Mastering Regular Expressions

Third Edition

Jeffrey E.F. Friedl

O'REILLY®

Регулярные выражения

Третье издание

Джеффри Фридл

Санкт-Петербург — Москва
2008

Джеффри Фридл

Регулярные выражения, 3-е издание

Перевод Е. Матвеева и А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Б. Попов*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Фридл Дж.

Регулярные выражения, 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008. – 608 с., ил.

ISBN-13: 978-5-93286-121-9
ISBN-10: 5-93286-121-5

Книга Джеффри Фридла «Регулярные выражения» откроет перед вами секрет высокой производительности. Тщательно продуманные регулярные выражения помогут избежать долгих часов утомительной работы и решить проблемы за 15 секунд. Ставшие стандартной возможностью во многих языках программирования и популярных программных продуктах, включая Perl, PHP, Java, Python, Ruby, MySQL, VB.NET, C# (и других языках платформы .NET), регулярные выражения позволят вам автоматизировать сложную и тонкую обработку текста.

В третье издание включена информация о PHP и его мощном механизме регулярных выражений. Кроме того, обновлены и дополнены сведения о других языках программирования, включая расширенное и углубленное описание пакета java.util.regex компании Sun, при этом особое внимание уделено различиям между Java 1.4.2 и Java 1.5/1.6. Рассматривается принцип действия механизма регулярных выражений, сравниваются функциональные возможности различных языков программирования и инструментальных средств, подробно обсуждается оптимизация, которая дает основную экономию времени! Вы научитесь правильно конструировать регулярные выражения для самых разных ситуаций и сможете сразу же использовать предлагаемые ответы для выработки элегантных и экономичных практических решений широкого круга проблем. Кроме того, автор демонстрирует наиболее распространенные ошибки и показывает, как их избежать.

ISBN-13: 978-5-93286-121-9
ISBN-10: 5-93286-121-5
ISBN 0-596-52812-4 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 25.07.2008. Формат 70×100 1/16. Печать офсетная.
Объем 38 печ. л. Тираж 2000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

МОЕЙ Фумиэ 文枝

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

Оглавление

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 примене-

ния 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 стареть. 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 книги. Единственное, о чем я сожалею, – это то, что я 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 с учетом новых возможностей Java 1.5 и Java 1.6.

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 электронной почты 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 регуляр-

ных 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 алфавитный указатель максимально полезным.

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 стр. 124) 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.

Конкретные программы

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

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.

Конкретные программы

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 литературном 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:

•	Символ пробела
TAB	Символ табуляции
NL	Символ новой строки
CR	Символ возврата каретки

- 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 (☞439)».

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

Safari® Enabled

Если на обложке книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.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» Мак-Клоски (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/*).

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

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}{sprintf "%.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,

[1] Материал книги относится к Perl версии 5.8.8.

знакомы с Perl (после чтения главы 2, вероятно, ваших 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» (☞ 347) 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» (☞ 355) 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» (☞ 366)

  «Оператор поиска» (☞ 370)

  «Оператор подстановки» (☞ 383)

  «Оператор разбиения» (☞ 386)
- В разделе «Специфические возможности Perl» (☞ 392) описаны некоторые средства из арсенала регулярных выражений Perl, в том

[1] Уолл Л., Кристиансен Т., Орвант Д. «Программирование на Perl», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2002.

числе возможность выполнения произвольного кода Perl в процессе поиска совпадения.

- Раздел «Проблемы эффективности в Perl» (☞ 416) 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, 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 (☞ 120), а также упоминаются многочисленные аспекты регулярных выражений, относящиеся к Perl, например проблемы кодировок, включая Юникод (☞ 140), режимы поиска (☞ 145) и длинный обзор метасимволов (☞ 149).
- 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.

Perl обладает исключительно богатыми возможностями, но при воплощении в относительно малом наборе операторов эта мощь может стать «палкой о двух концах».

Таблица 7.1. Общая поддержка регулярных выражений в Perl

Операторы	
`m/`*выражение*`/`*модификаторы* **(☞370)**	
`s/`*выражение*`/`*замена*`/`*модификаторы* **(☞381)**	
`qr/`*выражение*`/`*модификаторы* **(☞366)**	
`split(…)` **(☞386)**	
Директивы	
`use charnames ':full';` **(☞351)**	
`use overload;` **(☞409)**	
`use re 'eval';` **(☞404)**	
`use re 'debug';` **(☞431)**	
Вспомогательные функции	
`lc lcfirst uc ucfirst` **(☞351)**	
`pos` **(☞378)**	`quotemeta` **(☞351)**
`reset` **(☞372)**	`study` **(☞429)**
Модификаторы	
`/x /o`	Интерпретация регулярного выражения **(☞354, 417)**
`/s /m /i`	Интерпретация целевого текста **(☞354)**
`/g /c /e`	Прочее **(☞375, 380, 385)**
Переменные с информацией о совпадении (☞362)	
`$1`, `$2` и т. д.	Сохраненный текст
`$^N $+`	Последнее/старшее из присвоенных значений `$1`, `$2`…
`@- @+`	Массивы индексов в целевом тексте
`$` $& $'`	Предшествующий текст, текст совпадения и текст после совпадения (использовать не рекомендуется – см. «Проблемы эффективности в Perl», ☞426)
Вспомогательные переменные	
`$_`	Целевой текст по умолчанию **(☞372)**
`$^R`	Результат выполнения встроенного кода **(☞365)**

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 специфичес-

кой 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	
☞151 (C)	`\a \b \e \f \n \r \t \`*восьм* `\x`*шестн* `\x{`*шестн*`} \c`*символ*
Символьные классы и аналогичные конструкции	
☞155	Обычные классы: `[`⋯`]` и `[^`⋯`]` (допускаются конструкции стандарта POSIX `[:alpha:]` ☞166)
☞156	Любой символ, кроме символа новой строки: *точка* (с модификатором `/s` – любой символ)
☞157	Комбинационные последовательности Юникода: `\X`

[1] Несмотря на великое множество особых случаев, в этой главе мы попытаемся рассмотреть их все!

☞157	Принудительное однобайтовое совпадение (может быть рискованным): `\C`
☞158 (C)	Сокращенные обозначения классов:② `\w \d \s \W \D \S`
☞159 (C)	Свойства, алфавиты и блоки Юникода:③ `\p{`*свойство*`}`, `\P{`*свойство*`}`

Якорные метасимволы и другие проверки с нулевой длиной совпадения

☞169	Начало строки/логической строки: `^ \A`
☞169	Конец строки/логической строки: `$ \z \Z`
☞379	Конец предыдущего совпадения: `\G`
☞174	Границы слов:④ `\b \B`
☞175	Позиционная проверка:⑤ `(?=…) (?!…) (?<=…) (?<!…)`

Комментарии и модификаторы режимов

☞176	Модификаторы режимов:⑥ `(?`*модификатор*`)`. Допустимые модификаторы: `x s m i` (☞354)
☞177	Интервальное изменение режима `(?`*модификатор*`:…)`
☞177	Комментарии: `(?#…)` и `#…` (с модификатором `/x`, а также от символа `#` до новой строки или конца регулярного выражения)

Группировка, сохранение, условные и управляющие конструкции

☞178	Сохраняющие круглые скобки: `(…) \1 \2 . . .`
☞178	Группирующие круглые скобки: `(?:…)`
☞180	Атомарная группировка: `(?>…)`
☞181	Конструкция выбора: `|`
☞182	Условная конструкция: `(?`*if then*`|`*else*`)` – в части *if* может находиться встроенный фрагмент кода, позиционная проверка или `(`*число*`)`
☞183	Максимальные квантификаторы: `* + ? {n} {n,} {min,max}`
☞184	Минимальные квантификаторы: `*? +? ?? {n}? {n,}? {min,max}?`
☞393	Встроенный код: `(?{…})`
☞393	Динамическое регулярное выражение: `(??{…})`

Только в литералах регулярных выражений

☞351 (C)	Интерполяция переменных: `$`*имя* `@`*имя*
☞351 (C)	Преобразование регистра следующего символа: `\l \u`
☞352 (C)	Интервальное преобразование регистра: `\U \L … \E`
☞352 (C)	Литеральный текст: `\Q … \E`
☞351 (C)	Именованный символ Юникода: `\M{`*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 (☞ 174).

Восьмеричные коды могут состоять как из двух, так и из трех цифр.

Метапоследовательность ⌜\x*шестн*⌟ 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 (☞ 151).

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 (☞ 164). 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 стр. 160.

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 может 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 поддерживаются.

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 ☞ 354).

При многократном использовании регулярного выражения структура операндов (простой строковый литерал или интерполяция переменных) заметно влияет на эффективность программы. Эта тема рассматривается на стр. 418.

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 тексту, а не 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 стр. 409.

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 усмотре-

нию. 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 стр. 384.

- (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 (☞ 372).

- Как упоминалось на стр. 350, 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.

Первые четыре модификатора, описанные в главе 3, также могут использоваться внутри регулярных выражений в конструкциях общей модификации режима (☞ 176) и интервальной модификации (☞ 177). Если в регулярном выражении используются «внутренние» модифи-

каторы, 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).

Таблица 7.3. Базовые модификаторы, доступные во всех операторах регулярных выражений

/i	☞145	Игнорировать регистр символов при поиске
/x	☞146	Режим свободного форматирования
/s	☞146	Режим совпадения точки со всеми символами
/m	☞147	Расширенный режим привязки к границам строк
/o	☞418	Режим однократной компиляции

Пятый базовый модификатор, /o, связан в основном с эффективностью. Он рассматривается позднее в этой главе, начиная со стр. 418.

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 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

[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.

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.

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 полнос-

тью 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 `$Debug` из пакета `Acme::Widget` может использоваться *полностью уточненное имя* `$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 сло-

вом `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 некоторого 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 новой переменной*. 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 попрежнему 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 `UnrulyFunction` 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 (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 котором 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 работают 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 копии, а лишь ставит новую 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, $&

[1] Perl не позволяет использовать my с именами специальных переменных, поэтому сравнение чисто теоретическое.

(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 в том, 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 не-

вероятно 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.

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 `$'`, описанные далее) не рекомендуется по соображениям эффективности (подробнее об этом на стр. 426). В случае успешного совпадения переменной `$&` 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] 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.

Таблица 7.5. Примеры использования специальных переменных, значения которых задаются после совпадения

После выполнения команды

```
"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)

$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 конструкциях динамических регулярных выражений (☞ 393). В других случаях в самом регулярном выражении они не используются (для этого существуют ⌜\1⌟ и другие метасимволы из того же семейства). (Раздел «Можно ли использовать $1 в регулярном выражении?» на стр. 366.)

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.

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:

```
$ur1 =~ 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 стр. 413.

@- 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 конструкциях динамических регулярных выражений и не имеет смысла за пределами регулярного выражения. В ней хранится резуль-

[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 (☞142).

тат последней исполняемой части встроенного кода с единственным исключением: часть *if* условных конструкций ⌈(?*if then*|*else*)⌋ (☞ 182) не изменяет состояния 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 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 (☞ 393), 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 (☞ 106) и 6 (☞ 337) 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, заме-

ны 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 выше на стр. 352, 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 (☞ 106):

```
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 ее 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 $HostnameRegex следующим фрагментом из главы 5 (☞ 256):

```
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 стр. 355. 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 $WordRegex = qr/\b \w+ \b/; # Модификатор /x отсутствует!
    .
    .
    .
if (Stext =~ m/^($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 стр. 367:

```
(?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» (☞ 418).

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 (☞ 356), 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

Регулярное выражение-операнд 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 (☞ 352).

Если операнд задается в виде литерала, в команду могут включаться любые базовые модификаторы, перечисленные на стр. 355. 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 ☞ 397).

Регулярное выражение по умолчанию

Если регулярное выражение не задано, как в команде `m//` (или `m/$SomeVar/`, где переменная `$SomeVar` содержит пустую строку или имеет неоп-

ределенное 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 (☞ 366).

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 $result 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* (☞ 356) 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 со-

впадение. 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    perl5-porters@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',
  'perl5-porters@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 IP-адрес 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 @+ ☞ 365) 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 вебсервера, 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» не элегантен. 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 (☞ 171), а подробный пример – в главе 5 (☞ 265).

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 (☞ 329) 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" }
    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 провер-

[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⌟ (стр. 302).

ки 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 оправдано. 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 (☞ 172).

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 `@+` (☞ 362).
- В текущей области видимости устанавливается регулярное выражение *по умолчанию* (☞ 371).
- 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 ☞ 372).

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 (☞ 378).
- При использовании модификатора `/o` регулярное выражение «привязывается» к оператору, что предотвращает возможность его повторной компиляции (☞ 421).

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 (☞ 371).

`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» (☞ 429).

Оператор `m?…?` и `reset`

Удачный поиск и вызовы `reset` изменяют состояние флага «было/не было совпадений» для оператора `m?…?` (☞ 372).

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. ❖Переверните страницу и проверьте свой ответ.

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 стр. 355, 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 заключается 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 (☞ 352), 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.

Существует 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.

Ответ

❖*Ответ на вопрос со стр. 383.*

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.

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`»).

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-Z0-9])/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 (☞ 375). 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 стр. 355.

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 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 стр. 387.

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 состояния переменных $&, $`, $1 и т. д. Короче говоря, в отношении побочных эффектов оператор split полностью изолирован от остальной программы.[1]

[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.

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…' )
```

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 значение '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` ☞ 365).

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)` (☞ 182). В этом случае результат интерпретируется как логическая величина, в зависимости от значения которой применяется либо секция *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` (☞ 405).

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.

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:

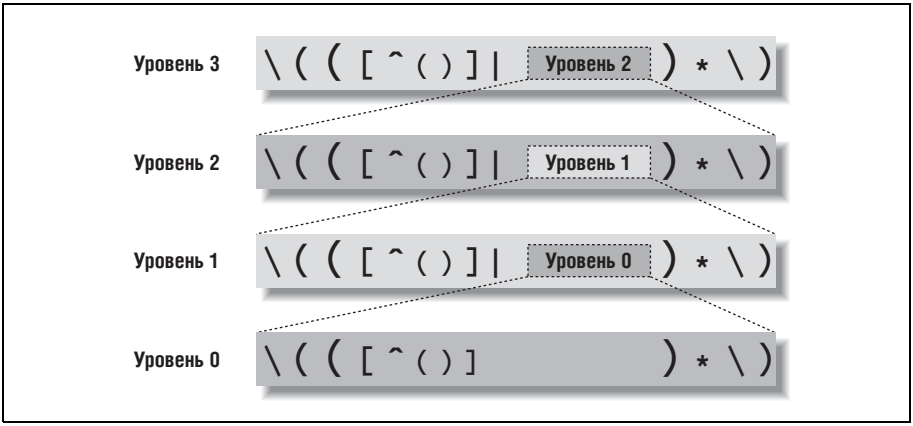

Рис. 7.1. *Поиск текста в круглых скобках*

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 ☞ 279).

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 стр. 411.

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 $' (☞ 362)[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 (☞ 191).

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;
```

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 исключе-

[1] Обычно я не рекомендую использовать специальные переменные $`, $& и $', так как они заметно снижают быстродействие программы (☞426), но для отладки они подойдут.

ния по начальному символьному классу (☞ 303) 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.

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.

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 стр. 225:

```
"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 ‘oneselfsufficient’.

Обратите внимание на одну тонкость: команда `print` выводит не «совпадение», а «совпадение *до данной позиции*». В нашем примере различия чисто академические, поскольку конструкция с 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 каждо-

го 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 становится 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 (☞ 225). 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*`)`⌋(☞ 182). В случае истинности условия регулярное выражение должно прекратить дальнейшие поиски, поэтому в части *then* используется заведомо несовпадающее подвыражение ⌈`(?!)`⌋ (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* (☞ 357) и разобраться в «крошечной аналогии» из главы 4, приведенной при обсуждении принципов работы механизма НКА, управляемого регулярными выражениями (☞ 202). Следующий хитроумный (но как вы вскоре убедитесь – 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 ⌜(?>…)⌟ (☞ 180) и происходящими возвратами, я поясню, что атомарная группировка используется для предотвращения бесконечного поиска (☞ 328) и не влияет на возвраты в границах конструкции – только на *возврат внутрь* конструкции после выхода из нее. 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 не уточненную глобальную переменную без ее «объявления» с ключевым словом our).

Интерполяция встроенного кода в Perl

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 `$RecordPossibleMatch` на стр. 400). 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; ☞431.)

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.

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 стр. 413.

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 `@` ☞ 365).

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 ра-

ботать 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 стр. 418). 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 стр. 413, 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 стр. 394 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 ⌜([…]+|…)*⌟ (☞ 280), 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 1 (незакрытых пар стало на 1 меньше). Если счетчик равен 0, 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*)⌟ (☞ 182), 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` (☞ 396).

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 в операторе 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* доста-

точно создать всего один раз (кстати, в самом файле *MyRegexStuff.pm* новые возможности недоступны, поскольку в нем отсутствует директива `use MyRegexStuff` – поверьте, делать это не стоит).

Поддержка захватывающих квантификаторов

Попробуем дополнить файл *MyRegexStuff.pm* и включить в него поддержку захватывающих квантификаторов типа ⌜x++⌟ (☞ 184). 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**)⌟ (☞ 220).

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` (стр. 396) 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 некоторые аспекты 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 стр. 172). 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 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 вмеши-

вается в обработку строки '\\>', обозначающей «символ '\', за которым следует '>'».

У перегрузки есть и другой 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*}⌟ ☞ 351).

Имитация именованного сохранения

Невзирая на все недостатки перегрузки, будет полезно рассмотреть сложный пример, в котором объединяются многие специальные конструкции. Именованное сохранение (☞ 180) в Perl не поддерживается, однако оно легко имитируется при помощи сохраняющих круглых скобок и переменной $^N (☞ 365), 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 стр. 367. 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 $+ ☞ 364), 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 стр. 406.

Приведенный ниже 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.

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;
```

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 (☞ 362). 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 других. 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 необязательно 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 ва-

рианте 8, позволяющая обойтись без круглых скобок, обходится слишком дорого. Мы вернемся к этой теме на стр. 425.

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 (☞ 296), 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» (☞ 354). 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 выра-

жения 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 стр. 405.

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 `$HttpUrl` на стр. 367 (с усовершенствованной версией `$HostnameRegex`). Хронометраж был организован так, чтобы тесты выводили затраты на предварительную обработку регулярного выражения (интерполяцию, проверку строк, компиляцию и другие вспомогатель-

ные 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 вы-

нужден 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/…/ (☞ 366).

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 основании 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, использующи-

ми 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 стр. 422.

Регулярное выражение по умолчанию

Регулярное выражение по умолчанию (☞ 371) 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 (☞ 362). 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 к 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 пользователь 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 `@+` (☞ 365) 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 на следующей врезке.

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` (☞431) 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 не известно 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 к информации 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 при-

менения 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 четыре 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 (☞ 303). 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 (☞ 312). 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 (☞ 286). 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 (☞ 294), но это далеко не все. Многие оптимизации применяются только в очень специфических ситуациях, поэтому для любого регулярного выражения исполь-

зуется 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 ☞ 404).

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 ☞ 300).

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 (☞ 426).

Отладочная информация на стадии выполнения

Выше уже приводился пример использования встроенного кода для получения информации о ходе поиска (☞ 398), 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 `debug` 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, для этого 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 (☞ 180), 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 (☞ 164), хотя ценой определенных усилий они имитируются на базе опережающей проверки (☞ 166).

Далее идут захватывающие квантификаторы (☞ 184). 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 стр. 402.

Наконец, как упоминалось на стр. 404, 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.