение библиотеки регулярных выражений с применением сборок, средства поиска вложенных конструкций, имеющиеся только в .NET, и объект `Capture`.

Сборки регулярных выражений

.NET позволяет инкапсулировать объекты `Regex` в сборках. Такая возможность может пригодиться для создания библиотеки регулярных выражений. Пример, приведенный во врезке на с. 542, показывает, как это делается.

В результате выполнения этого примера в каталоге *bin* проекта создается файл с именем *JfriedlsRegexLibrary.DLL*.

Созданная сборка может использоваться в других проектах. Сначала ссылка на сборку включается в Visual Studio .NET в диалоговом окне, вызываемом командой *Project > Add Reference*.

Чтобы получить доступ к классам сборки, следует импортировать их директивой

```
Imports jfriedl
```

Далее вы работаете с ними точно так же, как с любыми другими классами:

```
Dim FieldRegex as CSV.GetField = New CSV.GetField ' Создание нового объекта
Regex
    ⋮
Dim FieldMatch as Match = FieldReges.Match(Line) ' Применение регулярного
                                                 ' выражения к строке
While FieldMatch.Success
    Dim Field as String
    If FieldMatch.Groups(1) Success
        Field = FieldMatch.Groups("QuotedField").Value
        Field = Regex.Replace(Field, """""", """") ' Замена двух кавычек одной
    Else
        Field = FieldMatch.Groups("UnquotedField").Value
    End If

    Console.WriteLine("[" & Field & "]")
    ' Дальнейшие операции с переменной Field …
    FieldMatch = FieldMatch.NextMatch
End While
```

В этом примере классы импортируются только из пространства имен `jfriedl`, хотя они с таким же успехом могут импортироваться из пространства имен `jfriedl.CSV`. В этом случае объект `Regex` будет создаваться командой:

```
Dim FieldRegex as GetField = New GetField ' Создание нового объекта Regex
```

Выбор зависит в основном от стиля программирования.

Вы можете вообще ничего не импортировать и ссылаться на классы напрямую:

```
Dim FieldRegex as jfriedl.CSV.GetField = New jfriedl.CSV.GetField
```

При всей громоздкости эта запись ясно показывает, откуда берется используемый объект. Впрочем, и в этом случае выбор является делом стиля.

Поиск вложенных конструкций

Компания Microsoft включила в пакет регулярных выражений интересный и оригинальный инструмент для поиска сбалансированных конструкций (стоит напомнить, что эта задача не решается при помощи традиционных регулярных выражений). Раздел получился коротким, но разобраться в происходящем не так просто.

Суть дела проще всего понять на конкретном примере, поэтому мы начнем с рассмотрения следующего фрагмента:

```
Dim R As Regex = New Regex(" \(                          " & _
                           "   (?>                       " & _
                           "       [^()] +               " & _
                           "     |                       " & _
                           "       \( (?<DEPTH>)         " & _
                           "     |                       " & _
                           "       \) (?<-DEPTH>)        " & _
                           "   )*                        " & _
                           "   (?(DEPTH)(?!))            " & _
                           " \)                          " & _
RegexOptions.IgnorePatternWhitespace)
```

Такое выражение совпадает с первой парой круглых скобок правильной вложенности (например, с подчеркнутой частью '`before (nope (yes (here) okay) after`'. Первая круглая скобка не совпала, поскольку для нее в выражении не существует парной закрывающей скобки.

ПОСТРОЕНИЕ БИБЛИОТЕКИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Ниже приведен пример построения небольшой библиотеки регулярных выражений. Перед вами полная программа, которая создаст сборку (DLL) с тремя готовыми конструкторами `Regex`: `jfriedl.Mail.Subject`, `jfriedl.Mail.From` и `jfriedl.CSV.GetField`.

Первые два примера приведены просто для полноты. Только последний, действительно нетривиальный пример реально показывает сложность построения собственной библиотеки. Обратите внимание: флаг `RegexOptions.Compiled` не указывается, поскольку он автоматически применяется в процессе построения сборки.

```
Option Explicit On
Option Strict On

Imports System.Text.RegularExpressions
Imports System.Reflection

Module BuildMyLibrary
Sub Main()
    ' При вызовах RegexCompilationInfo передается регулярное выражение,
    ' параметры, имя в классе и логический признак открытости класса.
    ' Например, первый класс будет доступен для всех программ,
    ' использующих сборку, под именем "jfriedl.Mail.Subject".
    Dim RCInfo() as RegexCompilationInfo = {                              _
      New RegexCompilationInfo(                                           _
        "^Subject:\s*(.*)", RegexOptions.IgnoreCase,                      _
        "Subject", "jfriedl.Mail", true),                                 _
      New RegexCompilationInfo(                                           _
        "^From:\s*(.*)", RegexOptions.IgnoreCase,                         _
        "From", "jfriedl.Mail", true),                                    _
       New RegexCompilationInfo(                                          _
        "\G(?:^|,)                                        " &             _
        "(?:                                              " &             _
        "  (?# Любое поле в кавчках... )                  " &             _
        "  ""  (?# field's Открывающая кавычка )          " &             _
        "   (?<QuotedField>  (?> [^""]+ | """" )*   ) " &                 _
        "  ""  (?# Закрывающая кавычка )                  " &             _
        " (?# ...или... )                                 " &             _
        " |                                               " &             _
     "  (?# ...любой другой текст не в кавычках
              и не запятые...)                            " &
     "  (?<UnquotedField> [^"",]* )                       " &             _
     " )",                                                                _
     RegexOptions.IgnorePatternWhitespace,                                _
     "GetField", "jfriedl.CSV", true)                                     _
```

```
    }
    ' Служебные операции по построению и записи библиотеки
    Dim AN as AssemblyName = new AssemblyName()
    AN.Name = "JfriedlsRegexLibrary" ' Имя файла DLL
    AN.Version = New Version("1.0.0.0")
    Regex.CompileToAssembly(RCInfo, AN) ' Построение сборки
End Sub
End Module
```

В предельно сжатом виде это выражение работает так:

1. Для каждого совпавшего символа '(' конструкция ⌜(?<DEPTH>)⌟ увеличивает на единицу текущую глубину вложенности круглых скобок (по крайней мере после подвыражения ⌜\(⌟ в начале регулярного выражения).
2. Для каждого совпавшего символа ')' конструкция ⌜(?<DEPTH>)⌟ уменьшает на единицу текущую глубину вложенности.
3. Конструкция ⌜(?(DEPTH)(?!))⌟ убеждается в том, что текущая глубина равна нулю, прежде чем разрешать совпадение для завершающего литерала ⌜\)⌟.

Работа алгоритма основана на отслеживании успешно совпавших группировок в стеке возврата. ⌜(?<DEPTH>)⌟ — всего лишь разновидность ⌜()⌟ с именованным сохранением, которая всегда успешна. Поскольку эта конструкция находится сразу же после ⌜\(⌟, ее успешное совпадение (информация о котором остается в стеке до удаления) используется в качестве маркера для подсчета открывающих круглых скобок.

Таким образом, количество успешных группировок 'DEPTH', совпавших до настоящего момента, хранится в стеке возврата. Каждый раз, когда находится совпадение для закрывающей круглой скобки, это количество должно уменьшаться. Для этого используется специальная конструкция .NET ⌜(?<DEPTH>)⌟, которая удаляет из стека последнюю отметку об успешном совпадении DEPTH. Если в стеке таких отметок не окажется, ⌜(?<DEPTH>)⌟ не совпадет, что предотвращает возможные совпадения с лишними закрывающими скобками.

Наконец, ⌜(?(DEPTH)(?!))⌟ — обычное условие, которое применяет ⌜(?!)⌟ при успешной группировке 'DEPTH'. Если к моменту обработки этой конструкции группировка успешна, это говорит о наличии открывающей скобки, признак успеха которой не был компенсирован парной конструкцией ⌜(?<DEPTH>)⌟. В этом случае совпадение прерывается (несбалансированные элементы не должны включаться в совпадение), поэтому мы применяем ⌜(?!)⌟ (обычная негативная ретроспективная проверка пустого подвыражения, для которой заведомо не существует совпадения).

Надеюсь, вы получили некоторое представление о поиске парных конструкций в регулярных выражениях .NET.

Объект Capture

В объектной модели .NET присутствует еще один компонент — объект `Capture`, который мы еще не рассматривали. Одни считают, что он открывает новые возможности работы с результатами поиска, другие — что объект `Capture` только зря загромождает программу.

В определенном отношении объект `Capture` похож на объект `Group`; он также представляет текст, совпавший с парой сохраняющих круглых скобок. Как и объект `Group`, он содержит методы `Value` (совпавший текст), `Length` (длина совпавшего текста) и `Index` (смещение начала совпадения от начала строки в символах).

Главное отличие между объектами `Group` и `Capture` заключается в том, что каждый объект Group содержит коллекцию `Captures`, элементы которой представляют все *промежуточные* совпадения группы в процессе поиска, а также окончательный текст, совпавший с группой.

В следующем примере выражение `「^(..)+」` применяется к строке '`abcdefghijk`':

```
Dim M as Match = Regex.Match("abcdefghijk", "^(..)+")
```

Регулярное выражение совпадает с четырьмя экземплярами `「(..)」`, т. е. большей частью строки: '`abcdefghij`k'. Поскольку плюс находится за пределами круглых скобок, при каждой итерации квантификатора `+` совпадение сохраняется заново, а на последней итерации будет сохранен текст '`ij`' (т. е. `M.Groups(1).Value` вернет '`ij`'). Кроме того, `M.Groups(1)` также содержит коллекцию `Captures` со всеми промежуточными совпадениями '`ab`', '`cd`', '`ef`', '`gh`' и '`ij`', обнаруженными в процессе поиска:

```
M.Groups(1).Captures(0).Value is 'ab'
M.Groups(1).Captures(1).Value is 'cd'
M.Groups(1).Captures(2).Value is 'ef'
M.Groups(1).Captures(3).Value is 'gh'
M.Groups(1).Captures(4).Value is 'ij'
M.Groups(1).Captures.Count is 5
```

Обратите внимание: последний элемент содержит то же значение '`ij`', как и общее совпадение, `M.Groups(1).Value`. Оказывается, `Group.Value` всего лишь обеспечивает сокращенную запись для последнего промежуточного сохранения группы. Конструкция `M.Groups(1).Value` в действительности эквивалентна:

```
M.Groups(1).Captures( M.Groups(1).Captures.Count 1 ).Value
```

Остается сделать несколько дополнительных замечаний по поводу объектов Capture:

- `M.Groups(1).Captures` относится к типу `CaptureCollection` и, как любая коллекция, обладает свойствами `Items` и `Count`. Тем не менее свойство `Items` обычно опускается, а элементы индексируются напрямую в конструкциях вида `M.Groups(1).Captures(3)` (`M.Groups[1].Captures[3]` в C#).
- Объект `Capture` не имеет метода `Success`; для получения информации используется метод `Success` класса `Group`.
- Выше говорилось о том, что доступ к объектам `Capture` осуществляется через объект `Group`. Объект `Match` *тоже* обладает свойством `Captures`, хотя и относительно бесполезным. Свойство `M.Captures` обеспечивает прямой доступ к свойству `Capture` нулевой группы (иначе говоря, `M.Captures` является аналогом `M.Group(0).Captures`). Поскольку нулевая группа представляет все совпадение, промежуточные итерации отсутствуют, а коллекция состоит из одного объекта `Capture`. Так как содержимое `M.Captures` и `M.Group(0).Captures` в точности соответствует содержимому нулевого объекта `Group`, эти конструкции особой пользы не приносят.

Объект `Capture` — довольно любопытное новшество, которое выглядит более сложным и запутанным, чем есть на самом деле, из-за его «чрезмерной интеграции» в объектную модель. После того как я справился с документацией .NET и понял, как работают эти объекты, я отношусь к ним со смешанными чувствами. С одной стороны, новинка безусловно интересная. Я не берусь с ходу предложить практическое применение для объектов `Capture`, но это, вероятно, объясняется тем, что мой опыт работы с ними еще недостаточен. С другой стороны, все эти дополнительные группы конструируются в процессе поиска, а их инкапсуляция в объектах после поиска приводит к снижению эффективности; мне бы не хотелось платить такую цену, если дополнительная информация не используется в программе. В абсолютном большинстве программ группы `Capture` не используются, но, несмотря на это, все объекты `Group` и `Capture` (а также ассоциированные с ними объекты `GroupCollection` и `CaptureCollection`) все равно строятся при создании объекта `Match`. Таким образом, вы получаете эти объекты независимо от того, нужны они вам или нет. Впрочем, если вы придумаете достойное практическое применение для объектов `Capture` — используйте их.

10 PHP

Со второй половины 90-х годов, в период этапа бурного развития Веб, известного как «Web boom», язык PHP переживает период оглушительной популярности, рост которой продолжается и по сей день. Одна из причин такой популярности заключается в том, что даже неспециалисты после некоторой подготовки имеют возможность использовать основные его возможности. К тому же, несмотря на такую доступность, язык PHP обеспечивает широчайший круг функциональных возможностей, которые, вне всякого сомнения, придутся по душе даже бывалому программисту. Разумеется, PHP обладает поддержкой регулярных выражений, причем поддержка эта реализована в виде не менее *трех* механизмов регулярных выражений, никак не связанных друг с другом.

PHP включает в себя механизмы «preg», «ereg» и «mb_ereg». В этой книге будет рассматриваться семейство функций **preg**. В основе этого семейства лежит механизм НКА, а по своим возможностям и по производительности функции этого семейства опережают два других. (Слово «preg» следует произносить как «пи-рег».)

О предыдущих главах

Прежде чем представлять читателю содержимое этой главы, я должен подчеркнуть, что она в значительной степени основана на материале глав 1–6. Я понимаю, что некоторые читатели, программирующие только на языке PHP, могут начать сразу с этой главы, но я рекомендую ознакомиться с предисловием (особенно с системой условных обозначений) и предыдущими главами. В главах 1, 2 и 3 представлены многие концепции, возможности и приемы, используемые при работе с регулярными выражениями, а материал глав 4, 5 и 6 представляет ключевую информацию для понимания регулярных выражений, которая напрямую относится к механизму preg регулярных выражений PHP. Из наиболее важных тем, рассматривавшихся в предыдущих главах, можно упомянуть основы работы механизма регулярных выражений НКА, связанные с поиском совпадений, максимализмом, возвратами и вопросами эффективности.

Здесь мне хочется заметить, что, несмотря на наличие удобных таблиц, таких как на с. 548 в этой главе или на с. 165 и 169 в главе 3, данная книга не претендует на роль справочного руководства. Главная ее цель — научить вас *искусству* составления регулярных выражений.

Глава начинается с описания краткой истории развития механизма регулярных выражений preg, а вслед за этим следует обзор диалекта регулярных выражений, который он реализует. В последующих разделах вы найдете описание функционального интерфейса механизма preg, обсуждение вопросов эффективности, характерных для этого механизма, и в заключение будут приведены несколько расширенных примеров.

История развития механизма preg

Название «preg» происходит от префикса preg, с которого начинаются имена всех функций, составляющих интерфейс к этому механизму, и означает: «Perl Regular Expressions» (регулярные выражения Perl). Этот механизм был добавлен Андреем Змиевски (Andrei Zmievski), которого не устраивали ограничения, присущие стандартному на то время механизму ereg. (Название «ereg» расшифровывается как «extended regular expressions», т. е. «расширенные регулярные выражения» — POSIX-совместимый пакет, обладающий «расширенными» возможностями по сравнению с большинством обычных диалектов, однако с высоты современных стандартов он выглядит крайне минималистским.)

Андреем было добавлено семейство функций preg, которые составили интерфейс к PCRE («Perl Compatible Regular Expressions» — регулярные выражения, совместимые с Perl) — замечательной библиотеке поддержки регулярных выражений на базе механизма НКА, которая очень близко имитирует синтаксис и семантику регулярных выражений языка Perl и обеспечивает мощь, которой так не хватало Андрею.

Прежде чем наткнуться на библиотеку PCRE, Андрей взялся за изучение исходных текстов Perl, чтобы понять, что можно было бы позаимствовать для использования в PHP. Вне всяких сомнений, он был далеко не первым, кто взялся за это, и, конечно же, не первым, кто быстро пришел к выводу, что дело это не для слабонервных. Исходные тексты Perl, регулярные выражения которого поражают своей мощью и скоростью исполнения, за долгие годы неоднократно переделывались многими людьми и со временем превратились в нечто, выходящее за рамки понимания обычного человека.

К счастью, в свое время Филип Хейзель (Philip Hazel) из Кембриджского университета в Англии, также пораженный исходными текстами механизма регулярных выражений Perl, создал библиотеку PCRE (ссылка на которую приводилась в главе 3 на с. 127). Филип начал разработку на пустом месте, обладая знания-

ми семантики, которую он хотел имитировать. Для Андрея это стало большим плюсом, поскольку спустя несколько лет он обнаружил отлаженную, прекрасно документированную и высокопроизводительную библиотеку, которую смог включить в состав PHP.

На протяжении многих лет в соответствии с изменениями в Perl продолжала свое развитие и библиотека PCRE, а вместе с ней и PHP. В этой книге рассматриваются версии PHP 4.4.3 и 5.1.4. Обе эти версии включают в себя PCRE версии 6.6[1].

Для тех, кто еще не знаком с порядком нумерации версий PHP, замечу, что параллельно продолжают развиваться ветки *4.x* и *5.x*, при том, что PHP версий *5.x* был во многом переписан заново. Так как разработка и выпуск новых версий PHP в обеих ветках происходят независимо друг от друга, вполне возможна ситуация, когда версия PHP из ветки *5.x* будет содержать *более старую* версию PCRE, чем более свежий выпуск PHP из ветки *4.x*.

Диалект регулярных выражений PHP

Таблица 10.1. Общие сведения о диалекте preg регулярных выражений PHP

Сокращенные обозначения символов ①		
☞ 156	(C)	`\a [\b] \e \f\n \r \t \`*восьм* `\x`*шестн* `\x{`*шестн*`} \c`*символ*
Символьные классы и аналогичные конструкции		
☞ 161		Обычные классы: `[…]` и `[^…]` (допускаются конструкции стандарта POSIX `[:alpha:]` ☞ 166)
☞ 162		Любой символ, кроме символа новой строки: *точка* (с модификатором s — любой символ)
☞ 163	(U)	Комбинационные последовательности Юникода: `\X`
☞ 164	(C)	Сокращенные обозначения классов: `\w \d \s \W \D \S` (только для 8-битных символов)②
☞ 164	(C) (U)	Свойства, алфавиты и блоки Юникода:③ `\p{`*свойство*`}`, `\P{`*свойство*`}`
☞ 163		Принудительное однобайтовое совпадение (может быть рискованным):④ `\C`

[1] Занявшись исследованием различных версий PHP и PCRE, доступных к моменту написания этой главы, я обнаружил несколько ошибок, которые были исправлены в версиях PHP 4.4.3 и 5.1.4 (рассматриваемых в этой книге). Некоторые примеры из этой главы могут не работать с более ранними версиями.

Якорные метасимволы и другие проверки с нулевой длиной совпадения		
☞ 175		Начало строки/логической строки: `^ \A`
☞ 175		Конец строки/логической строки:⑤ `$ \Z\z`
☞ 177		Начало текущего совпадения: `\G`
☞ 180		Границы слов: `\b \B` (только для 8-битных символов)②
☞ 181		Позиционная проверка:⑥ `(?=…) (?!…) (?<=…) (?<!…)`
Комментарии и модификаторы режимов		
☞ 555		Модификаторы режимов: `(?`*мод-мод*`)`. Допустимые модификаторы: `x`⑦ `s m i X U`
☞ 555		Интервальное изменение режима: `(?`*мод-мод*`:…)`
☞ 183		Комментарии: `(?#…)` (с модификатором `x`, а также от символа # до
Группировка, сохранение, условные и управляющие конструкции		
☞ 184		Сохраняющие круглые скобки: `(…) \1 \2 ...`
☞ 186		Именованное сохранение: `(?P<`*имя*`>…) (?P=`*имя*`)`
☞ 185		Группирующие круглые скобки: `(?:…)`
☞ 187		Атомарная группировка: `(?>…)`
☞ 187		Конструкция выбора: `|`
☞ 593		Рекурсия: `(?R) (?`*число*`) (?P>`имя`)`
☞ 188		Условная конструкция: `(?`*if then*`|`*else*`)` — в части *if* может находиться позиционная проверка, `(R)` или (*число*)
☞ 189		Максимальные квантификаторы: `* + ? {n} {n,} {x, y}`
☞ 190		Минимальные квантификаторы: `*? +? ?? {n}? {n,}? {x, y}?`
☞ 190		Захватывающие квантификаторы: `*+ ++ ?+ {n}+ {n,}+ {x, y}+`
☞ 183	(C)	Литеральный текст: `\Q … \E`

(C) — может использоваться в символьном классе

(U) — только при совместном использовании с модификатором `u` ☞ 557

①...⑦ — дополнительные замечания приводятся в тексте

(Эта таблица может также служить описанием библиотеки PCRE, составляющей фундамент для функций семейства preg ☞ 127.)

В табл. 10.1 приводятся краткие сведения о диалекте регулярных выражений preg. Ниже приводятся дополнительные замечания к табл. 10.1.

① `\b` представляет символ backspace (забой) только в символьных классах; за их пределами он представляет границу слова (☞ 180).

Восьмеричные коды, соответствующие 8-битным значениям, могут состоять из двух или трех цифр. Специальная последовательность ⌜`\0`⌟, состоящая из одной цифры, соответствует символу NUL.

Метапоследовательность ⌜`\x`*шестн*⌟ может состоять из одной или двух цифр. Синтаксис ⌜`\x{`*шестн*`}`⌟ позволяет задавать шестнадцатеричные коды произвольной длины. Однако значения больше `\x{FF}` считаются допустимыми только при наличии модификатора `u` (☞ 555). Без этого модификатора использование значений больше чем `\x{FF}` будет расцениваться как ошибка.

② Даже в режиме использования кодировки UTF-8 (с модификатором `u`) метасимволы границ слов и символьные классы, такие как ⌜`\w`⌟, работают только с ASCII-символами. В случае необходимости работать с полным диапазоном символов Юникода вместо ⌜`\w`⌟ следует пользоваться конструкцией ⌜`\pL`⌟ (☞ 165), вместо ⌜`\d`⌟ — конструкцией ⌜`\pN`⌟, а вместо ⌜`\s`⌟ — конструкцией ⌜`\pZ`⌟.

③ PHP ориентируется на стандарт Юникода версии 4.1.0.

Поддержка алфавитов Юникода реализована без использования префиксов '`Is`' или '`In`', например: ⌜`\p{Cyrillic}`⌟ (☞ 165). Поддерживаются одно- и двухсимвольные сокращенные имена свойств, такие как ⌜`\p{Lu}`⌟, ⌜`\p{L}`⌟ и ⌜`\pL`⌟ (☞ 166). Длинные конструкции вида ⌜`\p{Letter}`⌟ не поддерживаются.

④ По умолчанию регулярные выражения механизма preg ориентированы на работу с однобайтовыми символами, а метасимвол ⌜`\C`⌟ по умолчанию означает то же, что и конструкция ⌜`(?s:.)`⌟ — **s**-модификация метасимвола ⌜`.`⌟. Однако при наличии модификатора **u** регулярные выражения механизма preg в состоянии работать с символами в кодировке UTF-8, т. е. с композитными символами, имеющим размер до 6 байтов. Но даже в этом случае ⌜`\C`⌟ продолжает соответствовать единственному байту. Предупреждение об этой особенности приводится на с. 163.

⑤ Оба метасимвола ⌜`\z`⌟ и ⌜`\Z`⌟ могут совпадать с самым концом целевой строки, но ⌜`\Z`⌟ может также соответствовать заключительному символу новой строки.

Смысл метасимвола ⌜`$`⌟ зависит от наличия модификаторов **m** и **D** (☞ 555): в отсутствие модификаторов ⌜`\$`⌟ является аналогом ⌜`\Z`⌟ (совпадение обнаруживается либо перед заключительным символом новой строки, либо в самом конце целевой строки); с модификатором **m** он также может совпадать с промежуточными символами новой строки; с модификатором **D** является аналогом ⌜`\z`⌟ (соответствие обнаруживается только в конце строки). При наличии обоих модификаторов, **m** и **D**, модификатор **D** игнорируется.

⑥ Ретроспективная проверка ограничивается подвыражениями, совпадающими с текстом фиксированной длины, при этом на верхнем уровне допускаются альтернативы различной фиксированной длины (☞ 181).

⑦ Модификатор **x** (режим свободного форматирования и комментариев) распознает только пропуски из набора ASCII. Остальные символы пропусков Юникода не распознаются.

Функциональный интерфейс механизма preg

Для доступа к механизму регулярных выражений язык PHP предоставляет исключительно процедурный интерфейс (☞ 132), который состоит из семи функций, перечисленных в верхней половине табл. 10.2. Кроме того, в таблице перечислены четыре дополнительные полезные функции, исходные тексты которых будут представлены ниже в этой же главе.

Таблица 10.2. Перечень функций языка PHP для доступа к механизму регулярных выражений Preg

Функция	Назначение
☞ 559 `preg_match`	Проверяет наличие совпадений в строке и извлекает данные из строки
☞ 565 `preg_match_all`	Извлекает данные из строки
☞ 571 `preg_replace`	Замещает найденное совпадение копией заданной строки
☞ 577 `preg_replace_callback`	Для каждого найденного совпадения вызывает заданную функцию
☞ 580 `preg_split`	Разбивает исходную строку на массив подстрок
☞ 586 `preg_grep`	Отбирает элементы массива, которые соответствуют/не соответствуют заданному регулярному выражению
☞ 587 `preg_quote`	Экранирует метасимволы регулярных выражений в строке
Следующие функции будут разработаны в этой главе; сюда они включены для удобства поиска	
☞ 566 `reg_match`	Версия `preg_match`, которая различает круглые скобки, не участвовавшие в совпадении
☞ 588 `preg_regex_to_pattern`	Создает строку шаблона preg из строки регулярного выражения
☞ 592 `preg_pattern_error`	Проверяет строку шаблона preg на наличие синтаксических ошибок
☞ 592 `preg_regex_error`	Проверяет строку регулярного выражения на наличие синтаксических ошибок

Результат работы каждой из этих функций во многом зависит от количества и типов входных аргументов, флагов и модификаторов, используемых в регулярном выражении. Прежде чем приступить к подробному обсуждению, рассмотрим несколько примеров, чтобы получить некоторое представление о том, как выглядят и как обрабатываются регулярные выражения в PHP:

```
/* Проверяет, является ли тег HTML тегом <table> */
if (preg_match('/^<table\b/i', $tag))
    print "tag is a table tag\n";

/* Проверяет, является ли строка изображением целого числа */
if (preg_match('/^-?\d+$/', $user_input))
    print "user input is an integer\n";

/* Извлекает из строки содержимое HTML-тега <title> */
if (preg_match('{<title>(.*?)</title>}si', $html, $matches))
    print "page title: $matches[1]\n";

/* Интерпретирует числа в строке как значения температуры по Фаренгейту
и замещает их значениями температуры по Цельсию */
$metric = preg_replace('/(-?\d+(?:\.\d+)?)/e',              /* шаблон */
                  'floor(($1-32)*5/9 + 0.5)', /* код, выполняющий замену */
                  $string);

/* Создает массив значений, полученных из строки в формате CSV */
$values_array = preg_split('!\s*,\s*!', $comma_separated_values);
```

В последнем примере, когда заданная строка содержит текст ‘Larry, Curly, Moe’, в результате будет получен массив из трех строк: ‘Larry’, ‘Curly’ и ‘Moe’.

Аргумент шаблон

Первым аргументом любой функции из семейства preg передается *шаблон*, представляющий собой регулярное выражение, окруженное парой разделителей, за которым могут следовать модификаторы шаблона. В первом примере выше шаблоном является строка `'/<table\b/i'`, которая представляет регулярное выражение `<table\b`, окруженное двумя символами слэша (разделителями), за которым следует модификатор шаблона **i** (поиск без учета регистра символов).

Строки в апострофах в языке PHP

В регулярных выражениях очень часто используются символы обратного слэша, поэтому наиболее удобной формой представления шаблонов в PHP являются

строки литералов в апострофах. Строки литералов уже рассматривались в главе 3 (☞ 141), однако в двух словах напомню, что при использовании строк в апострофах вам не придется добавлять массу дополнительных экранирующих символов обратного слэша для представления регулярного выражения в тексте программы. В языке PHP специальной интерпретации подвергаются всего две метапоследовательности, которые могут встречаться в строках, заключенных в апострофы. Это '\'' и '\\', которые позволяют включать в строки символы '' и '\' соответственно.

Одно общеизвестное исключение составляет последовательность ⌜\\⌟ внутри регулярного выражения, которая соответствует единичному символу обратного слэша. Внутри строк в апострофах для представления каждого ⌜\⌟ необходимо указывать с, поэтому, чтобы представить ⌜\\⌟, необходимо записать \\\\. И все это для того, чтобы найти совпадение с единственным символом обратного слэша!

(Пример подобного гипертрофированного использования символов обратного слэша вы найдете на с. 590.)

В качестве конкретного примера рассмотрим регулярное выражение, которое находит совпадение с именем диска в операционной системе Windows, например 'C:\'. Само регулярное выражение выглядит следующим образом: ⌜^[A-Z]:\\$⌟, однако после заключения в апострофы оно уже будет выглядеть так: '^[A-Z]:\\\\$'.

В примере на с. 248 (глава 5) мы уже видели, что для представления выражения ⌜^.*\\⌟ используется строка '/^.*\\\/' — с *тремя* символами обратного слэша. Помня об этом, я посчитал, что следующие примеры будут весьма поучительными:

```
print  '/^.*\/';      выведет  /^.*\/
print  '/^.*\\/';     выведет  /^.*\/
print  '/^.*\\\/';    выведет  /^.*\\/
print  '/^.*\\\\/';   выведет  /^.*\\/
```

Первые два примера дают одинаковые результаты, хотя достигается это разными способами. В первом случае последовательность '\/', завершающая строку, не имеет специального значения в строках, заключенных в апострофы, поэтому данная последовательность выводится как последовательность литералов. Во втором случае последовательность '\\' имеет специальную интерпретацию в строковых литералах и представляет одиночный символ '\'. Таким образом, в комбинации с завершающим символом слэша результат получается тем же самым, что и в случае с комбинацией '\/', которая используется в первом примере. Следуя той же логике, несложно понять, почему третий и четвертый примеры дают одинаковые результаты.

Безусловно, в языке PHP вы можете использовать и строки в кавычках, но они менее удобны для представления регулярных выражений, так как они поддерживают значительное число разнообразных метапоследовательностей, каждую из которых придется экранировать.

Разделители

Механизм preg требует наличия разделителей, окружающих регулярные выражения, потому что разработчики стремились обеспечить более близкую к Perl имитацию регулярных выражений, особенно при использовании модификаторов. Для некоторых программистов это обстоятельство может показаться осложняющим фактором в сравнении с возможностью передачи модификаторов каким-либо иным способом, но как бы то ни было, мы имеем то, что мы имеем. (Один из «худших» примеров вы найдете во врезке на с. 558.)

Наиболее часто в качестве разделителя используют символ слэша, однако можно употреблять любые не алфавитно-цифровые символы, за исключением пробела ASCII и символа обратного слэша. Помимо символа слэша в качестве разделителей нередко встречаются символы '`!`' и '`#`'.

Если в качестве первого разделителя используется один из «открывающих» знаков пунктуации:

```
{ ( < [
```

то в качестве завершающего разделителя должен использоваться парный «закрывающий» знак пунктуации:

```
} ) > ]
```

При использовании таких «парных» разделителей их можно применять для построения вложенных конструкций, т. е. в качестве строки шаблона вполне допустимо использовать такую комбинацию: `'((\d+))'`. В данном примере внешние скобки являются разделителями, ограничивающими шаблон, а внутренние скобки являются частью регулярного выражения. Однако во избежание путаницы я рекомендовал бы использовать более простую форму: `'/(\d+)/'`.

Символы, используемые в качестве разделителей, внутри строк шаблонов регулярных выражений необходимо экранировать. Так, запись вида `'/<B>(.*?)<\/B>/i'` считается допустимой, хотя в подобной ситуации удобнее было бы использовать запись `'!<B>(.*?)</B>!i'`, где в качестве разделителей выступают символы '`!…!`', или `'{<B>(.*?)</B>}i'`, где используются разделители `'{…}'`.

Модификаторы шаблонов

После закрывающего разделителя, а в некоторых случаях и внутри регулярного выражения, для настройки определенных аспектов использования шаблонов могут помещаться различные модификаторы режима (в терминологии PHP именуемые *модификаторами шаблонов*). В приведенных выше примерах нам уже встречался

модификатор шаблона **i**, включающий режим поиска без учета регистра символов. Ниже приводится перечень всех допустимых модификаторов.

Модификатор	Встроенный модификатор	Описание
`i`	`(?i)`	☞ 150 Поиск совпадений без учета регистра символов
`m`	`(?m)`	☞ 153 Расширенный режим привязки к границам строк
`s`	`(?s)`	☞ 152 Режим совпадения точки со всеми символами
`x`	`(?x)`	☞ 151 Режим свободного форматирования
`u`		☞ 556 Регулярное выражение и целевая строка интерпретируются как строки символов Юникода
`X`	`(?X)`	☞ 556 Разрешает использование дополнительных возможностей PCRE
`e`		☞ 572 Строка замены интерпретируется как программный код PHP (только для функции `preg_replace`)
`S`		☞ 597 Запускает попытку оптимизации регулярного выражения («study»)
Следующие модификаторы используются достаточно редко		
`U`	`(?U)`	☞ 557 Изменяет степень максимализма на противоположную для `*` и `*?` и других
`A`		☞ 557 Привязывает все совпадение к начальной позиции поиска
`D`		☞ 557 `$` совпадает с концом строки (EOS), но не с заключительным символом новой сроки. (Игнорируется при использовании модификатора `m`.)

Модификаторы шаблонов внутри регулярных выражений

Внутри регулярных выражений модификаторы шаблонов могут использоваться для временного включения или выключения режимов поиска (например, `(?i)` включает режим поиска без учета регистра символов, а `(?-i)` — выключает его ☞ 182). При таком использовании оказываемый модификатором эффект распространяется до конца объемлющих круглых скобок, если таковые имеются, либо до конца регулярного выражения.

Они могут использоваться также как *интервальные модификаторы режима* (☞ 183), например конструкция `(?i:…)` включает режим поиска без учета регистра символов, а конструкция `(?-sm:…)` отключает действие модификаторов **s** и **m** для заданного участка регулярного выражения.

Модификаторы режимов за пределами регулярных выражений

После заключительного разделителя модификаторы могут комбинироваться в любом порядке, например: 'si' — эта комбинация включает режимы поиска без учета регистра символов и совпадения точки со всеми символами:

```
if (preg_match('{<title>(.*?)</title>}si', $html, $captures))
```

Модификаторы, специфичные для PHP

Первые четыре модификатора из таблицы выше являются стандартными и уже обсуждались в главе 3 (☞ 150). Модификатор шаблона **e** используется исключительно вместе с функцией `preg_replace` и будет описан в этом разделе (☞ 572).

Модификатор шаблона **u** сообщает механизму регулярных выражений, что само регулярное выражение и целевая строка содержат символы в кодировке UTF-8. Наличие этого модификатора не приводит к изменению байтов, он просто воздействует на способ их интерпретации. По умолчанию (т. е. без модификатора **u**), механизм preg рассматривает полученные им данные как 8-битовые символы текущего локального контекста (☞ 125). Применяйте модификатор **u**, только если заранее известно, что данные будут содержать символы в кодировке UTF-8, в противном случае использовать этот модификатор не следует. Символы в кодировке UTF-8, коды которых не совпадают с кодами ASCII, кодируются несколькими байтами, и использование модификатора **u** гарантирует, что такие *многобайтовые* последовательности будут интерпретироваться как одиночные *символы*.

Модификатор шаблона **X** включает использование «дополнительных возможностей» PCRE, но пока из них доступна только одна: генерация ошибки в случае использования в регулярном выражении символа обратного слэша, когда он не является частью известного метасимвола. Например, по умолчанию метасимвол `\k` не имеет специального значения в PCRE, и он интерпретируется как `\k` (так как обратный слэш не является частью известной метапоследовательности, он просто игнорируется). Использование модификатора **X** в подобной ситуации приведет к фатальной ошибке «unrecognized character follows \» (неизвестный символ после \).

Вполне возможно, будущие версии PHP будут включать в себя версии библиотеки PCRE, в которых неизвестные ныне метасимволы будут интерпретироваться особым образом. Поэтому с целью обеспечения совместимости с будущими версиями (и удобочитаемости) лучше отказаться от использования экранированных символов, которые не имеют специального назначения. В этом случае модификатор **X** приобретает особую значимость, так как с его помощью без труда можно определить места опечаток.

Модификатор шаблона **S** запускает механизм «study» библиотеки PCRE, который выполняет предварительный анализ регулярного выражения, что в некоторых четко определенных ситуациях позволяет незначительно ускорить поиск совпадений. Подробнее об этом модификаторе будет рассказываться в разделе, посвященном вопросам повышения эффективности, который начинается со с. 597.

Остальные модификаторы шаблонов являются довольно экзотическими и используются достаточно редко:

- Модификатор шаблона **A** привязывает совпадение с точкой, откуда была начата первая попытка поиска, как если бы все регулярное выражение начиналось с метасимвола ⌜\G⌟. Используя аналогию с автомобилем из главы 4, действие этого модификатора заключается в отсоединении трансмиссии от двигателя.
- Использование модификатора шаблона **D** приводит к тому, что ⌜$⌟ превращается в ⌜\z⌟ (☞ 153), т. е. ⌜$⌟ находит совпадение непосредственно за окончанием строки и никогда — перед завершающим символом новой строки.
- Модификатор шаблона **U** меняет степень максимализма метасимволов: ⌜*⌟ интерпретируется как ⌜*?⌟ и наоборот, ⌜+⌟ интерпретируется как ⌜+?⌟ и наоборот, и т. д. На мой взгляд, основной эффект этого модификатора заключается в том, чтобы вносить путаницу, поэтому я настоятельно не рекомендую использовать его.

ОШИБКА «НЕИЗВЕСТНЫЙ МОДИФИКАТОР»

Однажды я столкнулся с ситуацией, когда программа, над которой я работал, начала постоянно генерировать ошибку «unknown modifier» (неизвестный модификатор). Я долго ломал голову в попытках отыскать причину такой ошибки, пока меня не осенило — я забыл добавить в регулярное выражение разделитель при создании шаблона.

Например, я пытался отыскать тег HTML следующей командой:

```
preg_match('<(\w+)([^>]*)>', $html)
```

Несмотря на то что, по моему замыслу, начальный символ '<' является частью регулярного выражения, функция `preg_match` рассматривала его как разделитель, открывающий регулярное выражение (в конце концов, она не виновата в моей забывчивости). В результате аргумент интерпретировался как '`<(\w+)([^>]*)>`', где серым фоном выделена та часть, которая интерпретировалась как регулярное выражение, а подчеркиванием — часть, воспринимавшаяся как модификаторы шаблона.

С точки зрения механизма регулярных выражений часть `(\w+)([^` является неверным регулярным выражением, но прежде чем добраться до нее и обнаружить ошибку, механизм сначала попытался интерпретировать '`]*)>`' как список модификаторов шаблона. Разумеется, ни один из символов в этой подстроке не является допустимым модификатором, поэтому механизм в первую очередь сгенерировал ошибку:

```
Warning: Unknown modifier ']'
```

Оглядываясь назад, совершенно очевидно, что я должен был окружить регулярное выражение разделителями:

```
preg_match('/<(\w+)(.*?)>/', $html)
```

Если только я постоянно не думаю о модификаторах *шаблонов* PHP, появление подобных ошибок не всегда вызывает у меня нужные ассоциации, и иногда проходит несколько мгновений, прежде чем я начинаю понимать, в чем дело. Всякий раз, когда это происходит, я чувствую себя глупцом; хорошо, что никто не догадывается, что я делаю такие глупые ошибки.

К счастью, последние версии PHP 5 дополнительно сообщают еще и имя функции:

```
Warning: preg_match(): Unknown modifier ']'
```

Имя функции сразу направляет ход моих мыслей в нужное русло, что позволяет немедленно отыскать причину возникшей проблемы. Однако опасность

потратить уйму времени на поиски ошибки, возникающей из-за отсутствия разделителей, по-прежнему остается, так как в некоторых ситуациях *ни о какой ошибке не сообщается*. Рассмотрим версию предыдущего примера:

```
preg_match('<(\w+)(.*?)>', $html)
```

Здесь я также забыл добавить разделители, но остальная часть строки `(\w+)(.*?)` является вполне допустимым регулярным выражением. Единственное, что свидетельствует об ошибке, — это то, что выражение не находит совпадений, которые я ожидал. Такого рода скрытые ошибки наиболее опасны.

Функции preg

В этом разделе во всех подробностях будет рассматриваться каждая функция, начиная с самой главной функции, `preg_match`, отвечающей на вопрос: «имеются ли в заданном тексте совпадения с регулярным выражением?».

preg_match

Синтаксис

```
preg_match(шаблон, текст [, совпадения [, флаги [, смещение ]]])
```

Описание аргументов

шаблон	Аргумент шаблон — это регулярное выражение, окруженное разделителями, с необязательными модификаторами (☞ 552)
текст	Целевая строка, в которой производится поиск
совпадения	Необязательная переменная, куда могут быть записаны найденные совпадения
флаги	Необязательные флаги, влияющие на некоторые аспекты поведения функции. В настоящее время существует единственный допустимый флаг PREG_OFFSET_CAPTURE (☞ 563)
смещение	Необязательное смещение в *тексте*, откуда будет начинаться поиск совпадений. Отсчет символов начинается с нуля (☞ 564)

Возвращаемое значение

В случае наличия совпадения возвращается значение true, в противном случае — false.

Пояснение

В простейшем случае,

```
preg_match($pattern, $subject)
```

функция возвращает true, если для шаблона `$pattern` обнаруживаются совпадения в строке `$subject`. Ниже приводятся несколько простых примеров:

```
if (preg_match('/\.(jpe?g|png|gif|bmp)$/i', $url)) {
    /* URL является ссылкой на изображение */
}
```

```
if (preg_match('{^https?://}', $uri)) {
    /* используемый протокол http или https */
}
```

```
if (preg_match('/\b MSIE \b/x', $_SERVER['HTTP_USER_AGENT'])) {
    /* Используемый браузер IE */
}
```

Сохранение совпадений

Необязательный третий аргумент функции `preg_match` — это переменная, которая принимает информацию о найденном совпадении. Переменная может иметь любое имя, но наиболее часто используется имя `$matches`. В этой книге, когда я использую имя «`$matches`» вне контекста конкретного примера, в действительности я подразумеваю «некоторую переменную, использующуюся в качестве третьего аргумента функции `preg_match`».

После успешной попытки поиска `preg_match` возвращает true, а в переменную `$matches` записывается следующая информация:

```
$matches[0] весь текст найденного совпадения
$matches[1] текст, совпавший с первой парой сохраняющих круглых скобок
$matches[2] текст, совпавший со второй парой сохраняющих круглых скобок
    ⋮
```

В случае использования конструкций именованного сохранения в переменную включаются соответствующие элементы (пример приводится в следующем разделе).

Ниже следует простой пример, который уже встречался нам в главе 5 (☞ 251):

```
/* Из полного пути извлекается имя файла */
if (preg_match('{ / ([^/]+) $}x', $WholePath, $matches))
    $FileName = $matches[1];
```

Переменную `$matches` (или любую другую, которая предназначена для сохранения информации о совпадении) можно использовать только в случае, когда `preg_match` возвращает true. Значение false возвращается в случае отсутствия совпадения или в случае ошибки (например, некорректный шаблон или неверный флаг). В случае одних ошибок переменная `$matches` очищается, в случае некоторых других — в ней остается информация, записанная ранее, поэтому не следует делать какие-либо выводы, основываясь лишь на наличии некоторых данных в `$matches` после вызова `preg_match`.

Ниже приводится более интересный пример, с тремя парами сохраняющих круглых скобок:

```
/* Извлечь протокол, имя хоста и номер порта из строки URL */
if (preg_match('{^(https?):// ([^/:]+) (?: :(\d+) )? }x', $url, $matches))
{
    $proto = $matches[1];
    $host  = $matches[2];
    $port  = $matches[3] ? $matches[3] : ($proto == "http" ? 80 : 443);

    print "Protocol: $proto\n";
    print "Host : $host\n";
    print "Port : $port\n";
}
```

Удаление завершающих элементов, не участвовавших в совпадении

Пара круглых скобок, не участвующая в окончательном совпадении, дает в результате пустую строку в соответствующем элементе массива `$matches`[1]. Однако элементы, соответствующие *завершающим сохраняющим скобкам, не участвовавшим в совпадении*, вообще не включаются в массив `$matches`. Применительно к предыдущему примеру это означает, что когда подвыражение `(\d+)` участвует в совпадении, в элемент `$matches[3]` записывается число, в противном случае элемент `$matches[3]` вообще не включается в массив.

[1] Если вам необходимо, чтобы вместо пустой строки соответствующий элемент содержал значение NULL, обратите внимание на пример во врезке на с. 566.

Именованные сохранения

Давайте перепишем предыдущий пример с использованием именованных сохранений (☞ 186). Это приведет к увеличению длины регулярного выражения, но сделает программный код более понятным:

```
/* Извлечь протокол, имя хоста и номер порта из строки URL */
if (preg_match('{^(?P<proto> https? ) ://
                  (?P<host> [^/:]+ )
            (?: : (?P<port> \d+ ) )? }x', $url, $matches))
{
  $proto = $matches['proto'];
  $host  = $matches['host'];
  $port  = $matches['port'] ? $matches['port'] : ($proto== "http" ? 80 : 443);
  print "Protocol: $proto\n";
  print "Host : $host\n";
  print "Port : $port\n";
}
```

Очевидность, которую несет в себе именованное сохранение, позволяет отказаться от копирования элементов массива `$matches` в отдельные переменные. В этом случае есть смысл присвоить переменной, используемой для сохранения результатов совпадения, имя, отличное от `$matches`. Ниже приводится версия примера, переписанного с учетом этого замечания:

```
/* Извлечь протокол, имя хоста и номер порта из строки URL */
if (preg_match('{^(?P<proto> https? ):://
                  (?P<host> [^/:]+ )
            (?: : (?P<port> \d+ ) )? }x', $url, $UrlInfo))
{
  if (! $UrlInfo['port'])
      $UrlInfo['port'] = ($UrlInfo['proto'] == "http" ? 80 : 443);
  echo "Protocol: ", $UrlInfo['proto'], "\n";
  echo "Host  : ", $UrlInfo['host'], "\n";
  echo "Port  : ", $UrlInfo['port'], "\n";
}
```

Нумерованные сохранения записываются в массив `$matches` независимо от того, использовались ли именованные сохранения. Например, после поиска совпадения в строке `$url`, содержащей адрес ‘`http://regex.info/`’, массив `$UrlInfo` из предыдущего примера будет содержать:

```
array
(
     0       => 'http://regex.info',
     'proto' => 'http',
     1       => 'http',
     'host'  => 'regex.info',
```

```
    2       => 'regex.info'
)
```

Такое дублирование приводит к неоправданному расходу памяти, но такова цена, которую приходится платить за удобство именованных сохранений. Для сохранения ясности я не рекомендовал бы использовать в программах одновременно именованные и числовые индексы при обращении к элементам массива `$matches`, за исключением элемента `$matches[0]`, в котором хранится полный текст совпадения.

Обратите внимание: в этом примере элементы `3` и '`port`' не были включены в массив, потому что соответствующая пара круглых скобок не участвовала в совпадении и оказалась завершающей (вследствие чего элемент был удален ☞ 561).

В настоящее время считается допустимым, но не рекомендуется использовать числа в качестве имен сохранений, например: `(?P<2>…)`. PHP 4 и PHP 5 по-разному интерпретируют эту неоднозначную ситуацию, что может приводить не к тем результатам, которые ожидаются. Поэтому лучше будет вообще не использовать числа для имен в именованных сохранениях.

Получение дополнительной информации о совпадении: PREG_OFFSET_CAPTURE

Если функции `preg_match` передается аргумент *флаги* и он содержит PREG_OFFSET_CAPTURE (которое является единственным допустимым значением при работе с функцией `preg_match`), в массив `$matches` вместо простых строк будут вставляться подмассивы, состоящие из двух элементов. Первый элемент каждого подмассива содержит текст совпадения, а второй элемент — смещение от начала целевой строки, где было обнаружено данное совпадение (или `-1`, если круглые скобки не участвовали в совпадении).

Смещение отсчитывается от начала строки, начиная с нуля. Даже если функции `preg_match` передается пятый аргумент `$offset` с позицией начала поиска, находящейся где-то внутри целевой строки, позиция найденного совпадения все равно отсчитывается от начала строки. Смещение всегда измеряется в *байтах*, даже при наличии модификатора шаблона **u** (☞ 557).

В качестве примера рассмотрим попытку извлечения атрибута HREF из тега `<a>`. Значение HTML-атрибута может заключаться в кавычки, апострофы или вообще не заключаться в кавычки какого-либо вида. Такие значения извлекаются с помощью первой, второй и третьей пар сохраняющих круглых скобок соответственно:

```
preg_match('/href \s*=\s* (?: "([^"]*)" ; \'([^\']*)\' ; ([^\s\'">]+) )/ix',
        $tag,
        $matches,
        PREG_OFFSET_CAPTURE);
```

Если предположить, что переменная `$tag` содержит строку

```
<a name=bloglink href='http://regex.info/blog/' rel="nofollow">
```

тогда массив `$matches` будет содержать:

```
array
(
    /* Текст всего совпадения */
    0 => array ( 0 => "href='http://regex.info/blog/'",
                 1 => 17 ),

    /* Текст совпадения для первой пары круглых скобок */
    1 => array ( 0 => "",
                 1 => 1 ),
    /* Текст совпадения для второй пары круглых скобок */
    2 => array ( 0 => "http://regex.info/blog/",
                 1 => 23 )
)
```

Элемент массива `$matches[0][0]` содержит текст всего совпадения с регулярным выражением, элемент `$matches[0][1]` — смещение найденного совпадения от начала целевой строки в байтах.

Для иллюстрации попробуем получить ту же строку, что находится в `$matches[0][0]`, иным способом:

```
substr($tag, $matches[0][1], strlen($matches[0][0]));
```

Элемент `$matches[1][1]` содержит `1`, это свидетельствует о том, что первая пара сохраняющих скобок не участвовала в совпадении. Третья пара скобок также не участвовала в совпадении, но, как уже отмечалось ранее (☞ 561), информация о заключительных парах скобок, не участвующих в совпадении, не включается в массив `$matches`.

Аргумент смещение

Если функции `preg_match` передается аргумент *смещение*, тогда попытка поиска начинается со смещением на указанное число байтов от начала целевой строки (или, если смещение отрицательное, со смещением на указанное число байтов от *конца* целевой строки). По умолчанию смещение равно нулю (т. е. поиск начинается с начала целевой сроки).

Обратите внимание: в любом случае величина смещения задается в байтах, даже при использовании модификатора шаблона **u**. Использование некорректного смещения (когда поиск начинается «внутри» многобайтового символа) приводит к тому, что попытка поиска терпит неудачу без выдачи сообщения об ошибке.

Ненулевое *смещение* не означает, что метасимвол ⌜^⌟ будет соответствовать начальной позиции поиска, смещение — это лишь позиция в целевой строке, откуда механизм регулярных выражений начинает поиск. Ретроспективная проверка, к примеру, может просматривать символы строки, расположенные левее позиции смещения.

preg_match_all

Синтаксис

```
preg_match_all(шаблон, текст, совпадения [, флаги [, смещение ]])
```

Описание аргументов

шаблон	Аргумент шаблон — это регулярное выражение, окруженное разделителями, с необязательными модификаторами (☞ 552)
текст	Целевая строка, в которой производится поиск
совпадения	*Обязательная* переменная, куда могут быть записаны найденные совпадения
флаги	Необязательные флаги, влияющие на некоторые аспекты поведения функции. PREG_OFFSET_CAPTURE (☞ 569). и/или один из: PREG_PATTERN_ORDER (☞ 567) PREG_SET_ORDER (☞ 568)
смещение	Необязательное смещение в *тексте*, откуда будет начинаться поиск совпадений. Отсчет символов начинается с нуля. (То же самое, что и аргумент *смещение* функции preg_match ☞ 564.)

Возвращаемое значение

preg_match_all возвращает число найденных совпадений.

Пояснение

Функция preg_match_all напоминает функцию preg_match, за исключением того, что она не останавливается после нахождения первого совпадения, а продолжает поиск последующих совпадений. Для каждого совпадения создается массив с соответствующими данными; таким образом, по окончании поиска переменная *совпадения* превращается в массив массивов, где каждый вложенный массив представляет данные об одном совпадении.

ПУСТОЕ СОВПАДЕНИЕ И ОТСУТСТВИЕ СОВПАДЕНИЯ

Функция `preg_match` возвращает в переменной `$matches` пустую строку для пары скобок, которая не участвовала в совпадении (за исключением *завершающих* пар скобок, не участвовавших в совпадении, информация о которых вообще не включается в массив `$matches`). Пустая строка неотличима от успешного пустого совпадения, поэтому я предпочитаю в элементы массива, соответствующие круглым скобкам, не участвовавшим в совпадении, записывать значение NULL.

Ниже приводится версия функции `preg_match` (я назвал ее `reg_match`), которая использует флаг PREG_OFFSET_CAPTURE для получения дополнительной информации обо всех совпадениях с подвыражениями в круглых скобках и затем на основе этой информации записывает значения NULL в соответствующие элементы массива `$matches`:

```
function reg_match($regex, $subject, &$matches, $offset = 0)
{
    $result = preg_match($regex, $subject, $matches,
                         PREG_OFFSET_CAPTURE, $offset);
    if ($result) {
        $f = create_function('&$X', '$X = $X[1] < 0 ? NULL : $X[0];');
        array_walk($matches, $f);
    }
    return $result;
}
```

Результат, возвращаемый функцией `reg_match`, не отличается от результата, который возвращает функция `preg_match` без флага, за исключением того, что в элементы, соответствующие *скобкам, не участвовавшим в совпадении*, вместо пустых строк записывается значение NULL.

Простой пример:

```
if (preg_match_all('/<title>/i', $html, $all_matches) > 1)
   print "whoa, document has more than one <title>!\n";
```

Третий аргумент (переменная, куда собирается информация об успешных совпадениях) в функции `preg_match_all` является обязательным, в отличие от функции `preg_match`. Именно поэтому в данном примере функции `preg_match_all` передается третий аргумент, несмотря на то что нигде в примере он больше не используется.

Сбор данных о совпадениях

Еще одно важное отличие от `preg_match` заключается в данных, которые помещаются в третий аргумент-переменную. Функция `preg_match` находит не более одного

совпадения, поэтому она помещает в переменную `$matches` единственный блок данных. Функция `preg_match_all` может обнаруживать более одного совпадения, поэтому она помещает в третий аргумент, в переменную `$matches`, множество блоков данных обо всех найденных совпадениях. Чтобы подчеркнуть отличие, при использовании функции `preg_match_all` в качестве имени переменной я применяю `$all_matches`, а не `$matches`, которое обычно используется при работе с функцией `preg_match`.

Существует возможность задать в `preg_match_all` способ упорядочения данных в `$all_matches`, выбрав один из двух взаимоисключающих флагов в четвертом аргументе: PREG_PATTERN_ORDER или PREG_SET_ORDER.

Флаг упорядочения по умолчанию PREG_PATTERN_ORDER

Ниже приводится пример, демонстрирующий порядок размещения данных при использовании флага PREG_PATTERN_ORDER (который я для краткости называю флагом «группировки»). Кроме того, такой порядок размещения используется по умолчанию, если не был задан ни один флаг, как в данном примере:

```
$subject = "
Jack A. Smith
Mary B. Miller";

/* Отсутствие флага упорядочения
   подразумевает наличие флага PREG_PATTERN_ORDER */
preg_match_all('/^(\w+) (\w\.) (\w+)$/m', $subject, $all_matches);
```

В результате массив `$all_matches` будет заполнен следующими данными:

```
array
(
    /* $all_matches[0]  массив полных совпадений */
    0 => array ( 0 => "Jack A. Smith", /* полный текст первого совпадения */
                 1 => "Mary B. Miller" /* полный текст второго совпадения */ ),

    /* $all_matches[1]  массив строк, сохраненных первой парой скобок */
    1 => array ( 0 => "Jack", /* первое совпадение с первой парой скобок */
                 1 => "Mary" /* второе совпадение с первой парой скобок */ ),

    /* $all_matches[2]  массив строк, сохраненных второй парой скобок */
    2 => array ( 0 => "A.",   /* первое совпадение со второй парой скобок */
                 1 => "B." /+ второе совпадение со второй парой скобок */ ),

    /* $all_matches[3]  массив строк, сохраненных третьей парой скобок */
    3 => array ( 0 => "Smith", /* первое совпадение с третьей парой скобок */
                 1 => "Miller" /* второе совпадение с третьей парой скобок */ )
)
```

В данном примере было найдено два совпадения, для каждого из которых было сохранено по одной строке с «полным текстом совпадения» и по три строки для каждой из пар сохраняющих круглых скобок. Я называю такой порядок размещения «сгруппированным» потому, что все полные совпадения сгруппированы в одном массиве (`$all_matches[0]`), все совпадения с первой парой скобок сгруппированы в другом массиве (`$all_matches[1]`) и т. д.

По умолчанию найденные совпадения группируются в отдельных элементах массива `$all_matches`, однако с помощью флага PREG_SET_ORDER можно изменить такой порядок размещения.

Флаг упорядочения PREG_SET_ORDER

При помощи флага PREG_SET_ORDER можно задать альтернативный вариант «послойного» размещения данных. В этом случае все данные, соответствующие первому совпадению, помещаются в `$all_matches[0]`, все данные, соответствующие второму совпадению, — в `$all_matches[1]`, и т. д. Такое размещение в точности соответствует порядку, который вы получили бы при выполнении поиска с помощью функции preg_match, помещая каждый раз содержимое `$matches` в массив `$all_matches`.

Ниже приводится версия предыдущего примера, в которой используется флаг PREG_SET_ORDER:

```
$subject = "
Jack A. Smith
Mary B. Miller";

preg_match_all('/^(\w+) (\w\.) (\w+)$/m', $subject,
               $all_matches, PREG_SET_ORDER);
```

В результате массив `$all_matches` будет заполнен следующими данными:

```
array
(
    /* $all_matches[0]  массив, аналогичный массиву $matches,
       который вернула бы функция preg_match */
    0 => array ( 0 => "Jack A. Smith", /* полный текст первого совпадения */
                 1 => "Jack",  /* первое совпадение с первой парой скобок */
                 2 => "A.",    /* первое совпадение со второй парой скобок */
                 3 => "Smith"  /* первое совпадение с третьей парой скобок */,

    /* $all_matches[1]  массив, также аналогичный массиву $matches,
       который вернула бы функция preg_match */
```

```
    1 => array ( 0 => "Mary B. Miller", /* полный текст второго совпадения */
                 1 => "Mary",   /* второе совпадение с первой парой скобок */
                 2 => "B.",     /* второе совпадение со второй парой скобок */
                 3 => "Miller"  /* второе совпадение с третьей парой скобок */,
)
```

Ниже приводится краткое описание двух способов упорядочения:

Тип	Флаг	Описание и пример
группировка	PREG_PATTERN_ORDER	Выполняется группировка сопоставимых частей каждого совпадения. `$all_matches[номер_пары_скобок][`**`номер_совпадения`**`]`
послойный	PREG_SET_ORDER	Все части одного совпадения хранятся вместе. `$all_matches[`**`номер_совпадения`**`][номер_пары_скобок]`

Функция preg_match_all и флаг PREG_OFFSET_CAPTURE

Функции `preg_match_all`, как и функции preg_match, можно передавать флаг PREG_OFFSET_CAPTURE, который превращает каждый оконечный элемент (leaf element) массива `$all_matches` в массив из двух элементов (текст совпадения и величина смещения в байтах), т. е. `$all_matches` превращается в массив массивов массивов (трехмерный массив). Если предполагается использовать оба флага — PREG_OFFSET_CAPTURE и PREG_SET_ ORDER, их можно объединить с помощью операции побитового «ИЛИ»:

```
preg_match_all($pattern, $subject, $all_matches
               PREG_OFFSET_CAPTURE | PREG_SET_ORDER);
```

Функция preg_match_all и именованные сохранения

При использовании именованных сохранений в массив `$all_matches` добавляются дополнительные элементы, индексы которых соответствуют именам сохранений (так же как и в случае с функцией preg_match ☞ 560). После вызова

```
$subject = "
Jack A. Smith
Mary B. Miller";

/* Отсутствие флага упорядочения
   подразумевает наличие флага PREG_PATTERN_ORDER */
preg_match_all('/^(?P<Given>\w+) (?P<Middle>\w\.) (?P<Family>\w+)$/m',
               $subject, $all_matches);
```

массив `$all_matches` будет заполнен следующими данными:

```
array
(
   0        => array ( 0 => "Jack A. Smith", 1 => "Mary B. Miller" ),
   "Given"  => array ( 0 => "Jack",     1 => "Mary" ),
   1        => array ( 0 => "Jack",     1 => "Mary" ),
   "Middle" => array ( 0 => "A.",       1 => "B." ),
   2        => array ( 0 => "A.",       1 => "B." ),
   "Family" => array ( 0 => "Smith",    1 => "Miller" ),
   3        => array ( 0 => "Smith",    1 => "Miller" )
)
```

Тот же самый пример с использованием флага PREG_SET_ORDER:

```
$subject = "
Jack A. Smith
Mary B. Miller";

preg_match_all('/^(?P<Given>\w+) (?P<Middle>\w\.) (?P<Family>\w+)$/m',
               $subject, $all_matches, PREG_SET_ORDER);
```

В результате массив `$all_matches` будет заполнен следующими данными:

```
array
(
   0   => array ( 0 => "Jack A. Smith",
                  Given => "Jack",
                  1 => "Jack",
                  Middle => "A.",
                  2 => "A.",
                  Family => "Smith",
                  3 => "Smith" ),
   1   => array ( 0 => "Mary B. Miller",
                  Given => "Mary",
                  1 => "Mary",
                  Middle => "B.",
                  2 => "B.",
                  Family => "Miller",
                  3 => "Miller" )
)
```

Лично я предпочел бы, чтобы при использовании именованных сохранений элементы с числовыми индексами не включались в массив с результатами, так как это позволило бы повысить эффективность, а результаты выглядели бы более логично. Но поскольку они все-таки существуют, вам остается просто игнорировать их, если вы их не используете.

preg_replace

Синтаксис

```
preg_replace(шаблон, замена, текст [, ограничение [, количество ]])
```

Описание аргументов

шаблон	Аргумент шаблон — это регулярное выражение, окруженное разделителями, с необязательными модификаторами. *Шаблоном* также может быть массив, содержащий строки шаблонов
замена	Строка замены или, если *шаблон* является массивом, аргумент *замена* также может быть представлен массивом строк замены. При наличии модификатора шаблона **e** строка (или строки) интерпретируется как программный код на языке PHP (☞ 572)
текст	Целевая строка, в которой производится поиск. Этот аргумент также может быть представлен массивом строк (которые обрабатываются по очереди)
ограничение	Необязательное целое число, ограничивающее максимальное число замен (☞ 573)
количество	Необязательная переменная, в которой возвращается число фактически выполненных замен (только в PHP 5 ☞ 574)

Возвращаемое значение

Если *текст* представлен единственной строкой, возвращаемое значение также является строкой (копия аргумента *текст*, возможно, измененная). Если *текст* представлен массивом строк, возвращаемое значение также является массивом (который содержит строки аргумента *текст*, возможно, измененные).

Пояснение

PHP предлагает несколько способов выполнения поиска с заменой в тексте. В случае, когда искомая часть может быть описана простой строкой, наилучшим вариантом будет использование функции `str_replace` или `str_ireplace`, для более сложных случаев верным выбором будет применение `preg_replace`.

В качестве простого примера разберем случай ввода номера кредитной карты или телефона в HTML-форме. Как часто вам приходилось видеть на подобных формах инструкцию к вводу: «дефисы и пробелы не вводить»? Не кажется ли вам странным такое глупое (но, по общему мнению, незначительное) предупреждение, возлагающее бремя ответственности на пользователя, в то время как для программиста не составит никакого труда позволить клиенту вводить информацию в ее естествен-

ном представлении, *вместе* с дефисами или другими знаками пунктуации?[1] В конце концов, это типичная задача «проверки» пользовательского ввода:

```
$card_number = preg_replace('/\D+/', '', $card_number);
/* после этого $card_number будет содержать только цифры или не будет
   содержать ничего */
```

В данном случае удаляются все нецифровые символы. Говоря буквально, в этом примере с помощью `preg_replace` создается копия строки `$card_number`, в которой все последовательности нецифровых символов заменяются «ничем» (пустой строкой), после чего копия, возможно, измененная, записывается обратно в переменную `$card_number`.

Простейший вариант использования preg_replace с одной строкой текста, одним шаблоном и одной строкой замены

Первые три аргумента (*шаблон*, *замена* и *текст*) могут быть представлены либо строками, либо массивами строк. В простейшем случае, когда все три аргумента представлены простыми строками, `preg_replace` создает копию целевого текста, отыскивает в нем первое совпадение с шаблоном, заменяет его копией строки замены, после чего повторяет попытки поиска последующих совпадений, пока не будет достигнут конец строки.

Внутри строки замены комбинация '`$0`' означает текст всего совпадения, '`$1`' — текст совпадения с первой парой сохраняющих круглых скобок, '`$2`' — со второй парой и т. д. Обратите внимание: здесь комбинация '*$число*' не является именем переменной, как в других языках программирования, это лишь простая комбинация символов, которая распознается и интерпретируется особым образом только функцией `preg_replace`. Кроме того, можно использовать форму с использованием фигурных скобок, окружающих число, например '`${0}`' или '`${1}`', что позволяет ликвидировать неоднозначность интерпретации, когда вслед за данной комбинацией сразу же следует число.

Ниже приводится простой пример, в котором все слова, состоящие только из заглавных символов, в HTML-странице окружаются тегом `<b>…</b>`:

```
$html = preg_replace('/\b[A-Z]{2,}\b/', '<b>$0</b>', $html);
```

В случае использования модификатора шаблона **e** (что допускается только в функции `preg_replace`), строка замены интерпретируется как программный код на языке PHP, который исполняется для каждого найденного совпадения, а полученный

[1] Определенно, в Сети полно ленивых программистов, о чем печально свидетельствует зал позора «No Dashes Or Spaces» моего брата. Увидеть ее можно по адресу: *http://www.unixwiz.net/ndos-shame.html*.

результат используется в качестве строки замены. Ниже приводится расширенная версия предыдущего примера, где символы слов, заключенных в теги …, преобразуются в нижний регистр:

```
$html = preg_replace('/\b[A-Z]{2,}\b/e',
                     'strtolower("<b>$0</b>")',
                     $html);
```

Например, если было найдено совпадение со словом 'HEY', это слово вставляется в строку замены на место сохраняющей ссылки '$0'. В результате строка замены приобретает вид: 'strtolower("HEY")', после чего она исполняется как программный код PHP, а полученный результат — 'hey' — используется как замещающий текст.

При наличии модификатора **e** сохраняющая ссылка в строке замены интерполируется особым образом: внутри интерполированной строки экранируются все кавычки и апострофы. Без такой специальной обработки кавычка внутри интерполированной строки может превратить получившуюся строку в ошибочный фрагмент программного кода PHP.

Если при использовании модификатора шаблона **e** в строке замены выполняется обращение к внешним переменным, тогда лучше будет заключать строку замены в апострофы, чтобы избежать преждевременной интерполяции этих переменных.

Следующий пример по своему действию напоминает встроенную в PHP функцию htmlspecialchars():

```
$replacement = array ('&' => '&amp;',
                      '<' => '&lt;',
                      '>' => '&gt;',
                      '"' => '&quot;');

$new_subject = pregRreplace('/[&<">]/eS', '$replacement["$0"]', $subject);
```

В этом примере очень важно заключить строку *замены* в апострофы, чтобы скрыть переменную $replacement от интерпретатора PHP до того момента, пока строка замены не будет обработана функцией preg_replace как программный код. Если бы строка замены была окружена кавычками, PHP выполнил бы интерполяцию переменной еще *до того*, как строка была бы передана функции preg_replace.

Модификатор шаблона **S** используется для повышения производительности (☞ 597).

Если функции preg_replace передается четвертый аргумент *ограничение*, то будет выполнено не более заданного числа замен (для каждого регулярного выражения,

для каждой строки — подробности в следующем разделе). Значение по умолчанию 1, которое означает «ограничение отсутствует».

Если функции передается пятый аргумент *количество* (не поддерживается в PHP 4), он должен представлять переменную, в которую `preg_replace` запишет число фактически произведенных замен. Чтобы просто узнать, были ли произведены замены, достаточно сравнить первоначальный целевой текст с полученным результатом, однако гораздо эффективнее будет выполнить проверку значения аргумента *количество*.

Несколько строк текста, шаблонов и строк замены

В предыдущем разделе уже говорилось, что аргумент *текст* часто представляет собой простую строку, как это было продемонстрировано в примерах выше. Однако целевой текст может быть представлен массивом строк. В этом случае поиск производится поочередно в каждой из строк, а в качестве результата возвращается массив строк.

Независимо от того, является целевой текст строкой или массивом строк, аргументы *шаблон* и *замена* также могут быть представлены массивами строк. Ниже приводится список возможных комбинаций и поясняется, что происходит в каждом случае.

Шаблон	Замена	Что происходит
строка	**строка**	Шаблон применяется к целевому тексту, каждое совпадение замещается строкой замены
массив	**строка**	Каждый из шаблонов поочередно применяется к целевому тексту, каждое совпадение замещается строкой замены
массив	массив	Каждый из шаблонов поочередно применяется к целевому тексту, каждое совпадение замещается строкой замены, соответствующей шаблону
строка	массив	Недопустимая комбинация

Если аргумент *текст* является массивом, действия, описанные в таблице выше, выполняются поочередно над каждой строкой целевого текста, а в качестве результата возвращается массив строк.

Обратите внимание: аргумент *ограничение* ограничивает число замен отдельно для каждого шаблона и каждой строки целевого текста, а не для всех шаблонов и всех строк целевого текста. Тем не менее в переменной, которая передается в качестве аргумента *количество*, возвращается *общее* число замен, произведенных во всех строках.

Ниже приводится пример вызова функции `preg_replace`, которой в качестве аргументов *шаблон* и *замена* передаются массивы строк. По своему действию этот вызов аналогичен встроенной в PHP функции `htmlspecialchars()`, которая выполняет «подготовку» текста к включению в HTML:

```
$cooked = preg_replace(
    /* Совпадения с…   */ array('/&/',   '/</', '/>/',  '/"/' ),
    /* Заменить на…    */ array('&amp;', '&lt;', '&gt;', '&quot;'),
    /* В копии текста… */ $text
);
```

Если в этом примере в переменную `$text` записать строку:

```
AT&T --> "baby Bells"
```

в переменную `$cooked` будет записан текст:

```
AT&amp;T &gt; &quot;baby Bells&quot;
```

Разумеется, можно создать аргументы-массивы заранее. Следующая версия работает совершенно идентично (и возвращает идентичные результаты):

```
$patterns   = array('/&/', '/</',  '/>/',  '/"/' );
$replacements = array('&amp;', '&lt;', '&gt;', '&quot;');

$cooked = preg_replace($patterns, $replacements, $text);
```

Очень удобно, что функция `preg_replace` может принимать массивы строк в качестве аргументов (это позволяет избежать необходимости писать в программе циклы для обхода шаблонов и строк целевого текста), но она не несет в себе никакой дополнительной функциональности. Она, к примеру, не выполняет «параллельную» обработку. Однако внутренняя реализация функции выполняет обработку данных быстрее, чем циклы на языке PHP, и к тому же при ее использовании программный код становится проще читать.

Рассмотрим еще один поучительный пример, в котором все аргументы являются массивами:

```
$result_array = preg_replace($regex_array, $replace_array, $subject_array);
```

По своему действию этот вызов сопоставим со следующим фрагментом:

```
$result_array = array();
foreach ($subject_array as $subject)
{
    reset($regex_array); // Подготовка к обходу в цикле двух массивов,
    reset($replace_array); // в их внутреннем порядке следования элементов
```

```
    while (list(,$regex) = each($regex_array))
    {
        list(,$replacement) = each($replace_array);
        // Регулярное выражение и строка замены готовы,
        // можно применить их к строке целевого текста
        $subject = preg_replace($regex, $replacement, $subject);
    }
    // Все регулярные выражения были использованы
    // для обработки этой строки целевого текста
    $result_array[] = $subject; // ... добавить ее в массив результатов.
}
```

Порядок следования элементов в аргументах-массивах

Когда оба аргумента, *шаблон* и *замена*, представлены массивами, каждый элемент одного массива соответствует одному элементу другого во внутреннем порядке их следования, т. е. в том порядке, в каком они добавлялись в массивы (элемент, добавленный первым в массив *шаблон*, соответствует элементу, добавленному первым в массив *замена*, и т. д.). Это означает, что «литеральные массивы», созданные и заполненные с помощью функции `array()`, как в следующем примере, обладают корректным порядком следования элементов:

```
$subject = "this has 7 words and 31 letters";

$result = preg_replace(array('/[a-z]+/', '/\d+/'),
                       array('word<$0>', 'num<$0>'),
                       $subject);

print "result: $result\n";
```

Регулярное выражение `/[a-z]+/` соответствует строке замены '`words<$0>`', а выражение `/\d+/` — строке '`num<$0>`', что в результате дает:

```
result: word<this> word<has> num<7> word<words> word<and> num<31>
word<letters>
```

С другой стороны, если массивы *шаблон* и *замена* создавались по частям в течение некоторого времени, внутренний порядок следования элементов в массиве может стать отличным от порядка следования ключей (т. е. от порядка, определяемого числовыми значениями ключей). По этой причине во фрагменте, имитирующем функцию `preg_replace` на предыдущей странице, для обхода элементов массивов используется конструкция `each`, которая гарантирует выборку элементов во внутреннем порядке следования в массиве независимо от значений ключей.

Если внутренний порядок следования элементов в массивах *шаблон* и *замена* отличается от желаемого, можно воспользоваться функцией `ksort()`, чтобы привести внутренний порядок следования элементов в соответствие с порядком следования их ключей.

Когда оба аргумента *шаблон* и *замена* представлены массивами, но в массиве *шаблон* больше элементов, чем в массиве *замена*, в качестве недостающих строк замены используются пустые строки.

Порядок следования элементов в массиве *шаблон* имеет большое значение, так как шаблоны применяются в том порядке, в каком они будут обнаружены в массиве. Например, каким получится результат, если порядок следования элементов в массиве *шаблон* (и соответственно в массиве *замена*) изменить на противоположный? То есть каким будет результат работы следующего фрагмента?

```
$subject = "this has 7 words and 31 letters";
$result = preg_replace(array('/\d+/',  '/[a-z]+/'),
                       array('num<\0>', 'word<\0>'),
                       $subject);
print "result: $result\n";
```

❖ Переверните страницу, чтобы узнать ответ.

preg_replace_callback

Синтаксис

`preg_replace_callback(`*`шаблон, функция, текст`* `[,` *`ограничение`* `[,` *`количество`* `]])`

Описание аргументов

шаблон	Аргумент *шаблон* — это регулярное выражение, окруженное разделителями, с необязательными модификаторами (☞ 552). *Шаблоном* также может быть массив, содержащий строки шаблонов
функция	Функция обратного вызова, которая будет вызываться для каждого найденного совпадения, чтобы сгенерировать текст замены
текст	Целевая строка, в которой производится поиск. Этот аргумент также может быть представлен массивом строк (которые обрабатываются по очереди)
ограничение	Необязательное целое число, ограничивающее максимальное число замен (☞ 573)
количество	Необязательная переменная, в которой возвращается число фактически выполненных замен (только в PHP 5.1.0 ☞ 574)

ОТВЕТ

❖ *Ответ на вопрос со с. 577.*

Фрагмент со с. 577 выведет следующее (разбито на две строки, чтобы уместить на странице книги):

```
result: word<this> word<has> word<num><7> word<words>
word<and> word<num><31> word<letters>
```

Если два фрагмента, выделенные жирным шрифтом, оказались для вас сюрпризом, вспомните, что `preg_replace` с несколькими регулярными выражениями (т. е., когда в виде аргумента *шаблон* передается массив) обрабатывает шаблоны не «параллельно», а по очереди.

В данном примере первая комбинация *шаблон/замена* добавит в целевой текст две последовательности `num<…>`, после чего слово '`num`' будет обнаружено вторым шаблоном, и каждое слово '`num`' превратится в '`word<num>`', что и приводит к, возможно, неожиданным результатам.

Мораль этой истории заключается в том, что нужно проявлять особую осторожность в выборе порядка следования шаблонов при использовании многошаблонной версии `preg_replace`.

Возвращаемое значение

Если *текст* представлен единственной строкой, возвращаемое значение также является строкой (копия аргумента *текст*, возможно, измененная). Если *текст* представлен массивом строк, возвращаемое значение также является массивом (который содержит строки аргумента *текст*, возможно, измененные).

Пояснение

По своему поведению функция `preg_replace_callback` похожа на `preg_replace`, за исключением того, что вместо строки (или массива строк) замены `preg_replace_callback` принимает функцию обратного вызова. Она больше похожа на `preg_replace`, использующую модификатор шаблона `e` (☞ 572), но отличается более высокой эффективностью (и легче воспринимается при чтении исходных текстов, чем сложное выражение в строке замены).

Более подробную информацию о *функциях обратного вызова* можно найти в документации PHP, но в двух словах замечу, что функции обратного вызова вызываются в предопределенных ситуациях, с предопределенными значениями аргументов и возвращают значение для предопределенного использования. В случае с `preg_replace_callback` обращение к функции обратного вызова производится при каждой успешной попытке найти совпадение, и ей передается единственный аргумент (соответствует массиву `$matches`). Функция должна вернуть значение, которое будет использовано в качестве строки замены.

Определить значение аргумента *функция* можно тремя способами. Первый: просто указать имя требуемой функции как обычную строку. Второй: определить анонимную функцию с помощью встроенной в PHP функции `create_function`. Ниже будут приведены примеры обоих способов определения аргумента функция, Третий способ определения аргумента функция, который нигде в книге больше не встречается, использующий особенности объектно-ориентированного программирования, заключается в использовании массива из двух элементов (имя класса и имя вызываемого метода).

Ниже приводится пример со с. 573, переписанный для использования функции `preg_replace_callback` и функции обратного вызова. В виде аргумента *функция* здесь передается строка с именем функции:

```
$replacement = array ('&' => '&amp;',
                      '<' => '&lt;',
                      '>' => '&gt;',
                      '"' => '&quot;');
/*
* В аргументе $matches хранится информация о найденном совпадении,
* где $matches[0] - это текстовый символ, который требуется
* преобразовать в HTML-эквивалент. Возвращаемое значение – соответствующая
* строка в формате HTML. Так как данная функция используется в строго
* контролируемом контексте, мы совершенно спокойно можем использовать входные
* аргументы без дополнительной проверки.
*/
function text2html_callback($matches)
{
    global $replacement;
    return $replacement[$matches[0]];
}
$new_subject = preg_replace_callback('/[&<">]/S',  /* шаблон */
                          "text2html_callback", /* функция обратного вызова */
                           $subject);
```

Если в переменной `$subject` передать текст

```
"AT&T" sounds like "ATNT"
```

в переменную `$new_subject` будет записана строка:

```
&quot;AT&amp;T&quot; sounds like &quot;ATNT&quot;
```

В данном примере `text2html_callback` — это самая обычная функция PHP, предназначенная для использования в качестве функции обратного вызова в функции `preg_replace_callback`, которая вызывает `text2html_callback` и передает ей единственный аргумент — массив `$matches` (конечно, вы можете выбрать любое другое имя, но я предпочитаю следовать соглашениям по использованию имени `$matches`).

Для полноты картины я приведу этот же пример, где используется анонимная функция (созданная с помощью встроенной в PHP функции `create_function`). В данной версии предполагается, что переменная `$replacement` содержит тот же текст, что и ранее. Тело самой функции обратного вызова не претерпело никаких изменений, только на этот раз это не *именованная* функция, и ее сможет вызвать только функция `preg_replace_callback`:

```
$new_subject = preg_replace_callback('/[&<">]/S',
                create_function('$matches',
                                'global $replacement;
                                 return $replacement[$matches[0]];'),
                $subject);
```

Функция обратного вызова и модификатор шаблона e

Для решения простых задач проще использовать модификатор шаблона **`e`** в паре с функцией `preg_replace`, чем функцию `preg_replace_callback`. Однако когда важна высокая эффективность, помните, что при использовании модификатора шаблона **`e`** всякий раз, когда будет найдено совпадение, аргумент *замена* заново интерпретируется как программный код PHP. Это может привести к увеличению накладных расходов, избежать которых можно при использовании `preg_replace_callback` (аргумент *функция* интерпретируется только один раз).

preg_split

Синтаксис

`preg_split(`*`шаблон, текст`* `[, ` *`ограничение`*`, [ ` *`флаги`* `]])`

Описание аргументов

шаблон	Аргумент *шаблон* — это регулярное выражение, окруженное разделителями, с необязательными модификаторами (☞ 552)
текст	Целевая строка, которую необходимо разделить на части
ограничение	Необязательное целое число, ограничивающее число частей, на которые будет разделен *текст*

флаги	Необязательные флаги, влияющие на некоторые аспекты поведения функции. Допустимыми считаются следующие флаги и любые их комбинации:
	`PREG_SPLIT_NO_EMPTY`
	`PREG_SPLIT_DELIM_CAPTURE`
	`PREG_SPLIT_OFFSET_CAPTURE`
	Обсуждение этих флагов начнется на с. 583. Комбинирование флагов производится с помощью оператора побитового «ИЛИ» (как в примере на с. 569)

Возвращаемое значение

Массив строк.

Пояснение

Функция `preg_split` разбивает копию исходной строки на несколько частей и возвращает их в виде массива. Необязательный аргумент *ограничение* позволяет ограничить число получаемых частей (при этом последняя часть будет включать в себя остаток строки). С помощью флагов можно определить, какие части будут возвращаться и как.

В некотором смысле `preg_split` являет собой противоположность функции `preg_match_all`, поскольку `preg_split` выделяет части строки, которые *не совпали* с регулярным выражением. Говоря обычным языком, функция `preg_split` возвращает части строки, оставшиеся после того, как из нее были удалены последовательности символов, совпавшие с регулярным выражением. Функция `preg_split` представляет собой более мощный (основанный на использовании регулярных выражений) эквивалент простой функции `explode`.

В качестве простого примера рассмотрим форму поиска на финансовом сайте, которая принимает список биржевых котировок, разделенных пробелами. Для выделения отдельных котировок можно было бы использовать функцию `explode`:

```
$tickers = explode(' ', $input);
```

но такой подход не годится для случаев, когда пользователь по небрежности может вставить между биржевыми котировками два или более пробела. Более грамотным будет использовать функцию `preg_split` с регулярным выражением `/\s+/` в качестве разделителя:

```
$tickers = preg_split('/\s+/', $input);
```

Однако, несмотря на наличие инструкции к форме поиска, которая гласит: «разделять пробелами», некоторые пользователи чисто интуитивно могут разделять отдельные элементы запятыми (или запятыми с пробелами), вводя такие строки, как 'YHOO, MSFT, GOOG'. Такая ситуация разрешается очень просто:

```
$tickers = preg_split('/[\s,]+/', $input);
```

В примере с вводом приведенной выше строки в переменную `$tickers` будет записан массив из трех элементов: 'YHOO', 'MSFT' и 'GOOG'.

Следуя той же логике, если входная строка содержит «теги», разделенные запятыми (а-ля «Web 2.0»), можно было бы использовать выражение `/\s*,\s*/`, чтобы сделать допустимыми пробелы с обеих сторон от запятой:

```
$tags = preg_split('/\s*,\s*/', $input);
```

Весьма поучительным будет сравнить действие выражения ⌜\s*,\s*⌟ с ⌜[\s,]+⌟ в этих примерах. Первое выражение выполняет разбиение по запятым (запятая необходима для разбиения), но при этом удаляет все пробелы, которые могут встретиться с любой стороны от запятой. Для переменной `$input`, содержащей строку '`123,,,456`', совпадение с этим выражением будет найдено трижды (по одному для каждой запятой), а массив результата будет состоять из четырех элементов: '`123`', два пустых элемента и '`456`'.

С другой стороны, ⌜`[\s,]+`⌟ выполняет разбиение по любым последовательностям из запятых, пробелов или их комбинаций. В примере со строкой '`123,,,456`' будет найдено одно совпадение с тремя идущими подряд запятыми, а массив результата будет содержать два элемента: '`123`' и'`456`'.

Аргумент ограничение

Аргумент *ограничение* сообщает функции `preg_split`, что она не должна разбивать исходную строку на количество частей большее, чем задано. Если заданное число частей было достигнуто раньше, чем функция успела дойти до конца строки, оставшаяся часть строки помещается в последнюю часть.

В качестве примера рассмотрим процесс разбора HTTP-ответа сервера вручную. В соответствии с требованиями стандарта заголовок должен отделяться от тела ответа последовательностью из четырех символов '`\r\n\r\n`', но на практике встречаются серверы, которые используют для этих целей последовательность '`\n\n`'. К счастью, `preg_split` упрощает обработку подобных ситуаций. Предположим, что ответ сервера хранится в переменной `$response`,

```
$parts = preg_split('/\r? \n \r? \n/x', $response, 2);
```

в результате заголовок ответа помещается в `$parts[0]`, а тело — в `parts[1]`. (Модификатор шаблона **S** используется для повышения эффективности ☞ 597.)

Третий аргумент, со значением `2`, говорит о том, что исходная строка должна быть разбита не более чем на две части. Если совпадение будет найдено, часть перед совпадением (а нам заранее известно, что это заголовок) станет первым элементом возвращаемого значения. Вторым элементом станет «остальная часть строки» (т. е. тело ответа), а поскольку будет достигнуто максимально возможное число частей, она останется нетронутой и станет вторым элементом возвращаемого значения.

Без ограничения числа частей (или со значением `-1` в аргументе *ограничение*, что то же самое) `preg_split` разобьет целевую строку на столько частей, на сколько потребуется, что приведет к разделению тела ответа на несколько частей. Установка ограничения вовсе не гарантирует, что полученный в результате массив будет содержать именно такое число элементов, но гарантирует, что число элементов массива не будет *превышать* указанного ограничения (хотя ниже в разделе с опи-

санием флага PREG_SPLIT_DELIM_CAPTURE рассматривается ситуация, когда это ограничение не соблюдается).

Существуют две ситуации, когда есть смысл устанавливать искусственное ограничение. С первой ситуацией мы уже познакомились: когда необходимо, чтобы последний элемент содержал «оставшуюся часть строки». В предыдущем примере после выделения первой части (заголовка) нам больше не нужно разбивать остальную часть строки (тело). Поэтому чтобы оставить тело ответа в неприкосновенности, было задано ограничение `2`.

Кроме того, ограничение можно устанавливать в ситуациях, когда заранее известно, что для работы потребуются не все части строки, которые могут быть созданы функцией `preg_split`. Например, представим, что переменная `$data` хранит строку, состоящую из множества полей, разделенных `/\s*,\s*/` (пусть это будут поля «имя», «адрес», «возраст» и т. д.), но для работы вам нужны только первые два. В этом случае можно было бы передать в аргументе *ограничение* число 3, чтобы остановить разбиение строки после того, как будут выделены первые два поля:

```
$fields = preg_split('/ \s*,\s*/x', $data, 3);
```

Эта команда поместит оставшуюся часть строки в третий элемент массива, который затем можно будет удалить с помощью функции `array_pop` или просто игнорировать.

Если предполагается передавать функции `preg_split` какие-либо флаги (которые обсуждаются в следующем разделе) и не ограничивать число получающихся частей, необходимо в аргументе *ограничение* передать значение `-1`, которое означает «без ограничения». С другой стороны, значение `1` в аргументе *ограничение* означает «без разбиения» и потому совершенно бесполезно. Смысл значения `0` или отрицательного значения, отличного от `-1`, явно не определен и потому не следует их использовать.

Флаги функции preg_split

Функция `preg_split` поддерживает три флага, которые оказывают влияние на ее поведение. Они могут использоваться отдельно или комбинироваться с помощью оператора побитового «ИЛИ» (пример приводится на с. 569).

PREG_SPLIT_OFFSET_CAPTURE

По аналогии с флагом `PREG_OFFSET_CAPTURE`, который используется с функциями `preg_match` и `preg_match_all`, использование этого флага приводит к тому, что каждый элемент массива результата превращается в массив из двух элементов — строку и смещение.

PREG_SPLIT_NO_EMPTY

Этот флаг заставляет `preg_split` игнорировать пустые строки, не возвращать их в массиве результата и не учитывать их при наличии ограничения на число частей разбиения. Пустые строки обычно появляются при наличии совпадений в самом начале и в самом конце целевой строки или при наличии «ничего» между двумя соседними совпадениями.

Возвращаясь к примеру с тегами «Web 2.0» (☞ 581), если переменная `$input` будет содержать текст '`party,,fun`', то в результате выполнения команды

```
$tags = preg_split('/ \s* , \s*/x', $input);
```

в переменную `$tags` будет записано три строки: '`party`', пустая строка и '`fun`'. Пустая строка в результатах соответствует пустой строке между двумя совпадениями с запятыми в целевой строке.

Если переписать этот пример с использованием флага `PREG_SPLIT_NO_ EMPTY`,

```
$tags = preg_split('/ \s* , \s*/x', $input, -1, PREG_SPLIT_NO_EMPTY);
```

в массиве результата будут присутствовать только строки '`party`' и'`fun`'.

PREG_SPLIT_DELIM_CAPTURE

При использовании этого флага в массив с результатами включается текст, совпавший с сохраняющими круглыми скобками регулярного выражения, выполняющего разбиение. Рассмотрим простой пример: предположим, что нам необходимо выполнить разбор строки из названий, в которой союзы '`and`' и '`or`' используются для их связки, например:

```
DLSR camera and Nikon D200 or Canon EOS 30D
```

Без использования флага `PREG_SPLIT_DELIM_CAPTURE` фрагмент

```
$parts = preg_split('/ \s+ (and|or) \s+ /x', $input);
```

запишет в переменную `$parts` следующий массив:

```
array ('DLSR camera', 'Nikon D200', 'Canon EOS 30D')
```

Все, что распознается как разделители, удаляется. Однако при наличии флага `PREG_SPLIT_DELIM_CAPTURE` (и со значением -1 в аргументе *ограничение*):

```
$parts = preg_split('/ \s+ (and;or) \s+ /x', $input, -1,
                    PREG_SPLIT_DELIM_CAPTURE);
```

в переменную $parts включаются разделители, совпавшие с круглыми скобками:

```
array ('DLSR camera', 'and', 'Nikon D200', 'or', 'Canon EOS 30D')
```

В этом случае для каждого разделителя в массив результата добавляется по одному элементу, совпавшему с круглыми скобками в регулярном выражении. После этого можно сделать обход элементов $parts и выполнить специальную интерпретацию элементов 'and' и 'or'.

Важно отметить, что при использовании несохраняющих круглых скобок (например, когда шаблон будет представлен строкой '/\s+ (?:and|or)\s+/') флаг PREG_SPLIT_DELIM_CAPTURE не будет оказывать никакого воздействия, поскольку он работает только при использовании сохраняющих круглых скобок.

Теперь вернемся к примеру с биржевыми котировками, который приводился на с. 581:

```
$tickers = preg_split('/[\s,]+/', $input);
```

Если в него добавить сохраняющие круглые скобки и флаг PREG_SPLIT_DELIM_CAPTURE,

```
$tickers = preg_split('/([\s,]+)/', $input, -1, PREG_SPLIT_DELIM_CAPTURE);
```

то из строки $input ничего не будет выброшено, она просто будет разбита на части в виде элементов массива $tickers. После этого при обработке массива $tickers вы будете знать, что элементы с нечетными индексами соответствуют выражению `([\s,]+)`. Такой прием может потребоваться, например, при отображении сообщения об ошибке, когда необходимо выполнить некоторую обработку отдельных частей, а затем опять объединить их для получения переработанной версии входной строки.

Кроме того, ограничение количества частей разбиения не распространяется на элементы, добавленные в массив результата вследствие использования флага PREG_SPLIT_DELIM_CAPTURE. Это единственный случай, когда количество элементов в массиве результата может превышать заданное ограничение на количество частей разбиения (причем превышать намного, если регулярное выражение содержит много сохраняющих круглых скобок).

Для завершающих круглых скобок, не участвовавших в совпадении, добавление соответствующих им элементов в массив результата не производится. Другими словами, пары сохраняющих круглых скобок, не участвовавших в окончательном совпадении (подробности на с. 561), могут добавлять (а могут и не добавлять) пустые строки в результирующий массив. Если за такой парой скобок имеется другая пара, которая *участвовала* в окончательном совпадении, то пустая строка добавляется, в противном

случае — нет. Обратите внимание: добавление флага `PREG_SPLIT_NO_EMPTY` ликвидирует эту проблему, так как он заставляет функцию `preg_split` игнорировать любые пустые строки независимо от природы их происхождения.

preg_grep

Синтаксис

`preg_grep(`*`шаблон, входной_массив`* `[,` *`флаги`* `])`

Описание аргументов

шаблон	Аргумент *шаблон* — это регулярное выражение, окруженное разделителями, с необязательными модификаторами
входной массив	Массив значений, которые будут скопированы в массив результата, если они совпадают с *шаблоном*
флаги	Необязательное значение `PREG_GREP_INVERT` или ноль

Возвращаемое значение

Массив, содержащий значения из аргумента *входной_массив*, совпавшие с *шаблоном* (или, наоборот, значения, **не** совпавшие с *шаблоном*, в случае использования флага `PREG_GREP_INVERT`).

Пояснение

Функция `preg_grep` используется для создания копии массива *входной_массив*, когда необходимо оставить только те элементы, значения которых совпадают (или не совпадают, при использовании флага `PREG_ GREP_INVERT`) с *шаблоном*. Значения ключей, ассоциированных с элементами, остаются неизменными.

Рассмотрим простой пример:

```
preg_grep('/\s/', $input);
```

который заполняет возвращаемый массив теми элементами из массива `$input`, которые содержат пробелы. Противоположную задачу решает команда:

```
preg_grep('/\s/', $input, PREG_GREP_INVERT);
```

Она заполняет возвращаемый массив теми элементами из массива `$in put`, которые не содержат пробелы. Обратите внимание: второй пример отличается от:

```
preg_grep('/^\S+$/', $input);
```

тем, что последний пример не включает в массив результата пустые элементы (строки с нулевой длиной).

preg_quote

Синтаксис

```
preg_quote(текст [, разделитель ])
```

Описание аргументов

текст	Строка, которая будет использоваться как последовательность литералов внутри аргумента *шаблон* (☞ 552)
разделитель	Необязательная односимвольная строка, которая будет использована в качестве разделителя при построении шаблона регулярного выражения

Возвращаемое значение

Функция `preg_quote` возвращает строку — копию аргумента *текст*, в которой будет выполнено экранирование всех метасимволов регулярных выражений. Если был передан аргумент *разделитель*, все его вхождения в строку также будут экранированы.

Обсуждение

Если имеется некоторая строка, которая должна использоваться внутри регулярного выражения в качестве литерального текста, тогда функция `preg_quote` поможет выполнить экранирование всех метасимволов регулярных выражений, которые могут содержаться в строке. Если дополнительно функции `preg_quote` передается символ, который предполагается использовать в качестве разделителей при создании шаблона, все вхождения этого символа также будут экранированы.

`preg_quote` — узкоспециализированная функция, которая используется достаточно редко, тем не менее ниже приводится пример ее использования:

```
/* Шаблон поиска сообщений электронной почты
   по заданной в $MailSubject теме сообщения */
$pattern = '/^Subject:\s+(Re:\s*)*' . preg_quote($MailSubject, '/') . '/mi';
```

Если предположить, что `$MailSubject` содержит такую строку:

```
**Super Deal** (Act Now!)
```

тогда переменная `$pattern` получит следующее значение:

```
/^Subject:\s+(Re:\s*)*\*\*Super Deal\*\* \(Act Now\!\)/mi
```

которое может использоваться как аргумент шаблон в preg-функциях.

Если в качестве разделителя указать символ '{', это не приведет к экранированию парного ему «закрывающего» символа (т. е. '}'), поэтому в подобных ситуациях не следует использовать парные разделители.

Кроме того, символы пропусков и '#' не экранируются, поэтому результат, скорее всего, будет непригоден для использования с модификатором **x**.

В таких ситуациях `preg_quote` может использоваться только для решения отдельных частей задачи представления произвольного текста в виде регулярного выражения. Она решает задачу представления «текста в регулярном выражении», но не выполняет шаг преобразования «регулярного выражения в шаблон», который мог бы использоваться в функциях семейства preg. Решение этой задачи будет представлено в следующем разделе.

«Недостающие» функции preg

Встроенные в PHP функции семейства preg предоставляют неплохой диапазон функциональности, но в ряде случаев ощущается нехватка некоторых возможностей. Один из примеров уже приводился в виде специализированной версии функции `preg_match` (☞ 565).

Другая область применений, где у меня возникала необходимость создавать свои собственные функции, касается ситуаций, когда регулярные выражения не могут быть представлены в тексте программы в виде литеральных строк шаблонов, а создаются на основе внешних данных (например, в результате чтения данных из файлов или на основе данных, поставляемых пользователем через веб-формы). Как будет показано в следующем разделе, задача преобразования строки регулярного выражения в соответствующий шаблон может оказаться довольно сложной.

Кроме того, перед использованием таких регулярных выражений далеко не лишним будет проверить корректность их синтаксиса. Этот вопрос мы также будем рассматривать.

Исходные тексты функций, описание которых начнется ниже, как и любые другие примеры программного кода в этой книге, доступны для загрузки с моего веб-сайта: *http://regex.info/*.

preg_regex_to_pattern

Если в вашем распоряжении имеется регулярное выражение в виде строки (возможно, полученное из файла с настройками или из веб-формы), которое предпо-

лагается использовать совместно с функциями семейства preg, его нужно окружить разделителями, чтобы превратить в соответствующий шаблон.

Задача

В большинстве случаев задача преобразования строки регулярного выражения в шаблон сводится к простому заключению этого регулярного выражения в символы слэша. При таком подходе, например, строка '`[a-z]+`' будет преобразована в строку '`/[a-z]+/`', пригодную для использования в качестве аргумента шаблона.

Однако преобразование осложняется наличием в регулярном выражении символов, которые используются в качестве разделителей. Например, если предположить, что строка регулярного выражения имеет вид '`^http://([^/:]+)`', то после простого заключения ее в символы слэша будет получен шаблон '`/^http://([^/:]+)/`', при использовании которого появится сообщение об ошибке «Unknown modifier /».

Как уже отмечалось во врезке на с. 558, это странное сообщение об ошибке появляется потому, что первый и второй символы слэша интерпретируются как разделители, а вся остальная часть строки (в данном случае часть строки, начинающаяся с третьего слэша) воспринимается как список модификаторов шаблона.

Решение

Существует два способа избежать появления конфликтов, связанных с внутренними разделителями. Один из них заключается в выборе такого символа разделителя, который отсутствует в регулярном выражении. Этот способ можно порекомендовать при составлении строки шаблона с модификаторами вручную. По этой причине я использовал `{...}` в качестве разделителей в примерах на с. 472, 560 и 561 (привожу лишь некоторые).

Однако совсем не просто (порой даже невозможно) выбрать разделитель, который отсутствовал бы в регулярном выражении, потому что текст может включать в себя все разделители или заранее может быть неизвестно, с каким текстом придется работать. Такое часто бывает, когда строка регулярного выражения создается программным путем, поэтому гораздо проще и легче использовать второй способ: выбрать определенный символ в качестве разделителя, а затем экранировать все вхождения этого символа в строке с регулярным выражением.

На самом деле реализация второго способа выглядит немного сложнее, чем может показаться на первый взгляд, потому что необходимо обратить внимание на некоторые важные детали. Например, особое внимание следует уделить экранированию в конце строки, чтобы не экранировать завершающий разделитель.

Ниже приводится исходный текст функции, которая принимает строку регулярного выражения и необязательную строку с модификаторами шаблона, а возвращает строку шаблона, готовую к использованию в функциях семейства preg. Лес из символов обратного слэша (экранированные символы в программном коде PHP и в строках регулярных выражений) — это одна из наиболее сложных для восприятия частей программного кода; этот пример — не самое легкое чтиво. (Если вам требуется освежить в памяти семантику строк в апострофах, обращайтесь к с. 552.)

```
/*
 * Принимает строку регулярного выражения (и, необязательно, строку
 * модификаторов шаблона), возвращает строку, пригодную к  использованию
 * в качестве шаблона с функциями семейства preg. Регулярное выражение
 * окружается разделителями, после чего к нему добавляется строка
 * с модификаторами.
 */
function preg_regex_to_pattern($raw_regex, $modifiers = "")
{
    /*
     * Чтобы преобразовать регулярное выражение в шаблон, необходимо
     * заключить исходную строку в разделители (здесь используются
     * символы слэша) и добавить к ней строку с модификаторами.
     * Необходимо также экранировать все вхождения разделителя
     * внутри строки с регулярным выражением и обратный слэш,
     * завершающий регулярное выражение, чтобы он не экранировал
     * замыкающий разделитель.
     *
     * Мы не должны вслепую экранировать все символы слэша в регулярном
     * выражении, потому что это может повредить регулярные выражения,
     * содержащие уже экранированные разделители. Например, если
     * регулярное выражение имеет вид '\/', слепое экранирование
     * превратит его в выражение '\\/', которое не будет работать
     * после заключения в разделители: '/\\//'.
     *
     * Поэтому для начала мы разобьем регулярное выражение на следующие
     * части: экранированные символы, неэкранированные слэши
     * (которые следует экранировать) и все остальное. Как частный
     * случай, нам необходимо проверить наличие символа обратного слэша
     * в конце регулярного выражения и экранировать его.
     */
  if (! preg_match('{\\\\(?:/|$)}', $raw_regex)) /* '\', за которым следует
                                                    '/' или конец строки */
  {
    /* Экранированные символы слэша отсутствуют, в конце строки
     * нет символа обратного слэша, поэтому здесь можно слепо
     * экранировать все символы слэша */
    $cooked = preg_replace('!/!', '\/', $raw)regex);
  }
  else
```

```
  {
    /* Следующий шаблон используется для разбора $raw_regex.
     * Две части, совпадения с которыми необходимо экранировать,
     * заключены в сохраняющие круглые скобки. */
    $pattern = '{ [^\\\\/]+ | \\\\. | ( / | \\\\$ ) }sx';

    /* Для каждого найденного совпадения $pattern в raw-redex
    /* вызывается функция обратного вызова. Если элемент $matches[1]
    * содержит непустое значение, возвращается его экранированная
    * версия. В противном случае возвращается неизмененная
    * копия элемента. */
   $f = create_function('$matches', '     // Это длинная
        if (empty($matches[1]))           // строка в апострофах,
             return $matches[0];          // которая превратится
        else                              // в программный код
             return "\\\\" . $matches[1]; // функции.
        ');
         * Применение шаблона $pattern к регулярному выражению $raw_regex,
         * и заполнение $cooked */
        $cooked = preg_replace_callback($pattern, $f, $rawRregex);
    }
    /* Теперь содержимое $cooked можно обернуть разделителями, добавить
     * модификаторы и вернуть */
    return "/$cooked/$modifiers";
}
```

Для решения поставленной задачи пришлось написать больше программного кода, чем я хотел бы вводить всякий раз, когда в этом появится необходимость. Именно по этой причине я оформил его в виде функции (одной из тех, которые я хотел бы видеть в составе встроенных функций семейства preg).

Весьма поучительно повнимательнее рассмотреть регулярное выражение, используемое при вызове функции `preg_replace_callback`, и разобраться с тем, как работают это выражение и функция обратного вызова в процессе обхода строки шаблона, чтобы экранировать неэкранированные символы слэша и оставить в неприкосновенности экранированные символы.

Проверка синтаксиса неизвестного шаблона

После того как регулярное выражение будет заключено в разделители, его можно без опаски использовать в качестве аргумента шаблона, но это не гарантирует синтаксической правильности исходного регулярного выражения.

Например, если первоначально строка регулярного выражения имела вид '`*.txt`' — возможно, просто потому, что кто-то вместо регулярного выражения случайно ввел шаблон имени файла (☞ 28), то результатом работы `preg_regex_to_pattern` будет

строка /*.txt/. Данная строка не является правильным регулярным выражением, поэтому попытка поиска совпадений будет терпеть неудачу с предупреждением (если включен режим вывода предупреждений):

```
Compilation failed: nothing to repeat at offset 0
```

В PHP отсутствует встроенная функция проверки аргумента шаблона на синтаксическую корректность регулярного выражения, однако я могу предложить вашему вниманию один из вариантов.

Функция preg_pattern_error проверяет шаблон простой попыткой использовать его — это строка с вызовом preg_match в середине функции. Остальная часть функции сосредоточена на решении административных задач по перехвату сообщения об ошибке, которая может возникнуть в функции preg_match.

```
/*
 *Возвращает сообщение об ошибке, если заданный шаблон или составляющее
 *его регулярное выражение содержат синтаксическую ошибку.
 *В противном случае (если ошибок не обнаружено) возвращается false.
 */
function preg_pattern_error($pattern)
{
   /* Чтобы определить корректность шаблона, мы просто попытаемся
    * использовать его. Определить и перехватить ошибку —
    * задача не из легких, особенно если желательно избежать побочных
    * влияний на глобальное состояние среды (например, на содержимое
    * переменной $php_errormsg). Для этого, если параметр
    * 'track_errors' включен, мы сохраним значение $php_errormsg
    * и по окончании проверки восстановим его. Если параметр
    * 'track_errors' выключен, мы включим его (это совершенно
    * необходимо), но перед завершением опять выключим.
    */
   if ($old_track = ini_get("track_errors"))
       $old_message = isset($php_errormsg) ? $php_errormsg : false;
   else
       ini_set('track_errors', 1);
   /* Теперь параметр track_errors включен. */
   unset($php_errormsg);
   @ preg_match($pattern, ""); /* испытание шаблона! */
   $return_value = isset($php_errormsg) ? $php_errormsg : false;

   /* Теперь у нас есть все, что нам нужно, и поэтому можно
    * восстановить прежнее состояние среды выполнения. */
   if ($old_track)
       $php_errormsg = isset($old_message) ? $old_message : false;
   else
       ini_set('track_errors', 0);
   return $return_value;
}
```

Проверка синтаксиса неизвестного регулярного выражения

В заключение ниже приводится функция, которая может использоваться для проверки имеющихся регулярных выражений (без разделителей и модификаторов). Если выражение содержит синтаксические ошибки, функция возвращает соответствующее сообщение об ошибке, в противном случае возвращается false.

```
/*
 * Возвращает сообщение об ошибке, если регулярное выражение
 * имеет неверный синтаксис.
 * В противном случае возвращается значение false.
 */
function preg_regex_error($regex)
{
    return preg_pattern_error(preg_regex_to_pattern($regex));
}
```

Рекурсивные регулярные выражения

Большинство аспектов механизмов регулярных выражений рассматривались в главе 3, однако данный диалект предлагает действительно интересный способ поиска совпадений с вложенными конструкциями — рекурсивные выражения.

Последовательность `(?R)` означает «рекурсивное применение всего выражения с данной позиции», а конструкция `(?`*число*`)` — «рекурсивное применение последовательности в сохраняющих круглых скобках с данным порядковым номером с данной позиции». Для версии с именованным сохранением используется конструкция `(?P>`*имя*`)`.

В следующих нескольких разделах будут продемонстрированы некоторые типичные случаи использования рекурсии. Кроме того, рекурсия занимает центральное место в улучшенном примере разбора «тегированных данных» (с. 601).

Поиск совпадений с вложенными круглыми скобками

Суть примера, использующего рекурсию, состоит в том, чтобы найти совпадения с вложенными парами круглых скобок. Один из способов использования рекурсии: `(?: [^()]++ | \( (?R) \) )*`.

Этому выражению соответствует любое число двух альтернатив. Первой альтернативе `[^()]++` соответствуют любые символы, за исключением круглых скобок.

Данная альтернатива использует захватывающую версию квантификатора +, чтобы избежать «бесконечного поиска» (☞ 289) из-за наличия объемлющей конструкции ⌜(?:…)*⌟.

Другая альтернатива, ⌜\((?R)\)⌟, — это наиболее интересная часть регулярного выражения. Второй альтернативе соответствует пара круглых скобок с любыми символами (в том числе и вложенные пары круглых скобок) между ними. «Любые символы между ними» — это как раз и есть то, что соответствует всему выражению, и поэтому мы просто можем использовать конструкцию ⌜(?R)⌟ для рекурсивного применения всего выражения.

Само по себе выражение прекрасно справляется с возложенной на него задачей, однако при добавлении в него новых конструкций следует проявлять особую осторожность, так как все, что будет в него добавлено, также будет применяться рекурсивно благодаря наличию конструкции ⌜(?R)⌟.

В качестве примера рассмотрим использование этого выражения для проверки всей строки на наличие непарных круглых скобок. Слишком велико искушение обернуть выражение в конструкцию ⌜^…$⌟, чтобы обеспечить проверку именно «всей строки». Однако это было бы ошибкой, потому что добавление якорных метасимволов начала и конца строки приведет к неудачной попытке рекурсивного поиска совпадений в середине строки.

Рекурсивные ссылки в сохраняющих круглых скобках

Конструкция ⌜(?R)⌟ создает рекурсивную ссылку на все регулярное выражение, но также, с помощью конструкции ⌜(?*число*)⌟, можно создать ссылку на определенное подвыражение. Конструкция ⌜(?*число*)⌟ позволяет создать рекурсивную ссылку на подвыражение, содержащееся внутри сохраняющих круглых скобок с порядковым номером *число*[1]. Развивая логически идею конструкции ⌜(?*число*)⌟, можно сказать, что ⌜(?0)⌟ является синонимом конструкции ⌜(?R)⌟.

Подобная ограниченная ссылка может применяться для решения проблемы из предыдущего раздела: прежде чем добавить в выражение конструкцию ⌜^…$⌟, его необходимо заключить в сохраняющие круглые скобки, а ⌜(?R)⌟ заменить на ⌜(?1)⌟. Сохраняющие круглые скобки использованы для выделения подвыражения, на которое ссылается ⌜(?1)⌟, что в точности соответствует выражению из предыдущего раздела, которому соответствуют вложенные пары круглых скобок. Метасимволы

[1] Подчеркнутая часть регулярного выражения находится внутри первой пары сохраняющих круглых скобок, поэтому она повторно применяется всякий раз, когда достигается ⌜(?1)⌟.

⌜^...$⌟ добавляются за пределами этих скобок, благодаря чему удалось избежать их рекурсивного применения: ⌜^((?:[^()]++|\((?1)\))*)$⌟.

Подчеркнутая часть регулярного выражения находится внутри первой пары сохраняющих круглых скобок, поэтому она повторно применяется всякий раз, когда достигается ⌜(?1)⌟.

Это регулярное выражение используется во фрагменте программного кода PHP, приведенного далее, который проверяет некоторый текст в переменной `$text` на наличие непарных круглых скобок:

```
if (preg_match('/^ ( (?: [^()]++ | \( (?1) \) )* ) $/x', $text))
    echo "text is balanced\n";
else
    echo "text is unbalanced\n";
```

Рекурсивные ссылки в именованных сохранениях

Если необходимо обеспечить рекурсивный вызов подвыражения, заключенного в именованные круглые скобки (☞ 186), тогда для создания рекурсивной ссылки вместо конструкции (?*число*) можно использовать конструкцию (?P>*имя*). В этом случае наш пример приобретает такой вид:

⌜^(?P<stuff> (?: [^()]++|\((?P>stuff)\))*)$⌟

В таком виде выражение выглядит очень сложным, но с помощью модификатора шаблона x его легко можно сделать более удобочитаемым (☞ 556):

```
$pattern = '{
     # Начало регулярного выражения...
     ^
       (?P<stuff>
            # Все, что совпадает с этими скобками, именуется как "stuff."
            (?:
               [^()]++          # все, что не является круглыми скобками
             |
               \( (?P>stuff) \) # откр. скобка, "все остальное" и закр. скобка
            )*
       )
      $
      # Конец регулярного выражения.
}x';   # 'x' - это модификатор шаблона.
if (preg_match($pattern, $text))
    echo "text is balanced\n";
else
    echo "text is unbalanced\n";
```

Дополнительно о захватывающих квантификаторах

Заключительный комментарий по поводу использования захватывающих квантификаторов в первоначальной версии выражения. Если бы во внешней конструкции `(?:…)*` использовался захватывающий квантификатор, то во вложенной конструкции `[^()]++` можно было бы использовать не захватывающую версию квантификатора. Чтобы избежать «бесконечного поиска», достаточно, чтобы любой (или оба сразу) из двух квантификаторов был захватывающим. Если бы в этом диалекте регулярных выражений захватывающие квантификаторы и атомарная группировка (☞ 328) были недоступны, то пришлось бы вообще удалить квантификатор из первой альтернативы: `(?:[^()]|\((?R)\))*`.

Это выражение менее эффективно, но в нем по крайней мере невозможна ситуация «бесконечного поиска». Чтобы снова повысить эффективность, можно воспользоваться методикой раскрутки цикла, описываемой в главе 6 (☞ 330), что приводит к выражению `[^()]*(?:\((?R)\) [^()]*)*`.

Никаких возвратов в рекурсии

Очень важный аспект семантики рекурсии в диалекте регулярных выражений preg состоит в том, что все совпадения, найденные в ходе рекурсии, интерпретируются как совпадения с атомарной группировкой (☞ 327). Это означает, что если некоторые результаты рекурсивного поиска препятствуют достижению общего успеха, то этого и не произойдет (результатом будет общая неудача).

Примечание «некоторые» в предыдущем предложении очень важно, потому что весь текст совпадения, полученный в результате рекурсивного вызова, рассматривается как единое целое и не сможет совпасть в результате возврата. Все, что отвергается в процессе рекурсии, вызывает возврат к точке *внутри* рекурсивного вызова.

Совпадение с парой вложенных скобок

Выше мы разобрали пример проверки строки на наличие непарных круглых скобок. Теперь, для полноты картины, я покажу, как найти совпадение с парой круглых скобок (внутри которых, возможно, содержатся другие пары круглых скобок): `\((?:[^()]++|(?R))*\)`.

В этом примере используются те же ингредиенты, что и ранее, изменился только порядок их следования. Как уже говорилось, если данное выражение необходимо будет использовать как часть более крупного регулярного выражения, его необходимо будет обернуть в сохраняющие круглые скобки и заменить

⌜(?R)⌟ рекурсивной ссылкой на данное подвыражение, такой как ⌜(?1)⌟ (число должно соответствовать порядковому номеру сохраняющих скобок во всем выражении).

Вопросы эффективности в PHP

Функции семейства preg в языке PHP используют библиотеку PCRE — оптимизированный механизм регулярных выражений НКА. Благодаря этому имеется возможность непосредственного использования большинства приемов, рассматривавшихся в главах 4–6, включая хронометраж критически важных участков программного кода, которые не только в теории, но и на практике оказывают влияние на скорость исполнения. Пример проведения хронометража в PHP приводится в главе 6 (☞ 298).

Если скорость работы кода имеет большое значение, помните, что функции обратного вызова вообще работают быстрее, чем модификатор шаблона **e** (☞ 580), а именованное сохранение слишком длинных строк может привести к необходимости копирования слишком больших объемов данных.

Компиляция регулярных выражений производится в том порядке, в каком они встречаются в ходе выполнения программы, но PHP обладает весьма объемным кэшем на 4096 записей (☞ 308), так что на практике компиляция строк шаблонов производится только при первом обнаружении.

Отдельного обсуждения заслуживает модификатор шаблона **S**: он «изучает» («studies») регулярное выражение для более быстрого достижения совпадения. (Этот модификатор никакого отношения не имеет к функции `study` в языке Perl, которая работает с целевым текстом, а не с регулярным выражением ☞ 449).

Модификатор шаблона S: «Study»

Использование модификатора шаблона **S** сообщает механизму регулярных выражений о необходимости потратить дополнительное время[1] на анализ выражения перед его использованием в надежде на то, что эти затраты окупятся более высокой скоростью поиска. Этот модификатор может вообще не дать никакого прироста в скорости, но в некоторых случаях скорость может быть увеличена на порядок.

[1] В действительности — весьма незначительное время. Для очень длинных и сложных регулярных выражений при среднем уровне нагрузки на систему дополнительное время, затраченное на анализ выражения с модификатором `S`, не превышает стотысячной доли секунды.

В настоящее время четко определены ситуации, в каких использование модификатора **S** может, а в каких не может дать прирост в скорости: это дополняет то, что в главе 6 называется *оптимизацией исключения по первому классу* (☞ 314).

Для начала замечу, что скорость поиска выходит на первое место, только когда регулярное выражение предполагается применять к большим объемам текста. Модификатор шаблона **S** следует использовать только в тех случаях, когда одно и то же регулярное выражение применяется к объемным фрагментам текста или к большому количеству маленьких фрагментов.

Стандартные оптимизации без модификатора шаблона S

Рассмотрим простое выражение `⌜<(\w+)⌟`. Из этого регулярного выражения видно, что каждое совпадение должно начинаться с символа '<'. Механизм регулярных выражений может (а в случае механизма preg так и происходит) извлечь из этого обстоятельства дополнительное преимущество, предварительно отыскав символ '<' и применив регулярное выражение только с этой позиции в строке (поскольку совпадение должно начинаться с `⌜<⌟`, бессмысленно применять его в позициях с другими символами).

Простой поиск символа в строке может выполняться намного быстрее, чем применение полного регулярного выражения, за счет чего и достигается оптимизация. Причем чем реже встречается рассматриваемый символ в целевом тексте, тем больше получается выигрыш. Кроме того, чем больший объем работы приходится выполнить механизму регулярных выражений для определения неудачи, тем больший эффект дает оптимизация. Например, для выражения `⌜<i>|</i>|<b>|</b>⌟` выигрыш от оптимизации существенно выше, чем для выражения `⌜<(\w+)⌟`, потому что в первом случае механизму регулярных выражений нужно проверить четыре альтернативы, прежде чем двинуться дальше. Это достаточный объем работы, которого можно избежать.

Усиление оптимизации с помощью модификатора шаблона S

Механизм preg достаточно грамотно использует этого рода оптимизацию для большинства выражений, содержащих единичный символ, с которого должно начинаться совпадение. Однако модификатор **S** сообщает механизму регулярных выражений, что тот должен выполнить предварительный анализ выражений, которые допускают наличие совпадений с одним из нескольких начальных символов.

Ниже приводится несколько регулярных выражений, встречавшихся ранее в этой главе, которые могут быть оптимизированы с помощью модификатора шаблона **S** именно таким образом:

Регулярное выражение	Возможные начальные символы
<(\w+) | &(\w+);	< &
[Rr]e:	R r
(Jan|Feb| |Dec) \b	A D F J M N O S
(Re: \s*) ? SPAM	R S
\s*, \s*	\x09 \x0A \x0C \x0D \x20
[&<”>]	& < “ >
\r?\n\r?\n	\r \n

В каких ситуациях применение модификатора S бесполезно

Весьма поучительным будет рассмотреть типы регулярных выражений, для которых использование модификатора шаблона **S** не дает преимуществ:

- Выражения, начинающиеся с якорных метасимволов (таких, как ⌜^⌟ и ⌜\b⌟), или когда с якорного метасимвола начинается альтернатива верхнего уровня. Это ограничение текущей реализации, в частности, для метасимвола ⌜\b⌟, теоретически может быть преодолено в последующих версиях.
- Выражения, которые допускают наличие пустых совпадений, например ⌜\s*⌟.
- Выражения, которые допускают наличие любого (или большинства) символа в начале совпадения, такого как ⌜**(?: [^()]++ \| \((?R) \))***⌟, которое встречалось в примере на с. 593. Совпадение с этим выражением может начинаться с любого символа, кроме ‘)’, поэтому дополнительная проверка вряд ли позволит исключить большое число позиций, с которых не может начинаться совпадение.
- Выражения, совпадение с которыми начинается с единственного возможного символа, так как такие выражения уже оптимизированы.

Предложение по применению

Модификатор шаблона **S** не приводит к существенным затратам времени на проведение дополнительного анализа регулярного выражения, поэтому будет совсем нелишним использовать его для организации поиска по относительно большим фрагментам текста. Если, по вашему мнению, использование модификатора позволит получить дополнительные выгоды — используйте его.

Расширенные примеры

Ниже приводятся два примера, завершающие главу.

Разбор данных в формате CVS в PHP

Ниже приводится PHP-версия примера разбора данных в формате CSV (данных, разделенных запятыми) из главы 6 (☞ 342). Регулярное выражение было дополнено использованием захватывающих квантификаторов (☞ 190) вместо атомарной группировки, что обеспечило более ясное его представление.

Сначала идет регулярное выражение:

```
$csv_regex = '{
     \G(?:^|,)
     (?:
        # Поле в кавычках...
        " # открывающая кавычка
         ( [^"]*+ (?: "" [^"]*+ )*+ )
        " # закрывающая кавычка
      | # ...или...
        # ...произвольный текст, кроме кавычек и запятых...
        ( [^",]*+ )
     )
}x';
```

А затем реализация разбора текста в формате CSV, который содержится в переменной `$line`:

```
/* Применить регулярное выражение, заполнить $all_matches всеми типами данных
*/
preg_match_all($csv_regex, $line, $all_matches);

/* В $Result будут храниться поля, которые мы извлечем из $all_matches */
$Result = array ();

/* Обойти все успешные совпадения... */
for ($i = 0; $i < count($all_matches[0]); $i++)
{
    /* Если это совпадение со второй парой сохраняющих скобок, использовать
     * значение как есть */
    if (strlen($all_matches[2][$i]) > 0)
        array_push($Result, $all_matches[2][$i]);
    else
    {
```

```
        /* Если это значение в кавычках, предварительно заменить
         * последовательности из двух кавычек одной кавычкой */
        array_push($Result, preg_replace('/""/', '"', $all_matches[1][$i]));
    }
}

/* Теперь массив $Result заполнен и готов к дальнейшему использованию */
```

Проверка тегированных данных на корректность вложенных конструкций

Следующий далее пример содержит в себе массу интересных моментов: он выполняет проверку данных в формате XML (XHTML или любых других тегированных данных) на отсутствие непарных или незакрытых тегов. При построении регулярного выражения я отталкивался от следующих положений: каждому открывающему тегу должен соответствовать парный закрывающий тег, данные могут содержать текст вне тегов и могут существовать самозакрывающиеся теги (например, `<br/>` — «теги без вложенных элементов» в языке XML), при условии, что поиск ведется по всей строке.

Ниже приводится полное регулярное выражение:

```
/^((?:<(\w++)[^>]*+(?<!/)>(?1)</\2>|[^<>]++|<\w[^>]*+/>)*+)$/
```

Этому выражению будет соответствовать строка, не содержащая непарные теги (дополнительные замечания приводятся ниже).

Выражение может показаться достаточно сложным, но после разделения на отдельные компоненты сопровождать его становится достаточно просто. Тело регулярного выражения окружено конструкцией `/^(…)$/`, чтобы гарантировать, что вся строка целевого текста участвует в совпадении. Основное тело выражения также окружено дополнительными парами сохраняющих круглых скобок, что позволяет рекурсивно ссылаться на «основное тело».

Основное тело выражения

Основное тело выражения содержит три альтернативы (каждая из которых выделена подчеркиванием для большей ясности), заключенных в конструкцию `/(?:…)*+/`, что допускает совпадение с ними в любой комбинации. Первая альтернатива соответствует парным тегам, вторая — тексту вне тегов и третья — самозакрывающимся тегам.

Совпадение с любой из альтернатив является уникальным по отношению к другим альтернативам (т. е. если текст совпадает с одной альтернативой, он не будет

совпадать ни с одной из других альтернатив). Зная, что последующий возврат не приведет к совпадению того же самого текста с другой альтернативой, можно взять этот факт на вооружение и повысить эффективность регулярного выражения за счет применения захватывающего квантификатора * к скобкам, «допускающим совпадения в любой комбинации». Благодаря этому механизм регулярных выражений даже не будет пытаться выполнить возврат, что ускорит достижение результата, когда совпадение не может быть найдено.

По той же причине альтернативы могут располагаться в произвольном порядке, так что я поместил на первое место альтернативу, совпадение с которой, на мой взгляд, будет происходить чаще (☞ 328).

Теперь рассмотрим каждую из альтернатив по отдельности...

Вторая альтернатива: текст вне тегов

Начнем со второй альтернативы как с наиболее простой: `[^<>]++`. Этой альтернативе соответствует текст, расположенный вне тегов. Использование захватывающего квантификатора здесь может показаться избыточным, если учесть, что объемлющая конструкция `(?:…)*+` также снабжена захватывающим квантификатором. Я предпочитаю использовать захватывающие квантификаторы, если знаю, что это не повредит. (Захватывающие квантификаторы часто используются из соображений эффективности, но при этом они изменяют семантику поиска совпадения. Такое изменение может быть полезным, но вы должны понимать его суть ☞ 328.)

Третья альтернатива: самозакрывающиеся теги

Третьей альтернативе, `<\w[^>]*+/>`, соответствуют самозакрывающиеся теги, такие как `<br/>` и `<img …/>` (самозакрывающиеся теги отличает символ '/', вслед за которым сразу же следует закрывающая угловая скобка). Как и прежде, использование захватывающего квантификатора может показаться излишним, но это никак не вредит делу.

Первая альтернатива: совпадение с парой тегов

Наконец, мы подошли к рассмотрению первой альтернативы: `<(\w++)[^>]*+(?<!/)>(?1)</\2>`.

Первая часть выражения (выделена подчеркиванием) соответствует открывающему тегу: конструкция `(\w++)` сохраняет имя тега и является второй парой сохраняющих круглых скобок всего выражения. (Использование захватывающего квантификатора в `(\w++)` — очень важный момент, на котором мы вскоре остановимся отдельно.)

Конструкция ⌜(?<!/)⌟ — это негативная ретроспективная проверка (☞ 181), которая гарантирует отсутствие символа '/' перед только что найденным совпадением. Эта конструкция была помещена в разделе «совпадение с открывающим тегом» непосредственно перед ⌜>⌟, чтобы убедиться, что это не самозакрывающийся тег, такой как <hr/> (самозакрывающиеся теги обслуживаются третьей альтернативой, которую мы уже рассмотрели).

После того как будет найдено совпадение с открывающим тегом, выполняется ⌜(?1)⌟ — рекурсивное применение подвыражения в первой паре сохраняющих круглых скобок. Это и есть вышеупомянутое «основное тело», которому соответствует фрагмент текста без непарных тегов. Как только это совпадение будет обнаружено, мы должны оказаться в закрывающем теге, парном открывающему тегу, найденному в первой части альтернативы (имя которого было сохранено во второй паре сохраняющих скобок). Начальные символы ⌜</⌟ в подвыражении ⌜</\2>⌟ гарантируют, что мы имеем дело с закрывающим тегом, а обратная ссылка ⌜\2⌟ — что мы имеем дело с *корректным* закрывающим тегом.

Если вы предполагаете использовать это выражение для проверки HTML или других данных, где регистр символов в именах тегов не имеет значения, добавьте в начало регулярного выражения конструкцию ⌜(?i)⌟ или используйте модификатор шаблона **i**.

Всё!

Захватывающие квантификаторы

Мне хотелось бы дополнительно прокомментировать использование захватывающего квантификатора ⌜\w++⌟ в первой альтернативе ⌜<(\w++)[^>]*+(?<!/)>⌟. Если бы мне пришлось использовать диалект регулярных выражений с меньшими выразительными возможностями, в котором отсутствуют захватывающие квантификаторы или атомарная группировка (☞ 187), я использовал бы в этой альтернативе ⌜\b⌟ после ⌜(\w+)⌟, чтобы обеспечить совпадение с именем тега: ⌜<(\w+)\b[^>]*(?<!/)>⌟.

Метасимвол ⌜\b⌟ позволит, к примеру, предотвратить совпадение начальных символов 'li' в последовательности «<link>…». Без него выражение оставило бы 'nk' для совпадения за пределами сохраняющих скобок и, как следствие, — усеченное имя тега для обратной ссылки ⌜\2⌟, которая следует далее.

Обычно этого не происходит, так как \w+ является максимальным и старается обеспечить соответствие полному имени тега. Однако если это регулярное выражение применить к тексту с неправильной вложенностью, для которого не должно быть совпадения, то возврат в процессе поиска совпадения может вынудить ⌜\w+⌟ вернуть часть символов и тем самым дать совпадение с неполным именем тега, как в примере с тегами '<link>…
'. Метасимвол ⌜\b⌟ предотвращает такую возможность.

К счастью, мощный механизм preg в PHP поддерживает захватывающие квантификаторы, а использование такого квантификатора в конструкции ⌜(\w++)⌟ означает «не допускать разделения имени тега в результате возврата», поэтому достигается эффект использования ⌜\b⌟, при этом более эффективно.

Настоящий XML

Формат XML имеет более сложную структуру, которую нельзя выразить простой концепцией парных тегов. Кроме всего прочего, необходимо также учитывать комментарии XML, секции CDATA и инструкции обработки.

Добавление поддержки комментариев XML выражается в добавлении четвертой альтернативы ⌜<!--.*?-->⌟ и использовании ⌜(?s)⌟ или модификатора шаблона **s**, чтобы обеспечить совпадение точки с символом новой строки.

Аналогично секции CDATA, которые имеют форму <![CDATA[…]]>, могут обслуживаться еще одной альтернативой ⌜<!\[CDATA\[.*?]]>⌟, а *инструкции обработки* XML, такие как '<?xml•version="1.0"?>', могут обслуживаться с помощью альтернативы ⌜<\?.*?\?>⌟.

Объявления сущностей, которые имеют вид <!ENTITY…>, можно обработать с помощью альтернативы ⌜<!ENTITY\b.*?>⌟. В языке XML существует еще ряд аналогичных структур, обработка которых может быть объединена путем замены ⌜<!ENTITY\b.*?>⌟ на ⌜<![A-Z].*?>⌟.

Некоторые проблемы остаются, но того, что мы уже обсудили, вполне должно хватить для работы с большинством XML-документов. Ниже приводится фрагмент, в котором учтены все сделанные замечания:

```
$html_regex = '{
    ^(
       (?: <(\w++) [^>]*+ (?<!/)> (?1) </\2> # соответствует парным тегам
         |[^<>]++                            # текст вне тегов
         | <\w[^>]*+/>                       # самозакрывающиеся теги
         | <!--.*?-->                        # комментарии
         | <!\[CDATA\[.*?]]>                 # блоки cdata
         | <\?.*?\?>                         # инструкции обработки
         | <![A-Z].*?>                       # объявления сущностей и пр.
       )*+
     )$
}sx';

if (preg_match($xml_regex, $xml_string))
     echo "block structure seems valid\n";
else
     echo "block structure seems invalid\n";
```

HTML?

В HTML-документах наиболее часто встречаются все виды проблем, которые делают нецелесообразной проверку, подобную этой: среди них непарные и незакрытые теги, неправильное использование символов ‘<’ и ‘>’. Однако даже в правильно оформленных HTML-документах имеются специальные случаи, которые требуют дополнительной обработки, — это комментарии и теги `<script>`.

Комментарии HTML обрабатываются аналогично комментариям XML — с помощью конструкции `<!--.*?-->` и модификатора шаблона `s`.

Секция `<script>` требует особого внимания, потому что в ее пределах допускается использовать символы ‘<’ и ‘>’, следовательно, нам необходимо разрешить появление любых символов и их последовательностей между открывающим тегом `<script>` и закрывающим тегом `</script>`. Сделать это можно с помощью выражения `<script\b[^>]*>.*?</script>`. Интересно, что если последовательность сценария не содержит запрещенных символов ‘<’ и'>’, она будет соответствовать первой альтернативе, потому что соответствует «совпадению с парой тегов». Если `<script>` будет содержать такие символы, первая альтернатива потерпит неудачу, оставив последовательность для совпадения с этой альтернативой.

Ниже приводится версия фрагмента программного кода PHP для проверки HTML:

```
$html_regex = '{
   ^(
      (?: <(\w++) [^>]*+ (?<!/)> (?1) </\2> # соответствует парным тегам
        |[^<>]++                            # текст вне тегов
        | <\w[^>]*+/>                       # самозакрывающиеся теги
        | <!--.*?-->                        # комментарии
        | <script\b[^>]*>.*?</script>       # блоки сценариев
      )*+
    )$
}isx';

if (preg_match($html_regex, $html_string))
    echo "block structure seems valid\n";
else
    echo "block structure seems invalid\n";
```

Джеффри Фридл

Регулярные выражения

3-е издание

Серия «Бестселлеры O'Reilly»

Перевели с английского *Е. Матвеев, А. Киселев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург,
ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.

Дата изготовления: 05.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 11.05.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 49,020. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*