

САМОУЧИТЕЛЬ PERL

В книге изложены основы современного языка Perl, популярность которого постоянно возрастает, особенно в таких областях, как обработка текста, CGI-программирование, системное администрирование. Язык описан по схеме от простого к сложному: типы данных, переменные, операции, операторы и т. д. Рассматривается объектно-ориентированная технология программирования. Приведенные в книге примеры и упражнения, которые авторы реализовали на различных платформах, помогут читателю разобраться в изложенном материале.

Содержание

Предисловие	1	4.2. Операции конкатенации и повторения	62
Глава 1. Введение в мир Perl	5	4.3. Операции отношения	65
1.1. История языка Perl	6	4.3.1. Числовые операции	65
1.2. Характерные черты Perl	8	отношения	
1.3. Области применения Perl	12	4.3.2. Строковые операции	67
Системная поддержка UNIX	13	отношения	
CGI-сценарии	13	4.4. Логические операции	68
Обработка почты	14	4.5. Побитовые операции	70
Поддержка узлов Web	14	4.5.1. Числовые операнды	70
Вопросы для самоконтроля	15	4.5.2. Строковые операнды	73
Глава 2. Структура программы	16	4.6. Операции присваивания	74
2.1. Простая программа	16	4.7. Ссылки и операция разыменования	77
2.2. Объявления и комментарии	21	4.8. Операции связывания	78
2.3. Выражения и операторы	22	4.9. Именованные унарные операции	79
Вопросы для самоконтроля	27	4.10. Операции ввода/вывода	79
Упражнения	27	4.10.1. Операция print	80
Глава 3. Типы данных	29	4.10.2. Выполнение системных команд	80
3.1. Алфавит языка	29	4.10.3. Операция \diamond	80
3.2. Скалярный тип данных	31	4.11. Разные операции	81
3.3. Массивы скаляров	42	4.11.1. Операция диапазон	81
3.4. Ассоциативные массивы	47	4.11.2. Операция запятая	86
3.5. Переменные	52	4.11.3. Операция выбора	87
Вопросы для самоконтроля	55	4.12. Списковые операции	88
Упражнения	56	4.13 Операции заключения в кавычки	88
Глава 4. Операции и выражения	57	4.13.1. Операция <code>q{ }</code>	89
4.1. Арифметические операции	58	4.13.2. Операция <code>qq{ }</code>	90
4.1.1. Бинарные арифметические операции	58	4.13.3. Операция <code>qx{ }</code>	90
4.1.2. Унарные арифметические операции	60		
4.1.3. Операции увеличения и уменьшения	61		

4.13.4. Операция <code>qw{ }</code>	91	7.2. Доступ к файлам	151
4.13.5. Операция "документ здесь"	91	7.3. Операции с файлами	162
4.14. Выражения	94	7.4. Получение информации о файле	168
4.14.1. Термы	95	7.5. Операции с каталогами	171
4.14.2. Приоритет операций	95	Вопросы для самоконтроля	174
4.14.3. Контекст	100	Упражнения	174
Вопросы для самоконтроля	102	Глава 8. Форматы	175
Упражнения	102	8.1. Объявление формата	175
Глава 5. Операторы	104	8.2. Использование нескольких форматов	182
5.1. Простые операторы	104	Вопросы для самоконтроля	186
5.2. Модификаторы простых операторов	105	Глава 9. Ссылки	187
5.2.1. Модификаторы <code>if</code> и <code>unless</code>	106	9.1. Виды ссылок	187
5.2.2. Модификаторы <code>while</code> и <code>until</code>	107	9.2. Создание ссылок	190
5.2.3. Модификатор <code>foreach</code>	109	9.2.1. Операция ссылки <code>"\"</code>	190
5.3. Составные операторы	110	9.2.2. Конструктор анонимного массива	191
5.3.1. Блоки	111	9.2.3. Конструктор анонимного ассоциативного массива	191
5.3.2. Операторы ветвления	113	9.2.4. Другие способы	192
5.4. Операторы цикла	116	9.3. Разыменование ссылок	194
5.4.1. Циклы <code>while</code> и <code>until</code>	116	9.3.1. Разыменование простой скалярной переменной	195
5.4.2. Цикл <code>for</code>	118	9.3.2. Блоки в операциях разыменования ссылок	195
5.4.3. Цикл <code>foreach</code>	122	9.3.3. Операция разыменования <code>"->"</code>	196
5.5. Команды управления циклом	125	9.4. Символические ссылки	198
5.5.1. Команда <code>last</code>	126	9.5. Использование ссылок	200
5.5.2. Команда <code>next</code>	128	9.5.1. Замыкания	201
5.5.3. Команда <code>redo</code>	129	9.5.2. Массив массивов	203
5.6. Именованные блоки	131	9.5.3. Другие структуры данных	204
5.7. Оператор безусловного перехода	133	Вопросы для самоконтроля	210
Вопросы для самоконтроля	134	Упражнения	210
Упражнения	135	Глава 10. Работа со строками	212
Глава 6. Операции ввода/вывода	137	10.1 Регулярные выражения	212
6.1. Операция ввода команды	137	10.1.1. Метасимволы	213
6.2. Операция <code><></code>	140	10.1.2.	215
6.3. Функция <code>print</code>	145	Метапоследовательности	
Вопросы для самоконтроля	147		
Упражнения	147		
Глава 7. Работа с файлами	148		
7.1. Дескрипторы файлов	148		

10.1.3. Атомы	217	деструктор пакета <i>BEGIN</i> и	
10.1.4. Обратные ссылки	218	<i>END</i>	
10.1.5. Расширенный	220	12.1.3. Автозагрузка	274
синтаксис регулярных		12.2. Библиотеки	275
выражений		12.2.1. Функция <i>require()</i>	275
10.1.6. Сводка результатов	223	12.2.2. Создание и	277
10.2. Операции с регулярными	226	подключение	
выражениями		библиотечного файла	
10.2.1. Операция поиска	226	12.3. Модули	278
10.2.2. Операция замены	231	12.3.1. Функция <i>use()</i>	279
10.2.3. Операция	231	12.3.2. Создание и	280
транслитерации		подключение модуля	
10.2.4. Операция заключения	233	12.3.3. Функция <i>no()</i>	282
в кавычки <i>qr//</i>		12.3.4. Стандартные модули	282
10.3. Функции для работы со	234	Perl	
строками		12.3.5. Прагма-библиотеки	283
Вопросы для самоконтроля	245	Вопросы для самоконтроля	285
Упражнения	247	Упражнения	286
Глава 11. Подпрограммы и	249	Глава 13. Объектно-	287
функции		ориентированное	
11.1. Определение	249	программирование в языке	
подпрограммы		Perl	
11.2. Вызов подпрограммы	251	13.1. Классы и объекты	288
11.3. Локальные переменные в	252	13.2. Методы	290
подпрограммах		13.2.1. Конструкторы	291
11.3.1. Функция <i>my()</i>	253	13.2.2. Методы класса и	292
11.3.2. Функция <i>local()</i>	253	методы объекта	
11.4. Передача параметров	255	13.2.3. Вызов метода	294
11.4.1. Передача по ссылке	256	13.2.4. Деструкторы	297
параметров-массивов		13.3. Обобщающий пример	297
11.5. В каких случаях функцию	260	Вопросы для самоконтроля	301
<i>local</i> нельзя заменить		Упражнение	301
функцией <i>my</i>		Глава 14. Запуск интерпретатора	302
11.6. Прототипы	263	и режим отладки	
11.7. Рекурсивные подпрограммы	265	14.1. Опции командной строки	302
Вопросы для самоконтроля	266	14.2. Отладчик Perl	308
Упражнения	267	14.2.1. Просмотр текста	309
Глава 12. Пакеты, библиотеки,	268	программы	
модули		14.2.2. Выполнение кода	310
12.1. Пакеты	268	14.2.3. Просмотр значений	311
12.1.1. Таблицы символов	271	переменных	
12.1.2. Конструктор и	272	14.2.4. Точки останова и	312

действия		15.4.4. Модуль CGI.pm	348
Вопросы для самоконтроля	318	Вопросы для самоконтроля	360
Глава 15. Язык Perl и CGI	319	Упражнение	360
программирование		Глава 16. Ресурсы Perl	363
15.1. Основные понятия	319	16.1. Конференции	363
15.2. HTML-формы	320	16.2. Специализированные Web-	364
15.2.1. Тэг <FORM>	321	узлы Perl	
15.2.2. Тэг <INPUT>	322	16.3. Архив CPAN	365
15.2.3. Тэг <SELECT>	325	Приложение 1. Стандартные	368
15.2.4. Тэг <TEXTAREA>	326	функции Perl	
15.2.5. Пример формы	327	Получение информации из	398
15.3. Передача информации CGI-	330	системных файлов	
программе		Межпроцессное взаимодействие	402
15.4. CGI-сценарии	332	Приложение 2. Модули Perl	406
15.4.1. Переменные среды	335	Стандартные модули	406
CGI		Модули CPAN	415
15.4.2. Обработка данных	337	Приложение 3. Специальные	417
формы		переменные	
15.4.3. Пример создания	340	Предметный указатель	425
собственного CGI-сценария			

Предметный указатель

Б	redo, 129
Блок, 111	Конструкторы
именованный, 132	массива скаляров, 42
В	Контекст, 53
Выражение, 57; 94	скалярный, 53
терм, 95	списковый, 53
побочные эффекты, 25	Л
Д	Литерал
Дескриптор, 148	строковый, 33
_, 169	числовой, 31
STDERR, 150	М
STDIN, 150	Модификатор, 105
STDOUT, 150	foreach, 109
предопределенный, 150	if, 106
файла, 148	unless, 106
К	until, 107
Ключевые слова, 31	while, 107
Команда управления циклом, 125	О
last, 126	Оператор, 104
next, 128	goto, 133

- ветвления, 113
- простой, 104
- составной, 110
- цикла, 116
- Оператор цикла
 - until, 116
 - while, 116
- Операция, 57
 - !~, 78
 - ", 77
 - %, 58
 - &, 71
 - *, 58
 - **, 58
 - , и =>, 87
 - . (конкатенация), 62
 - /, 58
 - ^, 72
 - |, 71
 - ~, 72
 - +, 58
 - ++, 61
 - <<, 72
 - <>, 80; 140; 156
 - <> без аргумента, 141
 - =, 74
 - =~, 78
 - >, 78
 - >>, 72
 - format, 176
 - lvalue, 74
 - qx, 137
 - арифметическая, 58
 - выбора, 87
 - диапазон, 81
 - "документ здесь", 91
 - заклЮчения в кавычки, 89
 - заклЮчения в обратные кавычки
 - ", 137
 - "запятая", 86
 - контекст, 100
 - логическая, 68
 - отношения, 65
 - побитовая, 70
 - приоритет, 96
 - разыменования ссылки, 78
 - составного присваивания, 75
 - сочетаемость, 99
 - списковая, 88
 - списковая операция, 98
 - ссылка, 77
 - укороченная схема вычисления, 69
 - унарный + и -, 60
 - x (повторение строки), 63
 - X (проверка файлов), 170
- П**
 - Параметры командной строки, 140
 - Переменная, 38; 52
 - \$\$, 154
 - \$. , 156
 - \$/ , 156
 - динамическая, 112
 - лексическая, 112
 - подстановка, 39
 - пространство имен, 53
 - скалярная, 39
 - хеш, 47
 - Перенаправление стандартного ввода/вывода, 150
 - Предопределенные файлы
 - STDERR, 140
 - STDIN, 140
 - STDOUT, 140
 - Простое слово, 41
- Р**
 - Работа с каталогами, 172
 - дескриптор каталога, 172
- С**
 - Структура индексного дескриптора, 168
- Т**
 - Тип данных, 29

массив, 42
хеш-массивы, 47

Ф

Файл

владелец, 163
двоичный, 171
жесткие ссылки, 165
закрытие, 155
запись, 149; 158
константы режимов доступа, 152
открытие, 149
переименование, 166
получение информации, 168
права доступа, 153; 164
режимы доступа, 151
символические ссылки, 165
текстовый, 171
текущая позиция, 158
удаление, 166
усечение, 166

Файловая система UNIX, 162

gid, 162
uid, 162

Формат, 175

верхний колонтитул, 181
переменная S%, 182
переменная S^L, 184
переменная \$~, 182
переменная \$=, 181
символы форматирования, 177
строка переменных, 176
строка шаблонов, 176
шаблоны, 177

Функция

chmod(), 164
chown(), 163
close(), 155
die (), 154
getc(), 159
glob, 114
length(), 160
link(), 165

local, 112
lstat(), 169
map, 111
my, 111
open(), 149; 151
print(), 145; 158
read(), 160
rename(), 166
seek(), 159
select(); 158
stat(), 168
symlink(), 165
sysopen(), 152
sysread(), 161
sysseek(), 161
syswrite(), 161
tell(), 158
truncate(), 166
unlink(), 166
utime(), 164
write(), 176

Э

Элементы языка блок, 26

выражение, 25
знаки операций, 24
идентификатор, 22
ключевые слова, 23
лексема, 22
литерал, 23
операторы, 22
пробельные символы, 24
простой оператор, 25
разделитель, 24

Предисловие



Предисловие, на наш взгляд, должно дать читателю информацию, на основании которой он решает, нужна ли ему эта книга.

О чем наша книга. Она, естественно, о языке Perl, потому что так заявлено в названии. Кому он нужен, этот Perl? Тем, кто создает CGI-сценарии, занимается администрированием системы при помощи написания скриптов, а не щелкая левой кнопкой мыши, обращает тексты, решает многие другие задачи из смежных областей и при этом нуждается в мощном, но простом в применении средстве, позволяющем создавать большие программы и маленькие программки и быстро их опробовать. Тем, кто преподает программирование, тоже полезно иметь представление об этом языке, так как он обладает интересными свойствами, отсутствующими в традиционных языках программирования, используемых в процессе обучения.

Нам нравятся некоторые особенности языка: зависимость результата от контекста, ассоциативные массивы, тип данных `typeglob`, пакеты, реализация объектно-ориентированного программирования и, конечно, средства обработки текста. Если вам не интересно хотя бы узнать, что все это означает, то можете книгу отложить. Если все перечисленное вам уже известно, то тоже можете ее отложить, потому что эта книга для тех читателей, кто еще только начинает изучать Perl самостоятельно.

Язык Perl создан системным программистом Ларри Уоллом (Larry Wall) как средство UNIX, позволяющее "склеивать" из программ, выполняющих отдельные функции, большие сценарии для решения комплекса задач, связанных с администрированием, обработкой текста и т. д. В дальнейшем он вышел за эти рамки, превратился в настоящий язык программирования, в котором нашли отражение многие тенденции, обозначившиеся в технологии программирования за последнее десятилетие, и получил широкое распространение в связи с развитием Internet. Perl является основным средством создания приложений CGI, удобен для решения задач администрирования Web-серверов, электронной почты и других систем. Благодаря скорости и легкости написания сценариев на этом языке он распространился и на другие платформы: DOS, Windows, OS/2, Mac, VMS и пр. Одно из основных достоинств языка Perl — его открытость и доступность. В сети Internet мож-

но получить совершенно бесплатно исходные тексты интерпретатора perl (язык Perl — интерпретируемый, что в некоторых случаях является преимуществом) и модулей его расширения.

Данная книга — самоучитель языка Perl, который изучается, что называется, с нуля, т. е. предполагается, что читатель не знаком с этим языком — все необходимое он узнает, последовательно изучив темы и закрепив пройденный материал, отвечая на вопросы и выполняя упражнения, приведенные в конце каждой главы. Повторим, что предлагаемый материал представляет всего лишь основы языка Perl. Наша книга ни в коей мере не претендует на учебник по программированию на языке Perl. В ней вы не найдете методологию программирования или готовые рецепты решения задач, в ней нет подробного описания наиболее часто используемых модулей и решения задач с их помощью, но, прочитав книгу, вы приобретете базовые знания, которые позволят разобраться в любом сценарии Perl.

При описании языка мы придерживались следующей схемы: сначала познакомить читателя с типами данных, затем с допустимыми операциями языка, далее с операторами и потом уже со специальными синтаксическими конструкциями (ссылки, регулярные выражения, подпрограммы, пакеты, модули и объектно-ориентированное программирование), т. е. мы старались идти от простых понятий к более сложным, что, как нам кажется, позволяет читателю получить целостное представление о языке программирования. Завершается изучение языка описанием того, как его использовать для создания CGI-сценариев — области Web-программирования, где он широко распространен.

Что осталось за рамками нашей книги? Очень многое, так как мир Perl велик. Это вопросы взаимодействия с базовой операционной системой, межпроцессное взаимодействие в UNIX, средства организации сетей, совместное использование Perl и других языков программирования, особенности работы на платформах, отличных от UNIX, работа с базами данных и еще многое другое. В главе о применении Perl в CGI-программировании неплохо было бы рассказать о протоколе HTTP и конфигурировании Web-сервера. По этим темам можно написать еще одну книгу, как минимум, такого же объема. Мы сознательно исключили данные-вопросы из рассмотрения, так как они являются достаточно специфическими.

Все примеры самоучителя рассчитаны на использование версии Perl 5.005, которая в процессе работы над книгой считалась последней устойчивой версией, и проверены в операционных системах Linux, Windows NT 4.0, Windows 95. Перед самым окончанием работы (конец марта 2000 года) появилась новая версия языка Perl 5.6, обладающая дополнительными возможностями по сравнению с предыдущей версией. Однако это не должно расстраивать читателя, поскольку основные понятия остались, естественно, без изменений.

Следует иметь в виду, что "родной" платформой Perl является система UNIX, поэтому результат выполнения некоторых примеров в других операционных системах может не соответствовать тому, что написано в книге.

Используемые обозначения

Курсив применяется для выделения терминов и ссылок на главы и разделы самоучителя.

Все тексты и консольный вывод программ Perl, переменные, операторы, функции и другие элементы языка представлены моноширинным шрифтом.

При описании синтаксических конструкций языка применяются следующие условные обозначения:

- {параметр} — необязательный параметр
- ... — повторение предыдущей конструкции

Литература

Наиболее известной и часто цитируемой книгой по программированию на языке Perl является издание *Larry Wall, Tom Christiansen, Randal Schwartz.— Programming Perl, 2nd Edition. O'Reilly & Associates, 1996, 646 pages.*

Подробная информация содержится также в документации, входящей в дистрибутивный комплект Perl.

Завершая предисловие, мы хотели бы выразить благодарность директору издательства "БХВ — Санкт-Петербург" Владимиру Сергееву и главному редактору Екатерине Кондуковой, которые предоставили нам возможность написать эту книгу.



Введение в мир Perl

Что такое Perl? Это сокращенное название языка программирования *Practical Extraction and Report Language* (Практический язык извлечений и отчетов). Что подразумевается под "извлечениями" и "отчетами"? Почему *практический* язык? Для чего он предназначен? Какие задачи можно решать с его помощью? Эти и многие другие вопросы возникают, естественно, у любого человека, хоть немного знакомого с информатикой, когда он впервые сталкивается с новым для него языком программирования. Эта глава и задумывалась как ответ на поставленные выше вопросы, так как зная, что может, для чего предназначен язык программирования (а время универсальных языков, кажется, миновало), программист, в конечном счете, решает, а стоит ли тратить время на его изучение. Хотя здесь также встают вопросы о легкости и скорости освоения нового языка, доступности компиляторов, существовании службы его поддержки, стоимости и т. д. Об этом также пойдет речь в этой главе, которая познакомит читателя с огромным миром Perl-программирования, и станет той отправной точкой, с которой он, мы надеемся, стремительно и без оглядки войдет в него и останется в нем навсегда.

Язык Perl родился в недрах операционной системы Unix как реакция одного талантливого программиста на ограниченную возможность стандартных средств системного администрирования в этой операционной среде. Авторы прекрасно осознают, что большинство читателей знакомы с Unix, возможно, только по названиям книг, лежащих на полках магазинов, так как традиция изучения информационных технологий в нашей стране связана больше с операционными системами семейства Microsoft Windows¹, чем с системой UNIX, которая является базой изучения информатики в западных университетах. Поэтому для воспитанных в традициях Windows читателей мы сделаем небольшое отступление и кратко охарактеризуем процедуру администрирования UNIX, которая радикально отличается от аналогичной работы в операционной системе Windows.

Под *администрированием* понимается настройка операционной системы через установку значений ее параметров таким образом, чтобы она отвечала потребностям отдельного пользователя или группы пользователей. В системах семейства Windows подобная работа выполняется с помощью Реестра,

¹ Под семейством операционных систем Microsoft Windows понимаются операционные системы Windows 95/98/NT.

представляющего собой базу данных двоичных данных, а для изменения параметров используется специальная программа `regedit`. В системе UNIX настройка осуществляется через специальные конфигурационные файлы, являющиеся обычными текстовыми файлами, и все изменения осуществляются выполнением команд, написанных на специальном языке оболочки (shell) и выполняемых, как правило, из командной строки. (Несколько лет назад на персональных компьютерах была широко распространена операционная система MS-DOS фирмы Microsoft, в которой для ввода команд также использовалась командная строка, поэтому читателю, работавшему в этой операционной системе, командная строка знакома.) В системе UNIX пользователь может создавать собственные команды на основе команд интерпретатора shell, сохранять их в обычных текстовых файлах и впоследствии выполнять также, как обычные стандартные команды операционной системы через командную строку. Следует отметить, что оболочка shell операционной системы UNIX является интерпретатором, в связи с чем команды пользователя имеют еще одно название — их называют *сценариями* или *скриптами* (script). Администратору операционной системы UNIX приходится писать большое количество скриптов, которые обрабатывают другие скрипты — текстовые файлы. Для этих целей обычно кроме командного языка оболочки shell используются специальные программы обработки текстовых файлов:

❑ `awk` — программа сопоставления с образцами и генератор отчетов;

❑ `sed` — пакетный редактор текстовых файлов.

Обе эти программы являются фильтрами, которые последовательно считывают строки входных файлов и выполняют применимые к строке действия, определенные с помощью команд этих программ. Основными действиями являются выделение цепочек символов по заданным шаблонам, замена их по определенным правилам и генерирование новых файлов.

Теперь можно перейти и к объекту нашего изучения — языку Perl, тем более что, как нам кажется, читателю уже должно быть понятно, почему он называется языком извлечений и отчетов. И начнем мы с истории его создания и разработки, которая, по существу, позволяет полнее понять его содержание.

1.1. История языка Perl

Perl был разработан Ларри Уоллом (Larry Wall) в 1986 году, когда он являлся системным администратором одного проекта UNIX, связанного с созданием многоуровневой безопасной сети, объединявшей несколько компьютеров, разнесенных на большие расстояния. Работа была выполнена, но потребовалось создание отчетов на основе большого числа файлов с многочисленными перекрестными ссылками между ними.

Первоначально Ларри предполагал использовать для этих целей фильтр `awk`, но оказалось, что последний не мог управлять открытием и закрытием

большого числа файлов на основе содержащейся в них же самих информации о расположении файлов. Его первой мыслью было написать специальную системную утилиту, решающую поставленную задачу, но вспомнив, что до этого ему уже пришлось написать несколько утилит для решения задач, не "берущихся" стандартными средствами UNIX, он принял кардинальное решение — разработать язык программирования, который сочетал бы в себе возможности обработки текстовых файлов (sed), генерации отчетов (awk), решения системных задач (shell) и низкоуровневое программирование, доступное на языке C. Результатом этого решения и явился язык Perl, интерпретатор для которого был написан на C.

По утверждению самого Ларри Уолла при создании языка Perl им двигала лень — не в прямом смысле, а в смысле того, что для решения стоявшей перед ним задачи следовало бы написать большое количество программ на разных языках, входящих в состав инструментальных средств UNIX, а это достаточно утомительное занятие.

Новый язык программирования сочетал в себе возможности системного администрирования и обработки файлов — две основные задачи, решаемые обычно при программировании в системе UNIX. Причем следует отметить, что язык Perl появился из практических соображений, а не из-за желания создать еще одно "красивое" средство для работы в UNIX, поэтому-то он и получил широкое распространение среди системных администраторов, когда Ларри Уолл предоставил его широкому кругу пользователей. С появлением языка Perl появилась возможность решать задачи с помощью одного инструмента, и не тратить время на изучение нескольких языков среды программирования UNIX.

Первая версия языка не содержала многих возможностей, которые можно найти в последней версии Perl, с которой читатель познакомится в нашей книге и которая идентифицируется как версия 5.005_03 и считается устойчивой. Первоначально язык включал:

- ☐ простой поиск по строковым образцам (шаблонам) в файлах;
- ☐ дескрипторы файлов;
- ☐ скалярные переменные;
- ☐ форматы.

Вся документация умещалась на 15 страницах, но Perl решал задачи быстрее, чем sed или awk, и быстро стал использоваться не только для решения задач системного администрирования.

В дальнейшем сам Ларри Уолл позаимствовал у Генри Спенсера (Henry Spencer) пакет для работы с регулярными выражениями и модифицировал его для языка Perl. Другие функциональные возможности были разработаны не только Ларри Уоллом, но и его друзьями и коллегами, и включены в состав языка. Опубликование в Internet привело к появлению сообщества еди-

номышленников, которые не только эксплуатировали, но и развивали язык. Он и по настоящее время продолжает интенсивно развиваться за счет разработки пакетов, реализующих новые применения языка к развивающимся информационным технологиям. В табл. 1.1 представлена динамика появления новых версий языка Perl, начиная с самой первой:

Таблица 1.1. Версии языка Perl и даты их выпуска

Версия	Дата выпуска
perl1	Январь, 1988
perl2	Июнь, 1988
perl3	Октябрь, 1989
perl4	Март, 1991
perl5	Октябрь, 1994

В настоящее время, как уже отмечалось ранее, устойчивой версией считается версия Perl 5.005_03, но уже существует версия 5.005_67. Их все можно получить с основного узла Web, поддерживающего язык Perl, по адресу <http://www.perl.com/>.

Замечание

В литературе по языку Perl принято, ссылаясь на сам язык, писать его с прописной буквой (Perl), а строчными буквами (perl) обозначать интерпретатор языка. По образному высказыванию самого Ларри Уолла: "perl — это ничего более как всего лишь интерпретатор Perl".

1.2. Характерные черты Perl

Perl — это интерпретируемый язык, оптимизированный для просмотра содержимого текстовых файлов, выделения из них информации и генерирования отчетов на основе этой информации, а также просто хороший язык для выполнения многих задач системного администрирования UNIX. Он обладает большим набором преимуществ как язык сценариев общего назначения, которые проявляются через его характерные черты и возможности.

Первым в цепочке достоинств языка Perl мы назовем его *интерпретируемость*. Конечно, некоторые программисты, прочитав это, скажут: "Ну вот, нашли себе достоинство. Посмотрим, как быстро будет выполняться программа Perl длиной, скажем, в тысячу операторов?". Что ж, замечание существенное, если рассматривать Perl как язык создания больших информационных систем, и совершенно не выдерживающее критики, если

вспомнить, для чего он предназначен — задач администрирования и обработки текстовых файлов — небольших по размерам сценариев, решающих нетрадиционные задачи, для программирования которых могло бы потребоваться взаимодействие нескольких специализированных языков. Разработка подобных решений с помощью компилируемых языков программирования потребовала бы на много больше времени, чем использование одного интерпретируемого: ведь цикл разработки программ на таком языке короче и проще, чем на компилируемом. Мы постепенно создаем программу, добавляя необходимые операторы, и сразу же получаем результаты, когда она завершена: интерпретатор perl постепенно компилирует все операторы во внутренний байт-код и программа готова к выполнению, как только в ней поставлена последняя точка (точнее точка с запятой, завершающая последний оператор). Для небольших по объему программ — это достаточное преимущество, так как отладка занимает много времени. Да, интерпретируемая программа, естественно, будет выполняться медленнее программы, представленной в формате двоичного файла и выполняющейся без предварительной обработки интерпретатором, но если в этом возникнет необходимость, то можно решение на языке Perl использовать в качестве прототипа для компилируемого языка, например C. Суммируя все сказанное, можно заключить, что Perl позволяет легко и быстро получить требуемое решение задачи, сочетая в себе элементы компилируемых и интерпретируемых языков программирования.

Замечание

Интерпретатор perl, как, вероятно, заметил внимательный читатель, отличается от традиционных интерпретаторов тем, что программа транслируется в промежуточный байт-код, и только после этого выполняется. В традиционных интерпретаторах каждый вводимый оператор интерпретируется и сразу же выполняется, что может приводить к синтаксическим ошибкам во время выполнения. Perl-программа свободна от этого "недостатка", так как все синтаксические ошибки обнаруживаются во время трансляции в байт-код.

Вторым преимуществом использования Perl для решения соответствующих задач (мы имеем в виду сетевые возможности) является его доступность для большинства серверных платформ:

- ☐ практически все варианты UNIX;
- ☐ MS-DOS;
- ☐ Windows NT;
- ☐ Windows 95/98;
- ☐ OS/2;
- ☐ Macintosh.

Для всех перечисленных платформ разработаны и свободно распространяются интерпретаторы perl вместе с документацией по их установке и работе, что приятно отличает его от других программных средств. И здесь уместно сказать несколько слов об условиях использования и распространения самого Perl и разработанных на нем программ.

(О том, где можно найти и получить интерпретатор perl, см. главу 16.)

Одним из способов распространения свободно распространяемого программного обеспечения, а именно таков интерпретатор perl, является использование Общей открытой лицензии GNU. По условиям этой лицензии файлы исходного текста программного продукта распространяются совершенно свободно и могут быть использованы любым лицом. Однако любые версии программы, созданные путем модификации этого кода, должны реализовываться также на условиях Общей открытой лицензии GNU, т. е. следует предоставлять файлы исходных текстов нового продукта любому, кто их захочет иметь. Этого зачастую вполне достаточно, чтобы защитить интересы автора первоначального программного продукта, однако может приводить к большому количеству производных версий исходного продукта, что приводит к "отчуждению" автора исходного продукта от процесса модификации его деталей. Более того, в связи с большим количеством разнообразных версий, пользователям становится трудно определить, какая версия пакета является на текущий момент окончательной, будут ли написанные им сценарии, если речь идет о perl, правильно работать с имеющейся у него версией, и т. п.

В связи с изложенными недостатками лицензии GNU, интерпретаторы языка Perl выпускаются на условиях лицензии Artistic License (Артистической лицензии), которая является некоторой вариацией лицензии GNU, и ее смысл заключается в том, что любой, кто выпускает пакет, полученный на основе Perl, должен ясно осознавать, что его пакет не является истинным пакетом Perl. Поэтому все изменения должны быть тщательно документированы и отмечены, выполнимые модули, в случае изменения, должны быть переименованы, а исходные модули должны распространяться вместе с модифицированной версией. Эффект от подобных условий заключается в том, что автор первоначального продукта всегда определяется как его владелец. При использовании Artistic License все условия Общей открытой лицензии GNU остаются в силе, т. е. она продолжает применяться.

Третьим преимуществом языка Perl можно назвать его практическую направленность, т. е. он создавался из практических соображений решения задач администрирования и разработки приложений для UNIX, а это означает, что он обладает следующими важными свойствами:

- ☐ полнотой;
- ☐ простотой использования;
- ☐ эффективностью.

Под полнотой Perl понимается его способность решать все возникающие в системе UNIX в связи с ее администрированием задачи. И это действительно так! Ведь язык Perl, как отмечалось выше, вобрал в себя все наилучшие возможности стандартных средств администрирования UNIX, перечисленных в табл. 1.2.

Таблица 1.2. Стандартные средства администрирования UNIX

Язык программирования	Характеристика
awk	Язык выделения по образцам информации из текстовых файлов
C	Компилируемый язык общего назначения для решения задач низкого уровня
shell	Основной командный язык запуска программ и скриптов, написанных на других языках программирования
sed	Потоковый редактор обработки текстовых файлов

Эти средства продолжают использоваться, так как каждое из них является прекрасным инструментом для выполнения тех задач, для которых они предназначены, однако все то, что можно выполнить, комбинируя эти средства, можно реализовать в одной Perl-программе, изучив только *один* язык. Но возможности Perl не ограничиваются только задачами администрирования. Подключаемые пакеты и модули позволяют легко и быстро решать и другие задачи, для которых, возможно, пришлось бы использовать язык программирования C. Начиная с версии 5.0, язык Perl поддерживает технологию объектно-ориентированного программирования, причем пакеты и модули можно оформить в виде объектов и использовать без знания содержащегося в них кода (хотя придется изучить большое количество объектных моделей со своими свойствами и методами).

Perl — это язык, на котором программист может *делать* свою работу, причем для выполнения одной и той же задачи Perl предлагает несколько средств ее реализации. Одни из них более сложны, другие — менее. Разработчик может выбрать то, которое ему более понятно и которое ему проще применить, не тратя времени на изучение более сложных возможностей. В этом заключается простота использования Perl, которая позволяет применять его как для реализации одноразовых утилит, так и для создания сложных, часто используемых приложений.

Perl является прямолинейным языком, а это означает, что простые программы не надо оформлять в виде головных процедур `main`, как это принято в большинстве процедурных языков программирования, или в форме класса, как принято в объектно-ориентированных языках программирования, т. е. не надо тратить время на дополнительное форматирование исходного

текста программы, а просто начинать писать операторы Perl, которые будут немедленно обрабатываться интерпретатором. Именно в этом заключена эффективность языка программирования Perl.

Четвертое преимущество использования Perl связана с его дополнительными возможностями, позволяющими выполнять не только традиционные задачи администрирования UNIX и обработки текстовых файлов.

И здесь, в первую очередь, следует обратить внимание на простое включение в Perl-программу вызовов библиотечных процедур языка C, что позволяет использовать огромное количество кода, написанного для этого популярного языка. В поставку Perl входят утилиты, конвертирующие заголовки библиотек C в соответствующие эквиваленты языка Perl. Конвертирование осуществляется с помощью XS-интерфейса, который представляет собой простой программный интерфейс, преобразующий среду вызова функций C в среду вызова подпрограмм Perl. Последующий вызов функций C ничем не отличается от вызова подпрограмм самого Perl. Более того, программы Perl версии 5.0 легко интегрируются в приложения C и C++ через интерфейс, реализованный в наборе функций `perl_call_*`.

Для работы с базами данных можно самому написать соответствующее приложение на языке C, а можно воспользоваться свободно распространяемыми модулями дополнительных расширений возможностей Perl, включающих работу с многочисленными популярными системами управления базами данных: Oracle, Ingres, Informix, Interbase, Postgre, Sybase 4 и др.

Способность Perl работать с сокетами TCP/IP сделала его популярным для реализации информационных систем взаимодействия с сетевыми серверами любых типов, использующих сокеты в качестве механизма обмена информацией. Именно эта возможность в сочетании с использованием Perl для создания CGI-сценариев послужила широкому распространению языка на других многочисленных платформах.

И в завершение перечисления достоинств Perl обратим внимание читателя на **пятое** преимущество его использования: так как изначально этот язык являлся свободно распространяемым, то вся наработанная документация также доступна совершенно бесплатно, а так как Perl, как язык сценариев очень популярен, то в Internet находится море документации по его применению для решения разнообразных задач.

(Некоторые адреса можно найти в главе 16.)

1.3. Области применения Perl

Наиболее широко Perl используется для разработки инструментов системного администрирования, однако в последнее время он получил огромную популярность в области разработки Internet-приложений: CGI-сценарии,

системы автоматической обработки электронной почты и поддержки узлов Web. В этом параграфе мы кратко охарактеризуем возможности Perl в каждой из указанных областей.

Системная поддержка UNIX

Как отмечалось ранее, именно задача соединения в одном языке программирования возможностей различных средств системного администрирования UNIX и послужила толчком к разработке и созданию языка Perl. Он и разрабатывался таким образом, чтобы оптимизировать решение именно этих задач, не прибегая к другим инструментам. На настоящий момент язык Perl является основным средством администрирования UNIX, который может выполнять работу нескольких других традиционных средств администрирования. Именно эта его универсальность и способствовала его широкому распространению среди системных администраторов и программистов UNIX, тем более, что он решает задачи обычно быстрее, чем другие аналогичные средства.

CGI-сценарии

Одной из первых, но продолжающей и по настоящее время широко применяться в Интернете технологией реализации динамических эффектов является технология CGI-сценариев, суть которой заключается в обработке информации, получаемой от пользователя, которую он вводит в поля формы страницы HTML, просматриваемой с помощью программы-обозревателя Internet. Информация из полей формы пересылается на сервер с помощью протокола HTTP либо в заголовке, либо в теле запроса и обрабатывается сценарием, который после анализа полученных данных выполняет определенные действия и формирует ответ в виде новой страницы HTML, отсылаемой обратно клиенту. Сценарий может быть написан, собственно говоря, на любом языке программирования, имеющем доступ к так называемым переменным среды, но сценарии Perl получили наибольшее распространение из-за легкости создания и оптимизационных возможностей языка Perl при обработке текстовых файлов. В Internet можно найти буквально тысячи примеров динамического CGI-программирования на Perl.

Его большая популярность для реализации подобных задач на UNIX-серверах Internet привела к тому, что разработчики серверов Internet, работающих в других операционных системах, стали включать возможность подключения сценариев Perl в свои системы. В настоящее время их можно использовать и на сервере Internet Information Server фирмы Microsoft для операционных систем семейства Windows, и на серверах Apache, NCSA и Netscape для операционной системы UNIX.

Обработка почты

Другая популярная область применения Perl — автоматическая обработка электронной почты Internet. Сценарии Perl можно использовать для фильтрации почты на основе адреса или содержимого, автоматического создания списков рассылки и для решения многих других задач. Одной из наиболее популярных программ для работы с электронной почтой является программа Majordomo, полностью реализованная средствами Perl.

Возможности Perl в этой области огромны и ограничиваются только фантазией разработчика. Можно, например, написать сценарий, который обрабатывает входящую почту и добавляет сообщения на заранее созданную страницу новостей, сортируя их по соответствующим тематикам, что позволяет быстро просматривать почту, не тратя время на чтение каждой полученной корреспонденции. По прошествии определенного времени сообщения удаляются со страницы.

Поддержка узлов Web

Узел Web — это ничто иное, как структурированное хранилище страниц HTML, которые являются обычными текстовыми файлами в определенном специальном формате, понимаемом программами просмотра их содержимого. Perl оптимизирован для обработки большого количества текстовых файлов, — поэтому его использование для анализа и автоматического изменения содержимого узла Web само собой вытекает из тех задач, для решения которых он специально и создавался. Его, например, можно использовать для решения задачи проверки правильности перекрестных ссылок на страницах узла Web, как, впрочем, и для проверки правильности ссылок на другие узлы (правда, здесь придется воспользоваться его сетевыми возможностями работы с сокетами).

Его возможности записи и чтения в/из сокетов позволяют использовать сценарии Perl для взаимодействия с другими узлами и получения информации на основе протокола HTTP. Следует отметить, что существуют даже серверы, написанные на Perl. Как упоминалось ранее, именно эти возможности Perl можно использовать для удаления со страниц HTML узла Web ссылок на несуществующие другие узлы.

Perl может работать и с протоколом FTP. Это позволяет автоматизировать получение файлов с других узлов, а в сочетании с его возможностями обработки текстовых файлов позволяет создавать сложные информационные системы.

* * *

В этой главе мы попытались кратко охарактеризовать сам язык Perl, очертить основные области его применения и привлечь внимание читателя к его

дальнейшему изучению и внедрению в собственную практику. В конечном счете только сам программист решает, нужен ему соответствующий язык или нет. Мы думаем, что наш уважаемый читатель уже сделал свой выбор и надеемся, что он не покинет нас до самой последней страницы книги.

Вопросы для самоконтроля

1. Назовите полное наименование языка Perl.
2. Что послужило толчком для разработки и создания Perl?
3. Каково назначение Perl-программы?
4. В чем заключаются преимущества и недостатки интерпретируемых языков?
5. Перечислите основные достоинства языка Perl.
6. Перечислите области применения Perl.

Глава 2



Структура программы

Изучение любого языка программирования начинается с его синтаксиса, одну из неотъемлемых частей которого составляет описание структуры программы, определяющей состав и порядок расположения разнообразных конструкций в теле программы. Мы не будем отступать от сложившихся традиций и объясним необходимые понятия на примере простой программы Perl, получающей информацию от пользователя и в ответ печатающей на экране монитора приветствие.

2.1. Простая программа

Язык Perl — достаточно простой язык программирования, семантика ключевых слов которого соответствует их значению в английском языке, поэтому даже начинающий его изучение программист, во всяком случае, так утверждают его разработчики, без особого труда может разобраться в простой программе Perl. Ну, что ж, может быть так оно и есть, но, как говорится, "лучше один раз увидеть, чем сто раз услышать". Итак, в примере 2.1 приведен текст программы, которая печатает на экране монитора приглашение ввести имя пользователя, а в ответ просто приветствует его.

Пример 2.1. Простая программа-приветствие

```
01 #! /bin/usr/perl  
02  
03 print "Ваше имя?\n";          # Приглашение ввести имя.  
04 $name = <STDIN>;              # Ввод имени с клавиатуры.  
05  
06 $_ = NAME_FORMAT;             # Назначение формата вывода.  
07 write;                        # Вывод приветствия.  
08  
09 $_ = NAME_FORMAT_BOTTOM;      # Вывод нижней разделительной черты.  
10 write;  
11  
12 format NAME_FORMAT=           # Начало описания формата.  
13 Привет, @>>>>>>>>>>!     # Строка вывода.
```

```
14 $name                                # Переменная, значение которой
                                         # подставляется в строку вывода.
15 .                                    # Завершение описания формата.
16
17 format NAME_FORMAT_TOP=             # Заголовок формата NAME_FORMAT.
18 =====
19     Сообщение Perl-программы
20
21 .
22
23 format NAME_FORMAT_BOTTOM=          # Формат вывода нижней разделительной черты
24 =====
25 .
```

Эта программа не совсем типичная для Perl-программы — в ней отсутствуют операторы ветвления, цикла, определение собственных подпрограмм и их вызов и многое другое, но она все же отражает предназначение языка Perl как языка генерирования отчетов, осуществляя вывод приветствия в соответствии с технологией создания отчетов путем определения формата вывода, который в большинстве современных языков либо отсутствует вообще, либо практически не используется программистами.

Если читатель по тексту программы уже понял, как она будет работать (надеюсь, что наши комментарии помогли ему в этом), то теперь самое время проверить его догадку — выполнить эту программу. Но прежде следует в обычном текстовом редакторе набрать ее текст и сохранить в файле с расширением `pl`. Программы Perl являются обычными текстовыми файлами и для их создания можно использовать любой текстовый редактор: в UNIX, например, всегда доступные `vi` и `ed`, а в Windows — Блокнот (`notepad.exe`) или редактор программы Far.

Замечание

Обычно расширение файла (до трех символов) используется для идентификации файлов определенной группы: выполняемые файлы программ (EXE), файлы текстового процессора Word (DOC) и т. д. Для сценариев Perl принято использовать двухбуквенное расширение `pl`.

Для выполнения программы примера 2.1, сохраненной в файле `program1.pl`, в любой операционной системе, имеющей командную строку, достаточно набрать в ней команду:

```
perl program1.pl
```

В результате выполнения программы на экране монитора появится приглашение ввести имя и после ввода имени отобразится приветствие Perl-

программы. Дамп экрана после выполнения нашей программы можно увидеть в примере 2.2, где мы полужирным шрифтом выделили ввод пользователя.

Пример 2.2. Вывод программы примера 2.1

Ваше имя?

Александр

=====

Сообщение Perl-программы

Привет, Александр!

=====

Соответствует ли он вашим соображениям относительно работы этой программы при анализе ее текста? Если да, то мы вас поздравляем, если нет — не надо отчаиваться, несколько наших комментариев поставят все на свои места.

Первый оператор — это специальный комментарий, который для системы UNIX определяет местонахождение интерпретатора perl. Дело в том, что в этой операционной системе можно сделать любой файл выполняемым с помощью команды

```
chmod +x program1.pl
```

и запустить на выполнение из командной строки:

```
./program1.pl
```

В этом случае первая строка программы сопоставляет файлу приложение, которое должно быть загружено для его обработки. Более того, в ней можно определить при необходимости параметры, или ключи, определяющие режим работы приложения, в нашем случае интерпретатора perl. Можно задать отображение предупреждающих сообщений или загрузку отладчика в случае обнаружения серьезной ошибки с помощью следующей строки:

```
#!/bin/usr/perl -w -d
```

(О режимах работы интерпретатора perl и соответствующих ключах см. главу 14.)

Замечание

При работе в операционной системе Windows можно ассоциировать с расширением файла определенную программу, которая будет вызываться при двойном щелчке мышью на любом файле с таким расширением. Программа установки интерпретатора языка Perl фирмы ActiveState Tool Corp., который назы-

вается Active Perl, автоматически устанавливает соответствие файлов с расширением `pl` и интерпретатора языка Perl. При двойном щелчке на файле сценарий действительно выполняется в окне DOS, которое, однако, автоматически закрывается после выполнения последнего оператора, что не позволяет просмотреть отображенные в нем результаты. Чтобы этого не происходило, следует заменить строку вызова приложения, генерируемую программой установкой, на следующую:

```
C:\WINDOWS\Dosprmpmt.pif /cПолный_путь\perl.exe
```

Для этого следует выполнить команду **Параметры** меню **Вид** программы Проводник, в диалоговом окне **Параметры** на вкладке **Типы файлов** в списке **Зарегистрированные типы** выделить **Perl File** и нажатием кнопки **Изменить** отобразить диалоговое окно **Изменение свойств типа файлов**. В этом окне в списке **Действия** выбрать **Ореп** и нажать кнопку **Изменить**. В появившемся диалоговом окне **Изменение действия для типа: Perl File** в поле **Приложение, исполняющее действие:** ввести указанную строку, задав в ключе `/c` полный путь к папке, где расположен двоичный исполняемый файл интерпретатора Active Perl, который называется `perl.exe`.

При работе с интерпретатором Active Perl нет необходимости задавать первую строку, однако если необходимо установить режимы работы интерпретатора, то ее следует задать, причем не обязательно указывать полный путь расположения интерпретатора, достаточно только его имя `perl` и необходимые ключи.

Внимание

Все примеры программ в этой книге проверены в интерпретаторе Active Perl 520 для Windows 95/NT и встроенном интерпретаторе Perl системы Linux RedHat 6.1, соответствующих текущей стабильной версии Perl 5.005.

Вообще любой комментарий в языке Perl начинается с символа `"#"` и распространяется до конца строки. Единственное исключение — это если сразу же после символа комментария следует символ `"!"`.

Внимание

В строке специального комментария не следует вводить ничего, кроме пути местонахождения интерпретатора и его ключей. В противном случае можно получить сообщение об ошибке.

В строке 3 функция `print` посылает на системное устройство вывода (обычно это монитор компьютера) содержимое списка своих параметров, при необходимости преобразуя его в символьное представление. В нашей программе при выполнении этого оператора на экране монитора отобразится содержимое строки, передаваемой функции `print` в качестве параметра. Для тех, кто не знаком с языком C, последовательность символов `"\n"` может показаться странной, тем более, что в дампе экрана (см. пример 2.2) она даже не отображается. Это одна из так называемых управляющих или ESC-

последовательностей, которые предназначены для представления неотображаемых символов — в данном случае символа перехода на новую строку.

Оператор строки 4 ожидает ввод со стандартного устройства ввода (обычно это клавиатура) и присваивает переменной `$name` введенное пользователем имя. Забегая вперед, скажем, что при работе с файлами в Perl определяются дескрипторы файлов, представляющие собой обычные идентификаторы, используемые в программе для ссылки на файлы, с которыми они ассоциированы. В языке имеется несколько predefined дескрипторов файлов, одним из которых является `STDIN`, ассоциированный со стандартным устройством ввода. Операция `<>` выполняет ввод из файла, заданного своим дескриптором.

Оператор строки 6 назначает системной переменной `$~` имя используемого формата для выполнения вывода на стандартное устройство вывода системной функцией `write` из строки 7. Сам формат задается в строках 12-15 оператором `format`, в котором определяется имя формата (`NAME_FORMAT`), завершающееся символом равенства `"=`". В последующих строках до строки 15, содержащей единственный символ точка `"."`, фиксирующий завершение объявления формата, задается сам формат, который обычно представляет собой повторяющуюся последовательность двух строк: строки вывода и строки, перечисляющей через запятую переменные, чьи значения при выводе подставляются вместо полей-держателей, объявленных в строке вывода. Строка вывода представляет собой именно строку, которая выводится функцией `write` в определяемый ее параметром файл (если параметр не задан, то вывод осуществляется на стандартное устройство вывода). В этой строке значащими являются все пробельные символы (сам пробел, символы перехода на новую строку и строки, символ табуляции). Поле-держатель — это специальная конструкция в строке вывода, начинающаяся с символа `"@"`, за которым следует определенное количество специальных символов, задающих длину поля, в которое выводится значение переменной, ассоциированной с полем-держателем и определенной в списке переменных, задаваемом в строке, непосредственно следующей за строкой вывода. Если в строке вывода полей-держателей нет, то строку со списком ассоциированных с ними переменных задавать не надо.

В формате `NAME_FORMAT` определена одна строка вывода с одним полем-держателем, который резервирует поле длиной в 12 символов и определяет, что выводимое значение должно быть прижато вправо (символ `">"`). Это означает, что если значение ассоциированной с этим полем-держателем переменной `$name` будет меньше 12 символов, то в этом поле при выводе они будут выровнены по правому краю. Если выводимое значение преобразуется в строку длиной более 12 символов, она обрезается по правому краю, т. е. отображаются первые 12 символов, считая слева направо.

Если для формата определен формат с таким же именем и суффиксом `_TOP`, то этот формат определяет вид заголовка страницы, который будет отобра-

жаться всякий раз при выводе новой страницы с использованием основного формата. Для формата `NAME_FORMAT` определен формат заголовка в строках 17-19.

Завершается вывод приветствия печатанием разделительной черты функцией `write` строки 10 с использованием формата `NAME_FORMAT_BOTTOM`. Обратите внимание, что для использования нового формата вывода нам пришлось изменить значение системной переменной `$~`. Это следует делать всякий раз, когда необходимо организовать вывод с помощью нового формата.

Итак, мы разработали и выполнили нашу первую программу на языке Perl. Теперь пришло время обратиться к синтаксису языка и рассказать в самых общих чертах, из чего состоит программа на языке Perl, т. е. описать структуру программы.

2.2. Объявления и комментарии

Программа Perl представляет собой последовательность операторов и объявлений, которые обрабатываются интерпретатором в том порядке, как они появляются в тексте программы. Во многих языках программирования обязательны объявления всех используемых переменных, типов и других объектов. Синтаксис Perl является "демократичным" в этом отношении и предписывает обязательные объявления только форматов и подпрограмм.

Объявления форматов и подпрограмм являются глобальными, а это означает, что они "видимы" из любого места сценария. Объявления могут располагаться в программе в любом месте, куда можно поместить оператор, но обычно их размещают либо в начале, либо в конце программы, чтобы быстро найти и откорректировать в случае необходимости. Объявления обрабатываются во время компиляции и не влияют на выполнение последовательности операторов, когда откомпилированный во внутренний код интерпретатора сценарий начинает свою работу.

В Perl нет специального оператора объявления переменной, она определяется при первом ее использовании, причем с помощью ключа `-w` интерпретатора можно задать режим отображения предупреждающих сообщений при попытке использования не инициализированной переменной. Переменные можно определять как глобальные, видимые из любой точки программы, так и с помощью функции `my` как локальные, видимые в определенной части программы — блоке.

(Объявления локальных переменных описываются в главе 3 и в главе 11.)

Можно объявить подпрограмму с помощью оператора `sub`, не определяя ее, т. е. не задавая операторы, реализующие ее функцию. После такого объяв-

ления подпрограммы ее имя можно использовать как операцию, действующую на список, определяемый передаваемыми ей параметрами.

(Более подробно объявления и определения подпрограмм рассматриваются в главе 11.)

Как уже отмечалось выше, если интерпретатор встречает символ `#`, то он игнорирует любой текст, расположенный за ним. Такая конструкция называется комментарием и ее действие распространяется до конца строки после символа `#`. Комментарии используются для документирования программы и не следует ими пренебрегать, так как они вносят ясность в понимание того, что выполняет программа. Однако и злоупотреблять ими не следует, так как слишком большое их количество может ухудшить читаемость программы — одну из важных характеристик любой программы.

Комментарии располагаются в любом месте программы. Их можно разместить непосредственно после оператора в той же самой строке, как в нашем примере 2.1, или занять ими всю строку, если первым символом в ней является символ комментария `#`. Если необходимо временно исключить из потока выполняемых операторов какой-либо из них, то его можно просто закомментировать, не забыв, правда, удалить символ комментария, когда этот оператор снова надо будет включить в поток вычислений.

2.3. Выражения и операторы

Оператор — это часть текста программы, которую интерпретатор преобразует в законченную инструкцию, выполняемую компьютером. С точки зрения синтаксиса языка (способов составления правильных конструкций, распознаваемых интерпретатором) оператор состоит из *лексем* — минимальных единиц языка, которые имеют определенный смысл для интерпретатора. Под минимальной единицей языка понимается такая его единица, которая не может быть представлена более мелкими единицами при дальнейшем ее синтаксическом разборе. В языке Perl лексемами могут быть идентификаторы, литералы, знаки операций и разделитель.

Мы дадим определения всем допустимым в языке лексемам. Хотя их семантика (смысл) может оказаться для начинающих программистов и не совсем ясна, но мы вернемся к некоторым определениям в последующих главах, где уточним их и синтаксис, и семантику в связи с вводимыми элементами языка. Дело в том, что, к сожалению, невозможно описать язык без ссылок вперед.

Идентификатор — это последовательность букв, цифр и символа подчеркивания `"_"`, начинающаяся с буквы или подчеркивания и используемая для именования переменных, функций, подпрограмм, дескрипторов файлов, форматов и меток в программе. Программист может использовать любые правиль-

ные идентификаторы для именования перечисленных объектов программы, если только они не совпадают с *ключевыми словами* языка — предопределенными идентификаторами, которые имеют специальное значение для интерпретатора языка Perl, например `if`, `unless`, `goto` и т. д. Примеры правильных и неправильных идентификаторов представлены в примере 2.3.

Пример 2.3. Правильные и неправильные идентификаторы

Правильные идентификаторы

`myName1`

`my_Name1`

`_myName_1`

Неправильные идентификаторы

`1myName` # Начинается с цифры.

`-myName` # Начинается не с символа буквы или подчеркивания.

`my%Name` # Используется недопустимый для идентификаторов символ

`my` # `my` является зарезервированным словом.

Замечание

Забегая вперед, скажем, что так как имена переменных Perl начинаются со специального символа ("\$", "@", "%"), определяющего их тип, после которого следует идентификатор, то в этом случае использование идентификатора, совпадающего с ключевым словом Perl, является правомочным и не вызывает ошибку интерпретатора. Так, следующие имена переменных являются допустимыми: `$print`, `@do`, `%if`, однако подобная практика не рекомендуется. Это замечание не относится к идентификаторам, используемым для именования дескрипторов файлов и меток, имена которых не начинаются с определенных символов.

(Как используются идентификаторы для объявления переменных см. главу 3.)

(Как используются идентификаторы в дескрипторах файлов см. главу 7.)

(Как используются идентификаторы для объявления форматов см. главу 8.)

Литерал, или **буквальная константа**, — символ или слово в языке программирования, определяющие в отличие от переменной свое собственное значение, а не имя другого элемента языка. Буквальные константы тесно связаны с типами данных, представимыми в языке, и являются, собственно говоря, их представителями. В Perl литералами являются числа и строки.

Пример 2.4. Литералы языка Perl

```
123          # Целое число.
23.56        # Вещественное число с фиксированной точкой.
2E+6         # Вещественное число с плавающей точкой.
"Язык Perl"  # Строковый литерал.
'Язык Perl'  # Строковый литерал.
```

(О литералах см. в главе 3.)

Знаки операций — это один или более специальных символов, определяющих действия, которые должны быть выполнены над величинами, называемыми операндами. Выполняемые действия называются операциями, которые могут быть унарными (применяются к одному операнду), бинарными (применяются к двум операндам) и тернарными (участвуют три операнда).

Пример 2.5. Операции языка Perl

```
++$n;          # Унарная операция (++)
23 * $n;        # Бинарная операция (*)
$n >= 3 ? print "true" : print "false"; # Тернарная операция (?:)
```

(Об операциях и используемых знаках операций см. в главе 4.)

Разделитель — это символ ";", которым завершается любой оператор и который сообщает об этом интерпретатору. Использование разделителя позволяет на одной строке задавать несколько операторов, хотя это и не принято в практике программирования, так как ухудшает читаемость текста программы.

В операторе все его лексемы могут отделяться любым числом *пробельных символов*, к которым относятся сам пробел, знак табуляции, символ новой строки, возврат каретки и символ перехода на новую строку. Поэтому один оператор можно записать на нескольких строках и для этого не надо использовать никакого символа продолжения, требующегося в других языках программирования. Например, оператор номер 4 присвоения данных, введенных с клавиатуры, из примера 2.1 можно записать и так:

Пример 2.6. Использование пробельных символов в операторе

```
$name
=
    <STDIN>
;
```

Можно вообще не использовать пробельные символы в операторе, но для обеспечения читаемости программы мы рекомендуем отделять лексемы одним пробелом, тем более что могут встречаться ситуации, когда интерпретатор не однозначно выделяет лексемы из непрерывного потока символов.

Замечание

Так как пробельные символы не являются значащими в Perl, то обычно их используют для структуризации текста программы, которая заключается в написании некоторой группы операторов, логически подчиненных некоторой конструкции языка, с некоторым отступом относительно этой конструкции. Например, подобный подход можно применить к блоку операторов, выполняющихся в конструкции цикла, сдвинув все их вправо относительно этой конструкции. Структуризация текста программы способствует ее лучшему прочтению и широко практикуется программистами многих языков программирования, например языка C.

Операторы в языке Perl могут быть простыми или составными. *Простой оператор* — это выражение, завершающееся разделителем точкой с запятой ";", и которое вычисляется исключительно ради своего побочного эффекта. Что такое побочный эффект выражения мы определим немного позже, а сейчас остановимся на понятии "выражение".

Выражение — последовательность литералов, переменных и функций, соединенных одной или более операцией, которые вычисляют скаляр или массив, т. е. при обработке интерпретатором выражения единственным действием является *вычисление* значения, а не выполнение некоторых других действий, например присвоение переменной нового значения.

При вычислении выражения могут проявляться *побочные эффекты*, когда при вычислении выражения меняется значение переменной, входящей в выражение. Они могут вызываться, например, операциями увеличения (++) и уменьшения (--) или при вызове функции, которая изменяет значение своего фактического параметра. Как упоминалось выше, простой оператор и выполняется, чтобы реализовать этот побочный эффект, иначе какой смысл просто вычислить выражение, значение которого никоим образом нельзя использовать.

Пример 2.7. Простые операторы

```
++$n;      # Значение переменной $n увеличивается на единицу.  
123*$n;    # Простой оператор без побочного эффекта.
```

Каждый простой оператор может иметь модификатор, который располагается после выражения перед завершающей точкой с запятой. Модификаторы представляют собой ключевые слова *if*, *unless*, *while* и *until*, за которыми следует некоторое выражение. Семантика использования модификатора за-

ключается в том, что простой оператор выполняется, если истинно или ложно выражение, стоящее после модификатора, в зависимости от используемого модификатора. Модификаторы употребляются так же, как и в обычном разговорном английском языке. Например, простой оператор

```
$n++ while <STDIN>;
```

будет выполняться, увеличивая всякий раз значение переменной `$n` на единицу, пока пользователь будет осуществлять ввод с клавиатуры. Остановить выполнение этого оператора можно вводом комбинации клавиш `<Ctrl>+<Z>` или `<Ctrl>+<C>`.

Замечание

Язык Perl вобрал в себя лучшие элементы других языков программирования, в основном C. Конструкция модификаторов заимствована из умершего языка BASIC/PLUS фирмы Digital Equipment Corp.

(Подробно все модификаторы простых операторов рассматриваются в главе 5.)

Чтобы определить конструкцию, называемую составным оператором, нам придется сначала ввести понятие "блок". Последовательность операторов Perl, определяющая область видимости переменных, называется *блоком*. После знакомства с переменными это определение не будет таким туманным, каким оно может показаться сейчас начинающему программисту. Для целей этой главы достаточно мыслить блок как последовательность операторов, заключенную в фигурные скобки:

```
{
оператор_1;
. . .
оператор_n;
}
```

Составной оператор определяется в терминах блока и может быть одного из следующих видов:

Пример 2.8. Составные операторы

```
if (выражение) БЛОК
if (выражение) БЛОК_1 else БЛОК_2
if (выражение_1) БЛОК_1 elsif (выражение_2) БЛОК_2 ... else БЛОК_n
МЕТКА while (выражение) БЛОК
МЕТКА while (выражение) БЛОК_1 continue БЛОК_2
МЕТКА for (выражение_1; выражение_2; выражение_3) БЛОК
```

МЕТКА foreach переменная (список) БЛОК

МЕТКА БЛОК_1 continue БЛОК_2

Обратим внимание читателя на то, что, в отличие от языков программирования C и Pascal, составные операторы Perl определяются в терминах блоков, а не в терминах операторов. Это означает, что там, где нужен блок, он всегда должен задаваться с помощью фигурных скобок. В составных операторах, если даже блок состоит из одного оператора, он должен быть заключен в фигурные скобки. Такой синтаксис не приводит к двусмысленностям и, например, во вложенных операторах условия всегда ясно, с каким if согласуется else или elsif. Метка, представляющая собой идентификатор с двоеточием ":", в составных операторах не обязательна, но если она присутствует, то имеет значение для операторов управления циклами next, last и redo.

(Подробно все составные операторы рассматриваются в главе 5.)

* * *

В этой главе мы познакомились с основными синтаксическими понятиями, используемыми для формирования правильных конструкций языка Perl. Узнали из каких основных элементов состоит Perl-программа, а также разработали и выполнили нашу первую программу.

Вопросы для самоконтроля

1. Что такое лексемы и какими элементами они представлены в языке Perl?
2. Что такое выражение и зачем оно создается?
3. Чем выражение отличается от оператора?
4. Каковы различия между объявлением и оператором?
5. Какие классы операторов имеются в языке Perl?
6. Что представляет собой Perl-программа?

Упражнения

1. Модифицируйте программу примера 2.1, чтобы она всегда приветствовала весь мир, т. е. при ее выполнении всегда отображалось бы "Привет, мир!".
2. Модифицируйте программу примера 2.1, используя для вывода оператор print, имеющий следующий синтаксис:

```
print выражение1, выражение2, . . . , выражениеn;
```

3. Исправьте ошибку в программе:

```
#!/usr/bin/perl Это строка показывает местонахождение интерпретатора
print "Perl";
```

4. Исправьте ошибки в программе:

```
#!/usr/bin/perl -w
$write = 24
$two = 3
$rez = $write * $two
```

5. Исправьте ошибку в программе:

```
#!/usr/bin/perl -w
$write = 24;
# Печать переменной! print $write;
```



Типы данных

Приступая к знакомству с любым языком программирования, мы прежде всего выясняем, а какие типы данных он позволяет обрабатывать, предоставляет ли язык механизм создания новых типов из уже существующих. Ведь язык программирования, как знаковая система обработки информации, как раз и предназначен для обработки информации, представленной данными и алгоритмами. Данные определяются своим *типом* — множеством значений и набором допустимых операций. Одни языки программирования предлагают большое число разнообразных типов данных, как, например, универсальный язык C, в котором даже тип данных, используемый для хранения символов, может быть со знаком или без знака; другие могут обходиться всего лишь двумя типами данных, как, например, VBScript, в котором для хранения и обработки любых скалярных данных (числовых, строковых и булевых) используется единственный вариантный тип данных и существует возможность создания массивов скалярных данных. В конечном счете, язык должен иметь такое разнообразие типов данных, чтобы программист мог с их помощью решать задачи, для которых предназначен язык программирования.

Язык Perl для решения своих задач предлагает всего три типа данных: скаляры, массивы скаляров и ассоциативные массивы скаляров, или хеши. В соответствии с допустимыми типами данных существует и три типа переменных, в которых можно хранить данные перечисленных типов. В этой главе определяются все допустимые в языке типы данных, вводятся числовые и строковые литералы, конструкторы массивов и ассоциативных массивов, а также обсуждаются переменные и их использование, но начнем мы наше изучение с вопроса, без которого невозможно вхождение в любой язык программирования — набора допустимых символов, или алфавита языка.

3.1. Алфавит языка

С чего начинается изучение любого языка? Конечно, с алфавита. Помните, как в первом классе мы усердно выводили в прописях все буквы русского алфавита, а в старших классах, приступая к изучению иностранного языка, мы сначала учили буквы его алфавита и какие звуки они обозначают, а потом из букв складывали слова, которые уже можно было использовать для составления предложений, несущих определенную информативность: "Мама мыла раму".

Аналогично происходит и при изучении языков программирования. Сначала мы должны выяснить, какие можно использовать символы для составления лексем (слов языка), из которых можно конструировать операторы (предложения языка).

В языке Perl можно использовать все буквы латинского алфавита (прописные и строчные), арабские цифры и знак подчеркивания "_". Perl относится к языкам, чувствительным к регистру. Это означает, что символы прописной и строчной буквы считаются *различными*. Поэтому, например, два идентификатора `one` и `One`, используемые для задания имени переменной, — это два различных идентификатора и следовательно, определяемые ими переменные одного и того же типа также являются различными.

Замечание

Буквы национальных алфавитов, в частности русского, можно использовать только в строковых данных. Идентификаторы переменных могут содержать буквы только латинского алфавита.

Кроме букв, цифр и символа подчеркивания, которые называются *алфавитно-цифровыми* символами, используется набор специальных символов, представленный в табл. 3.1.

Таблица 3.1. Специальные символы языка Perl

Символ	Название	Символ	Название
`	Обратный апостроф	~	Тильда
!	Восклицательный знак	@	Коммерческое АТ
#	Номер	\$	Знак доллара
%	Процент	^	"Крышка"
&	Амперсанд	*	Звездочка
-	Минус	+	Плюс
=	Равенство		Вертикальная черта
'	Прямой апостроф	"	Кавычки
<	Меньше	>	Больше
/	Косая черта	\	Обратная косая черта
,	Запятая	.	Точка
:	Двоеточие	;	Точка с запятой
[Квадратная левая скобка]	Квадратная правая скобка
{	Фигурная левая скобка	}	Фигурная правая скобка

Таблица 3.1 (окончание)

Символ	Название	Символ	Название
(Круглая левая скобка)	Круглая правая скобка
?	Вопросительный знак		Пробел

Анализ специальных символов показывает, что в языке Perl используются *все* символы, которые можно ввести с клавиатуры. Если буквенно-цифровые символы используются в составе идентификаторов, то специальные символы служат для определения знаков операций, уточнения синтаксиса выражений, а также именования специальных встроенных переменных языка Perl. Если читателю какой-то из перечисленных объектов сейчас и не совсем ясен, то при последующем изложении все встанет на свои места.

Алфавит языка используется для создания "правильных" (распознаваемых интерпретатором языка) лексем. Среди всего множества таких лексем существует подмножество предопределенных лексем, называемых *ключевыми словами* и используемых для создания правильных конструкций языка. Набор ключевых слов языка Perl не велик и представлен ниже:

```
if, elsif, else, unless, while, until,
foreach, for, next, continue, last,
do, eval, goto, sub, my, return
```

Кроме перечисленных ключевых слов, определяющих синтаксические языковые конструкции, в языке Perl сложился набор стандартных функций, который реализован в любом интерпретаторе perl. Имена этих функций можно тоже считать зарезервированными словами языка и не использовать в качестве имен пользовательских функций или меток в программе. Мы не будем здесь перечислять имена всех стандартных функций, так как их количество достаточно велико, да и мало прока будет от такого простого перечисления, а отошлем читателя к приложению А, где он может увидеть имена всех стандартных функций.

3.2. Скалярный тип данных

Скалярный тип данных в Perl предназначен для представления и обработки числовых данных (чисел) и последовательности символов, называемых строками. Для задания в программе перечисленных данных используются буквенные константы, или литералы: числовые и строковые.

Числовые литералы используются для представления обычных чисел, необходимых для реализации какого-либо алгоритма в программе Perl. Обычно используются числа с основанием десять, или десятичные числа, но язык

позволяет использовать и восьмеричные (с основанием восемь), и шестнадцатеричные (с основанием шестнадцать) числа, которые полезны при работе с содержимым памяти компьютера в процессе решения некоторых системных задач.

Десятичные числа могут быть целыми или вещественными с дробной частью, которые в программировании часто называют числами с плавающей точкой из-за способа их представления и хранения в памяти компьютера. Соответствующие им литералы ничем не отличаются от записи подобных чисел в математике: последовательность цифр без пробелов для целых чисел и последовательность цифр, в которой точка отделяет целую часть от дробной, для вещественных чисел (пример 3.1).

Пример 3.1. Числовые литералы

```
123          # Целое десятичное число.
234.89       # Вещественное число.
0.6780       # Вещественное с нулевой целой частью.
.678         # Незначащие нули можно не задавать.
1_000_000.67 # Для отделения разрядов в целой части числа
              # можно использовать символ подчеркивания.
```

Для вещественных чисел с плавающей точкой можно использовать и экспоненциальную форму записи:

```
[цифры] . [цифры] [E|e] [+|-] [цифры]
```

Эта форма записи означает, что для получения значения числа следует его мантиссу, заданную в форме действительного числа с точкой (`[цифры] . [цифры]`), умножить на десять в степени числа со знаком, заданного в его экспоненциальной части после символа `E` или `e` (пример 3.2).

Пример 3.2. Экспоненциальная форма записи вещественных чисел

```
10.67E56    # Знак "+" в экспоненте можно опускать.
10.67e+06    # Так экспонента лучше читаема.
1e-203       # Число близко к машинному нулю.
1e+308       # Число близко к бесконечно большому числу.
```

Замечание

Интерпретатор языка Perl представляет все числа (и целые, и вещественные) в формате чисел с плавающей точкой удвоенной точности. Это означает, что реально нельзя задать больше шестнадцати значащих цифр мантиссы, а экспо-

нента ограничена диапазоном от -323 до $+308$. Интерпретатор не сгенерирует ошибки, если мантисса будет превосходить 16 цифр, а экспонента 3 цифры, но при отображении таких чисел мантисса будет приведена к шестнадцати значащим цифрам. Если экспонента меньше нижнего предела, то будет выводиться нуль, а если больше верхнего предела, то используется специальный символ `1.#INF`, обозначающий бесконечно большое число. Подобный алгоритм представления очень больших и очень маленьких чисел не приводит к возникновению, соответственно, ошибок переполнения и исчезновения порядка, свойственной многим языкам программирования. Если задать целое число с числом значащих цифр больше 15, то при выводе оно будет отображаться как вещественное в экспоненциальной форме.

Некоторые системные установки или анализ некоторых системных параметров легче осуществлять с использованием чисел, представленных в восьмеричной или шестнадцатеричной системах счисления. Форма записи таких чисел аналогична их синтаксису в языке C: любое целое число, начинающееся с нуля "0", трактуется интерпретатором как восьмеричное целое число, а символы, непосредственно следующие за комбинацией "0x", рассматриваются как шестнадцатеричные цифры. При использовании восьмеричных чисел следует помнить, что в них не может быть цифры больше, чем 7, а в шестнадцатеричных числах кроме десяти цифр от 0 до 9 используются буквы A или a, B или b, C или c, D или d, E или e, F или f для обозначения недостающих цифр числа (пример 3.3).

Пример 3.3. Восьмеричные и шестнадцатеричные числа

```
010    # Восьмеричное 10, равное десятичному 8.
0x10   # Шестнадцатеричное 10, равное десятичному 16.
0239   # Вызовет ошибку интерпретации: нельзя использовать цифру 9.
0xA1Ff # Соответствует 41477 десятичному.
0xGA   # Вызовет ошибку интерпретации: нельзя использовать букву G.
```

Замечание

Задание шестнадцатеричных цифр — это единственный случай в Perl, когда прописные и строчные буквы идентичны. В других случаях их употребления, например в идентификаторах, они различны.

Внимание

Нельзя вместо последовательности символов "0x", идентифицирующей шестнадцатеричные числа, использовать последовательность "0X".

Строковые литералы, или просто *строки*, представляют последовательность символов, заключенную в одинарные ('), двойные (") или обратные (`) ка-

вычки, которая рассматривается как единое целое. Использование одинарных и двойных кавычек для задания строк аналогично их применению для этих же целей в системе UNIX.

В строке, ограниченной одинарными кавычками, нельзя использовать ESC-, или управляющие последовательности, а также в нее нельзя подставить значение переменной. Единственное исключение составляют две управляющие последовательности: (\') и (\\). Первая используется для отображения одинарной кавычки в самой строке, так как иначе интерпретатор рассматривал бы первую, встретившуюся ему одинарную кавычку как признак завершения строки, что не соответствовало бы ее включению в строку. Вторая последовательность используется для отображения самой обратной косой черты. Примеры задания строковых литералов, ограниченных одинарными кавычками, можно найти в табл. 3.2.

Таблица 3.2. Символьные литералы, ограниченные одинарными кавычками

Строка	Отображение	Комментарий
'Простая строка #1'	Простая строка #1	Строка без управляющих последовательностей
'\'perl.exe\''	'perl.exe'	Строка с одинарными кавычками
'D:\\perl.exe'	D:\\perl.exe	Строка с обратной дробной чертой
'Последовательность \\n'	Последовательность \\n	Управляющая последовательность \\n не влияет на отображение строки
'Завтрак	Завтрак	Многострочный символьный литерал отображается в нескольких строках
Бутерброд с ветчиной	Бутерброд с ветчиной	
Чашка кофе'	Чашка кофе	

Замечание

Esc-последовательности, состоящие из обратной косой черты (\), за которой следует буква или комбинация цифр. В них символ обратной косой черты рассматривается как символ, изменяющий значение буквы. Они вместе являются одним целым и выполняют определенное действие при выводе на устройство отображения, например, переход на новую строку (\n). Комбинация цифр трактуется как ASCII-код отображаемого символа. Название таких последовательностей происходит от английского слова "escape", означающего изменять смысл. Их еще называют *управляющие последовательности*.

Строковый литерал может распространяться на несколько строк программы (см. последний литерал табл. 3.2). Для этого при его вводе с клавиатуры следует использовать клавишу <Enter> для перехода на новую строку.

Многострочные литералы отображаются на стольких строках, на скольких они заданы. Это означает, что символ перехода на новую строку, введенный с клавиатуры, сохраняется в символьном литерале, ограниченном одинарными кавычками. Следует заметить, что это справедливо и для строковых литералов, ограниченных двойными кавычками.

Строки в двойных кавычках позволяют вставлять и интерпретировать управляющие последовательности, а также осуществлять подстановку значений переменных, содержащих скаляры или списки. Управляющие последовательности (табл. 3.3) при выводе строк могут интерпретироваться как символы новой строки, табуляции и т. п., а могут изменять регистр следующих за ними букв.

Таблица 3.3. Управляющие последовательности

Управляющая последовательность	Значение
\a	Звонок
\b	Возврат на шаг
\e	Символ ESC
\f	Перевод формата
\n	Переход на новую строку
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\$	Знак доллара
\@	Амперсанд или AT коммерческое
\0nnn	Восьмеричный код символа
\xnn	Шестнадцатеричный код символа
\cn	Эмулирует нажатие комбинации клавиш <Ctrl>+<n>. Например, \cC соответствует <Ctrl>+<C>
\l	Переводит следующий символ в нижний регистр
\u	Переводит следующий символ в верхний регистр
\L	Переводит следующую за ней последовательность символов, ограниченную управляющей последовательностью \E, в нижний регистр
\Q	В следующей за ней последовательности символов, ограниченной управляющей последовательностью \E, перед каждым не алфавитно-цифровым символом вставляет обратную дробную черту

Таблица 3.3 (окончание)

Управляющая последовательность	Значение
\U	Переводит следующую за ней последовательность символов, ограниченную управляющей последовательностью \E, в верхний регистр
\E	Ограничивает действие управляющих последовательностей \L, \Q и \U
\\	Символ обратной дробной черты
\"	Двойные кавычки
\'	Одинарные кавычки

Замечание

Если после обратной косой черты в строковом литерале, ограниченном двойными кавычками, следует символ, который не составляет с ней управляющую последовательность, то обратная косая черта не отображается при выводе строки на устройство отображения.

Строковые литералы в двойных кавычках удобно использовать для структурированного вывода текстовой информации, заданной одной строкой. Примеры строк в двойных кавычках представлены в табл. 3.4.

Таблица 3.4. Символьные литералы, ограниченные двойными кавычками

Строка	Отображение	Комментарий
"\Uline\E #1"	LINE #1	Управляющие последовательности перевода регистра \l, \u, \L и \U действуют только на буквы латинского алфавита и не применимы к буквам русского алфавита
"Конец страницы\f"	Конец страницы	При выводе на экран монитора или в файл в конце строки отображается символ перехода на новую страницу; при выводе на принтер печать начинается с новой страницы после вывода этой строки
"\tЗавтрак\nБутерброд с ветчиной\nЧашка кофе\n"	Завтрак Бутерброд с ветчиной Чашка кофе	Символьный литерал задан одной строкой с управляющими символами

Последней разновидностью строковых литералов являются строки в обратных кавычках, которые, по существу, не являются строками данных в том смысле, что содержащиеся в них символы не обрабатываются при выводе интерпретатором языка Perl как некоторый поток отображаемых символов. Встретив строку в обратных кавычках, интерпретатор передает ее на обработку операционной системе, под управлением которой он функционирует: Windows, UNIX или какая-либо другая, которая выполняет переданную ей команду и возвращает в программу Perl результаты ее выполнения в виде строки, которую в дальнейшем можно использовать для организации вычислений. Таким образом, строки в обратных кавычках должны содержать значимую для операционной системы последовательность символов: команду операционной системы, строку загрузки приложения и т. п. Например, при выводе строки ``dir`` оператором `print` отобразится не слово "dir", а результат выполнения команды `dir` операционной системы. В системе Windows эта команда отобразит содержимое текущей папки (пример 3.4).

Пример 3.4. Результат выполнения оператора `print `dir``;

Том в устройстве D не имеет метки

Серийный номер тома: 1F66-19F2

Содержимое каталога D:\PerlOurBook

```
.           <КАТАЛОГ>      09.01.00  16:01 .
..          <КАТАЛОГ>      09.01.00  16:01 ..
EXAMPLE PL          32  23.01.00  11:56 example.pl
01          <КАТАЛОГ>      11.01.00  14:12 01
02          <КАТАЛОГ>      11.01.00  14:12 02
03          <КАТАЛОГ>      11.01.00  14:12 03
PERLINF TXT         1 781  12.01.00  11:39 perlinf.txt
EXAMPLE1 PL         347  18.01.00  18:02 example1.pl
    3 файл(а,ов)                2 160 байт
    5 каталог(а,ов)             78 086 144 байт свободно
```

В Perl, как и в UNIX, строки в обратных кавычках используются для "ввода" в программу результатов выполнения не только системных команд, но и выводимых на экран монитора результатов выполнения другой программы, так как всегда можно передать на выполнение командной оболочке имя загружаемого модуля программы.

Замечание

Некоторые символы (их еще называют метасимволы) имеют специальное значение для командной оболочки. К ним относятся *, <, >, ?, | и &. В системе

UNIX, чтобы изменить интерпретацию метасимвола как символа со специальным значением, ставят перед ним обратную косую черту, которая изменяет (escape) его специальное назначение. Теперь он рассматривается командной оболочкой просто как символ, представляющий самого себя. Если во вводимой строке много таких специальных символов, то пользователю необходимо перед каждым поставить обратную косую черту, что приводит к плохой читаемости всей строки. Во избежание подобных "затруднений" в UNIX используются строки в одинарных кавычках, в которых все символы интерпретируются так, как они есть. Подобную же функцию одинарные кавычки выполняют и в языке Perl.

При работе в UNIX широко используется подстановка значений переменных в строку команды, которая передается на обработку оболочке shell. При задании команды в строке ввода используются строки в двойных кавычках, которые так же, как и строки в одинарных кавычках, отменяют специальные значения метасимволов, за исключением символа \$, который используется для подстановки значения переменной. Обратная косая черта перед ним изменяет его специальное значение. Именно этот механизм строк в двойных кавычках послужил прототипом для аналогичных конструкций в языке Perl.

Строка в обратных кавычках используется в UNIX для подстановки стандартного вывода команды, т. е. содержимое строки в обратных кавычках интерпретируется командной оболочкой как команда системы, которая должна быть выполнена, а результат ее выполнения подставлен вместо строки в обратных кавычках. В Perl эта конструкция перенесена без всяких изменений.

Все обрабатываемые программой данные хранятся в некоторой области памяти компьютера, определяемой своим адресом. Для удобства программирования доступа к данным языки высокого уровня, и Perl здесь не исключение, используют *переменные*, с помощью которых программист может ссылаться на данные, расположенные в памяти, или изменять их содержание. Переменная задается своим именем, которое используется программой для обращения к области памяти и извлечения хранящихся в ней данных или, наоборот, записи данных в область памяти. Обычно говорят, что в переменных хранятся данные, хотя, как мы видим, это не совсем так. Правильнее говорить, что переменная определяет именованную область памяти, в которой хранятся некоторые данные.

Более того, переменная определяет тип данных, хранимых в области памяти, на которую она ссылается. В большинстве языков программирования переменные до их использования в программе объявляются как переменные определенного типа, информируя транслятор, что в них можно хранить данные соответствующего типа. Как мы помним, в Perl нет операторов объявления переменных определенного типа; они автоматически объявляются при первом их использовании в конструкциях языка, например в операторе присваивания значения переменной. Любая переменная определяется заданием своего имени, которое представляет собой правильный идентификатор языка. В Perl имя любой переменной состоит из специального символа (префикса), определяющего тип переменной, за которым следует идентифи-

катор. Для переменных скалярного типа (пример 3.5), или просто скалярных переменных, таким определяющим символом является знак доллара "\$".

Пример 3.5. Скалярные переменные

```
# Правильные имена скалярных переменных.  
$Name; $name_surname; $name_1;  
  
# Неправильные имена скалярных переменных.  
$1_name; # Идентификатор не может начинаться с цифры.  
$Name@Surname; # Недопустимый символ @
```

Скалярная переменная может хранить только одно скалярное данное: числовое или строковое, причем не существует никакого способа определить, какой именно тип скалярных данных в ней содержится. Дело в том, что при использовании этих переменных в различных операциях хранимые в них данные *автоматически* преобразуются из одного типа в другой, т. е. в арифметических операциях строка преобразуется в число, а в строковых операциях числовое значение преобразуется в строковое. Преобразование строки в числовое значение осуществляется, если она содержит последовательность символов, которая интерпретируется как число, иначе интерпретатор генерирует ошибку. Шестнадцатеричные числа с префиксом "0x" и десятичные числа с символом подчеркивания для отделения триад в целой части числа, заданные как строки, не преобразуются в числа, а последовательность цифр, начинающаяся с 0, не интерпретируется как восьмеричное число. Для преобразования строк, содержащих представления шестнадцатеричных и восьмеричных чисел, в числовые значения следует пользоваться функцией `oct`.

Как отмечалось выше, строки в двойных кавычках не только интерпретируют управляющие символы, но и позволяют подставлять значения скалярных переменных. Это означает, что в строке можно задать имя переменной, которое при вычислениях заменяется значением, содержащимся в переменной на момент вычислений. (Подобную процедуру подстановки значения переменной в символьную строку в дальнейшем для краткости мы будем называть просто *подстановкой переменной*.) Например, следующая последовательность операторов

```
$s = "\$10";  
$n = "Книга стоит $s долларов.";   
print $n;
```

отобразит на экране монитора строку

Книга стоит \$10 долларов.

Замечание

Можно подставлять значения не только скалярных переменных, но и массивов скаляров, элементов массивов и хешей, а также сечений массивов и хешей. Об этом мы расскажем в следующих параграфах этой главы, определяя соответствующие типы данных и их переменные.

При подстановке переменной ее имя должно быть отделено разделителями от остальных символов строки, причем это правило не обязательно для первого символа имени переменной, так как интерпретатор, встретив символ "\$" в строке, ограниченной двойными кавычками, начинает выделять правильный идентификатор.

Разделителями могут быть пробелы или управляющие последовательности. Можно и явно указать идентификатор переменной, задав его в фигурных скобках. Подобная техника демонстрируется следующим фрагментом программы:

```
$day = 'пятницу';  
$number = 5;  
$html = "HTML";  
$s = "${html}-документ отослан в\n$day\t$number февраля.";   
print $s;
```

Результатом выполнения этого фрагмента будет отображение двух строк на экране монитора:

```
HTML-документ отослан в  
пятницу 5 февраля.
```

Переменная `$html` подставляется с явным указанием ее идентификатора, для выделения идентификаторов остальных переменных используются разделители.

Подставлять можно скалярные переменные, значения которых определены с помощью числовых и любых типов строковых литералов, причем строка в обратных кавычках интерпретируется как команда операционной системы.

Замечание

Рассмотренные в этом параграфе различные типы кавычек для задания строковых литералов на самом деле являются всего лишь удобной формой записи операций языка Perl: `q//`, `qq//`, `qx//`. (Эти операции подробно будут рассмотрены в главе 4).

Синтаксический анализатор языка Perl при анализе текста программы выделяет слова (не заключенная в кавычки последовательность алфавитно-цифровых символов) и определяет их принадлежность набору ключевых слов. Если слово не является ключевым, то интерпретатор рассматривает его

как строку символов, заключенную в кавычки. Это позволяет задавать строковые литералы, не заключая их в кавычки:

```
$day = Friday; # Тождественно оператору $day = 'Friday';
```

Такие слова без кавычек в тексте программы иногда еще называют *простыми словами* (barewords).

Внимание

Задание строковых литералов без кавычек возможно *только* для литералов, содержащих буквы латинского алфавита. Попытка применить подобную технику для литералов, содержащих буквы русского алфавита, приведет к ошибке компиляции.

Завершая разговор о литералах, следует упомянуть о специальных литералах языка Perl: `__LINE__`, `__FILE__`, `__END__` и `__DATA__`. Они являются самостоятельными лексемами, а не переменными, поэтому их нельзя вставлять в строки. Литерал `__LINE__` представляет номер текущей строки текста программы, а `__FILE__` — имя файла программы. Литерал `__END__` используется для указания логического конца программы. Информация, расположенная в файле программы после этого литерала, не обрабатывается интерпретатором, но может быть прочитана через файл с дескриптором `DATA`. Последний литерал `__DATA__` аналогичен литералу `__END__`, только дополнительно он открывает файл с дескриптором `DATA` для чтения информации, находящейся в файле программы после него. Программа примера 3.6 демонстрирует использование специальных литералов.

Пример 3.6. Использование специальных литералов

```
#!/perl520/bin/perl -w
$file = __FILE__;
$prog = __LINE__;
print "Находимся в строке: $prog\n",
      "Файл: $file";;
__END__
print "Текст после лексемы __END__";
```

Результат работы этой программы будет следующим, если файл программы хранится в файле `D:\PerlEx\example1.exe`:

```
Находимся в строке: 3
Файл: D:\PERLEX\EXAMPLE1.PL
```

Вывода последнего в программе оператора печати не наблюдается, так как он расположен после лексемы `__END__`.

3.3. Массивы скаляров

Массив, в отличие от скалярного типа данных, представляющего единственное значение, — это тип данных, предназначенный для хранения и обработки нескольких скалярных данных, ссылаться на которые можно с помощью индекса. Все массивы Perl одномерны и их можно мыслить как некий линейный список скаляров. Для извлечения какого-либо значения, хранимого в массиве, достаточно одного индекса. В программе массив задается с помощью специальной синтаксической конструкции языка Perl, называемой *конструктором* массива. Он представляет собой список скалярных значений, заключенный в круглые скобки. Элементы списка отделяются друг от друга запятыми и представляют элементы массива:

```
(скаляр_1, скаляр_2, ... , скаляр_n)
```

Как и в других языках программирования, массив Perl представляет набор *однотипных* данных — скаляров, но скалярные данные могут быть как числовыми, так и строковыми, поэтому, с точки зрения других языков программирования, в массивах Perl хранятся смешанные данные — числа и строки. В качестве скаляра в конструкторе массива может использоваться числовой или строковый литерал или скалярная переменная, чье значение и будет являться элементом массива:

```
(5, "умножить на", 4) # Используются только литералы.
```

```
($first, 'равно', $second) # Используются значения скалярных переменных.
```

Массив может состоять из неограниченного числа элементов, а может не иметь ни одного. В таком случае его называют *пустым массивом*. Конструктор пустого массива — круглые скобки без содержащегося в них списка `()`.

Как отмечалось в начале параграфа, массив характеризуется тем, что к любому его элементу можно обратиться при помощи индекса (целого литерала или скалярной переменной, принимающей целое значение), который задается в квадратных скобках непосредственно после конструктора массива. Индексы массивов в языке Perl начинаются с 0, а отрицательные индексы позволяют обратиться к элементам в обратном их заданию порядке:

```
(5, "умножить на", 4)[0] # Первый элемент массива: 5.
```

```
(5, "умножить на", 4)[2] # Последний элемент массива: 4.
```

```
(5, "умножить на", 4)[-1] # Последний элемент массива: 4.
```

```
(5, "умножить на", 4)[-3] # Первый элемент массива: 5.
```

Внимание

При использовании отрицательных индексов индекс `-1` соответствует последнему элементу массива, а индекс `-n`, где `n` — количество элементов массива, первому элементу массива.

Внимание

Нельзя использовать индекс для извлечения элементов списка, возвращаемого некоторыми функциями Perl. Индекс можно применять только к массивам или их конструкторам, а для этого достаточно заключить список в круглые скобки. Например, конструкция `stat(@m)[0]` вызовет ошибку компиляции, если возвращаемым значением функции `stat` является список, тогда как конструкция `(stat(@m))[0]` приведет к желаемому эффекту.

В качестве элемента списка в конструкторе массива можно использовать конструктор массива, но мы не получим в этом случае, как ожидает читатель, знакомый с другими языками программирования, массив массивов, т. е. многомерный массив, к элементам которого можно обратиться с помощью нескольких индексов. Дело в том, что если в качестве элемента списка в конструкторе массива используется конструктор массива, то его элементы добавляются к элементам массива, определяемого внешним конструктором, т. е. каждый его элемент становится элементом внешнего массива, увеличивая количество его элементов на количество элементов внутреннего массива. Например, массив

```
(1, (2, 3), 4)
```

эквивалентен массиву

```
(1, 2, 3, 4)
```

(О реализации многомерных массивов при помощи ссылок см. в главе 9.)

В программе массивы хранятся в специальных переменных, имена которых начинаются с символа `@`. Объявление переменной массива, или просто массива, чаще всего осуществляется в операторе присваивания (`=`), в правой части которого обычно задается конструктор массива:

```
@array = ($m1, '+', $m2, '=', $m1+$m2);
```

В этом примере также продемонстрировано, что в конструкторе массива можно использовать выражения, о которых речь пойдет в следующей главе.

Задать или получить значения элементов массива, хранящегося в переменной, можно и с помощью индекса. Однако "операцию" индексирования нельзя применять непосредственно к имени переменной массива, ее следует применять к переменной, в которой первый символ заменен на символ скалярной переменной `$`. Подобное "неудобство" связано с последовательным проведением в Perl использования первого символа переменной для указания ее типа: ведь элемент массива является ничем иным как скаляром. Примеры использования индекса с переменными массива представлены ниже:

```
$m1[0] = "Первый";           # Задает первый элемент массива @m1.  
$m1[1] = "Второй";          # Задает второй элемент массива @m1.  
@m2 = (1, 2, 3, 4, 5);      # Задание массива @m2.
```

```
print @m1, "\n", $m2[0]; # Печать массива @m1 и  
# первого элемента массива @m2.
```

Массивы в Perl являются динамическими. Добавление нового элемента в массив автоматически увеличивает его размер. С помощью индекса можно добавить элемент, отстоящий от последнего определенного элемента массива на любое число элементов, и это действие не приведет к ошибке. Просто все промежуточные элементы будут не определены (их значение будет равно пустой строке ""). При такой реализации массивов программисту не надо заботиться, как в других языках программирования, что индекс выйдет за размеры массива. В случае обращения к не существующему элементу массива Perl возвращает значение, равное нулевой строке.

Замечание

Если для интерпретатора perl включен режим отображения предупреждающих сообщений во время выполнения, то в случае обращения к не определенному или не существующему элементу массива будет отображаться сообщение, что значение элемента не определено.

В любой момент можно определить число элементов массива. Для этого следует воспользоваться синтаксической конструкцией

```
$#имя_массива
```

которая возвращает максимальный индекс массива или -1, если массив не определен. Чтобы получить количество элементов массива, следует к возвращаемому значению этой конструкции добавить 1, так как индексы массивов в Perl начинаются с 0.

При работе с массивами самой "утомительной" процедурой может оказаться процедура присваивания значений элементам массива. Хорошо, если все необходимые данные для заполнения массива большой размерности уже существуют в каком-либо текстовом файле. Тогда можно присвоить значения элементам массива, воспользовавшись операцией чтения из внешнего файла, но об этом в следующей главе. А если необходимо в программе создать массив натуральных чисел до 1000 включительно? Неужели надо писать 1000 операторов присваивания, или организовывать цикл, в котором осуществлять соответствующие присваивания, или создавать текстовый файл, содержащий эти числа? К счастью, нет! В Perl для подобных ситуаций предусмотрена операция диапазон, которую можно использовать в конструкторе массива. Например, чтобы создать массив натуральных чисел до 1000, достаточно одного оператора:

```
@naturalNumbers = (1..1000);
```

Общий синтаксис операции диапазон следующий:

```
первое_число..последнее_число
```

Она определяет набор чисел (целых или вещественных), начинающихся с `первое_число` и не превосходящих `последнее_число`, в котором последующее больше предыдущего на единицу. Эта операция действительна не только для чисел, но и для алфавитно-цифровых строк, но в этом случае увеличение на единицу означает увеличение на единицу ASCII-кода символа строки. Ниже показаны примеры использования операции диапазон со строковыми данными:

```
"a".."c"      # Соответствует: "a", "b", "c"
"BCY".."BDB" # Соответствует: "BCY", "BCZ", "BDA", "BDB"
```

Эта операция особенно удобна для задания целых чисел, начинающихся с нуля, например, для представления дня месяца в дате вида "01.02.2000":

```
@day_of_month = ("01".."31");
```

(Подробнее операция диапазон рассматривается в главе 4.)

Переменные массива и элемент массива можно подставлять в строки, ограниченные двойными кавычками. Если интерпретатор встречается в строке переменную массива, то он вставляет в нее значения элементов массива, отделенные друг от друга пробелами. Результатом выполнения следующих операторов

```
@m = (1..3);
print "Числа{@m}являются целыми";
```

будет строка

```
Числа1 2 3 являются целыми
```

Все правила определения идентификатора скалярной переменной при ее подстановке в строку переносятся и на переменную массива.

В связи с возможностью подстановки переменной массива можно указать быстрое решение проблемы распечатки содержимого массива. Дело в том, что если в операторе печати просто указать переменную массива, то его элементы выводятся сплошной строкой без пробелов между ними, что является неудобным при работе с числовыми данными. Если вставить переменную массива в строку, то при ее печати между элементами массива будут вставлены пробелы. Следующие два оператора печати

```
print @m, "\n";
print "@m\n";
```

отобразят массив `@m` предыдущего примера по-разному:

```
123
1 2 3
```

Подстановка в строку элемента массива с помощью индексного выражения ничем не отличается от подстановки скалярной переменной: элемент массива

ва, полученный с помощью индекса, является скалярной величиной. Однако здесь может возникнуть одна интересная проблема, если в программе определена скалярная переменная с таким же идентификатором, что и у массива. Значение этой переменной или значение соответствующего элемента массива будет вставлено в строку? Например, если в программе определена скалярная переменная `$var` и массив `@var` (о том, почему это возможно, мы расскажем в разделе 3.5 данной главы), то значение переменной или элемента массива будет вставлено в строку `"$var[0]"`.

Правильный ответ — значение элемента, так как при синтаксическом анализе интерпретатор будет рассматривать встретившуюся последовательность символов как лексему `$var[0]`, а не как имя переменной `$var` с последующим символом `"["`. Если необходимо использовать значение скалярной переменной `$var` в строке, то можно предложить три способа решения этой проблемы:

```
"${var}[0]"      # Фигурные скобки ограничивают символы, рассматриваемые
                  # интерпретатором как единое целое с символом $.
"$var\[0]"       # Обратная дробная черта ограничивает идентификатор
                  # переменной.
"$var" . "[0]"   # Конкатенация строк (операция "." ) позволяет однозначно
                  # интерпретировать переменную в первой строке.
```

Иногда для работы необходимо выделить некоторое подмножество элементов массива, которое мы будем называть *фрагментом массива*. Можно ее выполнить просто, но не эффективно: присвоить элементам некоторого нового массива значения соответствующих элементов старого, можно воспользоваться специальной конструкцией Perl для выделения фрагмента массива. Если после имени переменной массива в квадратных скобках задать список индексов некоторых элементов массива, то такая конструкция и будет определять фрагмент массива, причем индексы не обязательно должны идти в каком-то определенном порядке — их можно задавать произвольно. Для выделения фрагмента, состоящего из последовательно идущих элементов массива, можно использовать знакомую нам операцию диапазон. Фрагмент массива сам является массивом, и поэтому его можно использовать в правой части оператора присваивания. Несколько примеров создания фрагментов массива приведено ниже:

```
@m = (10..19);      # Исходный массив:
                    #      (10, 11, 12, 13, 14, 15, 16, 17, 18, 19).
@m[0, 2, 4, 6, 8];  # Фрагмент 1: (10, 12, 14, 16, 18).
@m[6, 4, 5, 8, 6];  # Фрагмент 2: (16, 14, 15, 18, 16).
@m[2..4];           # Фрагмент 3: (12, 13, 14).
@m[8, 2..4, 0];     # Фрагмент 4: (18, 12, 13, 14, 10).
```


Внимание

При выделении фрагмента массива используется имя переменной массива, начинающейся с символа "@", тогда как при ссылке на элемент массива префикс имени переменной заменяется на символ "\$". Здесь опять прослеживается последовательное использование префикса для задания типа переменной. Фрагмент массива является массивом, а потому следует использовать символ "@".

3.4. Ассоциативные массивы

Ассоциативные массивы, называемые также *хеш-массивами* или просто *хешами*, — это то, чем гордятся программисты на языке Perl. Они позволяют легко создавать динамические структуры данных — списки и деревья разных видов, с помощью которых можно реализовать функциональность простой системы управления базой данных. Подобной конструкции не найти ни в одном современном языке программирования.

Ассоциативные массивы отличаются от массивов скаляров тем, что в них для ссылки на элементы используются строки, а не числовые индексы, т. е. концептуально они представляют собой список не просто значений элементов массива, а последовательность ассоциированных пар ключ/значение. Ключ является строковым литералом, и именно он и используется для доступа к ассоциированному с ним значению массива.

В программе хеши задаются аналогично массивам скаляров с помощью конструктора, представляющего собой список, заключенный в круглые скобки, в котором пары ключ/значение следуют друг за другом:

(ключ_1, значение_1, ключ_2, значение_2, ... , ключ_n, значение_n)

Для хранения ассоциативных массивов, как и для других элементов данных, используются переменные, первым символом которых является символ процента "%". Переменные, в которых хранятся ассоциативные массивы, часто называют *хеш-переменными*. Ассоциативный массив создается во время операции присвоения такой переменной списка значений:

```
%m = ("Имя", "Ларри", "Фамилия", "Уолл");
```

Замечание

Для краткости мы будем иногда говорить, что при создании массива его переменной присваивается список, подразумевая при этом, что в правой части операции присваивания задан конструктор массива.

В ассоциативном массиве %m ключами являются строки "Имя" и "Фамилия", а ассоциированными с ними значениями, соответственно, "Ларри" и "Уолл". Теперь, чтобы получить значение, соответствующее какому-либо ключу, следует воспользоваться конструкцией:

```
$m{"ключ"}
```

Обратите внимание, что при работе с *элементом* ассоциативного массива, символ хеш-переменной "%" заменяется на символ скалярной переменной "\$". Аналогично мы поступали и при ссылке на элемент массива скаляров. Единственное отличие — ключ задается в фигурных скобках. Итак, чтобы, например, присвоить некоторой скалярной переменной значение элемента хеш-массива %m, следует воспользоваться следующим оператором:

```
$surname = $m{"Фамилия"};
```

Теперь скалярная переменная \$surname имеет в качестве своего значения строку Уолл.

Замечание

Интерпретация списка как последовательности пар ключ/значение происходит *только* при операции его присвоения хеш-переменной. Если список присваивается переменной массива скаляров, то он интерпретируется как простая последовательность значений элементов массива.

Инициализация хеш-массива с помощью списка, элементы которого отделяются друг от друга символом запятой ",", не очень удобно, так как в длинном списке трудно выделять соответствующие пары ключ/значение. Для улучшения наглядности пару ключ/значение можно соединить последовательностью символов "=>", заменив ей разделяющую запятую в списке. По правде говоря, и запятая ",", и символы "=>" представляют собой знаки операций в Perl, причем операция "=>" эквивалентна операции "запятая" с той лишь разницей, что ее левый операнд всегда интерпретируется как строковый литерал, даже если он не заключен в кавычки.

Замечание

Интерпретация левого операнда операции "=>" как строкового литерала справедлива для последовательности символов, в которой используются буквы латинского алфавита. Буквы русского алфавита вызовут ошибку интерпретации, так как последовательность символов не будет распознана как слово языка Perl.

Рассмотренный нами ранее хеш-массив %m можно инициировать и таким способом:

```
%m = (  
    "Имя"      => "Ларри",  
    "Фамилия"  => "Уолл"  
);
```

Если бы мы хотели в качестве индексов имени и фамилии использовать английские слова name и surname, то этот же ассоциированный массив можно было бы задать следующим оператором:

```
%m = (
    Name      => "Ларри",
    Surname   => "Уолл"
);
```

Добавить новый элемент ассоциативного массива или изменить значение существующего очень легко. Достаточно присвоить его элементу, определяемому заданным ключом, значение в операторе присваивания:

```
$m{"Имя"} = "Гарри";           # Изменили значение существующего элемента.
$m{"Телефон"} = "345-56-78";  # Добавили новый элемент.
```

Если при использовании подобной конструкции ассоциативный массив еще не существовал, то при выполнении операции присваивания сначала будет создан сам массив, а потом присвоится значение его элементу. Ассоциативные массивы, как и массивы скаляров, являются динамическими: все добавляемые элементы автоматически увеличивают их размерность.

Удалить элемент ассоциативного массива можно только с помощью встроенной функции `delete`:

```
delete($m{"Телефон"}); # Удалили элемент с ключом "Телефон".
```

Совет

При изменении значения элемента ассоциативного массива с помощью ключа следует проверять правильность его задания (отсутствие дополнительных пробелов, регистр букв), так как в случае несоответствия заданного ключа элемента с существующими в хеш-массив просто добавится новый элемент с заданным ключом.

При работе с ассоциативным массивом часто требуется организовать перебор по множеству всех его ключей или значений. В языке существуют две встроенные функции — `keys` и `values`, которые представляют в виде списка, соответственно, ключи и значения ассоциативного массива. Следующий фрагмент программы

```
print keys(%m), "\n";  # Печать ключей.
print values(%m), "\n"; # Печать значений.
```

отобразит на экране монитора строку ключей и строку значений массива `%m`

```
ФамилияИмяТелефон
УоллЛарри345-56-11
```

Обратите внимание, что они отображаются не в том порядке, как задавались с помощью конструктора массива.

Замечание

При создании элементов ассоциативного массива они сохраняются в памяти в порядке, удобном для их последующего извлечения. Поэтому при его печати

последовательность элементов не соответствует порядку их задания. Для упорядочивания значений хеш-массивов следует воспользоваться встроенной функцией сортировки.

Итак, хеш-массивы позволяют обращаться к своим элементам не с помощью числового индекса, а с помощью индекса, представленного строкой. Но что же здесь такого замечательного, какие возможности предоставляет подобная конструкция? Огромные. И первое, что приходит на ум, — это использовать ключи как аналог ключей реляционных таблиц. Правда, хеш-массивы не позволяют непосредственно хранить запись, а только один элемент, но и этого уже достаточно, чтобы создать достаточно сложные структуры данных (пример 3.7).

Пример 3.7. Использование ключей для получения связанной информации

```
#!/ perl -w

%friend = (
    "0001", "Александр Иванов",
    "0002", "Александр Петров",
    "0003", "Александр Сидоров"
);

%city = (
    "0001", "Санкт-Петербург",
    "0002", "Рязань",
    "0003", "Кострома"
);

%job = (
    "0001", "учитель",
    "0002", "программист",
    "0003", "управляющий"
);

$person = "0001";

print "Мой знакомый $friend{$person}\n";
print "живет в городе $city{$person}\n";
print "и имеет профессию $job{$person}.\n";
```

В этом примере создана простейшая база данных знакомых, в которой хранятся их имена, места жительства и профессии. Перечисленная информация содержится в разных хеш-массивах с одинаковым набором ключей, которые и связывают информацию по каждому человеку. Выполнение программы примера 3.6 приведет к отображению на экране монитора следующего текста:

Мой знакомый Александр Иванов
живет в городе Санкт Петербург
и имеет профессию учитель.

Если изменить значение переменной `$person` на другой ключ, то отобразится связанная информация о другом человеке.

Это только простейший пример, который может навести читателя на более плодотворные идеи применения хеш-массивов, одну из которых нам хотелось бы сейчас наметить: создание связанного списка.

Связанный список — это простейшая динамическая структура данных, расположенных в определенном порядке. Каждый элемент связанного списка состоит из некоторого значения, ассоциированного с данным элементом, и ссылки на следующий элемент списка. Последний элемент списка не имеет ссылки на следующий, что обычно реализуется в виде пустой ссылки. Для окончательного задания связанного списка следует объявить переменную, указывающую на первый элемент списка, которую называют заголовком. Для связанного списка определяются операции выбора, удаления и добавления элемента списка относительно заданного. Графически связанный список можно представить так, как показано на рис. 3.1, где указатель на следующий элемент обозначен серым цветом.

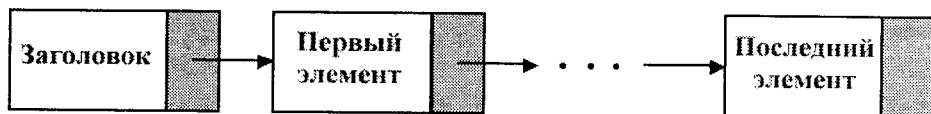


Рис. 3.1. Графическое представление связанного списка

С помощью хеш-массивов связанный список реализуется просто. Для этого следует значение элемента задать в качестве ключа для следующего за ним элемента списка, определив таким образом указатель на следующий элемент. Значением последнего элемента в хеш-массиве (с ключом, равным значению последнего элемента связанного списка) будет пустая строка `""`. Переменная-заголовок должна иметь значение, равное ключу первого элемента списка. В примере 3.8 показана реализация связанного списка, а также добавление нового элемента.

Пример 3.8. Реализация связанного списка

```
%linked_list = (  
"начало" => "первый",  
"первый" => "третий",
```

```

"третий" => ""
);
$header = "начало";

# Добавление элемента со значением "второй"
# после элемента со значением "первый".
$temp = $linked_list{"первый"};      # Запомнили старый указатель.
$linked_list{"второй"} = $temp;      # Добавили новый элемент.
$linked_list{"первый"} = "второй";  # Указатель на новый элемент.

$item = $header;

# Печать нового связанного списка.
while ($linked_list{$item}) {        # Пока не дойдем до пустой строки ""
print $linked_list{$item}, "\n";    # будем печатать значения элементов.
$item = $linked_list{$item};
}

```

Результатом выполнения программы примера 3.8 будет печать значений элементов нового связанного списка в следующем порядке:

```

первый
второй
третий

```

Этот пример только демонстрационный, чтобы показать легкость реализации подобной динамической структуры. При действительной реализации связанного списка следует все возможные действия оформить в виде функций, которые в дальнейшем использовать для работы со связанным списком.

3.5. Переменные

Итак, мы познакомились с тремя основными типами данных и соответственно тремя типами переменных языка Perl, используемых для их обработки и хранения в программе. В этом последнем параграфе данной главы мы суммируем наши знания о переменных и дополним некоторыми аспектами, связанными с их реализацией в интерпретаторе и использованием в простейшей операции присваивания.

Напомним, что переменную можно рассматривать как именованную область памяти, которая не только предоставляет возможность обращаться к содержимому этой области памяти по имени переменной, но и задает тип хранимых в ней данных, определяя, таким образом, формат хранения данных и набор применимых к ним операций.

Первый символ имени переменной языка Perl определяет ее тип. В языке Perl можно определить переменные трех типов: скаляр, массив и хеш-массив. Для каждого типа переменных интерпретатор создает собственное *пространство имен*, храня в нем идентификаторы объявленных в программе переменных заданного типа. Это позволяет создавать переменные разных типов с одинаковыми идентификаторами. В программе Perl бесконфликтно могут сосуществовать и скалярная переменная `$var`, и массив скаляров `@var`, и хеш-массив `%var`.

Так как имя переменной всегда начинается с одного из символов "\$", "@" или "%", то использование в качестве идентификатора "предопределенных" слов не приводит к конфликтам в процессе интерпретации программы.

Следует отметить, что в языке Perl существует достаточно большой набор специальных встроенных переменных, которые можно использовать для получения или изменения разнообразной системной информации. Большинство из них являются скалярными переменными с "идентификатором", состоящим из одного специального символа, например, `$~`, `$_` и т. д.

(Подробно специальные переменные рассматриваются в главе 14.)

Немного забегаая вперед, скажем, что переменные используются в выражениях, которые, в свою очередь, могут являться операндами определенных в языке операций. Интерпретация операций и значений их операндов в Perl зависит от *контекста*, в котором они вычисляются. Существует два основных контекста: *скалярный* и *списковый*. Например, если левым операндом операции присваивания является скалярная переменная, то правый операнд вычисляется в скалярном контексте, т. е. его значением должен быть скаляр; если левый операнд массив или хеш (или фрагмент массива или хеша), то правый операнд вычисляется в списке контексте, т. е. должен предоставить значение, являющееся списком.

Замечание

Использование конструктора массива с элементами, являющимися скалярными переменными, в качестве левого операнда операции присваивания предписывает вычислять правый операнд в списке контексте.

Переменные массивов и хешей, а также их конструкторы, используемые в качестве операндов операции присваивания (=), будут иметь разные значения в зависимости от используемого контекста.

В списке контексте конструктор массива будет иметь значение, представляющее собой все значения списка в заданном порядке, тогда как в скалярном контексте он будет иметь значение, равное значению последнего элемента списка:

```
@array = (0, 2, 4); # Массив скаляров @array
```

```
# содержит три элемента: 0, 2, 4.
```

```
$last = (0, 2, 4); # Значение скалярной переменной $last равно 4.
```

Переменная массива скаляров в списковом контексте возвращает список всех элементов массива, а употребленная в скалярном контексте, в отличие от своего конструктора, будет иметь значение, равное числу элементов массива:

```
@new_array = @array; # Массив @new_array
                        # содержит все элементы массива @array.
$number = @array;     # Скалярная переменная $number
                        # равна 3 - числу элементов массива @array.
```

В качестве левого операнда операции присваивания можно использовать заключенный в круглые скобки список, элементами которого являются скалярные переменные, а правым операндом может быть конструктор массива или переменная массива. В этом случае каждой переменной левого списка присваивается значение соответствующего элемента правого списка. Если количество элементов правого списка меньше числа скалярных переменных левого списка, то переменные, которым не хватило значений, будут не определены. Несколько примеров приводится ниже:

```
($a, $b) = (1, 2, 3); # $a = 1, $b = 2.
($a, $b, $c) = (1, 2); # $a = 1, $b = 2, $c = "".
```

В языке Perl каждая операция имеет вычисляемое значение. Значением операции присваивания со скаляром в качестве правого операнда является значение этого скаляра в любом контексте. Если правым операндом является конструктор массива или массив, то в списковом контексте значением операции присваивания будет список элементов массива, а в скалярном контексте — число элементов массива.

```
$x = ( @a = (1, 2) ); # $x = 2.
```

Хеш-переменные так же, как и массивы скаляров, ведут себя по-разному в разных контекстах. Употребленные в списковом контексте, они вычисляются в виде обычного списка, состоящего из элементов всех пар ключ/значение. Однако порядок элементов в списке может не соответствовать порядку задания пар в хеше. Это связано с тем, что внутренняя реализация хеш-массивов основана на использовании хеш-таблиц для ускорения поиска соответствия ключ/значение, поэтому при вычислении хеша в виде списка порядок следования "пар" ключ/значение и не соответствует порядку их задания, но после ключа всегда следует его значение.

```
%hash = ( 1 => 1, 2 => 2, 3 => 3, 4 => 4 );
@list = %hash; # @list = (blue, 3, green, 2, red, 1, black, 4)
```

В скалярном контексте вычисляемым значением хеш-массива является строка, состоящая из числа использованных участков записей (bucket) и числа выделенных участков записей, разделенных символом "/". Если при-

своить скалярной переменной хеш-массив `%hash` предыдущего примера, то ее значением будет строка "4/8", которая говорит, что 4 из 8 выделенных участков записей хеш-таблицы используется.

Любая переменная в Perl может находиться в состоянии: определена или не определена. Если ей присвоено какое-либо значение, то такая переменная считается определенной. Если ей не было присвоено значение до ее использования в каком-либо выражении, то такая переменная считается не определенной. Чтобы узнать, определена или нет переменная, можно воспользоваться встроенной функцией `defined`, которая возвращает 1 (Истина) в случае определенности переменной и пустую строку "", когда переменная не определена:

```
@m = (1,2);  
defined @m; # Возвращает 1, массив скаляров @m определен.  
defined $m; # Возвращает "", переменная $m не определена.
```

Определенную переменную можно в любой момент сделать неопределенной, выполнив функцию `undef` с параметром, равным имени этой переменной:

```
@m = (1,2);  
defined @m; # Возвращает 1, массив скаляров @m определен.  
undef @m;  
defined @m; # Возвращает "", массив скаляров @m не определен.
```

Если переменная сделана не определенной с помощью функции `undef`, то она, естественно, теряет присвоенное ранее ей значение.

И последнее, о чем нам здесь хотелось бы упомянуть в связи с обсуждением переменных Perl, — это области видимости переменных, т. е. области доступности переменных. Во всех приведенных примерах все переменные являются глобальными, они доступны из любой точки программы. Язык Perl, однако, позволяет создавать переменные с областью видимости, ограниченной блоком или телом подпрограммы. Это так называемые локальные, или синтаксические переменные, имена которых могут совпадать с именами глобальных переменных, и которые существуют, только пока выполняются операторы некоторого блока или подпрограммы. После завершения выполнения операторов блока, эти переменные уничтожаются.

(Более подробно локальные переменные описаны в главе 11.)

Вопросы для самоконтроля

1. Перечислите три встроенных типа данных языка Perl.
2. В чем отличие числового литерала от строкового.
3. Объясните различие между строкой, ограниченной одинарными кавычками, и строкой, ограниченной двойными кавычками.

4. Каким образом можно выполнить системную команду из программы Perl?
5. Что такое массив скаляров и ассоциативный массив?
6. Как задаются в программе массивы и хеш-массивы?
7. Как объявляются в программе переменные для хранения скалярных данных, массивов скаляров и хеш-массивов?
8. Что такое интерполяция переменной?
9. Можно ли интерполировать массивы скаляров и хеш-массивы?
10. Какие два контекста для операции присваивания вы знаете, и как ведут себя массивы скаляров и хеш-массивы в них?

Упражнения

1. Найдите ошибки в следующем фрагменте кода Perl:

```
$m = 'Исходные данные:\n';  
@data = ( 1, 2, 3, 4);  
print $m, 'Запись: @data';
```
2. Что напечатают следующие операторы и почему:

```
$m = "Скаляр \"$m\n";  
@m = ( 1, 2, 3);  
print "Значение равно $m[0]\n";  
print "Значение равно $m [0]";
```
3. Предположим, что есть группа слушателей курса по языку Perl, состоящая из 10 человек. В середине курса слушатели сдают промежуточный экзамен, а в конце — выпускную работу. За экзамен и за выпускную работу выставляется оценка по пятибалльной системе. По окончании курса каждый слушатель получает удостоверение, в котором указано, естественно, его имя, а также оценки за экзамен и выпускную работу. Разработайте базу данных слушателей курса, которую можно использовать для автоматизации подготовки удостоверений об успешном окончании курса. (Указание: воспользуйтесь хеш-массивами.)
4. Дополните программу примера 3.8 удалением первого и последнего элемента связанного списка. (Указание: воспользуйтесь функцией `delete()`.)
5. После выполнения упражнения 4 в связанном списке останется один элемент. Удалите его, распечатайте, а затем снова добавьте два элемента в список и распечатайте.



Операции и выражения

Язык программирования, предоставляя возможность определения разнообразных типов данных, должен обеспечивать их обработку, т. к. его основной целью является реализация алгоритмов обработки данных. Выполнение допустимых действий над данными осуществляется с помощью набора определенных в языке программирования операций. *Операция* — это выполнение определенного действия над операндами, результатом которого является новое значение. С точки зрения математики операцию можно рассматривать как некую функцию, которая по заданным переменным (операндам) вычисляет новое значение. Все знают со школьной скамьи четыре основных арифметических действия, выполняемых над числами: сложение (+), вычитание (−), умножение (*) и деление (/). Эти действия (операции) мы всегда выполняем над двумя числами (операндами), получая в результате новое число. Язык программирования определяет не только арифметические операции над числовыми данными, но и операции, применимые к другим допустимым типам данных. Это могут быть операции над строками, массивами и т. д. Важно только одно, что есть операция, определяемая своим знаком, и есть участвующие в ней операнды, в совокупности позволяющие получить (вычислить) новое значение, которое может принадлежать к одному из допустимых типов. В качестве операндов можно использовать литералы, переменные и выражения, представляющие собой комбинации основных операций. Общий синтаксис операции можно представить следующим образом:

операнд *знак_операции* операнд

Для некоторых операций один из операндов может быть опущен. В этом случае операция называется одноместной или унарной, например, вычисление противоположного по знаку значения операнда $-\$m$. Если в операции участвуют два операнда, то операция называется двуместной или бинарной. Практически все допустимые операции являются унарными или бинарными, но в некоторых современных языках программирования определена одна-единственная условная тернарная операция, требующая три операнда. Значением этой операции является второй или третий операнд, в зависимости от истинности и ложности первого:

операнд_1 ? операнд_2 : операнд_3

Синтаксически *выражение* представляется в виде последовательности операндов и знаков операций, которая обычно интерпретируется слева направо

с учетом порядка старшинства операций, т. е. правил, определяющих, в какой последовательности выполняются присутствующие в выражении операции. Для изменения порядка выполнения используются круглые скобки. Результатом обработки интерпретатором выражения является некоторое вычисленное значение, которое используется в других конструкциях языка, реализующих алгоритм обработки данных.

В этой главе подробно рассматриваются операции языка Perl и их использование при конструировании выражений. В языке Perl определено несколько десятков операций. Их можно сгруппировать по типу выполняемых действий (арифметические, побитовые, логические, отношения, присваивания), на какие данные они воздействуют (строковые, списковые, файловые), и не относящиеся ни к одному из перечисленных типов (операции запятая, ссылки, разыменования и выбора). Следуя именно такой классификации операций языка Perl, мы и познакомим с ними нашего читателя. Одни операции будут разобраны подробно, о других мы только дадим общее представление, отнеся более детальное описание в другие главы нашей книги.

4.1. Арифметические операции

Все *арифметические операции* можно разбить на три группы: бинарные, унарные и увеличения/уменьшения. Их основное назначение — выполнить определенные вычисления над числовыми данными, но во всех арифметических операциях в качестве операндов могут выступать и строковые данные, причем не обязательно, чтобы они конвертировались в числовые данные.

4.1.1. Бинарные арифметические операции

Бинарные арифметические операции — это известные всем четыре арифметических действия: сложение (+), вычитание (-), умножение (*) и деление (/), к которым добавляются еще два: остаток от деления двух целых чисел (%) и возведение в степень (**). Примененные к числовым данным или строковым, которые содержат правильные литералы десятичных чисел, они выполняют соответствующие арифметические действия (пример 4.1).

Пример 4.1. Арифметические операции над числовыми данными

```
3.14 + 123;      # Результат: 126.14
"3.14" + "123"; # Результат: 126.14
"3.14" + 123;   # Результат: 126.14
"3.14" * 10;    # Результат: 31.4
300 - 200;      # Результат: 100
300 / 200;      # Результат: 1.5
```

```
3 % 2;           # Результат: 1
2 ** 3;          # Результат: 8
(-2) ** 3;       # Результат: -8
2 ** (-3);       # Результат: 0.125
2.5 ** 1.5;      # Результат: -23.95284707521047
```

Как видим, бинарные арифметические операции "работают" именно так, как мы привыкли их использовать в обычных арифметических вычислениях в нашей повседневной жизни.

Замечание

Если операнд в операции получения остатка от деления целых чисел (%) является вещественным числом с дробной частью, то он преобразуется к целому простым отбрасыванием дробной части, после чего операция выполняется над целыми числами.

Замечание

Нельзя возводить отрицательное число не в целую степень. Если такое случается, то интерпретатор не выдает никакой ошибки, но результатом такой операции является ноль: $(-2.5) ** (1.3) = 0$.

В качестве операндов бинарных арифметических операций можно использовать строки, не содержащие правильные числовые литералы. В этом случае интерпретатор попытается выделить, начиная с первого символа, из содержимого строки число и использовать его в качестве соответствующего операнда заданной операции. Если не удастся выделить правильный числовой литерал, то операнд принимает значение, равное 0. Подобные ситуации демонстрируются в примере 4.2.

Пример 4.2. Строки в арифметических операциях

```
"3f14" + "12-30";   # Результат: 15 ("3" + "12")
"a120" + "12-30";   # Результат: 12 ("0" + "12")
"a120" + "-0012-30"; # Результат: -12 ("0" + "-12")
```

Замечание

Если установить режим отображения предупреждающих сообщений интерпретатора (ключ -w), то при попытке использовать в бинарных арифметических операциях строки, не содержащей правильные числовые литералы, будет отображено сообщение вида:

```
Argument "a120" isn't numeric in add at D:\EXAMPLE1.PL line 2.
```

Бинарные арифметические операции выполняются в скалярном контексте. Это означает, что операндами должны быть скалярные переменные, а переменные массивов скаляров и хеш-массивов принимают значения, равные, соответственно, количеству элементов массивов скаляров или количеству использованных в хеш-таблице записей в соответствии с требованиями скалярного контекста (пример 4.3).

Пример 4.3. Массивы в арифметических операциях

```
@m = (2, 4, 6, 8, 10);  
%m1 = ( 1 => "a", 2 => "b");  
$n = 100;  
$n + @m; # Результат: 105 (100 + 5)  
@m + %m1; # Результат: 7 (5 + 2)
```

Замечание

В скалярном контексте хеш-массив принимает строковое значение, состоящее из числа использованных участков записей в хеш-таблице и числа выделенных участков записей, разделенных символом "/" (см. главу 3). Используемое в арифметических операциях число получается выделением из этой строки числового литерала, который как раз и соответствует количеству использованных в хеш-таблице записей.

4.1.2. Унарные арифметические операции

В языке Perl есть только две унарные арифметические операции (+) и (-). Унарный плюс +, примененный к данным любого типа, представленным литералами или своими переменными, не имеет никакого семантического эффекта. Он полезен перед выражением в круглых скобках, стоящим непосредственно после имени функции, если необходимо чисто визуально акцентировать тот факт, что функция фактически является списковой операцией.

(Об использовании унарного плюса в вызовах функций см. раздел 4.14.2 и главу 11.)

Унарный минус (-) выполняет арифметическое отрицание числового операнда. Это означает, что если число было отрицательным, то оно станет положительным, и наоборот. Если операндом является идентификатор, то результатом выполнения этой операции будет строка, состоящая из символа "-", за которым следует идентификатор. Если операндом является строка, начинающаяся с символа минус или плюс, то результатом также будет строка, в которой минус заменен на плюс и наоборот. Для строк, не начинающихся с плюса или минуса, операция унарного минуса добавляет его первым символом.

лом в строку. Все перечисленные случаи употребления унарного минуса показаны в примере 4.4.

Пример 4.4. Унарный минус

```
- '12.09'; # Результат: -12.09
- (-12.09); # Результат: 12.09
-id;      # Результат: '-id'
-'+id';   # Результат: '-id'
-"-id";   # Результат: "+id"
-'a120';  # Результат: '-a120'
```

4.1.3. Операции увеличения и уменьшения

Операции увеличения (++) и уменьшения (--) аналогичны таким же операциям в языке C. (Авторы языка Perl не скрывают, что они многое заимствовали из этого языка.) Результат этих операций зависит от того, стоят ли они перед (префиксная форма) или после переменной (постфиксная форма). При использовании префиксной формы они, соответственно, увеличивают или уменьшают числовое значение переменной на единицу до возвращения значения. Постфиксная форма этих операций изменяет числовое значение переменной после возвращения ими значения. Действие этих операций на числовые переменные иллюстрируется примером 4.5 (операторы фрагмента программы выполняются последовательно).

Пример 4.5. Числовое увеличение и уменьшение

```
$n = 10.7;      # Начальное значение
$inf1 = --$n;   # Результат: $inf1 = 9.7 и $n = 9.7
$inf2 = ++$n;   # Результат: $inf2 = 10.7 и $n = 10.7
$post1 = $n--;  # Результат: $post1 = 10.7 но $n = 9.7
$post2 = $n++;  # Результат: $post2 = 9.7 но $n = 10.7
```

Операция увеличения (префиксная и постфиксная), примененная к переменной, содержащей строку определенного вида, выполняется несколько необычно. Если строка состоит только из латинских букв, то возвращаемым значением операции увеличения будет строка, в которой последняя буква заменена на следующую по порядку букву алфавита, причем строчная заменяется строчной, а прописная прописной. Если строка завершается идущими подряд буквами "z" или "Z", то все они заменяются соответственно на "a" или "A", а стоящая перед ними в строке буква заменяется на следующую букву алфавита. Если вся строка состоит из букв "z" и "Z", то кроме замены

этих букв в соответствии с предыдущим правилом, перед ними добавляется строчная или прописная буква "a" в зависимости от того, строчная или прописная буква "z" стояла первой в строке.

Аналогичные действия осуществляются, если строка завершается последовательностью цифр: последняя цифра увеличивается на единицу. Если строка завершается идущими подряд цифрами 9, то все они заменяются на 0, а примыкающий к ним символ "увеличивается" на единицу: для буквы он переходит в следующий по алфавиту, а для цифры в следующую по порядку цифру. Если последовательность целиком состоит из девяток, то все они заменяются на нули, перед которыми добавляется единица. Префиксная и постфиксная формы операции действуют как обычно. Несколько иллюстраций этих операций представлены в примере 4.6.

Пример 4.6. Действие операции увеличения на строки специального вида

```
$s = "abc"
$s1 = ++$s;      # Результат: $s1 = "abd"
$s = "abC";
$s1 = ++$s;      # Результат: $s1 = "abD"
$s = "abz";
$s1 = ++$s;      # Результат: $s1 = "aca"
$s = "abzZz";
$s1 = ++$s;      # Результат: $s1 = "acaAa"
$s = "ab09";
$s1 = ++$s;      # Результат: $s1 = "ab10"
$s = "99";
$s1 = ++$s;      # Результат: $s1 = "100"
```

Замечание

Операция уменьшения (`--`) работает со специальными строками так же, как и с обычными. Осуществляется попытка выделить числовой литерал, начиная с первого символа. Если такое оказывается возможным, то числовое значение строки приравнивается выделенному числовому литералу, если нет — ее значение считается равным 0. После этого применяется операция уменьшения к вычисленному числовому значению строки.

4.2. Операции конкатенации и повторения

Бинарная операция *конкатенации*, или *соединения* объединяет два строковых операнда в одну строку. Знаком этой операции служит точка ".":

```
"one_string"."two_string";      # Результат: "one_stringtwo_string"
```


В новой строке содержимое первого операнда и содержимое второго операнда соединяются без пробела между ними. Обычно эта операция используется для присваивания переменной некоторого нового значения. Если необходимо соединить две или более строки со вставкой пробелов между ними, то следует воспользоваться операцией `join` (см. гл. 10 "Работа со строками"). Можно, однако, для соединения строк со вставкой пробела (или любого другого символа между ними) воспользоваться свойством подстановки значения скалярной переменной в строку, ограниченную двойными кавычками:

```
$s1 = "one_string";
$s2 = "two_string";
$s = "$s1 $s2";      # Значение $s: "one_string two_string"
```

Можно использовать операцию конкатенации строк последовательно в одном выражении для соединения нескольких строк:

```
$s1 = "one";
$s2 = "two";
$s3 = "three";
$s = $s1.$s2.$s3;    # Значение $s: "onetwothree"
```

Операцию конкатенации можно применять и к числовым литералам, и к числовым данным, хранящимся в скалярных переменных. Результатом будет строка, содержащая символьные представления двух чисел:

```
$n1 = 23.5;
$n2 = 3e01;
$n = $n1.$n2;        # Значение $n: "23.530"
$n = 23.5.3e01;      # Значение $n: "23.530"
```

Заметим, что последний оператор выглядит несколько экзотично и его семантика не определяется с первого взгляда.

Для работы со строками в языке Perl предусмотрена еще одна операция — *повторение строки* `x` (просто символ строчной буквы "x"). Эта бинарная операция создает новую строку, в которой строка, заданная левым операндом, повторяется определяемое правым операндом количество раз:

```
"aA" x 2;           # Результат: "aAaA"
10.0 x "3";         # Результат: "101010"
101e-1 x 3;         # Результат: "101010"
$n = 010;
$n x 2;             # Результат: "88"
10.1 x 3.9;         # Результат: "10.110.110.1"
"101e-1" x 2;       # Результат: "101e-1101e-1"
```

Обратим внимание, что в качестве левого операнда можно использовать и числовые литералы, и переменные, содержащие числовые данные. Правым операндом, задающим число повторений, может быть любое число или строка, содержащая правильное десятичное число.

Эта операция удобна, если надо напечатать или отобразить на экране монитора повторяющийся символ или последовательность символов. Например, следующий оператор выведет на экран монитора строку, целиком состоящую из символов подчеркивания:

```
print "_" x 80;
```

Левым операндом этой операции может быть список, заключенный в круглые скобки. В этом случае операция повторения `x` работает как повторитель списка, т. е. ее результатом будет список, в котором список левого операнда повторяется заданное правым операндом количество раз:

```
(1) x 3;           # Результат: (1, 1, 1)
(1, 2) x 2;        # Результат: (1, 2, 1, 2)
```

Это пример использования операции Perl в разных контекстах: скалярном и списковом (о контекстах мы поговорим ниже в этой же главе). Операция повторения в списковом контексте удобна для задания массива скаляров с одинаковыми значениями элементов или групп элементов:

```
@array = ("a", "b") x 2; # Результат: @array = ("a", "b", "a", "b")
@array = ("a") x 3;      # Результат: @array = ("a", "a", "a")
```

Аналогично, эту операцию можно использовать для инициализации хеш-массива одинаковыми значениями:

```
@keys = ( one, two, three); # Определение ключей хеш-массива.
@hash{@keys} = ("a") x @keys; # Инициализация значений хеш-массива.
```

В последнем операторе присваивания в правой части массив скаляров `@keys` используется в списковом контексте и представляет список своих значений, тогда как в левой части он используется в скалярном контексте и имеет значение, равное числу своих элементов.

Внимание

Знак операции повторения `x` следует отделять пробелами от операндов, так как иначе он может быть воспринят интерпретатором, как относящийся к лексеме, а не представляющий операцию повторения. Например, при синтаксическом разборе строки

```
$nx$m;
```

интерпретатор определит, что в ней идут подряд две переменные `$nx` и `$m`, а не операция повторения содержимого переменной `$n`, что приведет к синтаксической ошибке.

4.3. Операции отношения

Для сравнения скалярных данных или значений скалярных переменных язык Perl предлагает набор бинарных операций, вычисляющих отношения равенства, больше, больше или равно и т. п. между своими операндами, поэтому эту группу операций еще называют *операциями отношения*. Для сравнения числовых данных и строковых данных Perl использует разные операции. Все они представлены в табл. 4.1.

Таблица 4.1. Операции отношения

Операция	Числовая	Строковая	Значение
Равенство	==	eq	Истина, если операнды равны, иначе ложь
Неравенство	!=	ne	Истина, если операнды не равны, иначе ложь
Меньше	<	lt	Истина, если левый операнд меньше правого, иначе ложь
Больше	>	gt	Истина, если левый операнд больше правого, иначе ложь
Меньше или равно	<=	le	Истина, если левый операнд больше правого или равен ему, иначе ложь
Больше или равно	>=	ge	Истина, если правый операнд больше левого или равен ему, иначе ложь
Сравнение	<=>	cmp	0, если операнды равны 1, если левый операнд больше правого -1, если правый операнд больше левого

Результатом операций отношения (кроме последней сравнения) является Истина, значение 1, или Ложь, пустая строка "".

Замечание

Значение истина в арифметических операциях интерпретируется как число 1, а в строковых как строка "1". Значение ложь в арифметических операциях интерпретируется как число 0, а в строковых как пустая строка "".

4.3.1. Числовые операции отношения

Числовые операции отношения применяются к числовым данным, причем один или оба операнда могут задаваться строкой, содержащей правильное де-

сятичное число. Если в числовых операциях отношения какой-либо из операндов задан строкой, содержимое которой не представляет правильное десятичное число, то его значение принимается равным 0 и отображается предупреждение о некорректном использовании операнда в числовой операции отношения (если включен режим отображения предупреждений интерпретатора Perl). Смысл операций отношения для числовых данных соответствует обычным математическим операциям сравнения чисел (пример 4.7).

Пример 4.7. Числовые операции отношения

```
123 > 89; # Результат: 1 (истина)
123 < 89; # Результат: "" (ложь)
123 == 89; # Результат: "" (ложь)
123 != 89; # Результат: 1 (истина)
89 <= 89; # Результат: 1 (истина)
23 >= 89; # Результат: "" (ложь)
23 <=> 89; # Результат: -1 (правый операнд больше левого)
89 <=> 23; # Результат: 1 (правый операнд больше левого)
```

Применение числовых операций сравнения не представляет сложности, однако при сравнении на равенство десятичных чисел с плавающей точкой могут проявиться эффекты округления, связанные с ограниченным количеством значащих цифр в мантиссе представления действительных чисел в компьютере и приводящие к "неправильной", с точки зрения пользователя, работе операций сравнения. Пример 4.8 иллюстрирует подобную ситуацию.

Пример 4.8. Ошибки округления

```
#!/perl -w
$z = 0.7;
$zz = 10+0.7-10;          # Переменная $zz содержит число 0.7
# Печать строки "z равно zz", если равны значения переменных $z и $zz
print "z равно zz\n" if ($z == $zz);
```

При попытке выполнить пример 4.8 мы с удивлением обнаружим, что наша программа ничего не напечатает. В чем же дело? Разгадка лежит в операторе вычисления значения переменной \$zz. При выполнении арифметических операций в результате ошибок округления получается значение 0.6999999999999999 (можете вставить оператор печати переменной \$zz и убедиться в этом), хотя и близкое к 0.7, но не равное ему в точности. Следовательно, операция сравнения отработала верно!

Совет

Не используйте операцию сравнения на равенство вещественных чисел, ее результат может не соответствовать ожидаемому с точки зрения математики. Если необходимо проверить равенство двух вещественных чисел, то лучше сравнивать абсолютное значение их разности с некоторым очень маленьким числом (в зависимости от требуемой точности):

```
abs($a-$b) <= 0.00000001; # Проверка равенства
```

4.3.2. Строковые операции отношения

Сравнение строковых данных базируется на их упорядочении в соответствии с таблицей кодов ASCII, т. е. символ с меньшим кодом ASCII предшествует символу с большим кодом. Сравнение строк осуществляется посимвольно слева направо. Это означает, что если равны первые символы строк, то сравниваются вторые и если они равны, то сравниваются третьи и т. д. Причем, если строки разной длины, то в конец строки меньшей длины добавляется недостающее для равенства количество символов с кодом 0. Следует отметить, что в отличие от некоторых других языков программирования в Perl замыкающие строку пробельные символы являются значимыми при сравнении строк. В примере 4.9 показаны сравнения строк, иллюстрирующие изложенные правила.

Пример 4.9. Сравнение строк

```
"A" lt "a"; # Результат: истина (код "A" - \101, код "a" - \141)
"a" lt "aa"; # Результат: истина (к строке "a" добавляется символ
# с кодом \000, который меньше кода \141
# второго символа "a" строки правого операнда)
"a" lt "a "; # Результат: истина (к строке "a" добавляется символ
# с кодом \000, который меньше кода \040
# замыкающего пробела строки правого операнда)
"12" lt "9"; # Результат: истина (код "1" - \061, код "9" - \071)
" 9" eq "09"; # Результат: ложь (код " " - \040, код "0" - \060)
```

Обратим внимание на две последние операции сравнения строковых литералов. Содержимое их операндов может быть преобразовано в правильные числа, и поэтому к ним применимы аналогичные числовые операции отношения. Однако их результат будет существенно отличаться от результата выполнения строковых операций отношения. При использовании операции < в предпоследнем выражении результат будет Ложь, а если в последнем выражении применить операцию ==, то результат будет Истина. Об этом всегда следует помнить, так как Perl автоматически преобразует символьные данные в числовые там, где это необходимо.

4.4. Логические операции

Рассмотренные в предыдущем параграфе операции сравнения используются в условном операторе `if` (о нем и других операторах Perl в следующей главе) для организации ветвления в программе. Однако, иногда желательно проверять одновременно результаты нескольких операций сравнения и предпринимать соответствующие алгоритму действия. Можно подобную ситуацию запрограммировать с помощью вложенных операторов `if`, а можно в одном операторе использовать сложное выражение, результатом вычисления которого будет, например, истинность двух или более каких-либо операций сравнения. Для формирования подобных проверок и служат *логические операции* языка Perl.

В языке определены бинарные операции логического сравнения `||` (ИЛИ), `&&` (И) и унарная операция логического отрицания `!`. Их действие аналогично действию соответствующих математических операций исчисления предикатов. Результатом операции `||` (логическое ИЛИ) является Истина, если истинен хотя бы один из операндов, в остальных случаях она возвращает Ложь (остальные случаи представляют единственный вариант, когда оба операнда ложны). Операция логического И `&&` возвращает в качестве результата Истину, только если оба операнда истинны, в противном случае ее результат Ложь. Операция логического отрицания `!` работает как переключатель: если ее операнд истинен, то она возвращает Ложь, если операнд имеет значение Ложь, то ее результатом будет Истина.

Замечание

В языке Perl нет специальных литералов для булевых значений Истина и Ложь. В качестве значения Истина принимается любое скалярное значение, не равное нулевой строке `""` или числу `0` (а также его строковому эквиваленту `"0"`). Естественно, нулевая `""` строка и `0` (вместе с его строковым эквивалентом `"0"`) представляют значение Ложь.

Начиная с Perl 5.001, в язык были введены логические операции `or`, `and`, `not` и `xor`. Первые три полностью аналогичны логическим операциям `||`, `&&` и `!`, тогда как операция `xor` реализует исключающее ИЛИ:

Истина	<code>xor</code>	Истина	=	Ложь
Истина	<code>xor</code>	Ложь	=	Истина
Ложь	<code>xor</code>	Истина	=	Истина
Ложь	<code>xor</code>	Ложь	=	Ложь

Единственное отличие этих логических операций от рассмотренных ранее заключается в том, что они имеют наименьший приоритет при вычислении сложных выражений.

(Старшинство, или приоритет операций при вычислении сложных выражений, рассматривается в разделе 4.14.2 этой главы.)

В Perl вычисление логических операций **ИЛИ** и **И** осуществляется по "укороченной схеме". Это непосредственно связано со смыслом этих операций. Если при вычислении операции **ИЛИ** определено, что значение ее первого операнда Истина, то при любом значении второго операнда результатом всей операции будет Истина, поэтому нет смысла вообще вычислять второй операнд. Аналогично для операции логического **И**: если значение первого операнда Ложь, то результат всей операции Ложь вне зависимости от значения второго операнда. В отличие от операций отношения, результатом которых может быть 0 (или пустая строка "") или 1, соответствующие булевым значениям Ложь и Истина, результатом логических операций является значение последнего вычисленного операнда. Пример 4.10 иллюстрирует вычисление логических операций.

Пример 4.10. Логические операции

```
$op1 = 0;
$op2 = "s";
$op3 = "";
$op4 = 25;
$op5 = "0";

$op4 || $op2; # Результат: истина. Значение: 25.
$op2 || $op4; # Результат: истина. Значение: "s".
$op1 && $op2; # Результат: ложь. Значение: 0.
$op2 && $op4; # Результат: истина. Значение: 25.
!$op2;       # Результат: ложь. Значение: "".
not $op3;    # Результат: истина. Значение: 25.
$op4 and $op5; # Результат: ложь. Значение: "".
```

Свойство логических операций языка Perl вычисляться по "укороченной схеме" можно использовать для управления некоторыми исключительными ситуациями, возникающими в программе в процессе вычислений. Например, можно достаточно элегантно избежать деления на ноль с помощью операции логического **ИЛИ**:

```
($x == 0) || ($m = 1/$x);
```

При вычислении результата этой операции сначала вычисляется левый операнд, который сравнивает значение переменной `$x` с нулем. Если это значение действительно равно нулю, то результатом операции сравнения будет Истина, а поэтому второй операнд операции логического **ИЛИ** не вычисляется, так его значение не влияет на результат выполнения логической опе-

рации, и не возникает ситуации деления на нуль. Если значение переменной $\$x$ не равно нулю, то результатом вычисления первого операнда операции `||` будет `Ложь`, и обязательно будет вычисляться ее второй операнд, в котором осуществляется деление на не равную нулю переменную $\$x$.

4.5. Побитовые операции

Данные в компьютере представляются в виде последовательности битов. В языке Perl определены бинарные операции *побитового* логического сравнения целых чисел и строк: `&` (И), `|` (ИЛИ) и `^` (исключающее ИЛИ), а также унарная операция логического отрицания `~`. Результат их вычисления зависит от того, к данным какого типа они применяются: числовым или строковым. Эти операторы различают числовые данные и строки, содержимое которых может быть преобразовано в число.

Кроме логических операций побитового сравнения, две операции сдвигают влево (`<<`) и вправо (`>>`) биты в представлении целых чисел. Эти операторы не работают со строками.

4.5.1. Числовые операнды

Если хотя бы один операнд в бинарных побитовых операциях является *числом*, то содержимое второго операнда также должно быть *числом*. Операнд, являющийся строкой символов, преобразуется в числовое значение. В случае несоответствия содержимого строки десятичному числу ее значение принимается равным 0 и отображается предупреждающее сообщение, если установлен соответствующий режим работы интерполятора. Все числовые операнды преобразуются к целым числам простым отбрасыванием дробной части, никакого округления не происходит.

Чтобы понять сущность побитовых операций над числовыми данными, необходимо представлять, как хранятся в программе целые числа. При задании чисел мы можем использовать одно из трех представлений: десятичное, восьмеричное или шестнадцатеричное. Однако в компьютере числа не хранятся ни в одном из указанных представлений. Они переводятся в двоичные числа — числа с основанием 2, цифры которых и называются битами. Двоичные числа представляют собой запись чисел в позиционной системе счисления, в которой в качестве основания используется число 2. Таким образом, двоичные цифры, или биты, могут принимать значения только 0 или 1. Например, десятичное число 10, переведенное в двоичное, представляется в виде 1010. Для обратного перевода этого числа в десятичную форму представления следует, в соответствии с правилами позиционной системы счисления, произвести следующие действия:

$$1 * (2^{**3}) + 0 * (2^{**2}) + 1 * (2^{**1}) + 0 * (2^{**0})$$

45.93 & 100

[illegible][illegible]

0000000000000000000000000000000100100

Замечание

45.93 | 100

[illegible]

И

[illegible]

дает следующую цепочку битов

0000000000000000000000000000000001101101

(десятичное 109:

$$2^{*6} + 2^{*5} + 2^{*3} + 2^{*2} + 2^{*1})$$

Побитовое исключающее ИЛИ \wedge при сравнении битов дает значение 1 тогда, когда *точно один* из операндов имеет значение равное 1. Следовательно, $1 \wedge 1 = 0$ и $0 \wedge 0 = 0$, в остальных случаях результат сравнения битов равен 0. Поэтому для тех же чисел результатом операции $45.93 \wedge 100$ будет десятичное число 73.

Операция логического отрицания \sim является унарной и ее действие заключается в том, что при последовательном просмотре битов числа все значения 0 заменяются на 1, и наоборот. Результат этой операции существенно зависит от используемого количества битов для представления целых чисел. Например, на 32-разрядной машине результатом операции

 ≈ 1

будет последовательность битов

11111111111111111111111111110

представляющая десятичное число $4294967294 = 2^{31} + 2^{30} + \dots + 2^1$, тогда как на 16-разрядной машине эта же операция даст число $6534 = 2^{31} + 2^{30} + \dots + 2^1$.

Бинарные операции побитового сдвига осуществляют сдвиг битов целого числа, заданного левым операндом, влево (<<) или вправо (>>) на количество бит, определяемых правым целочисленным операндом. При сдвиге вправо недостающие старшие биты, а при сдвиге влево младшие биты числа дополняются нулями. Биты, выходящие за разрядную сетку, пропадают. Несколько примеров операций сдвига представлено ниже:

Битовое представление числа 22: (000000000000000000000000000010110)

[illegible][illegible]

Все перечисленные операции работают и с отрицательными целыми числами, только при их использовании следует учитывать, что они хранятся в дополнительном коде. Двоичная запись неотрицательного целого числа называется прямым кодом. Обратным кодом называется запись, полученная по разрядной инверсии прямого кода. Отрицательные целые числа представляются в памяти компьютера в дополнительном коде, который получается прибавлением единицы к младшему разряду обратного кода. Например, представление числа -1 получается следующим образом:

[illegible]

111111111111111111111111111111110 # обратный код числа 1

[illegible]

и получаем представление числа -1

Именно с этим кодом числа -1 будут работать все побитовые операции, если оно будет задано в качестве операнда одной из них.

Внимание

В языке Perl, как отмечалось в гл. 3, в арифметических операциях используется представление всех чисел в виде чисел с плавающей точкой удвоенной точности. Там же говорилось, что целое число можно задавать с 15 значащими цифрами, т. е. максимальное положительное целое число может быть 999 999 999 999 999. Но это число не имеет ничего общего с представлением целых чисел в компьютере, для которых может отводиться 64, 32 или 16 битов, в зависимости от архитектуры компьютера. Во всех побитовых операциях можно предсказать результат только если операндами являются целые числа из диапазона $-2^{32}-1 \dots 2^{32}-1$, так как ясен алгоритм их представления. Вещественные числа, не попадающие в этот диапазон, преобразуются к целым, но алгоритм их преобразования не описан авторами языка.

4.5.2. Строковые операнды

Если *оба* операнда являются строковыми литералами или переменными, содержащими строковые данные, то операции побитового логического сравнения сравнивают соответствующие биты кода каждого символа строки. Для кодирования символов используется таблица ASCII-кодов. Если строки разной длины, то при сравнении полагается, что строка меньшей длины содержит необходимое число недостающих символов с восьмеричным кодом `\000`. Например, результатом сравнения двух строк `"++"` и `"3"` с помощью операции `|` побитового логического ИЛИ будет строка `";+"`. Операнды этой операции представляются следующими битовыми последовательностями (каждый символ представляется 8 битами):

```
00101011 00101011 # Восьмеричный код символа "+" равен 053.
```

```
00110011          # Восьмеричный код символа "3" равен 063.
```

Вторая строка дополняется восемью нулевыми битами и последовательно для каждой пары соответствующих бит двух строк одинаковой длины выполняется операция логического ИЛИ. Результатом выполнения этой процедуры является битовая последовательность

```
00111011 00101011
```

При ее интерпретации как строки символов получается последовательность двух символов с восьмеричными кодами 073 и 053, которые соответствуют символам `";"` и `"+"`.

Следует отметить, что в случае двух строковых операндов, содержимое которых можно преобразовать в числовые данные, подобное преобразование не происходит, и побитовые операции логического сравнения выполняются как с обычными строковыми данными:

```
45.93 | 100      # Результат: число 109.
"45.93" | 100    # Результат: число 109.
45.93 | "100"    # Результат: число 109.
"45.93" | "100"  # Результат: строка "55>93".
```

В первых трех операциях этого примера строки преобразуются в числа, а потом выполняется соответствующая операция побитового логического ИЛИ двух чисел.

Выполнение операции побитового логического отрицания `~` для строки ничем не отличается от соответствующей операции отрицания для чисел с той лишь разницей, что применяется она к битовому представлению символов строки:

```
~"1" # Результат: "+".
~"ab" # Результат: "ЮЭ".
```

4.6. Операции присваивания

Присваивание переменной какого-либо значения, определенного литералом, или присваивание одной переменной значения другой переменной является наиболее часто выполняемым действием в программе, написанной на любом языке программирования. В одних языках это действие определяется с помощью оператора, а в других — с помощью операции. Отличие заключается в том, что в языках, где присваивание является операцией, оно может использоваться в выражениях как его составная часть, так как любая операция вычисляет определенное значение, тогда как оператор всего лишь производит действие. В языке Perl присваивание является операцией, которая возвращает правильное *lvalue*. Что это такое, мы разъясним буквально в следующих абзацах.

Операция присваивания `=`, с которой читатель уже немного знаком, является бинарной операцией, правый операнд которой может быть любым правильным выражением, тогда как левый операнд должен определять область памяти, куда операция присваивания помещает вычисленное значение правого операнда. В этом случае и говорят, что левый операнд должен быть правильным *lvalue* (от английского *left value* — левое значение). А что мы можем использовать в программе для обозначения области памяти? Правильно, переменные. Следовательно, в качестве левого операнда операции присваивания можно использовать переменную любого типа или элемент любого массива. (В языке Perl существуют и другие объекты, которые можно использовать в качестве левого операнда операции присваивания, но об этом в свое время.) Следующая операция простого присваивания

```
$a = $b+3;
```

вычислит значение правого операнда и присвоит его переменной `$a`, т. е. сохранит в области памяти, выделенной для переменной `$a`. Возвращаемым значением этой операции будет адрес области памяти переменной `$a` (правильное *lvalue*), или говоря проще, имя скалярной переменной `$a`, которое снова можно использовать в качестве левого операнда операции присваивания. Таким образом, в языке Perl следующая операция присваивания

```
($a = $b) = 3;
```

является синтаксически правильной и в результате ее вычисления переменной `$a` будет присвоено значение 3, так результатом вычисления операции присваивания `$a = $b` будет присвоение переменной `$a` значения переменной `$b`, а возвращаемым значением можно считать *переменную* `$a`, которой в следующей операции присваивается значение 3. Читатель спросит: "А зачем городить такие сложности, если тот же самый результат можно получить простой операцией присваивания `$a = 3`?" Действительно, замечание справедливое. Но на этом примере мы показали, как можно использовать операцию присваивания в качестве правильного *lvalue*. Более интересные примеры мы покажем, когда определим составные операции присваивания, заимствованные из языка C.

Синтаксические правила языка Perl позволяют осуществлять присваивание одного и того же значения нескольким переменным в одном выражении:

```
$var1 = $var2 = $var1[0] = 34;
```

Очень часто при реализации вычислительных алгоритмов приходится осуществлять разнообразные вычисления с использованием значения некоторой переменной и результат присваивать этой же переменной. Например, увеличить на 3 значение переменной `$a` и результат присвоить этой же переменной `$a`. Это действие можно реализовать следующей операцией присваивания:

```
$a = $a+3;
```

Однако, язык Perl предлагает более эффективный способ решения подобных проблем, предоставляя в распоряжение программиста бинарную операцию *составного присваивания* `+=`, которая прибавляет к значению левого операнда, представляющего правильное *lvalue*, значение правого операнда и результат присваивает переменной, представленной левым операндом. Таким образом, оператор составного присваивания

```
$a += 3; # Результат: $a = $a+3
```

эквивалентен предыдущему оператору простого присваивания. Единственное отличие заключается в том, что его реализация эффективнее реализации простого присваивания, так как в составном операторе присваивания переменная `$a` вычисляется один раз, тогда как в простом ее приходится вычис-

лять дважды. (Под вычислением переменной понимается вычисление адреса представляемой ею области памяти и извлечение значения, хранящегося в этой области памяти.)

Для всех бинарных операций языка Perl существуют соответствующие составные операции присваивания. Все они, вместе с примерами их использования, собраны в табл. 4.2.

Таблица 4.2. Составные операции присваивания

Операция	Пример	Эквивалент с операцией простого присваивания
<code>**=</code>	<code>\$a **= 3;</code>	<code>\$a = \$a ** 3;</code>
<code>+=</code>	<code>\$a += 3;</code>	<code>\$a = \$a + 3;</code>
<code>-=</code>	<code>\$a -= 3;</code>	<code>\$a = \$a - 3;</code>
<code>.=</code>	<code>\$a .= "a";</code>	<code>\$a = \$a . "a";</code>
<code>*=</code>	<code>\$a *= 3;</code>	<code>\$a = \$a * 3;</code>
<code>/=</code>	<code>\$a /= 3;</code>	<code>\$a = \$a / 3;</code>
<code>%=</code>	<code>\$a %= 3;</code>	<code>\$a = \$a % 3;</code>
<code>x=</code>	<code>\$a x= 3;</code>	<code>\$a = \$a x 3;</code>
<code>&=</code>	<code>\$a &= \$b;</code>	<code>\$a = \$a & \$b;</code>
<code> =</code>	<code>\$a = 3;</code>	<code>\$a = \$a 3;</code>
<code>^=</code>	<code>\$a ^= 3;</code>	<code>\$a = \$a ^ 3;</code>
<code><<=</code>	<code>\$a <<= 3;</code>	<code>\$a = \$a << 3;</code>
<code>>>=</code>	<code>\$a >>= 3;</code>	<code>\$a = \$a >> 3;</code>
<code>&&=</code>	<code>\$a &&= \$b > 1;</code>	<code>\$a = \$a && \$b > 1;</code>
<code> =</code>	<code>\$a = \$b == 0;</code>	<code>\$a = \$a \$b == 0;</code>

Возвращаемым значением каждой из составных операций присваивания, как и в случае простого присваивания, является переменная левого операнда (правильное *lvalue*), поэтому их можно использовать в любом операнде других операций присваивания (пример 4.11).

Пример 4.11. Операции простого и составного присваивания

```
$b = 1;
$a = ($b += 3);          # Результат: $a = $b = 4
$a += ($b += 3);         # Результат: $a = $a+$b+3
(($a += 2) **= 2) -= 1;  # Результат: $a = ($a+2)**2-1
```

Замечание

При использовании операции присваивания (простой или составной) в качестве левого операнда другой операции присваивания обязательно ее заключение в круглые скобки. Иначе может сгенерироваться синтаксическая ошибка, или выражение будет интерпретировано не так, как задумывалось. При наличии нескольких операций присваивания в одном выражении без скобок интерпретатор perl начинает его разбор *справа*. Например, если последнее выражение примера 4.11 записать без скобок

```
$a += 2 **= 2 -= 1;
```

то при его синтаксическом анализе интерпретатор сначала выделит операцию присваивания

```
2 -= 1;
```

и сообщит об ошибке, так как ее синтаксис ошибочен (левый операнд не является переменной или элементом массива).

4.7. Ссылки и операция разыменования

При выполнении программы Perl она, вместе с используемыми ею данными, размещается в оперативной памяти компьютера. Обращение к данным осуществляется с помощью символических имен — переменных, что является одним из преимуществ использования языка высокого уровня типа Perl. Однако иногда необходимо получить непосредственно адрес памяти, где размещены данные, на которые мы ссылаемся в программе с помощью переменной. Для этого в языке определено понятие *ссылки*, или *указателя*, который содержит адрес переменной, т. е. адрес области памяти, на которую ссылается переменная. Для получения адреса переменной используется операция ссылки, знак которой "&" ставится перед именем переменной:

```
$m = 5;
```

```
$pm = &$m;           # Ссылка на скалярную величину
```

Ссылки хранятся в скалярных переменных и могут указывать на скалярную величину, на массив, на хеш и на функцию:

```
@array = (1,2,3);
```

```
$parray = \@array; # Ссылка на массив скаляров
```

```
%hesh = (one=>1, two=>2, three=>3);
```

```
$phesh = \%hesh; # Ссылка на массив скаляров
```

Если распечатать в программе переменные-ссылки `$pm`, `$parray` и `$phesh`, то мы увидим строки, подобные следующим:

```
SCALAR(0x655a74)
```

```
ARRAY(0x655b10)
```

```
HASH(0x653514)
```

В них идентификатор определяет тип данных, а в скобках указан шестнадцатеричный адрес области памяти, содержащей данные соответствующего типа.

Для получения содержимого области памяти, на которую ссылается переменная-указатель, требуется выполнить операцию *разыменования ссылки*. Для этого достаточно перед именем такой переменной поставить символ, соответствующий типу данных, на который ссылается переменная (\$, @, %):

```
@keys = keys(%$phash);           # Массив ключей хеша
@values = values(%$phash);        # Массив значений хеша
print "$$pm \n@$parray \n@keys \n@values";
```

Этот фрагмент кода для определенных в нем переменных-ссылок на скаляр, массив и хеш напечатает их значения:

```
5           # Значение скалярной переменной $$m
1 2 3       # Значения элементов массива скаляров @$array
three two one # Ключи хеша %hash
3 2 1       # Значения хеша %hash
```

Использование описанной выше простой операции разыменования может приводить к сложным, трудно читаемым синтаксическим конструкциям при попытке получить значения элементов сложных конструкций: массива массивов, массива хешей и т. п. Для подобных целей в языке Perl предусмотрена бинарная операция `->`, левым операндом которой может быть ссылка на массив скаляров или хеш-массив, а правым операндом индекс элемента массива или хеша, значение которого необходимо получить:

```
print "$parray->[0], $parray->[1], $parray->[2]\n";
print "$phash->{one}, $phash->{two}, $phash->{three}\n";
```

Эти операторы напечатают значения элементов массива `@array` и хеша `%hash`.

(Более подробно ссылки и операции разыменования рассматриваются в главе 9.)

4.8. Операции связывания

Операции сопоставления с образцом, используемые многими утилитами обработки текста в Unix, являются мощным средством и в языке Perl. Эти операции с регулярными выражениями включают поиск (`m//`), подстановку (`s//`) и замену символов (`tr//`) в строке. По умолчанию они работают со строкой, содержащейся в системной переменной `$_`. Операции `=~` и `!~` связывают выполнение сопоставления с образцом над строкой, содержащейся в переменной, представленной левым операндом этих операций:

```
$_ = "It's very interesting!";
s/very/not/;           # Переменная $_ будет содержать строку
                        # "It's not interesting!"
```



```
$m = "my string";  
$m =~ s/my/our/;      # Переменная $m будет содержать строку  
                        # "our string"
```

Возвращаемым значением операции `=~` является Истина, если при выполнении соответствующей ей операции сопоставления с образцом в строке была найдена последовательность символов, определяемая регулярным выражением, и Ложь в противном случае. Операция `!~` является логическим дополнением к операции `=~`. Следующие два выражения полностью эквивалентны:

```
$m !~ m/my/our/;  
not $m =~ m/my/our/;
```

(Более подробно регулярные выражения и операции связывания рассматриваются в главе 10.)

4.9. Именованные унарные операции

В языке Perl определено большое количество встроенных функций, выполняющих разнообразные действия. Некоторые из них, с точки зрения синтаксического анализатора языка, на самом деле являются унарными операциями, которые и называют именованными унарными операциями, чтобы отличить их от унарных операций со специальными символами в качестве знаков операций (например, унарный минус "-", операция ссылки "\", логического отрицания "!" и т. д.). Некоторые из именованных унарных операций перечислены ниже:

```
chdir, cos, defined, goto, log, rand, rmdir, sin, sqrt, do, eval, return
```

(Является ли функция унарной операцией, можно определить в приложении 1.)

К именованным унарным операциям относятся также все операции проверки файлов, синтаксис которых имеет вид:

`-символ [имя_файла|дескриптор_файла]`

Например, для проверки существования файла определена операция `-e`, выяснить возможность записи в файл можно операцией `-w`.

(Более подробно операции проверки файлов рассматриваются в главе 7.)

4.10. Операции ввода/вывода

Для взаимодействия и общения с внешним окружением в любом языке программирования предусмотрены операции ввода/вывода. Perl не является исключением. В нем определен ряд операций, обеспечивающих ввод и вывод данных в/из программы.

4.10.1. Операция *print*

С этой операцией вывода мы уже немного знакомы. Операция `print` — унарная операция, правым операндом которой служит задаваемый список значений, которые она отображает по умолчанию на экране монитора. Операцию, операндом которой является список, называют списковой операцией. Внешне ее можно задать как вызов функции, заключив ее операнд в круглые скобки. Следующие операции эквивалентны:

```
print "@m", "\n", $m, "\n";  
print("@m", "\n", $m, "\n");
```

(Более подробно эта операция рассматривается в главе 6.)

4.10.2. Выполнение системных команд

Операция заключения в обратные кавычки — это специальная операция, которая передает свое содержимое на выполнение операционной системы и возвращает результат в виде строковых данных:

```
$command = `dir`; # Переменная $command после выполнения операционной  
                  # системой команды 'dir' содержит результат ее  
                  # выполнения.
```

Содержимое строкового литерала в обратных кавычках должно быть, после подстановки значений переменных, которые могут в нем присутствовать, правильной командой операционной системы.

(Более подробно эта операция рассматривается в главе 6.)

4.10.3. Операция *<>*

При открытии файла с помощью функции `open()` одним из ее параметров является идентификатор, называемый дескриптором файла, с помощью которого можно в программе Perl сослаться на файл. Операция *ввода из файла* осуществляется заключением в угловые скобки его дескриптора `<дескриптор_файла>`. Результатом вычисления этой операции является строка файла или строки файла в зависимости от скалярного или спискового контекста ее использования. Следующие операторы иллюстрируют эту особенность данной операции:

```
open( MYFILE, "data.dat"); # Открытие файла "data.dat" и назначение ему  
                           # дескриптора MYFILE
```

```
$firstline = <MYFILE>; # Присваивание первой строки файла
```

```
@remainder = <MYFILE>; # Оставшиеся строки файла присваиваются  
                       # элементам массива скаляров.
```

Дескриптор файла можно использовать и в операции `print`, организовав вывод не на стандартное устройство вывода, а в файл, представляемый дескриптором:

```
print MYFILE @m;
```

Особый случай представляет операция чтения из файла с пустым дескриптором `<>`: информация считывается либо из стандартного файла ввода, либо из файлов, заданных в командной строке.

(Более подробно операции ввода/вывода из/в файл рассматриваются в главе 6.)

(Работа с файлами более подробно рассматривается в главе 7.)

4.11. Разные операции

В этом параграфе собраны операции, которые не вошли ни в одну из рассмотренных нами групп операций. Две из них упоминались при описании массивов и хешей (операции диапазон и запятая), а третья является единственной тернарной операцией языка Perl (операция выбора).

4.11.1. Операция диапазон

Бинарная операция *диапазон* `..` по существу представляет две различных операции в зависимости от контекста, в котором она используется.

В *списковом* контексте, если ее операндами являются числа (числовые литералы, переменные или выражения, возвращающие числовые значения), она возвращает список, состоящий из последовательности увеличивающихся на единицу целых чисел, начинающихся со значения, определяемого левым операндом, и не превосходящих числовое значение, представленное правым операндом. Операцию диапазон часто используют для задания значений элементов массивов и хешей, а также их фрагментов (см. главу 3). Она удобна для организации циклов `for` и `foreach`:

```
# Напечатает числа от 1 до 5, каждое на новой строке.
```

```
foreach $cycle (1..5){  
print "$cycle\n";  
}
```

```
# Напечатает строку "12345".
```

```
for(1..5){  
print;  
}
```

(Операторы цикла рассматриваются в главе 5.)

Замечание

Если значение какого-либо операнда не является целым, оно приводится к целому отбрасыванием дробной части числа.

Если левый операнд больше правого операнда, то операция диапазон возвращает пустой список. Подобную ситуацию можно отследить с помощью функции `defined()`, возвращающей истину, если ее параметр определен, или простой проверкой логической истинности массива, элементам которого присваивались значения с помощью операции диапазон:

```
$min = 2;
$max = -2;
@array = ($min .. $max); # Массив не определен.
print "@array array\n" if defined(@array); # Печати не будет!
print "@array array\n" if @array; # Печати не будет!
```

Замечание

В операции диапазон можно использовать и отрицательные числа. В этом случае возвращается список отрицательных чисел, причем значение левого операнда должно быть меньше значения правого операнда:

```
(-5..5) # Список чисел: (-2, -1, 0, 1, 2).
(-5..-10) # Пустой список.
```

Если операндами операции диапазон являются строки, содержащие буквенно-цифровые символы, то в списковом контексте эта операция возвращает список строк, расположенных между строками операндов с использованием лексикографического порядка:

```
@a = ("a".."d" ); # Массив @a: "a", "b", "c", "d"
@a = ("01".."31" ); # Массив @a: "01", "02", ... , "30", "31"
@a = ("a1".."d4" ); # Массив @a: "a1", "a2", "a3", "a4"
```

Если левый операнд меньше правого, с точки зрения лексикографического порядка, то возвращается единственное значение, равное левому операнду.

Замечание

Операция диапазон не работает со строками, содержащими символы национальных алфавитов. В этом случае она всегда возвращает единственное значение, соответствующее левому операнду.

В *скалярном* контексте операция диапазон возвращает булево значение Истина или Ложь. Она работает как переключатель и эмулирует операцию запятая ",", пакетного редактора **sed** и фильтра **awk** системы Unix, представляющую диапазон обрабатываемых строк этими программами.

Каждая операция диапазон поддерживает свое собственное булево состояние, которое изменяется в процессе ее повторных вычислений по следующей схеме. Она ложна, пока ложным остается значение ее левого операнда. Как только левый операнд становится истинным, операция диапазон переходит в состояние Истина и находится в нем до того момента, как ее правый операнд не станет истинным, после чего операция снова переходит в состояние Ложь. Правый операнд не вычисляется, пока операция находится в состоянии Ложь; левый операнд не вычисляется, пока операция диапазон находится в состоянии Истина.

В состоянии Ложь возвращаемым значением операции является пустая строка, которая трактуется как булева Ложь. В состоянии Истина при повторном вычислении она возвращает следующее порядковое число, отсчет которого начинается с единицы, т. е. как только операция переходит в состояние Истина, она возвращает 1, при последующем вычислении, если она все еще находится в состоянии Истина, возвращается 2 и т. д. В момент, когда операция переходит в состояние Ложь, к ее последнему возвращаемому порядковому числу добавляется строка "Е0", которая не влияет на возвращаемое значение, но может быть использована для идентификации последнего элемента в диапазоне вычислений. Программа примера 4.12 и ее вывод, представленный в примере 4.13, иллюстрируют поведение оператора диапазон в скалярном контексте. Мы настоятельно рекомендуем внимательно с ними ознакомиться, чтобы "почувствовать", как работает эта операция.

Пример. 4.12. Тест операции диапазон

```
#!/ perl -w

$left = 3;    # Операнд1
$right = 2;   # Операнд2

# Заголовок таблицы
print "\$i\tДиапазон\tОперанд1\tОперанд2\n";
print '-' x 48, "\n\n";

# Тест операции
for($i=1; $i <= 10; $i++) {
    $s = $left..$right;
    print "$i\t $s\t\t $left \t\t $right\n";
    $s10 = 3 if $i==5; # Когда переменная цикла $i равна 5,
                      # $s10 устанавливается равной 3.
    if ($right==0) {} else {--$right}; # Уменьшение $right на 1, пока
                                      # $right не достигла значения 0.
    --$left;
}
```

Замечание

В целях экономии времени мы не объясняем смысл незнакомых операторов примера 4.12, надеясь, что читатель сможет понять их смысл. Если все же это окажется для него сложным, мы рекомендуем снова вернуться к этой программе после прочтения главы 5.

Пример 4.13. Результаты выполнения теста операции диапазон "..."

\$i	Диапазон	Операнд1	Операнд2
<hr style="border-top: 1px dashed black;"/>			
1	1E0	3	2
2	1E0	2	1
3	1	1	0
4	2	0	0
5	3	-1	0
6	4E0	-2	2
7	1E0	-3	1
8	1	-4	0
9	2	-5	0
10	3	-6	0

Сделаем замечания относительно работы программы примера 4.12. На первом шаге цикла левый операнд операции диапазон истинен, следовательно сама операция находится в состоянии Истина и возвращает первое порядковое число (1). Но правый операнд становится также истинным ($\$right = 2$), следовательно она переходит в состояние Ложь и к возвращаемому ей значению добавляется строка "E0". На втором шаге цикла левый операнд истинен ($\$left = 2$) и операция переходит в состояние Истина, возвращая значение 1, к которому опять добавляется строка "E0", так как истинный правый операнд ($\$right = 1$) переводит операцию в состояние Ложь.

На третьем шаге операция становится истинной ($\$left = 1$), возвращая 1, и правый операнд со значением Ложь ($\$right = 0$) не влияет на ее состояние. На следующих шагах 4 и 5 правый операнд остается ложным, а операция возвращает соответственно следующие порядковые числа 2 и 3. На шаге 6 операция находится в состоянии Истина и возвращает 4, но правый операнд, принимая значение Истина ($\$right = 0$), переводит ее в состояние Ложь, в котором к возвращаемому значению добавляется строка "E0" и т. д.

Подобное поведение, связанное с переходом из состояния Истина в состояние Ложь и одновременным изменением возвращаемого значения (добавле-

нием строки "EO") эмулирует поведение операции запятая фильтра `awk`. Для эмуляции этой же операции редактора `sed`, в которой изменение возвращаемого значения осуществляется при следующем вычислении операции диапазон, следует вместо двух точек в знаке операции `".."` использовать три точки `"..."`. Результаты вывода программы примера 4.12, в которой осуществлена подобная замена, представлены в примере 4.14.

Пример 4.14. Эмуляция операции запятая редактора `sed`

\$i	Диапазон	Операнд1	Операнд2

1	1	3	2
2	2EO	2	1
3	1	1	0
4	2	0	0
5	3	-1	0
6	4EO	-2	2
7	1	-3	1
8	2	-4	0
9	3	-5	0
10	4	-6	0

Еще одно достаточно полезное свойство операции диапазон в скалярном контексте, используемое при обработке строк файлов, заключается в том, что если какой-либо операнд этой операции задан в виде числового литерала, то он сравнивается с номером прочитанной строки файла, хранящейся в специальной переменной `$.`, возвращая булево значение Истина при совпадении и Ложь в противном случае. В программе примера 4.15 иллюстрируется такое использование операции диапазон. В ней осуществляется пропуск первых не пустых строк файла, печатается первая строка после пустой строки и после этого завершается цикл обработки файла.

Пример 4.15. Операция диапазон в обработке строк файла

```
#!/ perl -w
open(POST, "file.txt") or die "Нельзя открыть файл file.txt!";
LINE:
while(<POST>) {
    $temp = 1..^$/;          # Истина, пока строка файла не пустая.
    next LINE if ($temp);    # Переход на чтение новой строки,
```

```

                                # если $temp истинна.

print $_;    # Печать первой строки файла после не пустой
last;        # Выход из цикла
}

close(POST);

```

В этой программе для нас интересен оператор присваивания возвращаемого значения операции диапазон переменной `$temp`. Прежде всего отметим, что эта операция используется в скалярном контексте, причем ее левый операнд представлен числовым литералом. На первом шаге цикла читается первая строка файла и специальной переменной `$.` присваивается ее номер 1, который сравнивается со значением левого операнда операции диапазон. Результатом этого сравнения является булево значение Истина, и операция диапазон переходит в состояние Истина, в котором она и остается, если первая строка файла не пустая, так как операция поиска по образцу `/^$/` возвращает в этом случае Ложь. Операция `next` осуществляет переход к следующей итерации цикла, во время которой читается вторая строка файла. Операция диапазон остается в состоянии Истина, если прочитанная строка файла пустая, увеличивая возвращаемое значение на единицу. Далее операция `next` снова инициирует чтение новой строки файла. Если строка файла пустая, то операция поиска по образцу возвращает значение Истина, переводя тем самым операцию диапазон в состояние Ложь, которое распознается при следующем ее вычислении, поэтому считывается еще одна строка файла (первая после пустой). Теперь операция `next` не выполняется, так как операция диапазон возвращает Ложь, печатается первая не пустая строка и операция `last` прекращает дальнейшую обработку файла.

4.11.2. Операция запятая

Бинарная операция *запятая* `,` ведет себя по-разному в скалярном и списковом контексте.

В *списковом* контексте она является всего лишь разделителем между элементами списка:

```

@a = (1, 2); # Создается массив скаляров
              # и его элементам присваиваются значения 1 и 2.

```

В *скалярном* контексте эта операция полностью соответствует аналогичной операции языка C: вычисляется левый операнд, затем правый операнд, вычисленное значение которого и является возвращаемым значением этой операции. Если в предыдущем примере заменить массив `@a` скалярной переменной `$a`, то ей будет присвоено значение 2:

```

$a = (1, 2); # Переменной $a присваивается значение 2.

```


Замечание

Надеемся, читателю теперь стало ясно, почему конструкторы массивов, о которых мы рассказывали в гл. 3, в скалярном и в списковом контексте ведут себя по-разному.

Для операции запятая в языке Perl существует удобный синоним — операция `=>`, которая полностью идентична операции запятая и удобна при задании каких-либо величин, которые появляются парами, например, ключ/значение в ассоциированных массивах. Правда, эта операция обладает еще одним свойством, достаточно удобным для ее использования при задании ассоциированных массивов: любой идентификатор, используемый в качестве ее левого операнда, интерпретируется как строка (см. раздел 3.4 главы 3).

4.11.3. Операция выбора

Единственная тернарная операция *выбора*

операнд1 ? операнд2 : операнд3

полностью заимствована из языка C и работает точно так же, как и ее двойник. Если операнд1 истинен, то возвращается значение операнд2, в противном случае операнд3:

```
($n == 1) ? $a : @array;
```

Скалярный или списковый контекст, в котором используется эта операция, распространяется и на возвращаемое значение этой операции:

```
$a = $yes ? $b : @b; # Скалярный контекст. Если возвращается
                     # массив @b, то присваивается количество его
                     # элементов.
```

Операцию выбора можно использовать в качестве левого операнда операции присваивания, если и второй, и третий ее операнды являются правильными *lvalue*, т. е. такими значениями, которым можно присвоить какое-либо значение, например, именами переменных:

```
($a = $yes ? $b : @b) = @c;
```

В связи с тем, что результатом операции выбора может оказаться правильное *lvalue*, следует использовать скобки для уточнения ее операндов. Например, если в следующем выражении

```
($a % 3) ? ($a += 2) : ($a -= 2);
```

опустить скобки вокруг операндов

```
$a % 3 ? $a += 2 : $a -= 2;
```

то оно будет откомпилировано следующим образом

```
(( $a % 3 ) ? ( $a += 2 ) : $a) -= 2;
```

4.12. Списковые операции

Списковая операция — это операция над списком значений, причем список не обязательно заключать в круглые скобки.

Мы уже знакомы с одной из таких операций — операцией вывода на стандартное устройство `print`. Иногда мы говорили об этой операции как о функции — и это справедливо, так как *все* списковые операции Perl выглядят как вызовы функций (даже их описание находится в разделе "Функции" документации по Perl), и более того, они позволяют заключать в круглые скобки список их параметров, что еще сближает их с функциями. Таким образом, списковую операцию (или функцию) `print` можно определить в программе любым из следующих способов:

```
print $a, "string", $b; # Синтаксис списковой операции.  
print($a, "string", $b); # Синтаксис вызова функции.
```

(Встроенные функции более подробно рассматриваются в главе 11.)

Замечание

Описание многих встроенных функций (списковых операций) можно найти в главах, в которых вводятся понятия языка, для работы с которыми и предназначены определенные встроенные функции. Например, функции `print`, `printf` и `write` описаны в главе 6.

4.13. Операции заключения в кавычки

Кавычки (одинарные, двойные и обратные) в Perl мы используем для задания строковых литералов, причем получающиеся результирующие строковые данные существенно зависят от используемого типа кавычек: символы строки в одинарных кавычках трактуются так, как они в ней заданы, тогда как некоторые символы (`$`, `@`) или даже последовательности символов (`\n`, `\t`) в строке в двойных кавычках выполняют определенные действия. Все дело в том, что в Perl кавычки — это всего лишь удобный синтаксический эквивалент определенных операций, выполняемых над символами строки.

В языке, кроме трех перечисленных операций заключения в кавычки, определен еще ряд операций, выполняющих определенные действия со строковыми данными и внешне похожих на операции заключения в кавычки, на которые мы будем в дальнейшем ссылаться так же, как на операции заключения в кавычки.

Все операции *заклЮчения в кавычки* представлены в табл. 4.3 с эквивалентным синтаксисом (если таковой существует) и кратким описанием действий, выполняемых при их выполнении.

Таблица 4.3. Операции *заклЮчения в кавычки*

Общая форма	Эквивалентная форма	Значение	Возможность подстановки
q{ }	" "	Строковый литерал	Нет
qq{ }	""	Строковый литерал	Да
qx{ }	``	Команда системы	Да
qw{ }	()	Список слов	Нет
m{ }	//	Поиск по образцу	Да
qr{ }		Образец	Да
s{ }{ }		Подстановка	Да
tr{ }{ }	y///	Транслитерация	Нет

При использовании общей формы операции *заклЮчения в кавычки* вместо фигурных скобок {}, представленных в табл. 4.3, можно использовать любую пару символов, выбранную в качестве разделителя. Если выбранный символ не является какой-либо скобкой (круглой, угловой, квадратной или фигурной), то он ставится в начале и в конце строки, к которой должна быть применена соответствующая операция, тогда как в случае использования скобок-разделителей сначала используется открывающая скобка, а в конце закрывающая. Между знаком операции и строками в символах-разделителях может быть произвольное число пробельных символов. Обычно в качестве разделителя программистами Perl используется косая строка "/", хотя это и не обязательно.

В табл. 4.3 в последнем столбце также указывается, осуществляет ли соответствующая операция подстановку значений скалярных переменных и массивов, а также интерпретацию управляющих символов.

В этом параграфе мы остановимся только на первых четырех операциях *заклЮчения в кавычки*. Остальные операции, как непосредственно связанные с регулярными выражениями, будут подробно рассмотрены в главе 10.

4.13.1. Операция *q{ }*

Эта операция аналогична заданию строкового литерала в одинарных кавычках. В нем каждый символ строки представляет самого себя, подстановка значений переменных не выполняется. Единственное исключение — обрат-

ная косая черта, за которой следует символ-разделитель или еще одна обратная косая черта. Эти последовательности символов позволяют ввести непосредственно в строку символ разделителя или обратную косую черту (хотя обратная косая черта и так представляет саму себя). Несколько примеров:

```
q<Дескриптор \<FILE\>>; # Строка символов: Дескриптор <FILE>
q!Каталог \\bin\usr\n!; # Строка символов: Каталог \bin\usr\n
'Каталог \\bin\usr\n'; # Эквивалентно предыдущей операции
```

4.13.2. Операция **qq{ }**

Эта операция аналогична заданию строкового литерала в двойных кавычках. При ее выполнении осуществляется подстановка в строку значений скалярных переменных, начинающихся с символа \$, и переменных массивов скаляров, начинающихся с символа @, а также осуществляется интерпретация управляющих последовательностей (см. главу 3). После выполнения указанных действий будут сформированы строковые данные. Для задания в строке символа разделителя, используемого в этой операции, можно воспользоваться обратной косой чертой перед этим символом. Несколько примеров:

```
qq(print\(\) - функция вывода); # Строка символов:
                                #      print() - функция вывода
$m = 123;
qq/Целое\t$m\n/;               # Строка символов:
                                # Целое    123
"Целое\t$m\n";                 # Эквивалентно предыдущей операции.
```

4.13.3. Операция **qx{ }**

Эта операция аналогична заданию строкового литерала в обратных кавычках. При ее вычислении сначала осуществляется подстановка значений скалярных переменных и переменных массивов скаляров (если таковые присутствуют) в строку, заданную между разделителями операции, а затем полученная строка, как некая команда, передается на выполнение командному интерпретатору операционной системы и результат ее выполнения подставляется в формируемое операцией qx{ } окончательное строковое значение. Таким способом можно ввести в программу Perl результаты выполнения определенных команд или пользовательских программ. Несколько примеров:

```
$file = "file.tmp";
qx(del $file);           # Удаление файла с именем file.tmp
$rez = qx(progl -a);     # Переменная $rez содержит результаты вывода
                        # на экран программы progl
$rez = `progl -a`;       # Эквивалентно предыдущей операции
```

4.13.4. Операция `qw{ }`

Эта операция возвращает список слов, выделенных из строки, заданной между разделителями операции. Разделителями между словами считаются пробельные символы:

```
@m = qw( one two ); # Эквивалентно: $m[0] = 'one'; $m[1] = 'two';
```

Действие операции `qw{СТРОКА}` эквивалентно действию встроенной функции

```
split(' ', q{СТРОКА});
```

(Описание функции `split` см. в главе 10.)

Внимание

Наиболее часто встречающаяся ошибка при использовании этой операции — отделить слова запятыми. При включенном режиме отображения предупреждений `-w` будет сгенерировано сообщение о том, что, возможно, запятая используется для разделения слов, а не входит в состав слова.

4.13.5. Операция "документ здесь"

В Perl реализована еще одна интересная возможность "ввода" строковых данных в программу, которая основана на синтаксисе "документ здесь" командного интерпретатора shell системы UNIX. Она позволяет определить в программе строковые данные большого объема, расположенные в нескольких последовательных строках текста программы, и использовать их в качестве операндов разных операций: присваивания, печати и т. п.

Ее синтаксис прост: после знака операции `<<` задается завершающий идентификатор, который служит признаком окончания задания строковых данных. Это означает, что все строки данных, расположенные между текущей строкой, содержащей операцию "документ здесь" и строкой, содержащей завершающий идентификатор, рассматриваются как единый фрагмент строковых данных:

```
$multi_line_string = <<LINES;
```

```
строка 1
```

```
строка 2
```

```
LINES
```

В приведенном фрагменте кода скалярная переменная `$multi_line_string` будет содержать строку "строка 1\nстрока 2\n". Как видим, введенные нами с клавиатуры символы перехода на новую строку сохраняются при использовании операции "документ здесь". По умолчанию операция "документ здесь" интерпретирует содержимое всех строк программы до завершающего идентификатора как строковые данные, заключенные в двойные кавычки, сохраняя в них символ перехода на новую строку `"\n"`. Perl позволяет явно

указать, как будут интерпретироваться данные при этой операции, заключив в двойные кавычки завершающий идентификатор операции. Следующие две операции "документ здесь" эквивалентны:

```
print <<m;
line 1
m
print <<"m";
line 1
m
```

Идентификатор можно задавать и в одинарных кавычках, и в обратных кавычках. В этом случае содержимое последующих строк программы до строки, содержащей завершающий идентификатор, трактуется как строковые данные в соответствующих кавычках.

Внимание

В строке, физически ограничивающей данные операции "документ здесь", завершающий идентификатор задается без каких-либо кавычек.

При задании завершающего идентификатора в кавычках на строковые данные распространяются все правила подстановок переменных и управляющих символов, применяемые к строкам, ограниченным соответствующим типом кавычек. Пример 4.16 демонстрирует использование различных кавычек в операции "документ здесь".

Пример 4.16. Кавычки в операции "документ здесь"

```
#!/ perl -w
```

```
$var = "Александр";
print <<FIRST;          # Отобразит:
Пользователь:          # Пользователь:
\t$var                 #      Александр
FIRST                  #

print <<'FIRST';         # Отобразит:
Пользователь:          # Пользователь:
\t$var                 # \t$var
FIRST                  #

$com1 = "echo Alex";
print <<`FIRST`;        # Отобразит:
```

```
$com1          # Alex
FIRST          #
```

Обратите внимание, что в примере 4.16 использовался одинаковый завершающий идентификатор `FIRST`. Это не приводит к двусмысленностям и ошибкам компиляции, так как компилятор ищет первую после операции `<<` строку с завершающим идентификатором. Главное, чтобы завершающий идентификатор в строке завершения был задан именно так, как он задан в самой операции.

Замечание

При использовании операции "документ здесь" с завершающим идентификатором в обратных кавычках в некоторых операционных системах может возникнуть проблема с обработкой потока команд, определяемого в нескольких строках. Не все командные интерпретаторы могут обрабатывать несколько строк команд. Обычно они ориентированы на ввод команды в строке ввода, выполнения ее и ожидания следующего ввода команды. Некоторые командные оболочки могут обрабатывать несколько команд, заданных в одной строке через разделитель, например, командный интерпретатор `cmd` системы Windows NT, в котором разделителем служит символ `&`.

Если завершающий идентификатор в операции `<<` задан без кавычек, то он должен следовать за знаком операции без каких-либо пробелов. Если такое случается, то Perl интерпретирует эту операцию с завершающим идентификатором пустая строка `"` и ищет в тексте программы первую пустую строку, ограничивающую строковые данные этой операции:

```
$var = "Александр";
print << x2;          # Отобразит 2 раза следующие 3 строки:
Пользователь:        # Пользователь:
\t$var               #          Александр
x2                   # x2
                     # Пустая строка завершает операцию <<
```

В этом фрагменте кода ошибочно поставлен пробел перед завершающим идентификатором `x2`. Компилятор разобрал строку с операцией печати `print` следующим образом: строковые данные, вводимые операцией `<<`, завершаются пустой строкой, после чего они просто повторяются 2 раза (последовательность символов `x2` понимается как операция повторения строки `x` с правым операндом равным 2).

Внимание

Этот пример подобран специально таким образом, чтобы он нормально откомпилировался. Если вместо идентификатора `x2` поставить, например `FIRST`, то компилятор сгенерирует ошибку.

Результат выполнения операции "документ здесь" можно использовать в качестве операнда строковых операций. Можно даже использовать несколько операций << в одном операторе, расположив строки их данных последовательно друг под другом, не забыв, конечно, строки с завершающим идентификатором:

```
$stack = <<"ONE".<<"TWO";
```

Первый

операнд

ONE

Второй

операнд

TWO

Значением скалярной переменной `$stack` будет следующая строка:

```
"Первый\nоперанд\nВторой\nоперанд"
```

4.14. Выражения

С помощью операций в программе можно выполнить определенные действия над данными. Если для реализации алгоритма решения поставленной задачи необходимо произвести несколько действий над определенными данными, то это можно осуществить последовательным выполнением операций, сохраняя при необходимости их результаты во временных переменных, а можно построить одно выражение, составленное из последовательности операций и их операндов, и получить необходимый результат.

Итак, *выражение* можно представить как последовательность операндов, соединенных одной или более операциями, результатом выполнения которой является единственное скалярное значение, массив или хеш. Операнды, в свою очередь, сами могут быть выражениями, что приводит к заданию в программе достаточно сложных выражений, вычисление которых начинается с их синтаксического разбора.

Когда компилятор начинает синтаксический разбор выражения Perl, он прежде всего выделяет в нем элементарные члены, называемые *термами*, а потом по определенным правилам соединяет их знаками операций, заданными в выражении, т. е. определяет последовательность выполнения операций над термами в соответствии с определенным в языке приоритетом операций. После такого разбора выражение вычисляется в соответствии с полученной последовательностью термов, соединенных знаками операций.

Таким образом, выражение можно мыслить как последовательность термов, соединенных знаками операций.

4.14.1. Термы

Чтобы понять, как вычисляется выражение, следует знать, что представляет собой терм — элементарный член арифметического или логического выражения. В Perl *термом* является любой литерал, любая переменная, выражение в круглых скобках, любая строка символов, к которой применена операция заключения в кавычки, а также любая функция с параметрами, заключенными в круглые скобки. В конечном итоге только терм может быть операндом вычисляемой операции в выражении.

Из всех перечисленных объектов языка, которые рассматриваются как термы, требует некоторого пояснения последний — функция с параметрами в круглых скобках.

В действительности в Perl отсутствуют истинные функции, понимаемые в смысле, например, языка C, в котором регламентировано обращение в программе к функции указанием ее имени с параметрами, заданными в круглых скобках. По существу, "функции" языка Perl являются списковыми и унарными именованными *операциями*, ведущими себя как функции, так как синтаксис языка позволяет заключать их параметры в круглые скобки. Вызывая в программе функцию (с заключенными или не заключенными в скобки параметрами), мы выполняем операцию (списковую или унарную именованную). Причем следует иметь в виду, что коль скоро выполняется операция, то она не только выполняет предписанные ей действия, но и возвращает определенный результат, который используется при вычислении выражения. Например, функция `print` возвращает Истину, если ее вывод завершен успешно, и Ложь в противном случае. Что напечатается при вычислении следующей операции?

```
print print "0";
```

Ответ — строка `01`, так как первая операция `print` должна напечатать результат вычисления второй операции `print`, которая успешно выводит на экран монитора `0`. Следовательно, ее результат Истина, а она представляется числом `1`.

При синтаксическом разборе выражений и операторов (о них в следующем разделе) как термы рассматриваются конструкции `do{}` и `eval{}`, вызовы подпрограмм и методов объектов, анонимные конструкторы массивов скаляров `[]` и хешей `{}`, а также все операции ввода/вывода. Все эти понятия будут рассмотрены в последующих главах книги, но мы сочли необходимым, в целях полноты изложения термов, просто привести их полный список.

4.14.2. Приоритет операций

После выделения и вычисления термов выражение разбирается с целью выявления последовательности выполнения операций в выражении: какая из

них должна быть выполнена раньше другой. Это достаточно ответственная процедура, так как порядок выполнения операций существенно влияет на результат вычисления всего выражения. Например, результатом вычисления выражения

$$4+3*2$$

будет 14, если сначала выполнить сложение, а потом умножение, и 10, если сначала выполнить умножение, а потом сложение. Дабы избежать подобных двусмысленностей в языках программирования, вводится *приоритет*, или *старшинство операций*, который учитывается при вычислении выражения. Приоритет операции умножения выше приоритета сложения, а поэтому наше арифметическое выражение будет однозначно вычислено равным 10.

В табл. 4.4 представлены все операции Perl в порядке убывания их приоритета, в ней также определен порядок выполнения операций с одинаковым приоритетом (столбец **Сочетаемость**).

Таблица 4.4. Приоритет и сочетаемость операций Perl

Приоритет	Операция	Сочетаемость
1	Вычисление термов и левосторонних списковых операций	Слева направо
2	->	Слева направо
3	++ --	Не сочетаются
4	**	Справа налево
5	! ~ \ унарные + и -	Справа налево
6	=~ !=	Слева направо
7	* / % x	Слева направо
8	+ - .	Слева направо
9	<< >>	Слева направо
10	Именованные унарные операции	Не сочетаются
11	< > <= >= lt gt le ge	Не сочетаются
12	== != <=> eq ne cmp	Не сочетаются
13	&	Слева направо
14	^	Слева направо
15	&&	Слева направо
16		Слева направо
17	Не сочетаются

Таблица 4.4 (окончание)

Приоритет	Операция	Сочетаемость
18	?:	Справа налево
19	= *= += -= . = *= /= %= x= &= = ^= <=> &&= =	Справа налево
20	, =>	Слева направо
21	Правосторонние списковые операции	Не сочетаются
22	not	Справа налево
23	and	Слева направо
24	or xor	Слева направо

Некоторые операции, приведенные в табл. 4.4, требуют пояснения. И первым в этом ряду стоят операции с наивысшим приоритетом: термы и левосторонние списковые операции. Термы мы определили в предыдущем разделе и там же разъяснили, что списковые операции и унарные именованные операции рассматриваются компилятором Perl как термы, если список их параметров заключен в круглые скобки. Так как умножение имеет больший приоритет, чем унарная именованная операция `sin`, то следующие операции вычисляются так, как указано в комментариях к ним:

```
use Math::Trig; # В пакете определена константа
                # pi = 3.14159265358979
sin 1 * pi;     # sin( 1 * pi) = 1.22460635382238e-016
sin (1) * pi;   # (sin 1) * pi = 2.64355906408146
```

В последнем выражении `sin (1)` рассматривается как *терм*, так как после имени операции первой распознаваемое лексемой стоит открывающая круглая скобка, а если это терм, — то и вычислять его надо в первую очередь, как операцию с наивысшим приоритетом.

Можно чисто визуально в тексте программы списковую или унарную именованную операцию с параметрами в круглых скобках сделать не похожей на вызов функции, поставив префикс `+` перед списком ее параметров:

```
sin +(1) * pi;   # (sin 1) * pi = 2.64355906408146
```

Этот префикс не выполняет никакой семантической роли в программе, даже не преобразует параметр в числовой тип данных. Он просто служит для акцентирования того факта, что `sin` не является функцией, а представляет собой унарную именованную операцию.

Если в списковой операции отсутствуют скобки вокруг параметров, то она может иметь либо наивысший, либо самый низкий (ниже только логические

операции `not`, `and`, `or` и `xor`) приоритет. Это зависит от того, где расположена операция относительно других операций в выражении: слева или справа. Все операции в выражении, расположенные *слева* от списковой операции (сама операция расположена *справа* от них), имеют более высокий приоритет относительно такой списковой операции, и вычисляются, естественно, раньше нее. Именно это имелось в виду, когда в табл. 4.4 вносилась строка с правосторонними списковыми операциями. Следующий пример иллюстрирует правостороннее расположение списковой операции:

```
$m = $n || print "Нуль, пустая строка или не определена!";
```

По замыслу программиста это выражение должно напечатать сообщение, если только значение переменной `$n` равно нулю, пустой строке или не определено. На первый взгляд, кажется, так и должно быть: выполнится операция присваивания и возвратит присвоенное значение. Если оно не равняется нулю, пустой строке или значению переменной `$n` не определено, то в булевом контексте операции логического ИЛИ (`||`) оно трактуется как Истина, а поэтому второй операнд этой логической операции (операция печати) не вычисляется, так как мы помним, что логическое ИЛИ вычисляется по укороченной схеме. Однако реально переменной `$m` *всегда* будет присваиваться 1, правда сообщение будет печататься именно тогда, когда переменная `$n` равна нулю, пустой строке или не определена.

В чем дело? Программист забыл о приоритете правосторонних списковых операций! В этом выражении списковая операция `print` расположена справа от всех остальных операций, поэтому она имеет самый низкий приоритет. Выражение будет вычисляться по следующему алгоритму. Сначала будет вычислен левый операнд операции `||`. Если он имеет значение Истина (переменная `$n` имеет значение, не равное нулю или пустой строке), то второй операнд этой операции (`print`) вычисляться не будет, а переменной `$m` будет присвоена Истина, т. е. 1. Если первый операнд вычисляется как Ложь (переменная `$n` равна нулю, пустой строке или не определена), то вычисляется второй операнд, выводящий сообщение на экран монитора. Но так как возвращаемым значением операции печати является Истина, то именно она и присваивается переменной `$m`.

Правильное решение — использовать низкоприоритетную операцию `or` логического ИЛИ:

```
$m = $n or print "Нуль, пустая строка или не определена!";
```

или скобками изменить порядок выполнения операций:

```
($m = $n) || print "Нуль, пустая строка или не определена!";
```

Теперь обратимся к случаю, когда списковая операция стоит *слева* от других операций в выражении (*левосторонняя списковая операция*). В этом случае, в соответствии с табл. 4.4, она имеет наивысший приоритет и *все*, что стоит справа от нее, она рассматривает как список своих параметров. Рассмотрим

небольшой пример. Предположим, что необходимо удалить из массива `@a` все элементы, начиная со второго, и вставить их в создаваемый массив `@m` после второго элемента. Списковая операция `splice` со списком параметров `@a, 1` удаляет из массива `@a` все элементы, начиная с элемента с индексом 1, т. е. со второго элемента до конца массива, и возвращает список удаленных элементов. Ее можно использовать в конструкторе нового массива для решения поставленной задачи:

```
@a = ("a1", "a2", "a3", "a4");  
@m = ("m0", "m1", splice @a, 1, "m2", "m3");
```

В конструкторе массива мы специально задали параметры операции `splice` без скобок. Если выполнить этот фрагмент и распечатать значения элементов массивов, то результат будет следующим:

```
@m: m0 m1  
@a: a1 m3 a2 a3 a4
```

Совершенно не то, что нам надо: в массив `@m` не вставлен фрагмент массива `@a`, да и из него самого не удалены элементы, начиная со второго. Все дело в том, что операция `splice` в этом выражении левосторонняя, и весь расположенный справа от нее список рассматривает как список своих параметров: `@a, 1, "m2", "m3"`. Ее третьим параметром должно быть число, определяющее количество удаляемых из массива элементов, начиная с элемента, индекс которого определен вторым параметром. В нашем случае третий параметр не является числовым, и функция завершается с ошибкой, возвращая Ложь. Исправить положение помогут опять скобки:

```
@m = ("m0", "m1", (splice @a, 1), "m2", "m3");
```

или

```
@m = ("m0", "m1", splice (@a, 1), "m2", "m3");
```

Завершая разговор о приоритете выполнения операций, следует объяснить свойство *сочетаемости операций* и его практическое применение. Сочетаемость важна при вычислении выражений, содержащих операции с одинаковым приоритетом, и определяет порядок их вычисления. Рассмотрим выражение:

```
$m += $n += 1;
```

Как следует его понимать? Как $(\$m += \$n) += 1$ или как $\$m += (\$n += 1)$? Ответ дает правило сочетаемости. Смотрим в табл. 4.4 и видим, что все операции присваивания сочетаются справа налево. Это означает, что сначала должно выполняться присваивание $\$n += 1$, а потом результат увеличенной на единицу переменной $\$n$ прибавляется к переменной $\$m$. Следовательно, это выражение эквивалентно следующему:

```
$m += ($n += 1);
```

Аналогично применяется правило сочетаемости и к другим операциям языка Perl:

`$a>$b<$c`; # Эквивалентно: `($a>$b)<$c`; Сочетаемость: слева направо.

`$a**$b**$c`; # Эквивалентно: `$a**($b**$c)`; Сочетаемость: справа налево.

Скобки изменяют порядок вычислений, определяемый по правилу приоритетов и сочетаемости. Любое, заключенное в скобки подвыражение, будет вычисляться с наивысшим приоритетом, так как Perl рассматривает его как терм, имеющий наивысший приоритет.

4.14.3. Контекст

Наш разговор о выражениях Perl был бы не полным, если бы обошли стороной такое понятие, как *контекст*. Каждая операция и каждый терм вычисляются в определенном контексте, который определяет поведение операции и интерпретацию возвращаемого ею значения. Существует два основных контекста: *скалярный* и *списковый*. В главе 3 мы уже немного познакомились с ними, когда определяли поведение конструктора массива и переменной массива в правой части оператора присваивания. Их можно "определить" так: если для выполнения операции требуются скалярные данные, то действует скалярный контекст, если необходимы массивы скаляров, то программа находится в списке контексте. Например, если левый операнд операции присваивания, скалярная переменная, то действует скалярный контекст, если же в левой части задан массив, хеш или фрагмент массива или хеша, то вычисления в правой части осуществляются в списке контексте. Присваивание списку скалярных переменных также инициирует список контекст для вычислений, осуществляемых в правой части операции.

Некоторые операции Perl распознают контекст, в котором они вычисляются, и возвращают список в списке контексте и скалярное значение в скалярном контексте. Обладает ли операция подобным поведением, можно всегда выяснить из ее описания в документации. Например, можно рассматривать префикс `@` перед идентификатором как унарную операцию объявления массива скаляров, распознающую контекст, в котором она вычисляется. В списке контексте она возвращает список элементов массива, а в скалярном — число элементов массива.

Некоторые операции поддерживают список контекст для своих операндов (в основном это списковые операции), и это также можно узнать из описания их синтаксиса, в котором присутствует СПИСОК (LIST). Например, известная уже нам операция создания фрагмента массива `splice` поддерживает список контекст для своих параметров, поэтому ее можно вызывать вот таким образом:

```
@s = (1, 2);
```

```
splice @m, @s; # Эквивалентно: splice @m, 1, 2;
```

Скалярный контекст можно подразделить на *числовой*, *строковый* и *безразличный*. Если скалярный и списковый контекст некоторыми операциями распознается, то ни одна операция не может определить, вычисляется ли она в числовом или строковом скалярном контексте. Perl просто при необходимости преобразует возвращаемые операцией числа в строки и наоборот. В некоторых случаях вообще не важно, возвращается ли операцией число или строка. Подобное, например, происходит при присваивании переменной какого-либо значения. Переменная просто принимает подтип присваиваемого значения. Такой контекст называется безразличным.

В языке существует *булевый* контекст — специальный тип скалярного контекста, в котором вычисленное значение выражения трактуется только как Истина или Ложь. Как уже отмечалось ранее, в Perl нет специального булева типа данных. Здесь любая скалярная величина трактуется как Истина, если только она не равна пустой строке "" или числу 0. (Строковый эквивалент ложности — строка из одного нуля "0"; любая другая строка, эквивалентная нулевому значению, считается в булевом контексте истинной, например, "00" или "0.0".)

Другим специфическим типом скалярного контекста является *void-контекст*. Он не только не заботится о подтипе возвращаемого значения — скалярный или числовой, но ему и не надо никакого возвращаемого значения. Этот контекст возникает при вычислении выражения без побочного эффекта, т. е. когда не изменяется никакая переменная программы. Например, следующие выражения вычисляются в void-контексте:

```
$n;  
"текст";
```

Этот контекст можно "обнаружить", если установить ключ -w компилятора Perl. Тогда можно получить предупреждающее сообщение следующего типа:

```
Useless use of a variable in void context at D:\P\EX.PL line 3.
```

(Бесполезное использование переменной в void-контексте в строке 3 программы D:\P\EX.PL)

Завершит наш рассказ о контексте *подстановочный контекст* (interpolative context), в котором вычисляются операции заключения в кавычки (кроме заключения в одинарные кавычки). В этом контексте вместо любой переменной, заданной в строке, в нее подставляется значение этой переменной, а также интерпретируются управляющие последовательности.

* * *

В этой главе мы изучили практически все скалярные операции языка Perl, чуть-чуть коснулись операций сопоставления по образцу, создания ссылок и операций ввода\вывода, познакомились с основами работы со списками

и унарными именованными операциями. Узнали, что такое выражение, а также в каком порядке вычисляются в нем операции на основе их приоритета и сочетаемости. Научились выделять термы в выражениях и выяснили, в каких контекстах могут вычисляться выражения.

Вопросы для самоконтроля

1. Какую роль выполняют операции в программе?
2. Какие основные группы операций существуют в Perl?
3. Объясните "укороченную схему" вычисления логических операций. Где она используется?
4. Что такое выражение?
5. Определите понятие терм. Что считается термом в языке Perl?
6. Что такое приоритет операций и как он применяется при вычислении выражений?
7. Когда необходимо применять свойство сочетаемости операции?
8. Объясните понятие "контекст". Какие два основных типа контекста используются в языке Perl?

Упражнения

1. Что будет отображено на экране монитора при вычислении выражения
`print print 1;`
2. Определите результат вычисления следующих выражений:
`print "0" || print "1";`
`print "0" or print "1";`
3. Что будет отображено на экране монитора и каковы будут значения элементов массива @m в результате выполнения следующей операции присваивания:
`@m = (print "p\n", 2, print 3, 4);`
4. Определите результат выполнения следующих операторов:
`$var0 = 2;`
`$var1 = 1;`
`$rez1 = $var0 ** 3 * 2 || 4 + $var1, $var1++;`
`$rez2 = ($var1++, $var0 ** 3 * 2 || 4 + $var1, "6");`
`@rez3 = ($var1++, $var0 ** 3 * 2 || 4 + $var1, "6");`

5. Что напечатает следующий фрагмент программы при вводе числа или строки и почему:

```
$input = <STDIN>;  
$hello = "Hello ";  
$hello += $input;  
print $hello;
```

6. Найдите ошибку в программе:

```
$first_number = 34;  
$second_number = 150;  
if( $first_number lt $second_number ) { print $first_number; }
```



Операторы

Perl является императивным языком программирования: его программа состоит из последовательности операторов, определяющих некоторые действия. *Оператор* — это завершенная инструкция интерпретатору на выполнение определенного действия. Все операторы языка Perl делятся на простые и составные. Простой оператор представляет собой выражение, возможно, снабженное модификатором. Составной оператор определяется в терминах блоков.

Каждый оператор Perl имеет возвращаемое значение. Для простого оператора — это значение вычисляемого в нем выражения, для составного оператора — значение последнего вычисленного в нем оператора.

Операторы обрабатываются в той последовательности, в которой они встречаются в программе. Однако среди множества допустимых операторов языка Perl, есть группа операторов, которая изменяет последовательность выполнения операторов в программе.

5.1. Простые операторы

Простой оператор представляет собой любое выражение, завершенное точкой с запятой ";". Символ точка с запятой обязателен. Он может отсутствовать, только если простой оператор является последним оператором в блоке, специальной синтаксической конструкции, о которой мы поговорим чуть-чуть позже.

Основное назначение простого оператора — вычисление выражения с побочным эффектом, который связан с изменением значения некоторых переменных в выражении при его вычислении. Обычно он реализуется операциями увеличения/уменьшения на единицу (`++`, `--`):

```
$n++;      # Переменная $n увеличивается на единицу.  
--$n**2;  # Переменная $n уменьшается на единицу.
```

Никакие другие операции Perl не вызывают побочных эффектов в выражении, если не считать операции присваивания (простое и составное), результатом вычисления которой является изменение значения левого операнда. Вызовы функций также могут приводить к побочным эффектам, но об этом подробнее мы расскажем в *главе 11*.

Простой оператор Perl может содержать выражение и без побочного эффекта. Все равно он будет рассматриваться интерпретатором как допустимый оператор, не выполняющий никакого действия, а только вычисляющий значение выражения. Однако если установлен флаг (-w) отображения интерпретатором предупреждающих сообщений, то будет получено сообщение о бесполезности использования соответствующей операции в void-контексте. Например, при выполнении оператора

```
($n*$m)**4 + 6;
```

будет отображено сообщение

```
Useless use of addition in void context at D:\PERL\EXAMPLE3.PL line 4.
```

Отметим, что в сообщении упоминается о последней выполненной операции в выражении.

Замечание

Подобное сообщение будет отображаться, даже если в сложном выражении присутствует операция, вызывающая побочный эффект. Сообщение не отображается только в случае выражения, составленного из одних операций уменьшения /увеличения, или выражения присваивания.

Читатель спросит, какой прок в операторе, не выполняющем никакого действия. Его можно использовать для задания возвращаемого пользовательской функцией значения. Забегая вперед, скажем, что если в функции явно не указано в операторе `return()` возвращаемое значение, то по умолчанию Perl считает таковым значение последнего вычисленного оператора. Именно это обстоятельство и используется многими программистами для определения возвращаемого значения:

```
sub mySub {  
    какие-то операторы  
    condition == true ? "Успех" : "Провал"; # Последний оператор  
}
```

Последний оператор функции `mySub` вычисляет операцию выбора, в которой второй и третий операнды представлены просто строковыми литералами — выражениями без побочного эффекта. Результат вычисления одного из них и является возвращаемым значением функции.

5.2. Модификаторы простых операторов

Каждый простой оператор может быть снабжен *модификатором*, представляющим ключевое слово `if`, `unless`, `while`, `until` или `foreach`, за которым следует выражение-условие. В самом операторе модификатор стоит непосредственно за выражением, составляющим простой оператор, перед завер-

шающим символом точка с запятой. Каждый простой оператор может иметь только *один* модификатор. Семантически роль модификатора сводится к тому, что оператор вычисляется при выполнении условия, определяемого модификатором. Например, следующий оператор присваивания

```
$n = $l/$m if $m != 0;
```

с модификатором `if` будет выполнен при условии, что переменная `$m` не равна 0. Общий синтаксис простого оператора с модификатором имеет следующий вид:

```
ВЫРАЖЕНИЕ ключ_слово_модификатора [ ( ) ВЫРАЖЕНИЕ-УСЛОВИЕ ( ) ] ;
```

5.2.1. Модификаторы *if* и *unless*

Модификаторы `if` и `unless` употребляются в прямом смысле их английского значения. Простой оператор с модификатором `if` выполняется, если `ВЫРАЖЕНИЕ-УСЛОВИЕ` истинно. Семантически простой оператор

```
ВЫРАЖЕНИЕ if ВЫРАЖЕНИЕ-УСЛОВИЕ;
```

эквивалентен следующему оператору условия:

```
if (ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; }
```

Замечание

В этом разделе мы представляем эквивалентные простым операторам с модификаторами соответствующие составные операторы языка Perl. Их синтаксис и применение будут детально разобраны в следующих параграфах, но мы уверены, что читатель поймет их и без дополнительных объяснений. Во всяком случае всегда можно вернуться к данному разделу, прочитав разделы, посвященные составным операторам языка Perl.

Модификатор `unless` является прямой противоположностью модификатора `if`: простой оператор выполняется, если `ВЫРАЖЕНИЕ-УСЛОВИЕ` не истинно. Общий синтаксис простого оператора с модификатором `unless` имеет следующий вид:

```
ВЫРАЖЕНИЕ unless ВЫРАЖЕНИЕ-УСЛОВИЕ;
```

Это всего лишь удобная форма записи оператора условия

```
if ( ! ВЫРАЖЕНИЕ-УСЛОВИЕ ) { ВЫРАЖЕНИЕ; }
```

Замечание

`ВЫРАЖЕНИЕ-УСЛОВИЕ` вычисляется в булевом контексте: оно трактуется как Ложь, если равно 0 или пустой строке "", и Истина — в любом другом случае.

Использование модификаторов `if` и `unless` показано в примере 5.1 — простой программе решения квадратного уравнения.

Пример 5.1. Модификаторы *if* и *unless*

```
# perl -w
# Решение квадратного уравнения a*x**2+b*x+c=0
$a = <STDIN>;
$b = <STDIN>;
$c = <STDIN>;
$d = $b**2 - 4*$a*$c;           # Вычисление дискриминанта уравнения
# Вычисление корней, если дискриминант положителен
( $x1 = (-$b+sqrt $d)/$a/2, $x2 = (-$b-sqrt $d)/$a/2 ) unless $d < 0;
# Печать результатов
print "Коэффициенты:\n a = $a b = $b c = $c";
print "\tРешение:\n\t$x1\t$x2" if defined $x1;
print "\tРешения нет!" unless defined $x1;
```

Наша программа решения квадратного уравнения, конечно, примитивна. Из всех возможных проверок в ней проверяется на положительность только дискриминант квадратного уравнения, хотя стоило бы проверить на нуль значение вводимого пользователем с клавиатуры старшего коэффициента уравнения, сохраняемого в переменной *\$a*.

Модификатор *unless* используется в операторах вычисления корней и печати сообщения об отсутствии решения. Обратите внимание, что в операторе печати проверяется, определена ли переменная *\$x1*, а будет она определена только в случае положительности дискриминанта *\$d*. В модификаторе *if* оператора печати корней уравнения также проверяется, определена ли переменная *\$x1*.

5.2.2. Модификаторы *while* и *until*

Эти два модификатора немного сложнее модификаторов *if* и *unless*. Они реализуют процесс циклического вычисления простого оператора. Их синтаксис таков:

ВЫРАЖЕНИЕ *while* ВЫРАЖЕНИЕ-УСЛОВИЕ;

ВЫРАЖЕНИЕ *until* ВЫРАЖЕНИЕ-УСЛОВИЕ;

Модификатор *while* повторно вычисляет ВЫРАЖЕНИЕ, пока истинно ВЫРАЖЕНИЕ-УСЛОВИЕ. Модификатор *until* противоположен модификатору *while*: ВЫРАЖЕНИЕ повторно вычисляется до момента, когда ВЫРАЖЕНИЕ-УСЛОВИЕ станет истинным, иными словами оно вычисляется, пока ВЫРАЖЕНИЕ-УСЛОВИЕ ложно.

Семантически эти модификаторы простых операторов эквивалентны следующему составному операторам цикла:

```
while (ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; }
```

```
until (ВЫРАЖЕНИЕ-УСЛОВИЕ) { ВЫРАЖЕНИЕ; }
```

Пример 5.2 дает представление о том, как работают модификаторы повтора `while` и `until`.

Пример 5.2. Модификаторы повторных вычислений `while` и `until`

```
# perl -w
$first = 10;
$second = 10;
$first++ while $first < 15; # $first увеличивается, пока не станет
                           # равной 15
--$second until $second < 5; # $second уменьшается, пока не станет
                           # равной 4
print "\$first $first\n";
print "\$second $second\n";
```

Оператор увеличения на единицу переменной `$first` будет выполняться, пока выражение `$first < 15` остается истинным. Когда значение переменной `$first` станет равным 15, выражение модификатора `while` становится ложным, и оператор завершает работу. Аналогично работает и следующий оператор уменьшения переменной `$second` на единицу. Единственное отличие от предыдущего оператора заключается в том, что он выполняется, пока выражение `$second < 5` модификатора `until` остается ложным. Два оператора печати выведут на экран значения переменных `$first` и `$second` равными соответственно 15 и 4.

Замечание

Модификаторы повтора следует применять к простым операторам, вычисление которых приводит к изменению условий модификаторов. Если это не так, то простой оператор либо вообще не будет выполнен, либо будет выполняться бесконечно. Например, следующий оператор

```
print $var while $var < 15;
```

либо ни разу не напечатает значение переменной `$var` (если `$var >= 15`), либо будет печатать бесконечно (если `$var < 15`). В последнем случае произойдет так называемое заикливание и только нажатием комбинации клавиш `<Ctrl>+<C>` можно будет остановить выполнение этого оператора.

Модификаторы `while` и `until` сначала проверяют истинность или ложность своих выражений-условий, а потом, в зависимости от полученного результа-

та, либо выполняют простой оператор, либо нет. Таким образом, они реализуют *цикл с предусловием*, при котором оператор, для которого они являются модификаторами, может не выполниться ни одного раза.

Существует единственное исключение из этого правила, когда модификаторы повтора применяются к синтаксической конструкции `do{}`, которая не является оператором (хотя внешне и похожа), а относится к термам (см. главу 4). Поэтому, если ее завершить точкой с запятой, то такая конструкция будет являться простым оператором, к которому можно применять все возможные модификаторы Perl. Семантика этой конструкции заключается в том, что она вычисляет операторы, заданные в фигурных скобках {}, и возвращает значение последнего выполненного оператора. Так вот, если к простому оператору `do{}`; применить модификаторы повтора, то сначала выполнятся операторы конструкции `do{}`, а потом будет проверено условие модификатора. Это позволяет написать следующий простой оператор, который сначала осуществит ввод с клавиатуры, а потом проверит введенную информацию на совпадение с символом завершения ввода:

```
$string = "";  
do{  
    $line = <STDIN>;  
    $string .= $line;  
} until $line eq ".\n";
```

Этот фрагмент кода будет накапливать вводимые пользователем строки в переменной `$string` до тех пор, пока не будет введена строка, состоящая из единственного символа точки, после чего оператор `do{}` `until`; завершит свою работу. Обратите внимание, что проверка в модификаторе `until` ведется на совпадение со строкой `".\n"`, в которой присутствует символ перехода на новую строку. Дело в том, что операция ввода `<STDIN>` передает этот символ в программу, так как пользователь именно этим символом завершает ввод строки (нажатие клавиши `<Enter>`).

5.2.3. Модификатор *foreach*

Модификатор `foreach` **ВЫРАЖЕНИЕ** относится к модификаторам цикла. Он повторно выполняет простой оператор, осуществляя итерации по списку значений, заданному в **ВЫРАЖЕНИЕ**. На каждой итерации выбранный элемент списка присваивается встроенной переменной `$_`, которую можно использовать в простом операторе для получения значения выбранного элемента списка. Например, следующий оператор распечатает все элементы массива `@m`:

```
print "$_ " foreach @m;
```

Общий синтаксис простого оператора с модификатором `foreach` следующий:

ВЫРАЖЕНИЕ `foreach` **ВЫРАЖЕНИЕ-СПИСОК**;

Простой оператор с модификатором `foreach` всего лишь удобная форма записи составного оператора `foreach`:

```
foreach (ВЫРАЖЕНИЕ-СПИСОК) { ВЫРАЖЕНИЕ; }
```

Эта форма составного оператора `foreach` в качестве переменной цикла использует встроенную переменную `$_` (см. раздел 5.4.3).

Обратим внимание читателя на то, что `ВЫРАЖЕНИЕ-СПИСОК` вычисляется в списковом контексте, поэтому все используемые в нем переменные ведут себя так, как они должны вести в списковом контексте. Например, хеш-массив представляет обычный список, составленный из последовательности его пар ключ/значение. Следующий фрагмент кода

```
%hash = ( one=>6, two=>8, three=>10 );  
print "$_ " foreach %hash;
```

напечатает строку

```
three 10 two 8 one 6
```

Эта строка и есть тот простой список, который возвращает хеш в списковом контексте.

Относительно модификатора `foreach` (это же относится и к его эквивалентному оператору `foreach`) следует сказать одну важную вещь. Дело в том, что переменная `$_` является не просто переменной, в которой хранится значение элемента списка текущей итерации, она является синонимом имени этого элемента. Это означает, что любое изменение переменной `$_` в простом операторе приводит к изменению текущего элемента списка в цикле. Пример 5.3 демонстрирует, как просто можно умножить каждый элемент массива на некоторое число:

Пример 5.3. Модификатор `foreach` изменяет значения элементов списка

```
# perl -w  
@array = (1, 2, 3);  
$_ *= 2 foreach @array; # Умножение каждого элемента на 2.  
print "@array";        # Напечатает строку: 2 4 6
```

5.3. Составные операторы

Составные операторы — это второй тип операторов языка Perl. С их помощью реализуют ветвления в программе и организуют циклические вычисления. Эти операторы, в отличие от аналогичных операторов других языков программирования, определяются в терминах блоков — специальном понятии языка Perl, задающим область видимости переменных. Именно с блоков мы и начнем изучение составных операторов.

5.3.1. Блоки

Блок — последовательность операторов, определяющая область видимости переменных. В программе блок обычно ограничен фигурными скобками {...}. Определяя синтаксис составных операторов, мы будем иметь в виду именно такой блок — последовательность операторов в фигурных скобках и обозначать его *влок*. Интерпретатор рассматривает *влок* как один оператор, вычисляемым значением которого является значение последнего выполненного оператора блока. Это означает, что там, где можно использовать один оператор, можно использовать и *влок*. Такая ситуация встречается при использовании функции `map()`. Она выполняет определяемый ее первым параметром оператор для всех элементов списка, заданного вторым параметром. Значение каждого элемента списка при вычислениях временно присваивается встроенной переменной `$_`. Возвращает эта функция список вычисленных значений оператора:

```
@rez = map $_ **= 2, @array; # Список квадратов элементов массива.
```

В качестве первого параметра этой функции можно использовать *влок*. Следующий оператор также вычисляет список квадратов элементов массива `@array`, одновременно подсчитывая количество его элементов:

```
@rez = map { ++$kol; $_ **= 2 } @array; # Список квадратов элементов  
                                         # массива и подсчет их количества.
```

Обратите внимание, что возвращаемым значением блока операторов в этом примере является значение последнего оператора блока, которое и попадает в возвращаемый функцией `map()` список.

Замечание

Блоки в Perl не ограничиваются только последовательностью операторов в фигурных скобках. Иногда блок может быть ограничен содержащим его файлом. Например, файлом, содержащим используемый в программе модуль Perl, или файлом самой программы.

Как сказано в начале этого раздела, блок определяет *область видимости переменных*. Это означает, что в блоке можно создать переменные, обращаться к которым можно только из операторов, расположенных в этом блоке. Пока мы в блоке, мы можем присваивать им новые значения, использовать в вычислениях и т. п., но как только мы вышли из блока, мы теряем с ними "связь", они становятся "не видимыми". Такие переменные еще называют *локальными* переменными.

Локальные переменные создаются с помощью функции `my()`. Ее параметром является список переменных, область видимости которых ограничена блоком, в котором вызывается эта функция. Если список переменных состоит из одной переменной, то скобки не обязательны. Созданные функци-

ей `my()` переменные называются также *лексическими* переменными, так как область их действия ограничена фрагментом текста программы — блоком операторов.

В языке Perl можно создавать другой тип локальных переменных, область действия которых определяется динамически во время выполнения программы. Они создаются функцией `local()` и называются локальными *динамическими* переменными. Однако именно переменные, созданные функцией `my()`, являются "истинными" локальными переменными: они создаются при входе в блок и уничтожаются при выходе из него (хотя существуют ситуации, когда Perl не уничтожает локальную лексическую переменную при выходе из ее области действия). Функция `local()` всего лишь временно сохраняет старое значение глобальной переменной при входе в блок и восстанавливает его при выходе из него.

(Более подробно лексические и динамические переменные рассматриваются в главе 11.)

Локальные переменные удобны для создания временных переменных, которые нигде больше не будут использоваться в программе, а только в одном определенном месте. Например, при отладке части кода часто приходится создавать временные переменные и выводить на печать их значения. Локальные переменные могут иметь такие же имена, как и глобально используемые переменные. Это не приводит к конфликту. После завершения операторов блока значение глобальной переменной имеет то же значение, которое она имела до начала выполнения операторов блока (пример 5.4).

Пример 5.4. Лексические переменные

```
# perl -w
$var = "outer";           # Глобальная переменная $var
$glob = "glob";          # Глобальная переменная $glob
my $lex = "outer_1";      # Лексическая переменная $lex
{
    my($var) = "inner";    # Внутренняя переменная $var
    my($var1) = "inner_1"; # Внутренняя переменная $var1
    print "В блоке \$var = $var\n"; # Напечатает inner
    print "В блоке \$var1 = $var1\n"; # Напечатает inner_1
    print "В блоке \$lex = $lex\n"; # Напечатает outer_1
    print "В блоке \$glob = $glob\n"; # Напечатает glob
}
print "Вне блока \$var = $var\n"; # Напечатает outer
print "Вне блока \$lex = $lex\n"; # Напечатает outer_1
print "Вне блока \$var1 = $var1\n"; # Напечатает пустую строку ""
```

Программа примера 5.4 демонстрирует области видимости лексических переменных. Внутри блока {...} "видны" переменные, созданные вне блока: и глобальные, и лексические (\$glob, \$lex), если только они не переопределены внутри блока (\$var). При выходе из внутреннего блока восстанавливаются значения переменных, которые были переопределены внутри блока (\$var). Доступ к локальным переменным блока извне невозможен (\$var1).

5.3.2. Операторы ветвления

Операторы программы Perl выполняются последовательно в порядке их расположения в программе. Для реализации простых алгоритмов этого вполне достаточно. Однако большинство реальных алгоритмов не укладываются в такую линейную схему. Практически всегда при реализации любого алгоритма возникают ситуации, когда одну группу операторов надо выполнить только при выполнении определенных условиях, тогда как другую группу при этих же условиях вообще не следует выполнять. В языке Perl для организации подобного ветвления в программе предусмотрены операторы `if`, которые мы и будем называть *операторами ветвления*.

Эти операторы вычисляют выражение, называемое условием, и в зависимости от его истинности или ложности выполняют или не выполняют некоторый блок операторов. Это означает, что выражения условия во всех операторах ветвления вычисляются в булевом контексте.

Иногда приходится делать выбор на основе проверки нескольких различных условий. Для подобных цепочек ветвлений существует специальная форма оператора `if`, реализующая множественные проверки.

В языке существует три формы оператора ветвления `if`:

```
if (ВЫРАЖЕНИЕ) БЛОК
```

```
if (ВЫРАЖЕНИЕ) БЛОК1 else БЛОК2
```

```
if (ВЫРАЖЕНИЕ1) БЛОК1 elsif (ВЫРАЖЕНИЕ2) БЛОК2 ... else БЛОКn
```

Обратим внимание читателя еще раз на тот факт, что все они определяются в терминах блоков операторов, заключенных в фигурные скобки, поэтому даже если в блоке содержится один оператор, он должен быть заключен в фигурные скобки. Такой синтаксис составных операторов Perl может оказаться не совсем привычным для программистов на языке C, в котором фигурные скобки в случае одного оператора в блоке не обязательны.

Первый оператор `if` реализует простейшее ветвление. Если `ВЫРАЖЕНИЕ` истинно, то выполняются операторы из `БЛОК`, в противном случае `БЛОК` просто пропускается:

```
$var = 10;
```

```
if ( $var == 5 ) {
```

```
print "Переменная \ $var = $var";  
}
```

Обратите внимание, что в этом примере **ВЫРАЖЕНИЕ** представляет операцию составного присваивания. Это может показаться необычным, так как в большинстве языков программирования здесь требуется выражение, возвращающее булево значение. В Perl можно использовать любое выражение, в том числе и присваивание. Результат его вычисления интерпретируется в булевом контексте: если вычисленное значение равно 0 или пустой строке "", то оно трактуется как Ложь, иначе — Истина. Возвращаемым значением операции присваивания является значение, присвоенное переменной левого операнда. В нашем примере это число 5, следовательно в булевом контексте оно трактуется как Истина, а поэтому оператор печати `print` будет выполнен. Если перед выполнением оператора `if` переменная `$var` будет равняться 5, то выражение условия будет вычислено равным Ложь и все операторы блока будут просто пропущены.

Обычно выражение условия представляет собой сложное выражение, составленное из операций отношения, связанных логическими операциями. Использование операции присваивания в выражении условия оператора `if` не совсем типично. Здесь мы его использовали, чтобы подчеркнуть то обстоятельство, что в Perl любое правильное выражение может быть использовано в качестве выражения условия, которое вычисляется в булевом контексте.

Вторая форма оператора `if` используется, когда необходимо выполнить одну группу операторов (**БЛОК1**) в случае истинности некоторого выражения (**ВЫРАЖЕНИЕ**), а в случае его ложности — другую группу операторов (**БЛОК2**):

```
if ($var >= 0)                                # ВЫРАЖЕНИЕ  
{  
    print "Переменная неотрицательна.";      # БЛОК1, если ВЫРАЖЕНИЕ истинно  
} else {  
    print "Переменная отрицательна.";        # БЛОК2, если ВЫРАЖЕНИЕ ложно  
}
```

По существу, первая форма оператора `if` эквивалентна второй форме, если **БЛОК2** не содержит ни одного оператора.

Последняя, третья форма оператора `if` реализует цепочку ветвлений. Семантика этого оператора такова. Выполняются операторы из **БЛОК1**, если истинно **ВЫРАЖЕНИЕ1**. Если оно ложно, то выполняются операторы из **БЛОК2** в случае истинности **ВЫРАЖЕНИЕ2**. Если и оно ложно, то проверяется **ВЫРАЖЕНИЕ3** и т. д. Если ни одно из выражений условия оператора `if` не истинно, то выполняются операторы блока, определяемого после ключевого слова `else` в случае его наличия. В противном случае выполняется следующий после оператора `if` оператор программы. При выполнении следующего оператора ветвления `if`

```
if( $var < 0) {                                # ВЫРАЖЕНИЕ1
    print "Переменная отрицательна"; # БЛОК1
} elsif ( $var == 0) {                        # ВЫРАЖЕНИЕ2
    print "Переменная равна нулю";   # БЛОК2
} else {
    print "Переменная положительна"; # БЛОК3
}
```

сначала проверяется условие отрицательности переменной `$var`. Если значение переменной строго меньше нуля (ВЫРАЖЕНИЕ1), то печатается сообщение из БЛОК1 и оператор завершает свою работу. Если значение переменной не меньше нуля, то оно проверяется на равенство (ВЫРАЖЕНИЕ2) и в случае истинности выполняется оператор печати из блока операторов `elsif` (БЛОК2). Если проверка на равенство нулю дала ложный результат, то выполняется оператор печати из блока операторов `else` (БЛОК3).

Замечание

Ключевое слово `else` вместе со своим блоком операторов может быть опущено.

В операторе `if` со множественными проверками может быть сколько угодно блоков `elsif`, но только один блок `else`.

Так как все операторы ветвления определяются в терминах блоков операторов, то не возникает двусмысленности при определении, какие операторы в какой части выполняются.

При работе с операторами ветвления важно помнить, что только один блок операторов будет выполнен — тот, для которого истинно соответствующее выражение условия.

Во всех операторах ветвления ключевое слово `if` может быть заменено на `unless`. В этом случае проверка выражения условия осуществляется на его ложность. Последний оператор `if` можно записать и так:

```
unless( $var >= 0) {                            # ВЫРАЖЕНИЕ1
    print "Переменная отрицательна"; # БЛОК1
} elsif ( $var == 0) {                        # ВЫРАЖЕНИЕ2
    print "Переменная равна нулю";   # БЛОК2
} else {
    print "Переменная положительна"; # БЛОК3
}
```

При этом нам пришлось заменить ВЫРАЖЕНИЕ1 на противоположное по смыслу.

Замечание

Все операторы `if` (`unless`) могут быть вложенными, т. е. в любом их блоке можно свободно использовать другие операторы ветвления.

5.4. Операторы цикла

Известно, что для реализации любого алгоритма достаточно трех структур управления: последовательного выполнения, ветвления по условию и цикла с предусловием. Любой язык программирования предоставляет в распоряжение программиста набор всех трех управляющих конструкций, дополняя их для удобства программирования другими конструкциями: цепочки ветвления и разнообразные формы цикла с предусловием, а также циклы с постусловием.

Мы уже познакомились с операторами ветвления Perl, а теперь пришло время узнать, какие конструкции цикла можно применять в Perl. Их всего три: `while`, `for` и `foreach`. Все они относятся к классу *составных* операторов и, естественно, определяются в терминах блоков БЛОК.

5.4.1. Циклы *while* и *until*

Цикл `while` предназначен для повторного вычисления блока операторов, пока остается истинным задаваемое в нем выражение-условие. Его общий синтаксис имеет две формы:

```
МЕТКА while (ВЫРАЖЕНИЕ) БЛОК
```

```
МЕТКА while (ВЫРАЖЕНИЕ) БЛОК continue БЛОК1
```

Все операторы цикла могут быть снабжены не обязательными метками. В Perl метка представляет правильный идентификатор, завершающийся двоеточием ":". Она важна для команды перехода `next`, о которой мы поговорим в следующем разделе.

Оператор `while` выполняется по следующей схеме. Вычисляется выражение-условия ВЫРАЖЕНИЕ. Если оно истинно, то выполняются операторы БЛОК. В противном случае оператор цикла завершает свою работу и передает управление следующему после него оператору программы (цикл 1 примера 5.5). Таким образом, оператор цикла `while` является управляющей конструкцией цикла с предусловием: сначала проверяется условие завершения цикла, а потом только тело цикла, определяемое операторами БЛОК. Поэтому может оказаться, что тело цикла не будет выполнено ни одного раза, если при первом вхождении в цикл условие окажется ложным (цикл 3 примера 5.5).

Вместо ключевого слова `while` можно использовать ключевое слово `until`. В этом случае управляющая конструкция называется циклом `until`, который отличается от разобранного цикла `while` тем, что его тело выполняется, только если выражение условия *ложно* (цикл 2 примера 5.5).

Пример 5.5. Операторы цикла while и until

```
# perl -w
# цикл 1
$i = 1;
while ($i <= 3) {
    $a[$i] = 1/$i;          # Присвоить значение элементу массива
    ++$i;
}
print "Переменная цикла $i = $i\n"; # $i = 4
print "Массив \@a: @a\n";          # @a = (1, 0.5, 0.3333333333333333)

# цикл 2, эквивалентный предыдущему
$i = 1;
until ($i > 3) {
    $a[$i] = 1/$i;          # Присвоить значение элементу массива
    ++$i;
}
print "Переменная цикла $i = $i\n"; # $i = 4
print "Массив \@a: @a\n";          # @a = (1, 0.5, 0.3333333333333333)

# цикл 3, тело цикла не выполняется ни одного раза
$i = 5;
while ($i <= 3) {
    $a[$i] = 1/$i;
    ++$i;
}
print "Переменная цикла $i = $i\n"; # $i = 5

# цикл 4, бесконечный цикл (не изменяется выражение условия)
$i = 1;
while ($i <= 3) {
    $a[$i] = 1/$i;
}
```

Обратим внимание на то, что в теле цикла должны присутствовать операторы, вычисление которых приводит к изменению выражения условия. Обычно это операторы, изменяющие значения переменных, используемых в вы-

ражении условия. Если этого не происходит, то цикл `while` или `until` будет выполняться бесконечно (цикл 4 примера 5.5).

Замечание

Цикл с постусловием реализуется применением модификатора `while` к конструкции `do{}`, и рассматривался нами в разделе 5.2.2 "Модификаторы `while` и `until`".

Блок операторов `блок1`, задаваемый после ключевого слова `continue`, выполняется всякий раз, когда осуществляется переход на выполнение новой итерации. Это происходит после выполнения последнего оператора тела цикла или при явном переходе на следующую итерацию цикла командой `next`. Блок `continue` на практике используется редко, но с его помощью можно строго определить цикл `for` через оператор цикла `while`.

Пример 5.6 демонстрирует использование цикла `while` для вычисления степеней двойки не выше шестнадцатой. В этом примере оператор цикла `while` функционально эквивалентен циклу `for`. Блок `continue` выполняется всякий раз по завершении очередной итерации цикла, увеличивая переменную `$i` на единицу. Он эквивалентен выражению увеличения/уменьшения оператора `for` (см. следующий раздел).

Пример 5.6. Цикл `while` с блоком `continue`

```
# perl -w
# Вычисление степеней числа 2
$i = 1;
while ($i <= 16) {
    print "2 в степени $i: ", 2**$i, "\n";
} continue {
    ++$i;    # Увеличение переменной $i перед выполнением следующей итерации
}
```

5.4.2. Цикл `for`

При выполнении циклов `while` и `until` заранее не известно, сколько итераций необходимо выполнить. Их количество зависит от многих факторов: значений переменных в выражении условия до начала выполнения цикла, их изменении в теле цикла, виде самого выражения условия и т. п. Но иногда в программе необходимо выполнить заранее известное количество повторов определенной группы операторов. Например, прочитать из файла 5 строк и видоизменить их по определенным правилам. Конечно, можно такую задачу запрограммировать операторами цикла `while` и `until`,

но это может выглядеть не совсем выразительно. В том смысле, что при прочтении программы придется немного "пошевелить" мозгами, прежде чем понять смысл оператора цикла. Для решения подобных задач с заранее известным числом повторений язык Perl предлагает специальную конструкцию цикла — цикл `for`:

МЕТКА `for` (ВЫРАЖЕНИЕ1; ВЫРАЖЕНИЕ2; ВЫРАЖЕНИЕ3) БЛОК

ВЫРАЖЕНИЕ1 используется для установки начальных значений переменных, управляющих циклом, поэтому его называют *инициализирующим* выражением. Обычно это одна или несколько операций присваивания, разделенных запятыми.

ВЫРАЖЕНИЕ2 определяет условие, при котором будут повторяться итерации цикла. Оно, как и выражение-условие цикла `while`, должно быть истинным, чтобы началась следующая итерация. Как только это выражение становится ложным, цикл `for` прекращает выполняться и передает управление следующему за ним в программе оператору.

ВЫРАЖЕНИЕ3 отвечает за увеличение/уменьшение значений переменных цикла после завершения очередной итерации. Обычно оно представляет собой список выражений с побочным эффектом или список операций присваивания переменным цикла новых значений. Его иногда называют *изменяющим* выражением.

Алгоритм выполнения цикла `for` следующий:

1. Вычисляется инициализирующее выражение (ВЫРАЖЕНИЕ1).
2. Вычисляется выражение условия (ВЫРАЖЕНИЕ2). Если оно истинно, то выполняются операторы блока БЛОК, иначе цикл завершает свое выполнение.
3. После выполнения очередной итерации вычисляется выражение увеличения/уменьшения (ВЫРАЖЕНИЕ3) и повторяется пункт 2.

Как отмечалось в предыдущем разделе, цикл `for` эквивалентен циклу `while` с блоком `continue`. Например, следующий цикл

```
for ($i = 1; $i <= 10; $i++) {  
    . . .  
}
```

эквивалентен циклу `while`

```
$i = 1;  
while ($i <= 10) {  
    . . .  
} continue {  
    $i++;  
}
```

Существует единственное отличие между этими двумя циклами. Цикл `for` определяет лексическую область видимости для переменной цикла. Это позволяет использовать в качестве переменных цикла локальные переменные, объявленные с помощью функции `my`:

```
$i = "global";
for (my $i = 1; $i <= 3; $i++) {
    print "Внутри цикла \ $i: $i\n";
}
print "Вне цикла \ $i: $i\n ";
```

При выполнении этого фрагмента программы оператор печати будет последовательно отображать значения 1, 2 и 3 локальной переменной цикла `$i`. При выходе из цикла локальная переменная `$i` будет уничтожена и оператор печати вне цикла напечатает строку `global` — значение глобальной переменной `$i`, определенной вне цикла `for`.

Все три выражения цикла `for` являются необязательными и могут быть опущены, но соответствующие разделители ";" должны быть оставлены. Если опущено выражение условия, то по умолчанию оно принимается равным Истина. Это позволяет организовать бесконечный цикл:

```
for (;;) {
    . . .
}
```

Выход из такого цикла осуществляется командами управления, о которых речь пойдет в следующем параграфе.

Инициализировать переменную цикла можно и вне цикла, а изменять значение переменной цикла можно и внутри тела цикла. В этом случае инициализирующее и изменяющее выражения не обязательны:

```
$i = 1;
for (; $i <= 3;) {
    . . .
    $i++;
}
```

Совет

Хотя существует возможность изменения переменной цикла в теле цикла, не рекомендуется ею пользоваться. Цикл `for` был введен в язык именно для того, чтобы собрать в одном месте все операторы, управляющие работой цикла, что позволяет достаточно быстро изменить его поведение.

Цикл `for` позволяет использовать несколько переменных для управления работой цикла. В этом случае в инициализирующем и изменяющем выра-

жениях используется операция запятой. Например, если мы хотим создать хеш, в котором ключам, представляющим цифры от 1 до 9, соответствуют значения этих же цифр в обратном порядке от 9 до 1, то эту задачу можно решить с помощью цикла `for` с двумя переменными цикла:

```
for ($j = 1, $k = 9; $k > 0; $j++, $k--) {  
    $hash{$j} = $k;  
}
```

Этот же пример показывает, что в цикле `for` переменная цикла может как увеличиваться, так и уменьшаться. Главное, чтобы выражение условия правильно отслеживало условия продолжения итераций цикла.

Цикл `for` достаточно гибкая конструкция, которую можно использовать не только для реализации цикла с заранее заданным числом итераций. Он позволяет в инициализирующем и изменяющем выражениях использовать вызовы встроенных и пользовательских функций, а не только определять и изменять переменные цикла. Основное — чтобы изменялось выражение условия завершения цикла. Пример 5.7 демонстрирует именно такое использование цикла `for`.

Пример 5.7. Ввод строк циклом `for`

```
# perl -w  
for (print "Введите данные, для завершения ввода нажмите <Enter>\n";  
     <STDIN>;  
     print "Введите данные, для завершения ввода нажмите <Enter>\n")  
{  
    last if $_ eq "\n";  
    print "Ввели строку: $_";  
}
```

В этом примере пользователь вводит в цикле строки данных. Перед вводом новой строки отображается подсказка с помощью функции `print()`, которая определена в изменяющем выражении цикла. Выражение условия представляет операцию ввода из файла стандартного ввода `<STDIN>`. Так как это выражение вычисляется всякий раз, когда цикл переходит на очередную итерацию, то на каждом шаге цикла программа будет ожидать ввода с клавиатуры. Выход из цикла осуществляется командой `last`, вызываемой в случае ввода пользователем пустой строки. Введенные данные сохраняются во встроенной переменной `$_`, причем в ней сохраняется и символ перехода на новую строку, являющийся признаком завершения операции ввода данных. Поэтому при вводе пустой строки на самом деле в переменной `$_` хранится

управляющая последовательность "\n", с которой и осуществляется сравнение для реализации выхода из цикла.

Пример 5.8 демонстрирует программу, читающую 3 строки файла err.err. Операция чтения из файла задается в инициализирующем и изменяющем выражении вместе с определением и изменением переменной цикла \$i.

Пример 5.8. Чтение заданного числа строк из файла

```
# perl -w
open (FF, "err.err") or die "Ошибка открытия файла";
for ($line=<FF>, $count = 1; $count <=3; $line=<FF>, $count++)
{
    print "Строка $count:\n $line\n";
}
close(FILE);
```

5.4.3. Цикл *foreach*

Одно из наиболее частых применений циклов в языках программирования — организация вычислений с элементами массивов: найти максимальный элемент, распечатать элементы массива, выяснить, существует ли элемент массива, равный заданному значению, и т. п. Подобные задачи легко решаются с помощью циклов while и for. В примере 5.9 определяется максимальный элемент массива (в предположении, что он содержит числовые данные).

Пример 5.9. Определение максимального элемента массива циклом for

```
#!/ perl -w
@array = (1,-6,9,18,0,-10);
$max = $array[0];
for ($i = 1; $i <= $#array; $i++) {
    $max = $array[$i] if $array[$i] > $max;
}
```

После выполнения программы примера 5.9 переменная \$max будет иметь значение 18, равное максимальному элементу массива \$array. Обратим внимание читателя на то, что в цикле for (как и в цикле while) доступ к элементам массива организуется с помощью индекса.

В Perl списки и массивы, являющиеся, по существу, также списками, являются столь полезными и часто используемыми конструкциями, что для ор-

организации цикла по их элементам в языке предусмотрен специальный оператор `foreach`, имеющий следующий синтаксис:

МЕТКА `foreach` ПЕРЕМЕННАЯ (СПИСОК) БЛОК

МЕТКА `foreach` ПЕРЕМЕННАЯ (СПИСОК) БЛОК `continue` БЛОК

Он реализует цикл по элементам списка `СПИСОК`, присваивая на каждом шаге цикла переменной `ПЕРЕМЕННАЯ` значение выбранного элемента списка. Блок операторов `continue` выполняется всякий раз, как начинается очередная итерация, за исключением первой итерации, когда переменная `$temp` равна первому элементу списка. Список можно задавать или последовательностью значений, разделенных запятыми, или массивом скаляров, или функцией, возвращаемым значением которой является список. Определение максимального элемента массива можно переписать с циклом `foreach` (пример 5.10).

Пример 5.10. Определение максимального элемента массива циклом `foreach`

```
#!/ perl -w
@array = (1,-6,9,18,0,-10);
$max = $array[0];
foreach $temp (@array) {
    $max = $temp if $temp > $max;
}
print "$max";
```

На каждом шаге цикла переменная `$temp` последовательно принимает значения элементов массива `$array`. Обратите внимание на внешний вид программы — в отсутствии индексов массива она стала лучше читаемой.

Отметим несколько особенностей цикла `foreach`. Прежде всего следует сказать, что ключевое слово `foreach` является синонимом ключевого слова `for`. Цикл из примера 5.10 можно было бы записать и так:

```
for $temp (@array) {                                # Ключевое слово foreach синоним for.
    $max = $temp if $temp > $max;
}
```

Однако, как нам кажется, использование `foreach` лучше отражает семантику этого оператора цикла, так как в самом ключевом слове уже отражена его сущность (`for each` — для каждого).

Следующая особенность оператора `foreach` связана с переменной цикла. По умолчанию эта переменная является локальной, область видимости которой ограничена телом цикла. Она создается только на время выполнения оператора `foreach`, доступна внутри тела цикла и уничтожается при выходе из цикла.

Обычно программисты, работающие на языке Perl, вообще не применяют в циклах `foreach` переменную цикла. Это связано с тем обстоятельством, что в отсутствии явно заданной переменной цикла Perl по умолчанию использует специальную переменную `$_`. На каждом шаге цикла именно она будет содержать значение элемента списка или массива. С учетом этого факта цикл `foreach` примера 5.10 можно переписать так:

```
foreach (@array) {                # В качестве переменной цикла используется $_.
    $max = $_ if $_ > $max;
}
```

Последняя особенность оператора `foreach`, которая также связана с переменной цикла, заключается в том, что фактически на каждом шаге выполнения цикла эта переменная является синонимом того элемента списка, значение которого она содержит. Это означает, что ее изменение в цикле приводит к изменению значения соответствующего элемента списка. Это свойство цикла `foreach` удобно для изменения значений элементов списка. Отметим, что его можно применять к спискам, хранящимся в массивах. Например, возвести в квадрат каждый элемент списка можно следующим оператором `foreach`:

```
foreach $temp (@array) {
    $temp **= 2;
}
```

Список, по элементам которого организуется цикл, может быть задан не только явно конструктором или переменной массива, но и функцией, возвращаемым значением которой является список. Канонический способ печати хеш-массива в упорядоченном порядке представлен в примере 5.11.

Пример 5.11. Упорядочение и печать хеш-массива

```
# perl -w
%array = (
    blue   => 1,
    red    => 2,
    green  => 3,
    yellow => 3
);

foreach (sort keys %array) {
    print "$_\t => $array{$_}\n";
}
```

Эта программа напечатает пары ключ/значение хеш-массива `%array` в соответствии с алфавитным порядком его ключей:

```
blue    => 1
green   => 3
red     => 2
yellow  => 3
```

Замечание

Цикл `foreach` выполняется быстрее аналогичного цикла `for`, так как не требует дополнительных затрат на вычисление индекса элемента списка.

5.5. Команды управления циклом

Каждый цикл в программе завершается при достижении некоторого условия, определяемого самим оператором. В циклах `while` и `for` это связано с ложностью выражения-условия, а в цикле `foreach` с окончанием перебора всех элементов списка. Иногда возникает необходимость при возникновении некоторых условий завершить выполнение всего цикла, либо прервать выполнение операторов цикла и перейти на очередную итерацию. Для подобных целей в языке Perl предусмотрены три команды `last`, `next` и `redo`, которые и называют *командами управления циклом*.

Синтаксис этих команд прост — ключевое слово, за которым может следовать необязательный идентификатор метки:

`last ИДЕНТИФИКАТОР_МЕТКИ;`

`next ИДЕНТИФИКАТОР_МЕТКИ;`

`redo ИДЕНТИФИКАТОР_МЕТКИ;`

Семантика этих команд также проста. Они изменяют порядок выполнения циклов, принятый по умолчанию в языке, и передают управление в определенное место программы, завершая выполнение цикла (`last`), переходя на следующую итерацию цикла (`next`) или повторяя выполнение операторов тела цикла при тех же значениях переменных цикла (`redo`). Место перехода задается *меткой*. Помните синтаксис операторов цикла? Каждый из них может быть помечен. Именно идентификаторы меток операторов цикла и используются в командах управления для указания места передачи управления.

Внимание

Метка в программе Perl задается идентификатором, за которым следует двоеточие. В командах управления циклом используется именно *идентификатор метки*, а не *метка*.

Несколько слов о терминологии. Читатель, наверное, обратил внимание, что мы не называем команды управления циклом операторами. И это справедливо. Они не являются операторами, хотя могут использоваться как операторы. Их следует считать *унарными операциями*, результатом вычисления которых является изменение последовательности выполнения операторов. Поэтому команды управления циклом можно использовать в любом выражении Perl. Заметим, что их следует использовать в таких выражениях, где имеет смысл их использовать, например в выражениях с операцией "запятая":

```
open (INPUT_FILE, $file)
    or warn ("Невозможно открыть $file: $!\n"), next FILE;
```

Приведенный оператор может являться частью программы, которая в цикле последовательно открывает и обрабатывает файлы. Команда `next` инициирует очередную итерацию цикла с меткой `FILE`, если не удалось открыть файл в текущей итерации. Обратите внимание, что она используется в качестве операнда операции "запятая". В таком контексте эта команда имеет смысл. Следующий оператор является синтаксически правильным, но использование в нем команды `redo` не имеет никакого смысла:

```
print "qu-qu", 5 * redo OUT, "hi-hi\n";
```

Результатом выполнения этого оператора будет повторение вычислений операторов цикла с меткой `OUT`, т. е. простое выполнение команды `redo OUT`.

Относительно команд управления циклом следует сказать, что к ним можно применять модификаторы, так как употребленные самостоятельно с завершающей точкой с запятой они рассматриваются как *простые* операторы:

```
next if $a == 2;
```

Переход на следующую итерацию цикла осуществится только, если переменная `$a` равна 2.

5.5.1. Команда *last*

Команда `last` немедленно прекращает выполнение цикла, в котором она задана, и передает управление на оператор, непосредственно следующий за оператором цикла. Ее целесообразно использовать для нахождения одного определенного значения в массиве (пример 5.12).

Пример 5.12. Выход из цикла командой *last*

```
#!/ perl -w
@letters = ("A".."Z");
for ($index=0; $index<@letters; $index++) {
```



```

    last if $letters[$index] eq "M";
}
print $index;

```

Цикл в программе примера 5.12 будет выполняться, пока перебор элементов массива `$letters` не достигнет элемента, содержащего символ "M". После чего будет выполнен первый после оператора `for` оператор программы. В результате будет напечатано число 12 — индекс элемента, содержащего символ "M".

Метка используется для конкретизации передачи управления в случае вложенных циклов: управление передается непосредственно на оператор, следующий за оператором цикла с указанной меткой (пример 5.13).

Пример 5.13. Команда `last` с меткой

```

CYCLE_1: while (...) {

    CYCLE_2: for(...) {

        CYCLE_3: foreach (...) {
            last CYCLE_2;
        }
        Операторы цикла CYCLE_2
    }
    Операторы цикла CYCLE_1    # Сюда передает управление
                               # оператор last CYCLE_2;
}

```

Если в команде `last` указать метку `CYCLE_1`, то управление будет передано на первый после самого внешнего цикла оператор программы. Если в команде `last` задать метку `CYCLE_3` (или задать ее вообще без метки), то управление будет передано на первый оператор группы Операторы цикла `CYCLE_2`.

Внимание

Передача управления командой `last` осуществляется не на оператор цикла с соответствующей меткой, а на оператор, непосредственно следующий за ним.

Команда `last` осуществляет выход из цикла, не выполняя никаких блоков операторов `continue`.

5.5.2. Команда *next*

Команда *next* позволяет пропустить расположенные после нее в теле цикла операторы и перейти на следующую итерацию цикла. Если оператор цикла содержит блок *continue*, то его операторы выполняются до проверки условия окончания цикла, с которой начинается следующая итерация. Одно из применений этой команды — обработать определенные элементы массива, ничего не делая с другими. Программа примера 5.14 присваивает всем элементам массива, содержащим четные числа, символ звездочка "*".

Пример 5.14. Использование команды *next*

```
#!/ perl -w
@array = (2, 5, 8, 4, 7, 9);
print "До: @array\n";
foreach (@array) {
    next if $_ % 2;
    $_ = "*";
}
print "После: @array\n";
```

Результат выполнения программы примера 5.14 показан ниже:

```
До:      2 5 8 4 7 9
После:   * 5 * * 7 9
```

Если элемент массива нечетное число, то результат операции `$_ % 2` равен 1 (Истина) и команда *next* инициирует следующую итерацию цикла *foreach*, не изменяя значение текущего элемента массива. Если значением элемента массива является четное число, то команда *next* не выполняется и значение элемента меняется на символ "*".

Команда *next*, употребленная совместно с идентификатором метки, прерывает выполнение цикла, в теле которого она находится, и начинает новую итерацию цикла с указанной меткой, выполнив предварительно его блок *continue*, если таковой имеется (пример 5.15).

Пример 5.15. Команда *next* с меткой

```
#!/ perl -w
$out = 0;
OUT: while ($out < 2) {
    print "Начало внешнего цикла\n";
    for($in=0; $in<=2; $in++) {
```

```

print "\$out: $out\t\$in: $in\n";
next OUT if $in ==1;
}

print "\$out: $out\n";    # Никогда не выполняется!
} continue {
print "Блок continue внешнего цикла\n";
$out++;
}

```

Вывод этой программы будет следующим:

```

Начало внешнего цикла
$out: 0 $in: 0
$out: 0 $in: 1
Блок continue внешнего цикла
Начало внешнего цикла
$out: 1 $in: 0
$out: 1 $in: 1
Блок continue внешнего цикла

```

Обратите внимание, что количество итераций внутреннего цикла `for` равно двум, так как на второй его итерации выполняется команда `next OUT`, прекращающая его выполнение и инициализирующая выполнение очередной итерации внешнего цикла `OUT`. Оператор печати этого цикла пропускается, выполняется блок операторов `continue`, проверяется условие и если оно истинно, то тело цикла выполняется. Таким образом, оператор печати внешнего цикла `OUT` не выполняется ни одного раза, что подтверждается приведенным выводом из программы примера 5.15.

5.5.3. Команда *redo*

Команда `redo` повторно выполняет операторы тела цикла, не инициализируя следующую итерацию. Это означает, что ни выражение изменения цикла `for`, ни операторы блока `continue`, если он присутствует, ни выражение условия не вычисляются. Операторы тела цикла, расположенные за оператором `redo`, пропускаются и снова начинается выполнение тела цикла со значениями переменных, которые они имели перед выполнением этой передачи управления. Программа примера 5.16 демонстрирует использование команды `redo`.

Пример 5.16. Команда *redo* в цикле

```

#! perl -w
$notempty = 0;

```

```

$total = 0;
for (;;) {                                # Бесконечный цикл
    $line=<STDIN>;                          # Ввод строки
    chop($line);
    last if $line eq "END"; # Выход из цикла
    ++$total;
    redo if $line eq "";    # Возврат на чтение строки
    ++$notempty;
}
print "Всего прочитано строк: $total\nИз них не пустых: $notempty\n";

```

Эта программа в бесконечном цикле ожидает ввода пользователем на клавиатуре строки данных и в переменной `$total` подсчитывает количество введенных строк. В переменной `$notempty` вычисляется количество введенных не пустых строк. Если введена пустая строка, то команда `redo` начинает повторное выполнение операторов тела цикла, не увеличивая на единицу переменную `$notempty`. Для завершения бесконечного цикла следует ввести строку `END`. В этом случае выполняется команда `last`.

Функция `chop` используется для удаления из введенной пользователем строки символа перехода на новую строку `"\n"`, поэтому в программе она сравнивается со строками без завершающего символа перехода на новую строку (сравни с примером 5.7).

Если команда `redo` используется с идентификатором метки, то ее действие аналогично действию команды `next` с той лишь разницей, что она просто передает управление на первый оператор тела цикла с указанной меткой, не иницилируя следующей итерации и не вычисляя операторов блока `continue`. В качестве иллюстрации такого использования команды `redo` перепишем программу примера 5.16 следующим образом:

Пример 5.17. Команда `redo` с идентификатором метки

```

#!/perl -w
$notempty = 0;
$total = 0;
OUT: while (1) {
    print "Введи строки\n";    # Сюда передает управление команда redo OUT;
    for (;;) {
        $line=<STDIN>;
        chop($line);

```

```
last OUT if $line eq "END";      # Выход из всех циклов
++$total;
redo OUT if $line eq "";
++$notempty;
}
}
print "Всего прочитано строк: $total\nИз них не пустых: $notempty\n";
```

В примере 5.17 мы ввели внешний бесконечный цикл `OUT` и изменили команды `redo` и `last`, добавив к ним метку на внешний цикл. Теперь в случае, если пользователь вводит пустую строку, команда `redo OUT` передает управление на первый оператор внешнего цикла, и программа печатает приглашение ввести строки.

5.6. Именованные блоки

В Perl блок операторов, заключенный в фигурные скобки, семантически эквивалентен циклу, выполняющемуся *только* один раз. В связи с этим обстоятельством можно использовать команду `last` для выхода из него, а команду `redo` для повторного вычисления операторов блока. Команда `next` также осуществляет выход из блока, но отличается от команды `last` тем, что вычисляются операторы блока `continue`, который может задаваться для блока операторов в фигурных скобках:

```
BLOCK1: {
    $i = 1;
    last BLOCK1;
} continue {
    ++$i;
}
print "Переменная \$$i после BLOCK1: $$i\n";

BLOCK2: {
    $i = 1;
    next BLOCK2;
} continue {
    ++$i;
}
print "Переменная \$$i после BLOCK2: $$i\n";
```

Первый оператор `print` этого фрагмента кода напечатает значение переменной `$i` равным 1, тогда как второй оператор `print` напечатает 2, так как при

выходе из блока BLOCK2 будет выполнен оператор увеличения на единицу переменной \$i из блока continue.

Замечание

Если в простом блоке операторов задан блок continue, то при нормальном завершении простого блока (без использования команд управления циклом) блок операторов continue также будет выполнен.

Блоки могут иметь метки, и в этом случае их называют *именованными блоками*. Подобные конструкции используются для реализации переключателей — конструкций, которые не определены в синтаксисе языка Perl. Существует множество способов создания переключателей средствами языка Perl. Один из них представлен в примере 5.18.

Пример 5.18. Реализация переключателя

```
#!/ perl -w
$var = 3;
SWITCH: {
    $case1 = 1, last SWITCH if $var == 1;
    $case2 = 1, last SWITCH if $var == 2;
    $case3 = 1, last SWITCH if $var == 3;
    $nothing = 1;
}
```

После выполнения именованного блока операторов SWITCH переменная \$case1 будет равна 1, если \$var равна 1, \$case2 будет равна 2, если \$var равна 2 и, наконец, \$case3 будет равна 3, если \$var равна 3. В случае, если переменная \$var не равна ни одному из перечисленных значений, то переменная \$nothing будет равна 1. Конечно, это простейший переключатель, разработанный всего лишь для демонстрации возможности быстрого создания переключателя в Perl. Для выполнения группы операторов в переключателе можно использовать не модификатор if, а оператор выбора if.

Блоки могут вложенными друг в друга. Именованные блоки и команды управления циклом, используемые для выхода из внутренних блоков, позволяют создавать достаточно прозрачные конструкции, реализующие сложные алгоритмы. Например, можно организовать бесконечный цикл без использования какого-либо оператора цикла:

```
$notempty = 0;
$total = 0;
INPUT: {
```

```
$line=<STDIN>;
chop($line);
last INPUT if $line eq "END";      # Выход из бесконечного цикла
++$total;
redo INPUT if $line eq "";
++$notempty;
redo INPUT;
}
```

Узнаете программу примера 5.16? Действительно, это реализация без оператора цикла программы ввода строк и подсчета общего числа введенных, а также непустых строк. Единственное, что нам пришлось добавить — еще одну команду `redo` в конце блока операторов.

5.7. Оператор безусловного перехода

Оператор безусловного перехода `goto`, возможно, самый спорный оператор. Много копий было поломано в дебатах о его целесообразности и полезности. Однако практически в любом языке программирования можно обнаружить оператор безусловного перехода. Не является исключением и язык Perl. В нем есть три формы этого оператора:

```
goto МЕТКА;
goto ВЫРАЖЕНИЕ;
goto &ПОДПРОГРАММА;
```

Первая форма `goto МЕТКА` передает управление на оператор с меткой `МЕТКА`, который может быть расположен в любом месте программы, за исключением конструкций, требующих определенных иницилирующих действий перед их выполнением. К ним относятся цикл `foreach` и определение подпрограммы `sub`.

Замечание

Компилятор Perl не генерирует никаких ошибок, если в операторе `goto` задана не существующая метка, или он передает управление в конструкцию `foreach` или `sub`. Все ошибки, связанные с этим оператором, возникают во время выполнения программы.

Во второй форме оператора безусловного перехода `goto ВЫРАЖЕНИЕ` возвращаемым значением выражения должен быть метка, на которую и будет передано управление в программе. Эта форма оператора `goto` является аналогом вычисляемого `goto` языка FORTRAN:

```
@label = ("OUT", "IN");  
goto $label[1];
```

В приведенном фрагменте кода выражение в операторе `goto` будет вычислено равным строке `IN` и именно на оператор с этой меткой будет передано управление.

Последняя форма оператора `goto` «ПОДПРОГРАММА» обладает магическим свойством, как отмечают авторы языка. Она подставляет вызов указанной в операторе подпрограммы для выполняемой в данный момент подпрограммы. Эта процедура осуществляется подпрограммами `AUTOLOAD()`, которые загружают одну подпрограмму, скрывая затем, что на самом деле сначала была вызвана другая подпрограмма.

Описание оператора `goto` приведено нами исключительно для полноты изложения. В программах его следует избегать, так как он делает логику программы более сложной и запутанной. Намного лучше использовать структурированные команды управления потоком вычислений `next`, `last` и `redo`. Если в процессе программирования выяснится, что не обойтись без оператора безусловного перехода, то это будет означать только одно: на этапе проектирования программы она была не достаточно хорошо структурирована. Вернитесь снова к этапу проектирования и постарайтесь реструктурировать ее таким образом, чтобы не требовалось использовать оператор `goto`.

В этом разделе мы познакомились с основными операторами языка Perl, которые используются для написания программ. Узнали, что операторы могут быть простыми и составными.

Выполнением простых операторов можно управлять с помощью модификаторов, которые вычисляют простой оператор при выполнении некоторого условия. Некоторые модификаторы организуют повторное вычисление в цикле оператора, к которому они применены.

Составные операторы представляют собой операторы, управляющие потоком вычислений в программе. Они определяются в терминах блоков. К ним относятся операторы выбора и цикла. Команды управления циклом позволяют изменить порядок выполнения операторов цикла.

Вопросы для самоконтроля

1. Как определяются простые операторы Perl?
2. Что такое модификаторы простых операторов и как они влияют на выполнение простых операторов?
3. Перечислите составные операторы языка Perl.
4. Что такое блок операторов и что он определяет в программе?

5. Определите лексическую переменную.
6. Какой оператор цикла удобнее для перебора всех элементов списка и почему?
7. Какие команды используются в Perl для управления выполнением циклов?
8. Как реализуются в Perl переключатели?

Упражнения

1. Какие из следующих операторов являются простыми, а какие составными:

```
"abc" if 1;
if ($a) { print $a; }
do{ $a++; $b--; } until $b;
while( $a eq "a") { $a--; }
```
2. Найдите ошибку в программе:

```
# perl -w
$a = "true";
$b = "false";
if ($a)
    $a = $b;
elsif ($b) $b == $a;
```
3. Напишите программу, которая по заданному числу STEP печатает лесенку из STEP ступеней (каждая следующая ступень на один символ "-" шире предыдущей):

```
-
|                               (первая ступень)
--
|                               (вторая ступень)
---
|                               (третья ступень)
. . . . .
```
4. Напишите программу, которая во вводимой пользователем строке подсчитывает количество слов, количество не пробельных символов и количество пробельных символов. Словом считать непрерывную последовательность алфавитно-цифровых символов, ограниченных пробельными символами ("\n", "\t", " "). Для завершения программы пользователь должен ввести пустую строку.

5. Напишите программу, которая читает целую величину ROW и печатает первые ROW строк треугольника Паскаля:

```
      1
    1 1
  1 2 1
1 3 3 1
1 4 6 4 1
. . . . .
```



Операции ввода/вывода

Ни одна программа не может функционировать сама по себе, не получая и не посылая информацию во внешнюю среду. Perl предоставляет несколько способов получения программой данных извне и вывода информации из выполняющегося сценария. В процессе функционирования программы может потребоваться выполнить некоторую команду операционной системы и проанализировать результаты ее выполнения, прочитать данные из внешнего файла или группы файлов, записать результаты вычислений во внешний файл или отобразить их на экране монитора — все эти действия реализуются разнообразными операциями и функциями языка Perl.

Простейшее взаимодействие с операционной системой, в которой выполняется программа Perl, реализуется операцией заключения строки данных в обратные кавычки. Содержимое такой строки передается на выполнение операционной системы, которая возвращает результат выполнения команды в эту же строку.

Для чтения из файла используется операция "ромб" `<>`, которой в качестве операнда передается дескриптор файла. В этой главе мы не будем обсуждать ввод из файла через его дескриптор, отнеся рассмотрение этого вопроса в следующую главу, полностью посвященную работе с файлами. Здесь мы расскажем о том, как работает операция "ромб" в случае отсутствия операнда, представляющего дескриптор файла. В этом случае эта операция может читать записи из стандартного файла ввода `STDIN` или получать информацию, передаваемую программе через командную строку.

Для отображения в стандартный файл вывода `STDOUT` используется уже знакомая нам функция `print`, которая, однако, может выводить информацию и в файл, определенный своим дескриптором.

6.1. Операция ввода команды

Заключенная в обратные кавычки `" "` строка символов является всего лишь удобной формой записи операции ввода команды операционной системы `qx{ }`, с которой мы уже знакомы (см. главу 4).

Когда интерпретатор Perl встречает строковый литерал в обратных кавычках, он осуществляет подстановку в нее значений скалярных переменных и переменных массивов и передает получившуюся строку, как команду, на вы-

полнение операционной системе. Последняя выполняет ее и возвращает в строковый литерал результаты вывода команды на стандартное устройство вывода, которым обычно является экран монитора. В связи с таким "поведением" строкового литерала в обратных кавычках его иногда называют псевдолитералом.

Операция ввода команды различает скалярный и списковый контексты, в которых она может выполняться. В скалярном контексте возвращается одна строка, содержащая весь вывод на экран монитора выполненной команды, включая символы новой строки в случае многострочного вывода. В списке контексте возвращается список значений, каждое из которых содержит одну строку вывода. Пример 6.1 демонстрирует использование операции ввода команды в соответствующих контекстах.

Пример 6.1. Операция ввода команды в скалярном и списке контекстах

```
#!/perl -w
$command = "dir";
$scalar = `$command`;    # Скалярный контекст.
@list = ` $command`;     # Списковый контекст.
print $scalar;
print $list[0], $list[1];
```

При выполнении операции заключения в кавычки сначала осуществляется подстановка значения скалярной переменной `$command`, а потом полученная строка передается на выполнение операционной системы. Переменная `$scalar` (скалярный контекст) содержит весь вывод на экран монитора текущего текущего каталога, поэтому при ее печати мы увидим все, что вывела команда `dir`. Когда результаты выполнения команды присваиваются массиву `@list` (списковый контекст), то каждая строка вывода команды становится элементом массива, поэтому последний оператор печати примера 6.1 выводит первую и вторую строки.

В списке контексте разбиение вывода команды операционной системы на элементы списка осуществляется в соответствии со значением встроенной переменной `$/`, которое используется в качестве разделителя. По умолчанию эта переменная содержит символ конца строки `"\n"` — поэтому и разбиение на элементы происходит по строкам. Присваивая этой переменной новое значение, мы тем самым определяем новое значение разделителя, которое будет использоваться при формировании элементов списка. Разделителем может быть любая последовательность символов. Например, в примере 6.2 в качестве разделителя задается строка `"<КАТАЛОГ>"`.

Пример 6.2. Задание разделителя элементов списка

```
#!/ perl -w
$/ = "<КАТАЛОГ>";
@list = `dir`;      # Списковый контекст.
print $list[1], $list[2];
```

Теперь, в отличие от примера 6.1, элемент массива `$list[0]` содержит не первую строку вывода команды `dir`, а вывод команды до первой встретившейся в нем последовательности символов "<КАТАЛОГ>". Аналогично, элемент `$list[1]` содержит вывод команды до следующей встретившейся последовательности символов "<КАТАЛОГ>" и т. д.

Команда, содержащаяся в псевдолитерале, выполняется всякий раз, как вычисляется этот псевдолитерал. Встроенная переменная `$?` содержит числовое значение состояния выполненной команды.

(Об интерпретации значений встроенной переменной `$?` см. главу 14.)

Хотим обратить внимание читателя еще раз на тот факт, что операция ввода команды возвращает вывод на стандартное устройство вывода операционной системы. При выполнении команды можно направить ее вывод на другое устройство, например, в файл. Для этого в строке после имени команды и всех необходимых для ее выполнения параметров следует задать символ ">", после которого ввести имя файла. В этом случае на экран монитора ничего выводиться не будет, а следовательно и ничего не будет возвращаться в псевдолитерал, т. е. после выполнения такой команды псевдолитерал будет содержать неопределенное значение (пример 6.3).

Пример 6.3. Выполнение команды с перенаправлением вывода

```
#!/ perl -w
$/ = "<КАТАЛОГ>";
$list = `dir >file.dat`;      # Вывод осуществляется в файл file.dat
print $list;                  # Оператор ничего не напечатает!
```

Замечание

Обобщенная форма операции заключения в обратные кавычки `qx{}` работает точно так же, как и операция заключения в обратные кавычки `"`"`.

6.2. Операция <>

Для нашего читателя эта операция не является совсем уж новой. Несколько слов о ней было сказано в *главе 4*; в некоторых примерах мы ее использовали для ввода пользователем данных в программу Perl. Основное ее назначение — прочитать строку из файла, дескриптор которого является операндом этой операции. (Операнд операции <> расположен внутри угловых скобок.) Мы не будем сейчас объяснять, что такое дескриптор файла, зачем он нужен и какую функцию выполняет в программах Perl. Эти вопросы будут нами подробно рассмотрены в следующей главе, посвященной работе с файлами. Здесь же мы остановимся на специальном случае использования этой операции — операции с пустым операндом <>. В этом случае ввод осуществляется или из стандартного файла ввода, или из каждого файла, перечисленного в командной строке при запуске программы Perl. Но прежде чем перейти к описанию функционирования операции ввода с пустым операндом, остановимся на некоторых понятиях, необходимых для понимания дальнейшего.

Для обеспечения единообразия работы программ Perl в разных операционных системах при их запуске открывается несколько стандартных файлов. Один из них предназначен для ввода данных в программу и связан со стандартным устройством ввода — клавиатурой. Этот файл и называется стандартным файлом ввода и имеет дескриптор `STDIN`. Для вывода информации из программы создается стандартный файл вывода, также связанный со стандартным устройством вывода операционной системы, которым является экран монитора компьютера. Этому стандартному файлу назначается дескриптор `STDOUT`. Для отображения разнообразных сообщений о возникающих в процессе выполнения программы ошибках создается стандартный файл ошибок, который связан со стандартным устройством вывода. Этот файл имеет дескриптор `STDERR`. Эти файлы не надо создавать и открывать — они уже существуют, когда наша программа начинает выполняться. Иногда их называют предопределенными файлами ввода/вывода. Таким образом, если мы, например, говорим о том, что ввод осуществляется из стандартного файла (или стандартного устройства), мы имеем в виду стандартный файл ввода с дескриптором `STDIN`.

При запуске программы Perl в системе UNIX или из командной строки Windows ей можно передать параметры. Эти параметры задаются после имени файла, содержащего программу Perl, и отделяются от него и друг от друга пробелами:

```
perl program.pl par1 par2 par3
```

Параметрами могут быть ключи (обычно символ с лидирующим дефисом, например, `-v`), которые устанавливают определенные режимы работы программы, или имена файлов, содержимое которых должна обработать программа. Все передаваемые в программу параметры сохраняются в специаль-

ном встроенном массиве @ARGV. Если не передается ни одного параметра, то этот массив пуст.

Операция <> без операнда, употребленная в циклах while и for, при первом своем вычислении проверяет, пуст ли массив @ARGV. Если он пуст, то в первый элемент этого массива \$ARGV[0] заносится символ "-" и операция ожидает ввода пользователя из стандартного файла ввода STDIN. Если массив @ARGV не пуст, то он содержит параметры, переданные программе при ее запуске. Операция <> трактует каждый из них как имя файла и в цикле передает в программу последовательно все строки всех файлов, указанных в командной строке. Рассмотрим простейшую программу (пример 6.4), состоящую из одного цикла while с операцией <>, и рассмотрим ее поведение при разных способах запуска.

Пример 6.4. Применение оператора <> без операнда

```
#!/ perl -w
while ($line = <>) {
    print $line;
}
```

При ее запуске без параметров она будет ожидать ввода пользователя с клавиатуры и в цикле распечатывать вводимые им строки, пока пользователь не завершит ввод комбинацией клавиш <Ctrl>+<Z>, интерпретируемой как конец файла.

Если при запуске передать ей имя файла, например, файла, содержащего саму же программу,

```
perl examp6_4.pl examp6_4.pl
```

то программа примера 6.4 распечатает его содержимое:

```
#!/ perl -w
while ($line = <>) {
    print $line;
}
```

Замечание

Предполагается, что программа примера 6.4 сохранена в файле с именем examp6_4.pl.

Если эту же программу запустить, задав в командной строке дважды имя файла программы,

```
perl examp6_4.pl examp6_4.pl examp6_4.pl
```

то программа дважды распечатает свой собственный текст.

Замечание

В операционной системе Windows в именах файлов можно использовать пробелы. Для передачи в программу Perl файла с таким именем его следует заключать в двойные кавычки: "Name with blanks.dat".

При выполнении операции ввода из файла встроенная переменная `$.` на каждом шаге цикла хранит номер прочитанной строки файла. В случае задания нескольких имен файлов в командной строке при последовательном вводе их строк операцией `<>` эта переменная продолжает увеличивать свое значение при переходе на чтение строк очередного файла, т. е. она рассматривает содержимое всех файлов как один-единственный файл.

Операцию `<>` и массив `@ARGV` можно совместно использовать для ввода в программу содержимого нескольких файлов, не связывая их с заданием имен файлов в командной строке. В любом месте программы перед первым использованием в цикле операции ввода `<>` можно в массив `@ARGV` занести имена файлов, содержимое которых необходимо обработать:

```
@ARGV = ("file1.dat", "file2.dat", "file3.dat");
```

```
for (;<>); {
```

Операторы обработки строк файлов

```
}
```

Этот фрагмент программы в цикле `for` последовательно обработает строки трех файлов `file1.dat`, `file2.dat` и `file3.dat`. Здесь же продемонстрирована еще одна интересная особенность операции ввода `<>`. Обычно прочитанная этой операцией строка присваивается скалярной переменной, как это происходило в примере 6.4, но если эта операция одна представляет выражение условия цикла, то результат ее выполнения сохраняется в специальной встроенной переменной `$_`. Цикл `while` программы примера 6.4 можно записать и так:

```
while (<>) {
```

```
    print;
```

```
}
```

Здесь также используется то обстоятельство, что функция `print` без параметров по умолчанию выводит содержимое переменной `$_`.

Если мы хотим передать в программу некоторые ключи, устанавливающие режим ее работы, то в начале программы следует поместить цикл, который проверяет содержимое массива `@ARGV` на наличие ключей в командной строке вызова программы. Один из способов подобной проверки приводится в примере 6.5, где предполагается, что программе могут быть переданы ключи `-d`, `-s` и `-e`.

Пример 6.5. Проверка наличия ключей, переданных в программу

```
#!/perl -w
while ($_ = $ARGV[0], /^-/) {
    if(/^-d/) { print $ARGV[0],"\n"; }
    if(/^-s/) { print $ARGV[0],"\n"; }
    if(/^-e/) { print $ARGV[0],"\n"; }
    shift;
}
```

При вычислении выражения условия цикла `while` осуществляется присваивание переменной `$_` значения первого элемента массива `@ARGV` и проверка присутствия дефиса `"-"` в качестве первого символа содержимого этой переменной (операция `/^-/`). Операторы `if` проверяют содержимое переменной `$_` на соответствие известным ключам и отображают их. (В реальных программах в этих операторах обычно определяют некоторые переменные, которые в дальнейшем используются для выполнения действий, присущих соответствующим ключам.) Функция `shift` удаляет из массива `@ARGV` первое значение, сдвигая оставшиеся в нем элементы на одну позицию влево: второй становится первым, третий вторым и т. д. Цикл повторяется до тех пор, пока переданные через командную строку параметры начинаются с дефиса.

Еще одно применение операции `<>` связано с получением в программе имен файлов определенного каталога, удовлетворяющих заданному шаблону. Если в качестве операнда этой операции используется шаблон имен файлов, то в скалярном контексте она возвращает первое найденное имя файла в текущем каталоге, в списковом контексте — список имен файлов, удовлетворяющих заданному шаблону. (В шаблоне можно использовать метасимволы: `*` для произвольной цепочки символов, `?` для произвольного одиночного символа.) Если в каталоге не найдены файлы с именами, удовлетворяющими шаблону, то операция возвращает неопределенное значение. Например, выполнение следующей операции

```
$first = <*.pl>;
```

приведет к сохранению в переменной `$first` имени первого файла из списка всех файлов текущего каталога с расширением `pl`, если таковые файлы в каталоге есть, иначе эта переменная будет иметь неопределенное значение. В списке файлы упорядочены в алфавитном порядке.

Эта же операция в списковом контексте

```
@files = <*.pl>;
```

возвращает список всех файлов с расширением `pl`. После выполнения этой операции элементы массива `@files` содержат имена всех файлов с расширением `pl`.

Замечание

Имена подкаталогов текущего каталога считаются файлами без расширения. Например, в возвращаемом операцией `<*. *>` списке файлов будут содержаться и имена подкаталогов текущего каталога.

Если при задании шаблона файла явно указать каталог, то эта операция возвратит список файлов из указанного каталога, имена которых удовлетворяют заданному шаблону. Например, операция

```
@files = </perl/*.pl>;
```

сохранит в массиве `@files` имена всех файлов каталога `/perl` с расширением `pl`.

Замечание

В системе Windows эта операция найдет все файлы с расширением `pl` в каталоге `/perl` текущего диска. Для задания конкретного диска следует использовать принятый в Windows синтаксис для полного имени файла: `<d:/perl/*. *>`. Эта операция возвратит список всех файлов каталога `/perl`, расположенного на диске `d:`.

При использовании этой операции в выражении условия цикла `while` или `for` она последовательно на каждом шаге цикла возвращает очередное имя файла, удовлетворяющее заданному шаблону:

```
while ($file = <*.pl>) {  
    print "$file\n";  
}
```

Употребленная в выражении условия самостоятельно, эта операция возвращает очередное имя файла в переменной `$_`. Например, предыдущий фрагмент можно переписать следующим образом:

```
while (<*.pl>) {  
    print $_, "\n";  
}
```

Операция получения имен файлов, соответствующих заданному шаблону, реализуется с помощью внутренней функции `glob`, единственным параметром которой является шаблон имен файлов. Эту функцию можно использовать самостоятельно для получения соответствующих имен файлов:

```
@scripts = glob "/*.pl";
```

В скалярном контексте она возвращает имя первого файла, удовлетворяющего заданному шаблону, в списковом — список имен всех файлов. Употребленная без параметра, она использует в качестве параметра специальную переменную `$_`.

6.3. Функция *print*

Функция `print` наиболее часто используемая функция для вывода информации из сценария Perl. Ее синтаксис имеет следующий вид:

```
print ДЕСКРИПТОР СПИСОК;
```

Здесь `ДЕСКРИПТОР` представляет дескриптор файла, в который функция выводит строковые данные, представленные списком вывода `СПИСОК`. Он может состоять из переменных, элементов массивов и выражений, вычисляемых как строковые данные. Дескриптор файла создается функцией `open()`, о которой мы поговорим в следующей главе. Он может быть опущен, и в этом случае вывод осуществляется в стандартный файл вывода `STDOUT`, если только функцией `select()` не выбран другой файл вывода по умолчанию. Как уже отмечалось ранее, обычно стандартное устройство вывода — экран монитора компьютера.

Функция `print` при выводе своего списка не завершает его символом новой строки `"\n"`. Это означает, что следующая функция `print` начнет вывод на экран непосредственно после последнего выведенного предыдущей функцией `print` символа. Если такое поведение не желательно, то следует список вывода каждой функции `print` явно завершать строкой, содержащей символ новой строки, или включать его последним символом последнего элемента списка вывода. Пример 6.6 демонстрирует вывод с помощью функции `print`.

Пример 6.6. Вывод функцией `print`

```
#!/ perl -w
print "String 1:";
print "String 2:\n";
print "String 3:", "\n";
print STDOUT "String 4:\n";
print FILEOUT "String 4:\n";
```

Вывод первых четырех функций `print` примера 6.6 представлен ниже:

```
String 1:String 2:
String 3:
String 4:
```

Вторая функция `print` начинает свой вывод на той же строке, на которой завершила вывод первая функция, в которой в списке вывода нет символа перехода на новую строку. В четвертой функции явно указан дескриптор стандартного файла вывода `STDOUT`. Относительно пятой функции скажем, что она ничего ни в какой файл, определенный дескриптором `FILEOUT`, не

выведет, так как с этим дескриптором не связан никакой файл. Для этого следовало бы до выполнения последней функции `print` открыть файл функцией `open()` и связать с ним дескриптор `FILEOUT`. Мы отложим эти вопросы до следующей главы.

Функция `print`, как и большинство других функций, определенных в языке Perl, является списковой операцией, и все элементы списка вывода вычисляются в списковом контексте. Это обстоятельство следует учитывать при использовании в качестве элементов списка вывода выражений с вызовами подпрограмм.

Все, что было сказано относительно списковых операций и их использования в качестве термов выражений в главе 4, относится, естественно, и к функции `print`. Если ее параметры, включая дескриптор файла, заключены в круглые скобки, то такая синтаксическая конструкция считается *термом* и в выражении имеет наивысший приоритет вычисления. Например, следующий оператор

```
print ($m + $n) ** 2;
```

напечатает сумму значений переменных `$m` и `$n`, а не их сумму, возведенную в квадрат. Компилятор perl, обнаружив после лексемы `print` левую круглую скобку, найдет правую круглую скобку и будет рассматривать их содержимое как список параметров функции `print`. А так как такая конструкция есть терм, то сначала будет выполнена операция печати суммы значений переменных, а потом результат этой операции (Истина = 1) будет возведен в квадрат. Добавление необязательного дескриптора стандартного файла вывода `STDOUT` исправит подобную ошибку:

```
print STDOUT ($m + $n) ** 2; # Выведет ($m + $n) ** 2
```

Если в функции печати `print` не задан список вывода, то она по умолчанию выводит содержимое специальной переменной `$_` в файл, определенный параметром ДЕСКРИПТОР:

```
print;           # Выводится содержимое переменной $_ на экран монитора.
print STDOUT;    # Эквивалентен предыдущему оператору.
print FILEOUT;   # Выводится содержимое переменной $_ в файл
                  # с дескриптором FILEOUT
```

* * *

В этой главе мы познакомились с основными возможностями ввода/вывода, предоставляемыми языком Perl. Узнали, как легко и просто можно выполнить команду операционной системы и получить результаты ее вывода на экран монитора непосредственно в программу Perl. Операция `<>` позволяет не только считывать записи внешних файлов, но и автоматически обрабатывать содержимое нескольких файлов, заданных в командной строке при за-

пуске сценария Perl. Эта же операция позволяет осуществлять поиск файлов, чьи имена удовлетворяют заданному шаблону. Для вывода информации из программы используется функция `print()`, которая может выводить ее не только на экран монитора (стандартное устройство вывода), но также и во внешний файл.

Вопросы для самоконтроля

1. Каким образом можно получить результаты выполнения команды операционной системы в программе Perl?
2. Какая операция позволяет прочитать содержимое всех файлов, переданных сценарию Perl через командную строку?
3. Где хранятся имена параметров, переданных сценарию Perl через командную строку?
4. Можно ли получить в программе Perl имена файлов определенного каталога, удовлетворяющих заданному шаблону?
5. Какая списковая операция осуществляет вывод на экран монитора?
6. Какая списковая операция осуществляет вывод во внешний файл?

Упражнения

1. Напишите программу, которая копирует один файл в другой. Имена файлов передаются в программу при ее запуске как параметры командной строки. (Подсказка: используйте системную команду `copy`.)
2. Напишите программу, которая отображает на экране содержимое файлов, имена которых задаются в командной строке. Отображение содержимого каждого файла должно предваряться строкой, содержащей имя файла. (Подсказка: использовать операцию `<>`.)
3. Напишите программу Perl, которая удаляет файлы определенного каталога. Имена файлов задаются шаблоном, который вместе с именем каталога передается в программу через командную строку при ее запуске.



Работа с файлами

Когда в программе мы создаем переменные и храним в них разнообразные данные, мы теряем их по завершении работы программы. Если нам необходимо сохранить данные и использовать их в разрабатываемых программах, мы создаем файл, записываем в него данные и сохраняем его на диске. Практически любой язык программирования предоставляет программисту средства манипулирования файлами и хранимыми в них данными.

Доступ к файлам в программе Perl осуществляется через специально создаваемые дескрипторы, которые можно рассматривать как некоторый особый вид переменных. Один дескриптор в каждый момент времени может быть связан с одним и только одним файлом, хотя на протяжении всей программы один и тот же дескриптор можно последовательно связывать с разными файлами.

Более того, дескриптор можно связать не только с файлом, но и с программным каналом, обеспечивающим связь между процессами. В этой главе мы не будем касаться вопросов взаимодействия программ с другими процессами, а рассмотрим только работу с файлами и их содержимым. Поэтому дескрипторы мы иногда будем называть дескрипторами файлов.

7.1. Дескрипторы файлов

Дескриптор — это символическое имя, которое используется в программе Perl для представления файла, устройства, сокета или программного канала. При создании дескриптора он "присоединяется" к соответствующему объекту данных и представляет его в операциях ввода/вывода. Мы дали наиболее полное определение дескриптора, чтобы читатель понимал, что дескриптор позволяет работать не только с данными файлов, но и с данными других специальных программных объектов, реализующих специфические задачи получения и передачи данных. Когда дескриптор присоединен к файлу, мы будем называть его дескриптором файла.

Замечание

При открытии файла в системе UNIX ему также назначается файловый дескриптор, или дескриптор файла, который ничего общего не имеет с файловым дескриптором Perl. В UNIX дескриптор файла является целым числом, тогда

как в Perl это символическое имя, по которому мы можем сослаться на файл. Чтобы получить числовой файловый дескриптор в программе Perl, можно воспользоваться функцией `fileno()`.

В программе дескриптор файла чаще всего создается при открытии файла функцией `open()`, которой передаются два параметра — имя дескриптора и строка с именем файла и режимом доступа:

```
open( LOGFILE, "> /temp/logfile.log");
```

Этот оператор создает дескриптор с именем `LOGFILE` и присоединяет его к файлу с указанным именем, который открывается в режиме записи (строка второго параметра начинается с символа ">"). В этом разделе мы не будем касаться вопросов, связанных с режимом открытия файла, а сконцентрируем наше внимание на дескрипторах. В следующем разделе режимы открытия файла будут рассмотрены нами подробнее.

Дескриптор, как указывалось, является символическим именем файла и представляет собой правильный идентификатор, который не может совпадать с зарезервированными словами Perl. В нашем примере создается дескриптор `LOGFILE`, "замещающий" в операциях ввода/вывода файл, к которому он присоединен (`/temp/logfile.log`). Например, известной нам функцией `print()` мы можем теперь записать в этот файл значение какой-либо переменной:

```
print LOGFILE $var;
```

Любой созданный дескриптор попадает в символьную таблицу имен Perl, в которой находятся также имена всех переменных и функций. Однако дескриптор не является переменной, хотя некоторые авторы и называют его файловой переменной. В имени дескриптора не содержится никакого префикса, присущего переменным Perl (`$`, `@` или `%`). Поэтому его нельзя непосредственно использовать в операции присваивания и сохранить в переменной или передать в качестве параметра в функцию. Для подобных целей приходится использовать перед его именем префикс `*`, который дает ссылку на глобальный тип данных. Например, предыдущий оператор печати в файл, определенный дескриптором `LOGFILE`, можно осуществить с помощью следующих операторов, предварительно сохранив ссылку на дескриптор в переменной `$logf`:

```
$logf = *LOGFILE;  
print $logf $var;
```

В операции `print` первая переменная `$logf` замещает дескриптор файла `LOGFILE`, в который выводится значение второй переменной `$var`.

(Ссылки на глобальные имена более подробно рассматриваются в главе 9.)

Замечание

В программах Perl принято в именах дескрипторов использовать прописные буквы. Подобная практика позволяет легко обнаруживать их в программе и не приводит к конфликтам с зарезервированными именами функций, которые обычно определены строчными буквами.

В любой программе Perl всегда существуют три предопределенные дескриптора (STDIN, STDOUT и STDERR), которые связаны со стандартными устройствами ввода/вывода и используются некоторыми функциями Perl в качестве уменьшаемых дескрипторов файлов ввода или вывода. Как мы уже знаем, дескриптор STDIN связан со стандартным устройством ввода (обычно клавиатура), STDOUT и STDERR — со стандартным устройством вывода (обычно экран монитора). Стандартное устройство ввода используется операцией `<>`, если в командной строке вызова сценария Perl не задан список файлов. Дескриптор STDOUT по умолчанию используется функциями `print` и `die`, а STDERR — функцией `warn`. Другие функции также используют предопределенные дескрипторы файлов для вывода своей информации.

При вызове программ в среде Unix и DOS можно перенаправлять стандартный ввод и вывод в другие файлы, задавая в командной строке их имена с префиксами `>` для файла вывода и `<` для файла ввода:

```
perl program.pl <in.dat >out.dat
```

При выполнении программы `program.pl` все исходные данные должны быть подготовлены в файле `in.dat`. Вывод будет сохранен в файле `out.dat`, а не отображаться на экране монитора.

Перенаправление стандартного ввода и вывода, а также стандартного отображения ошибок, можно осуществлять непосредственно в программе Perl. Для этого следует функцией `open()` связать соответствующий предопределенный дескриптор с некоторым дисковым файлом:

```
open(STDIN, "in.dat");  
open(STDOUT, ">out.dat");  
open(STDERR, ">err.dat");
```

Теперь весь стандартный ввод/вывод будет осуществляться через указанные в операторах `open()` файлы. Обратите внимание, что при переопределении стандартных файлов вывода и ошибок перед именами файлов стоит префикс `>`, указывающий на то, что файлы открываются в режиме записи.

Замечание

Перенаправление стандартного ввода/вывода в программе можно производить только один раз. Это переназначение действует с момента перенаправления ввода/вывода и до конца программы, причем функцией `open()` нельзя вернуть первоначальные установки для дескрипторов STDIN, STDOUT и STDERR.

7.2. Доступ к файлам

Как мы уже знаем, для доступа к файлу из программы Perl необходим дескриптор. Дескриптор файла создается функцией `open()`, которая является списковой операцией Perl:

```
open ДЕСКРИПТОР, ИМЯ_ФАЙЛА;  
open ДЕСКРИПТОР;
```

При выполнении операции `open` с заданным в параметрах именем файла открывается соответствующий файл и создается дескриптор этого файла. В качестве дескриптора файла в функции `open()` можно использовать выражение — его значение и будет именем дескриптора. Имя файла задается непосредственно в виде строкового литерала или выражения, значением которого является строка. Операция `open` без имени файла открывает файл, имя которого содержится в скалярной переменной `$ДЕСКРИПТОР`, которая не может быть лексической переменной, определенной функцией `my()`. Пример 7.1 демонстрирует использование операции `open()` для открытия файлов.

Пример 7.1. Открытие файла

```
#!/ perl -w  
$var = "out.dat";  
$FILE4 = "file4.dat";  
open FILE1, "in.dat";           # Имя файла задано строкой  
open FILE2, $var;               # Имя файла задано переменной  
open FILE3, "/perlourbook/01/" . $var; # Имя файла вычисляется в выражении  
open FILE4;                     # Имя файла в переменной $FILE4
```

Замечание

Если задано не полное имя файла, то открывается файл с указанным именем и расположенный в том же каталоге, что и программа Perl. Можно задавать полное имя файла (см. третий оператор `open` примера 7.1), однако следует иметь в виду, что оно зависит от используемой операционной системы. Например, в Windows следует обязательно задавать имя диска: `d:/perlourbook/01/Chapter1.doc`.

Замечание

В системе UNIX можно открыть достаточно много файлов, тогда как в DOS и Windows количество открытых файлов зависит от установленного значения переменной окружения `FILE` и варьируется от 20 до 50 одновременно открытых файлов.

Любой файл можно открыть в одном из следующих режимов: чтения, записи или добавления в конец файла. Это осуществляется присоединением соответ-

ствующего префикса к имени файла: < (чтение), > (запись), >> (добавление). Если префикс опущен, то по умолчанию файл открывается в режиме чтения. Запись информации в файл, открытый в режиме записи (префикс >), осуществляется в начало файла, что приводит к уничтожению содержащейся в нем до его открытия информации. Информация, содержащаяся в файле, открытом в режиме добавления (префикс >>), не уничтожается, новые записи добавляются в конец файла. Если при открытии файла в режиме записи или добавления не существует файла с указанным именем, то он создается, что отличает эти режимы открытия файла от режима чтения, при котором файл должен существовать. В противном случае операция открытия завершается с ошибкой и соответствующий дескриптор не создается.

Perl позволяет открыть файл еще в одном режиме — режиме чтения/записи. Для этого перед префиксом чтения <, записи > или добавления >> следует поставить знак плюс +. Отметим различия между тремя режимами чтения/записи +<, +> и +>>. Первый и третий режимы сохраняют содержимое открываемого файла, тогда как открытие файла с использованием второго режима (+>) сначала очищает содержимое открываемого файла. Третий режим отличается от первых двух тем, что запись в файл всегда осуществляется в конец содержимого файла.

Замечание

Некоторые операционные системы требуют устанавливать указатель чтения/записи файла при переключении с операций чтения на операции записи. В Perl для этого предназначена функция `seek()`, описание которой будет дано несколько позже в этом же параграфе.

Открытие файла и создание для него дескриптора функцией `open()` охватывает все практически важные режимы работы с файлом. Однако возможности этой функции не позволяют задать права доступа для создаваемых файлов, а также вообще решить, следует ли создавать файл, если его не существует. Для подобного "тонкого" открытия файлов можно использовать функцию `sysopen()`, которая позволяет программисту самому задать отдельные компоненты режима работы с файлом: чтение, запись, создание, добавление, очистка содержимого и т. д. Синтаксис этой функции таков:

```
sysopen ДЕСКРИПТОР, ИМЯ_ФАЙЛА, ФЛАГ [, РАЗРЕШЕНИЕ];
```

Здесь параметр `ИМЯ_ФАЙЛА` представляет имя файла без префиксов функции `open()`, определяющих режим открытия файла. Последний задается третьим параметром `ФЛАГ` — числом, представляющим результат операции побитового ИЛИ (`|`) над константами режимов, определенными в модуле `Fcntl`. Состав доступных констант зависит от операционной системы. В табл. 7.1 перечислены константы режима, встречающиеся практически во всех операционных системах.

Таблица 7.1. Константы режима доступа к файлу

Константа	Значение
<code>O_RDONLY</code>	Только чтение
<code>O_WRONLY</code>	Только запись
<code>O_RDWR</code>	Чтение и запись
<code>O_CREAT</code>	Создание файла, если он не существует
<code>O_EXCL</code>	Завершение с ошибкой, если файл уже существует
<code>O_APPEND</code>	Добавление в конец файла
<code>O_TRUNC</code>	Очистка содержимого файла

Права доступа (необязательный параметр `РАЗРЕШЕНИЕ`) задаются в восьмеричной системе и при их определении учитывается текущее значение маски доступа к процессу, задаваемого функцией `umask()`. Если этот параметр не задан, то Perl использует значение `0666`.

(О правах доступа читайте документацию Perl для установленной на вашем компьютере операционной системе.)

Совет

Если возникают затруднения с установкой прав доступа, то придерживайтесь следующего правила: для обычных файлов передавайте `0666`, а для каталогов и исполняемых файлов `0777`.

В примере 7.2 собраны операции открытия файлов функцией `open()` и эквивалентные им открытия с помощью функции `sysopen()`.

Пример 7.2. Открытие файлов

```
use Fcntl;

# Только чтение
open FF, "< file.txt";
sysopen FF, "file.txt", O_RDONLY;

# Только запись (создается, если не существует,
#                  и очищается содержимое, если существует)
open FF, "> file.txt";
sysopen FF, "file.txt", O_WRONLY | O_CREAT | O_TRUNC;

# Добавление в конец (создается, если не существует)
```

```
open FF, ">> file.txt";
sysopen FF, "file.txt", O_WRONLY | O_CREAT | O_APPEND;

# Чтение/запись (файл должен существовать)
open FF, "< file.txt";
sysopen FF, "file.txt", O_RDWR;

# Чтение/запись (файл очищается)
open FF, "> file.txt";
sysopen FF, "file.txt", O_RDWR | O_CREAT | O_TRUNC;
```

При открытии файла функции `open()` и `sysopen()` возвращают значение 0, если открытие файла с заданным режимом произошло успешно, и неопределенное значение `undef` в противном случае. Всегда следует проверять успешность выполнения операции открытия файла, прекращая выполнение программы функцией `die()`. Эта функция отображает список передаваемых ей параметров и завершает выполнение сценария Perl:

```
open(FF, "< $file") or die "Нельзя открыть файл $file: $!";
```

Обратите внимание, в сообщении функции `die()` используется специальная переменная `$!`, в которой хранится системное сообщение или код ошибки. Эта информация помогает обнаружить и исправить ошибки в программе. Например, если переменная `$file` содержит имя не существующего файла, то при выполнении предыдущего оператора пользователь может увидеть сообщение следующего вида:

```
Нельзя открыть файл file.txt: No such file or directory at D:\PERL\EX2.PL
line 4.
```

Английский текст этого сообщения представляет информацию, содержащуюся в переменной `$!`.

Для полноты описания работы с функцией `open()` следует сказать, что если имя файла представляет строку `"-"`, то открываемый файл соответствует стандартному вводу `STDIN`. Это означает, что ввод с помощью созданного дескриптора файла осуществляется со стандартного устройства ввода. Если имя файла задано в виде строки `">-"`, то это соответствует выводу на стандартное устройство вывода, представленное в программе дескриптором `STDOUT`.

Замечание

Если стандартный ввод или вывод были перенаправлены (см. раздел 7.1), то ввод/вывод с помощью дескрипторов, соответствующих файлам `"-"` и `">-"`, будет осуществляться в файл, определенный в операции перенаправления стандартного ввода или вывода.

Последнее, что нам хотелось бы осветить в связи с дескрипторами файлов, — это создание дескриптора-дубликата. Если в строке имени файла после префикса режима открытия следует амперсанд "&", то ее оставшаяся часть рассматривается как имя дескриптора файла, а не как имя открываемого файла. В этом случае создается независимая копия этого дескриптора с именем, заданным первым параметром функции `open()`. Оба дескриптора имеют общий указатель текущей позиции файла, но разные буферы ввода/вывода. Закрытие одного из дескрипторов не влияет на работу другого. В программах Perl возможность создания копии дескриптора в основном применяется для восстановления стандартных файлов ввода/вывода после их перенаправления на другие файлы (пример 7.3).

Пример 7.3. Перенаправление и восстановление стандартного вывода

```
#!/ perl -w
# Создание копии дескриптора STDOUT
open(OLDOUT, ">&STDOUT");
# Перенаправление стандартного вывода
open(STDOUT, "> file.out") or die "Невозможно перенаправить STDOUT: $!";
# Печать в файл file.out
print "Информация в перенаправленный STDOUT\n";
# Закрытие перенаправленного дескриптора стандартного вывода
close(STDOUT) or die "Невозможно закрыть STDOUT: $!";
# Восстановить файл стандартного вывода
open(STDOUT, ">&OLDOUT") or die "Невозможно восстановить STDOUT: $!";
# Закрыть копию дескриптора стандартного вывода STDOUT
close(OLDOUT) or die "Невозможно закрыть OLDOUT: $!";
# Печать в восстановленный файл стандартного вывода
print "Информация в восстановленный STDOUT\n";
```

Замечание

В программах следует избегать работу с одним файлом через несколько дескрипторов-копий.

По завершении работы с файлом он закрывается функцией `close()`. Единственным необязательным параметром этой функции является дескриптор, ассоциированный с файлом:

```
close ДЕСКРИПТОР;
```

Эта функция возвращает значение Истина, если успешно очищен буфер ввода/вывода и закрыт системный дескриптор файла. Вызванная без пара-

метра, функция `close` закрывает файл, связанный с текущим дескриптором, установленным функцией `select()`.

Следует отметить, что закрывать файлы в программе функцией `close()` не обязательно. Дело в том, что открытие нового файла с дескриптором, уже связанным с каким-либо файлом, закрывает этот старый файл. Более того, при завершении программы все открытые в ней файлы закрываются. Однако такое неявное закрытие файлов таит в себе потенциальные ошибки из-за невозможности определить, завершилась ли эта операция корректно. Может оказаться, что при записи в файл переполнится диск, или будет разорвана связь с удаленным устройством вывода. Подобные ошибки можно "отловить", если использовать явное закрытие файла и проверять содержимое специальной переменной `$!`:

```
close( FILEIO ) or die "Ошибка закрытия файла: $!";
```

Существует еще один нюанс, связанный с явным закрытием файлов. При чтении из файла специальная переменная `$.` (если ее значение не изменено явным образом в программе) хранит номер последней прочитанной записи файла. При явном закрытии файла функцией `close()` значение этой переменной обнуляется, тогда как при неявном закрытии оно остается равным номеру последней прочитанной записи старого файла и продолжает увеличиваться при операциях чтения из нового файла.

Чтение информации из файла осуществляется операцией `<>`, операндом которой является дескриптор файла. В скалярном контексте при первом выполнении эта операция читает первую запись файла, устанавливая специальную переменную `$.`, отслеживающую количество прочитанных записей, равной 1. Последующие обращения к операции чтения из файла с тем же дескриптором приводят к последовательному чтению следующих записей. В списковом контексте эта операция читает все оставшиеся записи файла и возвращает список, элементами которого являются записи файла. Разделитель записей хранится в специальной переменной `$/`, и по умолчанию им является символ новой строки `"\n"`. Perl позволяет задать и другой разделитель записей обычной операцией присваивания переменной `$/` нового символа разделителя записей. В примере 7.4 демонстрируются некоторые приемы чтения из файла.

Пример 7.4 Чтение из файла

```
#!/perl -w

open(F1, "in.dat") or die "Ошибка открытия файла: $!";
open(F2, "out.dat") or die "Ошибка открытия файла: $!";
```

```
$line1 = <F1>;           # Первая запись файла in.dat
$line2 = <F1>;           # Вторая запись файла in.dat
```

```
@rest = <F1>;          # Оставшиеся записи файла in.dat

$/ = ":";              # Задание другого разделителя записей файла
@f2 = <F2>;

# Печать прочитанных записей файла out.dat
for($i=0; $i<=$#f2; $i++) {
    print "$f2[$i]\n";
}

$/ = "\n";             # Восстановление умалчиваемого разделителя записей
close(F1) or die $!;
close(F2) or die $!;

open(F3, "out.dat") or die "Ошибка открытия файла: $!";
print <F3>;            # Печать всего файла
close(F3) or die $!;
```

Несколько комментариев к программе примера 7.4. В переменные `$line1` и `$line2` читаются соответственно первая и вторая строка файла `in.dat`, так как используется умалчиваемый разделитель записей `"\n"`. Элементы массива `@rest` хранят строки с третьей по последнюю этого же файла: в операторе присваивания операция чтения `<F1>` выполняется в списковом контексте.

Перед чтением записей файла `out.dat` устанавливается новый разделитель записей — символ `":"`. Если файл `out.dat`, например, содержит только одну строку

```
111: 222: 333: Конец
```

то элементы массива `@f2` будут содержать следующие значения:

```
$f2[0] = "111:"
$f2[1] = "222:"
$f2[2] = "333:"
$f2[3] = "Конец"
```

Замечание

Если при создании файла `out.dat` его единственная строка завершена переходом на новую строку (нажата клавиша `<Enter>`), то `$f2[3]` будет содержать строку `"Конец\n"`.

При достижении конца файла операция `<>` возвращает неопределенное значение, которое трактуется как `Ложь`. Это обстоятельство обычно используется для организации чтения записей файла в цикле:

```
while($line = <F1>) {
    print $line;          # Печать очередной строки связанного
                          # с дескриптором F1 файла
}
```

Запись в файл, открытый в режиме записи или добавления, осуществляется функцией `print()` с первым параметром, являющимся дескриптором файла:

```
print ДЕСКРИПТОР СПИСОК_ВЫВОДА;
```

Эта операция записывает содержимое элементов списка в том порядке, в котором они определены в вызове функции, и не добавляет в конец списка разделителя записей. Об этом должен позаботиться сам программист:

```
$/ = " ";                # Разделитель записей
print F1 @rec11, $/;     # Запись в файл первой записи
print F1 @rec12, $/;     # Запись в файл второй записи
```

Замечание

Между дескриптором и первым элементом списка вывода *не должно* быть запятой. Если такое случится, то компилятор `perl` выдаст ошибку:

```
No comma allowed after filehandle
```

Если в функции `print` не указан дескриптор файла, то по умолчанию вывод осуществляется в стандартный файл вывода с дескриптором `STDOUT`. Эту установку можно изменить функцией `select()`. Вызванная без параметров, она возвращает текущий умалчиваемый дескриптор для вывода функциями `print()` и `write()`. Если ей передается единственный параметр, то этот параметр должен быть дескриптором файла. В этом случае она также возвращает текущий умалчиваемый дескриптор и меняет его на дескриптор, определенный переданным ей параметром.

```
$oldfilehandle = select(F1); # Сохранение текущего дескриптора по
                             # умолчанию и назначение нового F1
print $line;                # Вывод в дескриптор F1
select($oldfilehandle);     # Восстановление старого дескриптора
                             # по умолчанию
print $line;                # Вывод в старый дескриптор
```

Файлы в Perl интерпретируются как неструктурированные потоки байтов. При работе с файлом через дескриптор отслеживается его *текущая позиция*. Операции чтения/записи выполняются с текущей позиции файла. Если, например, была прочитана запись длиной 80 байт, то следующая операция чтения или записи начнется с 81 байта файла. Для определения текущей позиции в файле используется функция `tell()`, единственным параметром которой может быть дескриптор файла. Она возвращает текущую позицию в

связанном с дескриптором файле. Эта же функция без параметра возвращает текущую позицию в файле, для которого была в программе выполнена последняя операция чтения.

Текущая позиция в файле автоматически изменяется в соответствии с выполненными операциями чтения/записи. Ее можно изменить с помощью функции `seek()`, которой передаются в качестве параметров дескриптор файла, смещение и точка отсчета. Для связанного с дескриптором файла устанавливается новая текущая позиция, смещенная на заданное параметром СМЕЩЕНИЕ число байт относительно точки отсчета:

```
seek ДЕСКРИПТОР, СМЕЩЕНИЕ, ТОЧКА_ОТСЧЕТА;
```

Параметр `ТОЧКА_ОТСЧЕТА` может принимать одно из трех значений: 0 — начало файла, 1 — текущая позиция, 2 — конец файла. Смещение может быть как положительным, так и отрицательным. Обычно оно отрицательно для смещения относительно конца файла и положительно для смещения относительно начала файла. Для задания точки отсчета можно воспользоваться константами `SEEK_SET`, `SEEK_CUR` и `SEEK_END` из модуля `IO::Seekable`, которые соответствуют началу файла, текущей позиции и концу файла. Естественно, необходимо подключить этот модуль к программе с помощью ключевого слова `use`. Например, следующие операторы устанавливают одинаковые текущие позиции в файлах:

```
use IO::Seekable;  
seek FILE1, 5, 0;  
seek FILE2, 5, SEEK_SET;
```

Для перехода в начало или в конец файла следует использовать нулевое смещение относительно соответствующих точек отсчета при обращении к функции `seek()`:

```
seek FILE1, 0, 0; # Переход в начало файла  
seek FILE1, 0, 2; # Переход в конец файла
```

Кроме операции чтения записей файла `<>`, `Perl` предоставляет еще два способа чтения информации из файла: функции `getc()` и `read()`. Первая читает один байт из файла, тогда как вторая читает записи фиксированной длины.

Функция `getc` возвращает символ в текущей позиции файла, дескриптор которого передан ей в качестве параметра, или неопределенное значение в случае достижения конца файла или возникновении ошибки. Если функция вызывается без параметра, то она читает символ из стандартного файла ввода `STDIN`.

```
getc;      # Чтение символа из STDIN  
getc F1;   # Чтение символа в текущей позиции файла с дескриптором F1
```

Функции `read()` передаются три или четыре параметра и ее синтаксис имеет вид:

```
read ДЕСКРИПТОР, ПЕРЕМЕННАЯ, ДЛИНА [,СМЕЩЕНИЕ];
```

Она читает количество байтов, определенное значением параметра `ДЛИНА`, в скалярную переменную, определяемую параметром `ПЕРЕМЕННАЯ`, из файла с дескриптором, заданным первым параметром `ДЕСКРИПТОР`. Возвращаемое значение — действительное количество прочитанных байтов, 0 при попытке чтения в позиции конца файла и неопределенное значение в случае возникновения ошибки. Параметр `СМЕЩЕНИЕ` определяет количество сохраняемых байтов из содержимого переменной `ПЕРЕМЕННАЯ`, т. е. запись прочитанных из файла данных будет добавлена к содержимому переменной после байта, определяемого значением параметра `СМЕЩЕНИЕ`. Отрицательное значение смещения `-n` (`n` — целое число) означает, что из содержимого переменной `ПЕРЕМЕННАЯ` отбрасываются последние `n` байтов и к оставшейся строке добавляется запись, прочитанная из файла. Пример 7.5 демонстрирует чтение записей фиксированной длины в предположении, что файл `in.dat` содержит три строки данных:

One

Two

Three

Пример 7.5. Чтение записей фиксированной длины

```
#!/ perl -w
open(F1, "in.dat") or die "Ошибка открытия файла: $!";
$string = "1234567890";
read F1, $string, 6;           # Чтение шести байт в переменную без смещения
print $string, "\n";          # $string = "One\nTw"
read F1, $string, 6, length($string);
print $string, "\n";          # $string = "One\nTwo\nThre"
```

Функция `length()` возвращает количество символов (байтов) в строковых данных, хранящихся в скалярной переменной, переданной ей в качестве параметра. После выполнения первой операции чтения содержимое переменной `$string` было уничтожено, так как эта функция `read()` вызывалась без смещения. Тогда как при втором чтении хранившиеся данные в переменной `$string` были полностью сохранены.

Операции `<>`, `print`, `read`, `seek` и `tell` относятся к операциям буферизованного ввода/вывода, т. е. они для повышения скорости выполнения используют буферы. Perl для выполнения операций чтения из файла и записи в файл предлагает также аналоги перечисленных функций, не используя

щие буферы при выполнении соответствующих операций с содержимым файла.

Функции `sysread` и `syswrite` являются не буферизованной заменой операции `<>` и функции `print`, а функция `sysseek` заменяет функции `seek` и `tell`.

Функции не буферизованного чтения и записи получают одинаковые параметры, которые соответствуют параметрам функции `read`:

```
sysread ДЕСКРИПТОР, ПЕРЕМЕННАЯ, ДЛИНА [,СМЕЩЕНИЕ];
```

```
syswrite ДЕСКРИПТОР, ПЕРЕМЕННАЯ, ДЛИНА [,СМЕЩЕНИЕ];
```

Смысл всех параметров аналогичен параметрам функции `read()`. Возвращаемым значением этих функций является истинное количество прочитанных/записанных байт, 0 в случае достижения конца файла или `undef` при возникновении ошибки.

Параметры функции `sysseek()` полностью соответствуют параметрам функции `seek()`:

```
sysseek ДЕСКРИПТОР, СМЕЩЕНИЕ, ТОЧКА_ОТСЧЕТА;
```

Все, сказанное относительно использования функции `seek()`, полностью переносится и на ее не буферизованный аналог.

Функциональность буферизованной операции `tell()` реализуется следующим вызовом функции `sysseek`:

```
$position = sysseek F1, 0, 1; # Текущая позиция указателя файла
```

Пример 7.6 демонстрирует использование не буферизованных функций ввода/вывода для обработки содержимого файла.

Пример 7.6. Не буферизованный ввод/вывод

```
#!/ perl -w
use Fcntl;

# Открытие файла в режиме чтение/запись
sysopen F1, "in.dat", O_RDWR;

# Чтение блока в 14 байт
$read = sysread F1, $string, 14;
warn "Прочитано $read байт вместо 14\n" if $read != 14;

# Установка текущей позиции (на 15 байт)
$position = sysseek F1, 0, 1;
die "Ошибка позиционирования: $!\n" unless defined $position;

# Запись строки в текущей позиции
$string = "Новое значение";
$written = syswrite F1, $string, length($string);
```

```
die "Ошибка записи: $!\n" if $written != length($string);  
# Закрытие файла  
close F1 or die $!;
```

При работе с не буферизованными функциями ввода/вывода следует всегда проверять завершение операции чтения, записи или позиционирования. Стандартная система ввода/вывода, через которую реализуется буферизованный ввод/вывод, сама проверяет и отвечает за завершение указанных операций, если процесс был прерван на середине записи. При не буферизованном вводе/выводе об этом должен позаботиться программист.

Совет

При работе с одним и тем же файлом не следует смешивать вызовы буферизованных и не буферизованных функций ввода/вывода. Подобная практика может приводить к непредсказуемым коллизиям.

7.3. Операции с файлами

Перед изучением функций, выполняющих действия с целыми файлами, мы напомним читателю основные положения, связанные с организацией файловой системы UNIX и процедур доступа к файлам. Функции Perl разрабатывались для работы именно с этой файловой системой, хотя в определенной степени многое из того, о чем пойдет речь, применимо и к файловым системам других платформ.

Работа пользователя в UNIX начинается с процедуры регистрации в системе, во время которой он вводит свое регистрационное имя и пароль. Регистрационное имя назначается администратором системы и хранится в специальном учетном файле. Пароль задает сам пользователь.

Регистрационное имя легко запоминается пользователем, но для системы удобнее вести учет пользователей, идентифицируя их не по символическим регистрационным именам, а по числовым идентификаторам. Поэтому каждому пользователю системы UNIX помимо мнемонического регистрационного имени присваивается также числовой идентификатор пользователя (uid — User Identifier) и идентификатор группы (gid — Group Identifier), к которой он относится. Значения uid и gid приписываются процессу, в котором выполняется командный интерпретатор shell, запускаемый при входе пользователя в систему. Эти же идентификаторы передаются и любому другому процессу, запускаемому пользователем во время его сеанса работы в UNIX.

Файловая система UNIX представляет собой дерево, промежуточные вершины которого соответствуют каталогам, а листья файлам или пустым каталогам. Каждый файл идентифицируется своим уникальным полным име-

нем, которое включает в себя полный путь (pathname) от корня файловой системы через промежуточные вершины (каталоги) непосредственно к файлу. Корневой каталог имеет предопределенное имя, представляемое символом "/". Этот же символ используется и для разделения имен каталогов в цепочке полного имени файла, например /bin/prog.exe.

Каждый файл в файловой системе UNIX характеризуется значительно большим объемом информации, чем, например, файл в файловой системе FAT. Эта информация включает, в частности, данные о владельце файла, группе, к которой принадлежит владелец файла, о том, кто имеет право на чтение файла, запись в файл, на выполнение файла и т. д. Эта информация позволяет задавать разные права доступа к файлу для следующих категорий пользователей: владелец файла, члены группы владельца, прочие пользователи. Вся существенная информация о файле хранится в специальной структуре данных, называемой индексным дескриптором (inode). Индексные дескрипторы размещаются в специальной области диска, формируемой при его форматировании в системе UNIX.

При запуске процесса с ним связываются два идентификатора пользователя: действительный (real) и эффективный (effective) и два аналогичных идентификатора группы пользователей. Действительные идентификаторы пользователя и группы — это постоянные идентификаторы, связываемые со всеми процессами, запускаемыми пользователем. Эффективные идентификаторы — это временные идентификаторы, которые могут устанавливаться для выполнения определенных действий. Например, при изменении пользователем пароля программа passwd автоматически устанавливает эффективные идентификаторы процесса таким образом, чтобы обеспечить права записи в файл паролей.

Как только с процессом связаны соответствующие идентификаторы, для него начинают действовать ограничения доступа к файлам. Процесс может получить доступ к файлу только в случае, если это позволяют хранящиеся при файле ограничения доступа.

Для каждого зарегистрированного пользователя системы создается так называемый "домашний" (home) каталог пользователя, к которому он имеет неограниченный доступ, а также и ко всем каталогам и файлам, содержащимся в нем. Пользователь может создавать, удалять и модифицировать каталоги и файлы из своего домашнего каталога. Потенциально возможен доступ и ко всем другим файлам, однако он может быть ограничен, если пользователь не имеет достаточных привилегий.

Любой пользователь, создавший собственный файл, считается его *владельцем*. Изменить владельца файла из сценария Perl можно функцией `chown()`. Параметром этой функции является список, первые два элемента которого должны представлять новые числовые идентификаторы `uid` и `gid`. Остальные элементы списка являются именами файлов, для которых изменяется владе-

лец. Эта функция возвращает количество файлов, для которых операция изменения владельца и группы прошла успешно.

```
@list = ( 234, 3, "file1.dat", "file2.dat");  
$number = chown(@list);  
warn "Изменился владелец не у всех файлов!" if $number != @list-2;
```

Замечание

Изменить владельца файла может только сам владелец или суперпользователь (обычно системный администратор) системы UNIX. В операционных системах с файловой системой отличной от UNIX (DOS, Windows) эта функция отработывает, но ее установки не влияют на доступ к файлу.

Функция `chmod()` изменяет права доступа для файлов, представленных в списке, передаваемом ей в качестве параметра. Первым элементом этого списка должно быть трехзначное восьмеричное число, задающее права доступа для владельца, пользователей из группы, в которую входит владелец, и прочих пользователей. Каждая восьмеричная цифра определяет право на чтение файла, запись в файл и его выполнение (в случае если файл представляет выполняемую программу) для указанных выше групп пользователей. Установленные биты ее двоичного представления отражают соответствующие права доступа к файлу. Например, если установлены все три бита (восьмеричное число 7), то соответствующая группа пользователей обладает всеми перечисленными правами: может читать из файла, записывать в файл и выполнять его. Значение равное 6 определяет право на чтение и запись, 5 позволяет читать из файла, выполнять его, но не позволяет записывать в этот файл и т. д. Обычно не выполняемый файл создается с режимом доступа 0666 — все пользователи могут читать и записывать информацию в файл, выполняемый файл — с режимом 0777. Если владелец файла желает ограничить запись в файл пользователей не его группы, то следует выполнить следующий оператор:

```
chmod 0664, "file.dat";
```

Возвращаемым значением функции `chmod()`, как и функции `chown()`, является количество файлов из списка, для которых операция изменения прав доступа завершилась успешно.

Замечание

В операционных системах DOS и Windows имеет значение только установка режимов доступа владельца.

В структуре индексного дескриптора файла существует три поля, в которых хранится время последнего обращения (`atime`) к файлу, его изменения (`mtime`) файла и изменения индексного дескриптора (`ctime`). Функцией `utime()` можно изменить время последнего обращения и модификации

файла. Ее параметром является список, содержащий имена обрабатываемых файлов, причем первые два элемента списка — числовые значения нового времени последнего доступа и модификации:

```
@files = ("file1.dat", "file2.dat");  
$now = time;  
utime $now, $now, @files;
```

В этом фрагменте кода время последнего доступа и модификации файлов из списка @files изменяется на текущее время, полученное с помощью функции time.

Отметим, что при выполнении функции utime() изменяется и время последней модификации индексного дескриптора (ctime) — оно устанавливается равным текущему времени. Возвращаемым значением является количество файлов, для которых операция изменения времени последнего доступа и модификации прошла успешно.

Файловая система UNIX позволяет создавать ссылки на один и тот же файл. Это реализуется простым указанием одного и того же индексного дескриптора для двух элементов каталога. Такие ссылки называются *жесткими* (hard) *ссылками*, и операционная система не различает элемент каталога, созданный при создании файла, и ссылок на этот файл. При обращении к файлу по ссылке и по имени изменяются поля индексного дескриптора. Физически файл уничтожается только тогда, когда уничтожается последняя жесткая ссылка на файл.

В UNIX существует еще один тип ссылок на файл — *символические ссылки*. Эти ссылки отличаются от жестких тем, что они косвенно ссылаются на файл, имя которого хранится в блоке данных символической ссылки.

Жесткие ссылки создаются в Perl функцией link(), а символические — функцией symlink(). Синтаксис этих функций одинаков — их два параметра представляют имя файла, для которого создается ссылка, и новое имя файла-ссылки:

```
link СТАРЫЙ_ФАЙЛ, НОВЫЙ_ФАЙЛ;  
symlink СТАРЫЙ_ФАЙЛ, НОВЫЙ_ФАЙЛ;
```

При успешном создании жесткой ссылки функция link() возвращает Истина, иначе Ложь. Создание символической ссылки функцией symlink() сопровождается возвратом ею числа 1 в случае успешного выполнения операции и 0 в противном случае.

Замечание

В версиях Perl для DOS эти функции не реализованы, и при попытке их вызова интерпретатор выдает фатальную ошибку:

The Unsupported function link function is unimplemented at D:\EX2.PL line 2.

The symlink function is unimplemented at D:\EX2.PL line 2.

Удалить существующие ссылки на файл можно функцией `unlink()`. Эта функция удаляет одну ссылку на каждый файл, заданный в списке ее параметров. Если ссылок на файл не существует, то удаляется сам файл. Функция возвращает количество файлов, для которых успешно прошла операция удаления. Вызов функции `unlink` без списка параметров использует содержимое специальной переменной `$_` в качестве списка параметров. Следующий фрагмент кода удаляет все резервные копии файлов текущего каталога:

```
unlink <*.bak>;
```

В структуре индексного дескриптора поле `nlink` содержит количество жестких ссылок на файл. Его можно использовать совместно с функцией `unlink()` для удаления всех ссылок на файл. Если ссылок нет, то это поле имеет значение 1 (только имя файла, определенное при его создании, ссылается на индексный дескриптор файла).

Замечание

Каталоги в UNIX являются файлами специального вида. Однако их нельзя удалить функцией `unlink`, если только вы не суперпользователь или при запуске `perl` не используется флаг `-U`. Для удаления каталогов рекомендуется использовать функцию `rmdir()`.

Две последние операции, связанные с файлами, — это переименование и усечение файла. Функция `rename()` меняет имя файла, заданного первым параметром, на имя, определяемое вторым параметром этой функции:

```
rename "old.dat", "new.dat";
```

Этот оператор переименует файл `old.dat` в файл `new.dat`. Функция переименования файла возвращает 1 при успешном выполнении этой операции и 0 в противном случае.

Функция `truncate()` усекает файл до заданной длины. Для задания файла можно использовать как имя файла, так и дескриптор открытого файла:

```
truncate ДЕСКРИПТОР, ДЛИНА;
```

```
truncate ИМЯ_ФАЙЛА, ДЛИНА;
```

Функция возвращает значение Истина, если длина файла успешно усечена до количества байт, определенных в параметре `длина`, или неопределенное значение `undef` в противном случае. Под усечением файла понимается не только уменьшение его длины, но и увеличение. Это означает, что значение второго параметра функции `truncate()` может быть больше истинной длины файла, что позволяет делать "дыры" в содержимом файла, которые в

дальнейшем можно использовать для записи необходимой информации, не уничтожая уже записанную в файл (пример 7.7).

Пример 7.7. Файл с "дырами"

```
#!/ perl -w
# Создание файла с "дырами"
for($i=1;$i<=3;$i++){
    open(F, ">>out.dat") or die $!;
    print F "Запись".$i;
    close F;

    open(F, ">>out.dat") or die $!;
    truncate F, 19*$i;
    close F;
}
# Запись информации в "дыры"
open(F, "+<out.dat") or die $!;
for($i=1;$i<=3;$i++){
    seek F,0,1;
    read F,$recl,7;
    seek F,0,1;
    print F "<CONTENTS:". $i.">";
}
close F;
```

На каждом шаге первого цикла `for` примера 7.7 в конец файла `out.dat` записывается информация длиной 7 байтов, а потом его длина увеличивается на 12 байтов, образуя пустое пространство в файле. Следующий цикл `for` заносит в эти созданные "дыры" информацию длиной 12 байтов, не затирая хранящуюся в файле информацию. Обратите внимание, что для изменения длины файла функцией `truncate` приходится закрывать его и снова открывать. Это связано с тем обстоятельством, что функция `truncate()` добавляет пустое пространство в начало файла, сдвигая в конец его содержимое, если применять ее, не закрывая файл. Можете поэкспериментировать с программой примера 7.7, открыв файл перед выполнением первого цикла `for`, и закрыв его после завершения цикла. Содержимое файла даст вам наглядное представление о работе функции `truncate` в этом случае. У нас же после выполнения первого цикла `for` содержимое файла `out.dat` выглядит так:

По завершении всей программы файл будет содержать следующую строку:

Запись1<CONTENTS:1>Запись2<CONTENTS:2>Запись3<CONTENTS:3>

7.4. Получение информации о файле

Мы знаем, что в файловой системе UNIX информация о файле хранится в его индексном дескрипторе (inode). Структура индексного дескриптора состоит из 13 полей, для которых используются специальные обозначения. Все они перечислены в табл. 7.2.

Таблица 7.2. Структура индексного дескриптора

Поле	Описание
dev	Номер устройства в файловой системе
ino	Номер индексного дескриптора
mode	Режим файла (тип и права доступа)
nlink	Количество жестких ссылок на файл (в отсутствии ссылок равно 1)
uid	Числовой идентификатор владельца файла
gid	Числовой идентификатор группы владельца файла
rdev	Идентификатор устройства (только для специальных файлов)
size	Размер файла в байтах
atime	Время последнего обращения к файлу с начала эпохи
mtime	Время последнего изменения файла с начала эпохи
ctime	Время изменения индексного дескриптора с начала эпохи
blksize	Предпочтительный размер блока для операций ввода/вывода
blocks	Фактическое количество выделенных блоков для размещения файла

Замечание

Начало эпохи датируется 1 января 1970 года 0 часов 0 минут.

Замечание

Не все перечисленные в табл. 7.2 поля структуры индексного дескриптора поддерживаются всеми файловыми системами.

Для получения значений полей структуры индексного дескриптора файла в Perl предназначена функция `stat()`. Ее единственным параметром может быть либо имя файла, либо дескриптор открытого в программе файла. Она

возвращает список из 13 элементов, содержащих значения полей структуры индексного дескриптора файла в том порядке, как они перечислены в табл. 7.2. Типичное использование в программе Perl представлено ниже

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,  
$atime,$mtime,$ctime,$blksize,$blocks) = stat($filename);
```

Присваивание значений полей списку скалярных переменных с идентификаторами, соответствующими названиям полей, способствует лучшей читаемости программы, чем присваивание массиву скаляров:

```
@inode = stat($filename);
```

В последнем случае получить значение соответствующего поля можно только с помощью индекса, что не совсем удобно, так как надо помнить номер нужного поля структуры.

Если при обращении к функции `stat()` не указан параметр, то она возвращает структуру индексного дескриптора файла, чье имя содержится в специальной переменной `$_`.

Функция получения информации о файле при успешном выполнении в списковом контексте возвращает список значений полей структуры индексного дескриптора файла или пустой список в случае неудачного завершения. В скалярном контексте она возвращает булево значение Истина или Ложь в зависимости от результатов своего выполнения.

Для удобства использования информации о файле функция `stat()` при успешном выполнении кэширует полученные значения полей. Если вызвать эту функцию со специальным дескриптором файла `_` (символ подчеркивания), то она возвратит информацию, хранящуюся в кэше от предыдущего ее вызова. Это позволяет проверять различные атрибуты файла без повторного вызова функции `stat()` или сохранения результатов ее выполнения в переменных программы.

Функцию `stat()` можно использовать для получения структуры индексного дескриптора не только файла, но и жестких ссылок на него, а также каталогов, так как они являются также файлами, блоки данных которых содержат имена файлов каталога и их числовых индексных дескрипторов. Для получения информации о символических ссылках следует использовать функцию `lstat()`, которая возвращает список значений полей структуры индексного дескриптора самой ссылки, а не файла, на который она ссылается. Эта функция работает аналогично функции `stat()`, включая использование специального дескриптора `_`.

Замечание

Если операционная система не поддерживает символические ссылки, то обращение к функции `lstat()` заменяется обращением к функции `stat()`.

Кроме двух этих функций, позволяющих получать информацию о файлах системы, в Perl предусмотрен набор унарных операций, возвращающих значение только одного поля структуры индексного дескриптора. Эти операции в документации называются "операциями -х", так как их названия состоят из дефиса с последующим единственным символом. Все они являются унарными именованными операциями и имеют свой приоритет в сложных выражениях, о котором мы рассказывали в гл. 3. Полный перечень унарных операций проверки атрибутов файлов представлен в табл. 7.3.

Таблица 7.3. Унарные именованные операции проверки файлов

Операция	Проверяемый атрибут
-r	Файл может читаться эффективным uid/gid
-w	Записывать в файл может эффективный uid/gid
-x	Файл может выполняться эффективным uid/gid
-o	Владельцем файла является эффективный uid
-R	Файл может читаться действительным uid/gid
-W	Записывать в файл может действительный uid/gid
-X	Файл может выполняться действительный uid/gid
-O	Владельцем файла является действительный uid
-e	Файл существует
-z	Размер файла равен нулю
-s	Размер файла отличен от нуля (возвращается размер)
-f	Файл является обычным (plain) файлом
-d	Файл является каталогом
-l	Файл является символической ссылкой
-p	Файл является именованным программным каналом (FIFO) или проверяемый дескриптор связан с программным каналом
-S	Файл является сокетом
-b	Файл является специальным блочным файлом
-c	Файл является специальным символьным файлом
-t	Дескриптор файла связан с терминалом
-u	У файла установлен бит setuid
-g	У файла установлен бит setgid
-k	У файла установлен бит запрета (sticky bit)
-T	Файл является текстовым файлом.

Таблица 7.3 (окончание)

Операция	Проверяемый атрибут
-B	Файл является двоичным (противоположным текстовому)
-M	Возраст файла в днях на момент выполнения программы
-A	То же для времени последнего обращения к файлу
-C	То же для времени последней модификации индексного дескриптора файла

Унарные операции применяются к строке, содержащей имя файла, к выражению, вычисляемым значением которого является имя файла, или к файловому дескриптору Perl. Если параметр операции не задан, то она тестирует файл, чье имя содержится в специальной переменной `$_`. Каждая операция проверки атрибута файла возвращает 1, если файл обладает соответствующим атрибутом, пустую строку "" в противном случае и неопределенное значение `undef`, если указанный в параметре файл не существует.

Несколько слов об алгоритме определения текстовых и двоичных файлов (операции `-t` и `-b`). Эти операции анализируют содержимое первого блока файла на наличие "странных" символов — необычных управляющих последовательностей или байтов с установленными старшими битами. Если обнаружено достаточно большое количество подобных символов (больше 30%), то файл считается двоичным, иначе текстовым. Любой файл с пустым первым блоком рассматривается как двоичный.

Если эти операции применяются к файловым дескрипторам Perl, то проверяется содержимое буфера ввода/вывода, а не первого блока файла. Обе эти операции, примененные к файловым дескрипторам, возвращают булево значение Истина, если связанный с дескриптором файл пуст или установлен на конец файла.

При выполнении унарных именованных операций проверки файла на самом деле неявно вызывается функция `stat()`, причем результаты ее вычисления кэшируются, что позволяет использовать специальный файловый дескриптор `_` для ускорения множественных проверок файла:

```
if( -s("filename") && -T _) {
# Что-то делаем для текстовых файлов не нулевого размера.
. . . . .
}
```

7.5. Операции с каталогами

Как мы отмечали ранее, в UNIX каталоги являются файлами специального формата, помеченными в структурах своих индексных дескрипторов как ка-

талог (поле `rdev`). Содержимым блоков данных каталогов является множество пар, состоящих из объекта, содержащегося в каталоге, и числового значения его индексного дескриптора.

Для работы с каталогами в Perl предусмотрены функции открытия, закрытия и чтения содержимого каталога, синтаксис и семантика которых аналогичны синтаксису и семантике соответствующих операций с файлами:

```
opendir ДЕСКРИПТОР, ИМЯ_КАТАЛОГА;
```

```
closedir ДЕСКРИПТОР;
```

```
readdir ДЕСКРИПТОР;
```

Доступ к содержимому каталога осуществляется, как и в случае с файлом, через создаваемый функцией `opendir()` *дескриптор каталога*. Отметим, что для дескрипторов каталогов в таблице символов Perl создается собственное пространство имен. Это означает, что в программе могут существовать, совершенно не конфликтуя между собой, дескрипторы файла и каталога с одинаковыми именами:

```
open FF, "/usr/out.dat" # Дескриптор файла
```

```
opendir FF, "/usr"      # Дескриптор каталога
```

Замечание

Использование одинаковых имен для дескрипторов файла и каталога может запутать самого пользователя. Однако для perl такой проблемы не существует: интерпретатор всегда точно знает, какой дескриптор следует использовать.

Функция `readdir()` для открытого каталога в списковом контексте возвращает список имен всех файлов каталога или пустой список, если все имена уже были прочитаны. Эта же функция в скалярном контексте возвращает следующее имя файла каталога или неопределенное значение `undef`, если были прочитаны все имена файлов.

Функцией `rewinddir()` текущая позиция в каталоге устанавливается на начало, что позволяет осуществлять повторное чтение имен файлов каталога, не закрывая его. Единственным параметром этой функции является дескриптор открытого каталога.

Программа примера 7.8 проверяет, являются все файлы каталога двоичными (содержимое вложенных каталогов не проверяется).

Пример 7.8. Проверка содержимого каталога

```
#!/ perl -w
opendir FDIR, "/usr/prog";
while( $name = readdir FDIR) {
    next if -d $name;    # Каталог
```

```
print("$name: двоичный\n") if -B $name; # Двоичный файл
}
closedir FDIR;
```

Функция `readdir()` возвращает относительное имя файла. Для получения полного имени файла следует создать его в программе самостоятельно. Например, добавить имя проверяемого каталога в примере 7.8:

```
print("/usr/prog/$name: двоичный\n") if -B $name; # Двоичный файл
```

Для создания нового каталога следует воспользоваться функцией `mkdir()`, параметрами которой являются имя каталога и режим доступа (восьмеричное число):

```
mkdir ИМЯ_КАТАЛОГА, РЕЖИМ;
```

Если задается не полное имя каталога, то он создается в текущем каталоге, устанавливаемом функцией `chdir()`. Возвращаемым значением функции создания нового каталога `mkdir()` является Истина, если каталог создан, и Ложь, если произошла какая-то ошибка. В последнем случае в специальной переменной `$!` хранится объяснение не выполнения операции создания каталога.

Совет

Для каталогов рекомендуется задавать режим доступа равным `0777`.

Удалить каталог можно функцией `rmdir()` с параметром, содержащим строку с именем каталога. Если параметр не задан, то используется специальная переменная `$_`. Как и функция создания каталога, эта функция возвращает значение Истина в случае успешного удаления каталога и Ложь в противном случае, записывая в переменную `$!` объяснение возникшей ошибки.

Замечание

Функция `rmdir()` удаляет только пустой каталог. Если он содержит другие пустые каталоги, их надо удалить ранее.

* * *

В этой главе мы узнали, как получить доступ к содержимому файлов и каталогов через соответствующие дескрипторы. Научились читать и записывать информацию в файлы, создавать и удалять каталоги. Познакомились с большим набором унарных именованных операций для получения информации об атрибутах файлов из полей структуры индексного дескриптора.

Вопросы для самоконтроля

1. Как можно получить доступ к файлу из программы Perl?
2. Перечислите операции, позволяющие читать содержимое файла и записывать в него информацию.
3. Какие существуют режимы открытия файла и чем они отличаются?
4. Что такое режим доступа файла и как его можно изменить в программе Perl?
5. Что такое дескриптор каталога и зачем он нужен?
6. Как можно получить имена всех файлов определенного каталога?

Упражнения

1. Найдите ошибки в программе:

```
#!/ perl -w
$var = (stat "file1.dat")[7];
open FILE1 ">file1.dat";
print FILE1, "Длина файла: " . $var . "\n";
```

2. Напишите программу, которая удаляет каталог, имя которого передается через командную строку. Если сценарий запущен без параметров, то предусмотрите отображение сообщения о правильном синтаксисе его вызова. (Указание: сначала следует удалить все файлы из нижележащих каталогов, если таковые имеются.)
3. Напишите программу копирования одного файла в другой. Предусмотрите ввод имен файлов как через командную строку, так и с экрана монитора.
4. Напишите программу копирования содержимого одного каталога в другой каталог. Предусмотрите ввод имен каталогов как через командную строку, так и с экрана монитора.
5. Напишите программу чтения строки текстового файла с заданным номером. Предусмотрите случаи, когда номер заданной строки превосходит количество строк в файле. Если номер строки отрицательный, то следует прочитать все строки, начиная со строки с номером, равным абсолютно-му значению введенного отрицательного значения.



Форматы

Как мы помним, дословный перевод аббревиатуры языка Perl включает в себя слова "язык отчетов", т. е. язык Perl предназначен не только для извлечения и обработки информации из текстовых файлов, но и для генерирования отчетов на основе этой информации. Пока что мы для вывода информации использовали функцию `print()`, которая не очень-то удобна для создания отчетов — определенным образом отформатированной выходной информации. (Можно было бы воспользоваться функцией форматированного вывода `printf()`, но мы решили не нагружать нашего читателя изучением языковых средств, которыми он редко будет пользоваться, тем более что всегда можно обратиться к документации Perl.)

Для создания простых отчетов в Perl предусмотрены *форматы*, которые позволяют в тексте программы практически визуализировать внешний вид выводимого отчета, так как определение формата в Perl очень близко к тому, что отображается при выводе. Форматы позволяют задавать верхний колонтитул каждой страницы отчета, куда можно поместить название документа, номер страницы и другую полезную информацию. Perl может отслеживать количество строк на странице отчета и автоматически переходить на новую страницу по заполнению всех строк предыдущей.

Использование форматов для создания отчетов очень просто. Первое, что необходимо сделать, — определить формат и переменные, которые в нем используются. Далее следует инициализировать в программе эти переменные и осуществить форматированный вывод в файл, определенный своим дескриптором, с помощью функции `write()`.

Эти и другие, связанные с форматами, вопросы и являются предметом изучения в этой главе. Начнем мы с основного вопроса — объявление форматов в программе Perl.

8.1. Объявление формата

Формат — это одна из двух языковых единиц (вторая — подпрограмма `sub`), которая требует обязательного объявления в программе Perl. Он используется в качестве "руководства" функцией `write()`, которая выводит на экран монитора, принтер или в файл информацию из программы в соответствии с записанными в формате "инструкциями" форматирования строк вы-

вода. При объявлении формата определяется, как должна быть отформатирована каждая его строка при отображении на устройстве вывода.

Формат объявляется в программе с помощью ключевого слова `format`, после которого следуют "инструкции" по форматированию определенных в нем строк. Завершается объявление формата строкой, первым символом которой является точка ".". Общий синтаксис конструкции объявления формата следующий:

```
format ИМЯ_ФОРМАТА =  
ФОРМАТЫ_СТРОК
```

Параметр `ИМЯ_ФОРМАТА` представляет собой правильный идентификатор Perl. Он должен в точности соответствовать имени дескриптора файла, который используется в качестве единственного параметра в функции вывода `write()`. Например, если форматированный отчет выводится в файл, определенный в программе дескриптором `FILE`, то и имя формата должно быть также `FILE`. Функцию `write()` можно вызывать без параметра. В этом случае вывод осуществляется на стандартное устройство вывода (`STDOUT`), и имя формата в этом случае должно быть равным `STDOUT`. Если функцией `select()` установлен дескриптор файла вывода по умолчанию, то вывод функцией `write()` без параметра будет осуществляться в этот файл, причем имя формата вывода должно быть изменено на имя дескриптора файла.

Замечание

Так как операция `format` не является вычисляемой операцией Perl, то объявление формата может осуществляться в любом месте программы. Обычно все объявления форматов задают либо в начале, либо в конце программы.

Замечание

Имя формата, как отмечалось, может быть любым правильным идентификатором Perl, однако обычно его определяют прописными буквами, что способствует лучшей читаемости программы.

В теле формата (до завершающей строки с точкой) определяются форматы для *каждой* строки вывода. Формат строки состоит из двух строк: первая, называемая *строкой шаблонов*, определяет, как отображается информация, вторая, называемая *строкой переменных*, задает переменные, содержащие выводимую информацию. Вместе эти две строки определяют формат и содержимое одной строки вывода функцией `write()`.

Строка шаблонов печатается точно так, как она выглядит в тексте программы (включая пробельные символы), за исключением некоторых полей, в которые подставляются значения переменных из строки переменных. Эти

поля (иногда их называют *шаблоны*, что и дало название соответствующей строке формата) начинаются с символа "@" или "^", за которым следуют *символы форматирования* (табл. 8.1), определяющие ширину поля вывода значения переменной в символах и выравнивание выводимого значения внутри поля. Количество символов форматирования определяет ширину поля вывода, причем для одного поля все символы должны быть одинакового типа.

Переменные, определяющие значения для полей строки шаблонов, задаются через запятую в строке переменных. Порядок их задания соответствует порядку задания полей вывода в строке шаблонов: значение первой переменной выводится в первое поле, второй — во второе и т. д. Все переменные в строке переменных вычисляются в списковом контексте. Это позволяет задавать выводимые значения в элементах массива скаляров.

Замечание

Если строка шаблонов не содержит полей, то для нее не надо задавать строку переменных. Она отображается в точности так, как она задана в формате.

Таблица 8.1. Символы форматирования

Символ	Описание
>	Определяет символьное поле, в котором выводимое значение выровнено по правому краю
<	Определяет символьное поле, в котором выводимое значение выровнено по левому краю
	Определяет символьное поле, в котором выводимое значение выровнено по центру
#	Определяет числовое поле (выводимое значение должно быть числом)
.	Определяет положение десятичной точки в числовом поле (###.##)

Небольшой пример прольет свет на все вышесказанное — лучше один раз увидеть, чем сто раз услышать. Предположим, что у нас имеется файл (назовем его `books`), содержащий информацию о книгах, продаваемых неким книжным магазином. Каждая строка этого файла содержит информацию об одной книге: автор(ы), название, издательство, год выпуска и стоимость. Все поля записи разделены символом двоеточие ":". Одна из строк этого файла может выглядеть так:

В.Долженков Ю.Колесников:Excel 2000:BHV:1999:90

Нам необходимо распечатать отчет о всех продаваемых книгах. Воспользуемся форматами Perl. Программа примера 8.1 реализует поставленную задачу.

Заменим формат `STDOUT` программы примера 8.1 на следующий:

Теперь вывод нашей программы будет выглядеть так:

В.Долженков Ю.Колесников	Excel 2000	BHV	1999	90.00р.
А.Матросов А.Сергеев	HTML 4.0	BHV	1999	70.00р.
М.Чаунин				
Т.Кристиансен Н.Торкингтон	Perl	Питер	2000	100.00р.

Символ тильды "~" в конце строки шаблона подавляет вывод пустых строк. Если не поставить его, то между первой и второй книгой в нашем отчете появится дополнительная строка, как если бы была выведена вторая строка шаблона с пустым значением переменной `%author`. Символ подавления вывода пустых строк можно задавать в любом месте строки шаблона, помня, что при выводе он отображается, как пробел.

В нашем примере мы знали, что данные в переменной `$author` не займут более двух строк при выводе. Поэтому в формате мы использовали эту информацию, добавив еще одну строку шаблона с переменной `$author`. А что делать, если не известно количество строк продолжения в которых будут выводиться данные? Можно воспользоваться двумя идущими подряд символами тильда вместо одного. В этом случае алгоритм буферизации данных по словам будет продолжаться до завершения вывода всех данных переменной. Если наш формат изменить на следующий

```
format STDOUT =  
^<<<<<<<<<<<<<<<<<<<<<< | @>>>>>>>>>>>> | @||||||| | @#### | @###.##p.  
$author, $title, $pub, $year, $price
```

\$author

Вернемся к разработке формата для вывода нашего отчета. Пока что отчет был достаточно маленьким и помещался на одной странице. Реальные отчеты, конечно, на одной странице не поместятся. Наша программа напечатает и несколько страниц отчета. Дело в том, что создание отчетов в Perl предполагает их вывод на принтер, а поэтому после вывода определенного количества строк оператором `write()` Perl автоматически выведет символ перехода на новую страницу и печать продолжится на следующей странице. По умол-

чанию количество строк на странице установлено равным 60. Эта величина хранится в специальной переменной `$=`, значение которой может быть изменено в любое время.

Итак, мы теперь знаем, что переход на новую страницу происходит автоматически, но нам хотелось бы, чтобы на каждой странице печатался *верхний колонтитул*, в котором отображалось бы наименование отчета и печатались номера страниц. И это возможно в Perl. Следует только задать формат со специальным именем, добавив к имени формата, по которому мы выводим информацию (в нашей программе `STDOUT`), суффикс `_TOP`. Этот формат будет выводиться каждый раз, как начинается печать новой страницы.

Добавим в программу примера 8.1 следующее объявление формата

```
format STDOUT_TOP =
```

```

                                Книги на складе                                @>>>>>
                                "стр. " . $%
Автор                          Название      Издатель      Год      Цена
=====
```

и явно зададим количество строк на странице, добавив перед циклом `while` оператор

```
$= = 6;
```

Теперь наша программа напечатает две страницы отчета, причем на каждой из них будет напечатан колонтитул:

```

                                Книги на складе                                стр. 1
Автор                          Название      Издатель      Год      Цена
=====
В.Долженков Ю.Колесников      |   Excel 2000 |   BHV       |   1999 |   90.00p.
А.Матросов А.Сергеев          |   HTML 4.0  |   BHV       |   1999 |   70.00p.
М.Чаунин                      |              |             |       |
-----разрыв страницы-----
```

```

                                Книги на складе                                стр. 2
Автор                          Название      Издатель      Год      Цена
=====
Т.Кристиансен Н.Торкингтон    |   Perl      |   Питер     |   2000 |  100.00p.
```

Вернемся к объявлению формата для колонтитула. Во-первых, при его задании мы использовали выражение `"стр. " . $%` в строке переменных. Действительно, хотя формат и не вычисляется, но во время выполнения программы вычисляются значения переменных и все выражения строки переменных

формата. Во-вторых, мы использовали специальную переменную `$%`, которая хранит текущий номер выводимой страницы. Это позволило нам в колонтитуле напечатать номера страниц.

8.2. Использование нескольких форматов

Как мы уже знаем, форматы Perl позволяют без каких-либо усилий создавать верхние колонтитулы — следует только объявить формат с суффиксом `_TOP`. Для создания полноценного документа не мешало бы еще иметь возможность создавать нижние колонтитулы страницы и печатать, например, в конце заказа общую стоимость. К сожалению, такой возможности Perl не предоставляет, но он позволяет переключать вывод с одного формата на другой и в специальной переменной хранит строку, которую печатает перед переходом на новую страницу. А это и позволит нам создать и напечатать и нижний колонтитул, и общую стоимость заказа.

Но прежде мы еще немного поговорим о специальных переменных Perl, которые используются для управления форматом. В переменной `$~` хранится имя формата, который используется при выводе функцией `write()` без параметра:

```
write; # Эквивалентно оператору write STDOUT;
```

По умолчанию в ней хранится имя формата `STDOUT`, но и вывод функцией `write()` без параметра происходит на стандартное устройство вывода `STDOUT`. (Мы помним, что имя формата должно совпадать с именем дескриптора файла в вызове функции `write()`, а именно такая ситуация по умолчанию и реализуется.) Если мы изменим значение переменной `$~` на имя другого формата, то вывод в стандартный файл функцией `write()` без параметра будет осуществляться в соответствии с указанным форматом, который, конечно, должен быть объявлен в программе. Например, следующий оператор `write` выводит на стандартное устройство вывода в соответствии с форматом `NEW`:

```
$~ = NEW;
write;
format NEW =
. . . .
.
```

Таким образом, меняя значение переменной `$~`, можно переключать вывод с одного формата на другой. Этим другим форматом как раз и может быть формат общей стоимости заказа.

Пример 8.2. Заказ с итоговой суммой

```
format STDOUT_TOP =

```

Автор	Название	Издатель	Год	Цена	Заказ № @# \$number
=====					
.					

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<< | @>>>>>>>>> | @| | | | | | | | @#### | @###.##p.
$author, $title, $pub, $year, $price
^<<<<<<<<<<<<<<<<<<<<<<<<<< | | | | | | | | ~
$author
.
format TOTAL =
=====
Итого: @###.##p.
$total
=====
```

В этой программе после форматной печати содержимого файла books осуществляется переключение на другой формат, по которому выводится стро-

ка с общей суммой заказа, подсчитанной в переменной \$total. Полученный с помощью этой программы заказ показан ниже

				Заказ №	1
Автор	Название	Издатель	Год	Цена	
В.Долженков Ю.Колесников	Excel 2000	BHV	1999	90.00р.	
А.Матросов А.Сергеев	HTML 4.0	BHV	1999	70.00р.	
М.Чаунин					
Т.Кристиансен Н.Торкингтон	Perl	Питер	2000	100.00р.	
				Итого:	260.00р.

В завершение разговора о создании отчетов в Perl мы модифицируем программу примера 8.1, приспособив ее для печати отчета на основании информации о книгах из файла books, в котором в записи о книгах добавлено еще одно поле, содержащее краткую аннотацию книги:

В.Долженков Ю.Колесников:Excel 2000:BHV:1999:90:Аннотация книги

Отчет, формируемый этой программой (пример 8.3), также печатает нижний колонтитул на каждой странице. Для этого мы воспользуемся специальной переменной \$^L, содержимое которой Perl печатает перед переходом на новую страницу во время форматного вывода. При этом следует уменьшить на количество строк, заданных в этой переменной, количество строк на странице, хранящееся в специальной переменной \$=, иначе строки из переменной \$^L попадут не в конец текущей страницы, а будут напечатаны на следующей странице, не создав никакого нижнего колонтитула.

Пример 8.3. Полный отчет по книгам

```
#!/ perl -w
open FILE, "<books" or die $!;
open REPORT, ">report" or die $!;

select REPORT;
$~ = STDOUT;
$= = 24;

$ftime = localtime;
$^L = (" " x 73)."\n"."Книготорговая база \"БЕСЫ\"".
(" " x 24)."$ftime\n\f";
```

Вывод отчета осуществляется в файл с именем `herort`. Обратите внимание на задание переменной `$^L`. В ней используется переменная `$ltime`, в которой хранится текущая дата, полученная обращением к функции `localtime`. Одна страница отчета будет выглядеть следующим образом:

Книги на складе					стр. 1
Автор	Название	Издатель	Год	Цена	Аннотация
1. В.Долженков Ю.Колесников	Excel 2000	BHV	1999	90.00р.	Книга является справочным пособием по MS Excel 2000. В ней рассматриваются следующие основные темы - настройка интерфейса и его основные элементы.
2. А.Матросов А.Сергеев М.Чаунин	HTML 4.0	BHV	1999	70.00р.	Представлен весь спектр технологий создания Web-документов (начиная от простейших -

статических - и до
документов на
основе
динамического
HTML), включая
форматирование
текста, создание
списков.

Книготорговая база "БЕСЫ"

Sat Mar 18 19:01:37 2000

Замечание

Представленные в этой главе отчеты являются снятыми копиями экрана монитора, вывод на который осуществляется с использованием моноширинного шрифта. Если вывод осуществляется на принтер, то чтобы отчеты выглядели так, как они должны выглядеть, следует также использовать моноширинный шрифт, например Courier. Если используется пропорциональный шрифт, принятый на многих принтерах по умолчанию, то сформированные сценарием Perl отчеты "поползут", так как в этих шрифтах каждый символ имеет собственную ширину, тогда как в моноширинных все символы имеют одинаковую ширину.

* * *

Отчеты в Perl создаются с помощью форматов, определяющих внешний вид строк вывода и содержащуюся в них информацию. Печатаются отчеты функцией `write()`, которая осуществляет вывод как на стандартное устройство вывода, так и в файл, открытый в программе. В процессе печати отчета можно переключаться между существующими форматами.

Вопросы для самоконтроля

1. Опишите процедуру создания отчетов в Perl.
2. Какой синтаксис имеет оператор `format`?
3. Какая функция используется для активизации форматного вывода?
4. Как создается верхний колонтитул для страниц отчета?
5. Как создается нижний колонтитул для страниц отчета?
6. Каким образом осуществляется переключение между форматами?
7. Перечислите специальные переменные Perl, которые используются для управления форматным выводом.



Ссылки

Данные, используемые программой, размещаются в оперативной памяти компьютера. Каждая переменная имеет свой адрес и свое значение, которое хранится по этому адресу. Адрес переменной является информацией, которую также можно использовать в программе.

Ссылка на некоторую переменную содержит адрес этой переменной в оперативной памяти. Говорят, что ссылка *указывает* на переменную. Ссылки широко используются в современных языках программирования, таких как Pascal, C/C++. Вместо слова "ссылка" для обозначения термина может применяться слово "указатель". Основной областью применения ссылок является создание сложных структур данных, способных изменяться во время выполнения программы. Для ссылок используются специальные обозначения. В языке C это символ "*" перед именем переменной. В языке Pascal существует специальный тип данных для описания ссылок-переменных. Признаком этого типа является символ "^" перед идентификатором, описывающим базовый тип данных. Ссылка может быть переменной или константой.

Ссылка в языке Perl — это обычная скалярная величина, в которой хранится некоторый адрес в оперативной памяти.

9.1. Виды ссылок

Ссылка в языке Perl может указывать на любой фрагмент данных. Фрагментом данных здесь мы называем любую переменную, константу или часть кода программы. Тип ссылки определяется типом данных, на которые она указывает. Таким образом, существуют следующие типы ссылок: ссылка на скалярную величину, ссылка на массив, ссылка на хеш, ссылка на функцию. Нельзя использовать ссылку одного типа там, где контекст выражения требует присутствия ссылки другого типа, например, использовать ссылку на массив вместо ссылки на хеш-массив. Поскольку сама ссылка является скалярной величиной, то, естественно, существует ссылка на ссылку. Имеется еще один вид ссылок, который мы в свое время рассмотрим подробнее. Это ссылки на данные типа `typeglob`. Тип `typeglob` является внутренним типом языка Perl, который служит для обозначения переменных разных типов, имеющих общее имя. Принадлежность к типу `typeglob`, обозначается пре-

фиксом `"*"`. Например, запись `*abc` обозначает всю совокупность, а также любую из следующих переменных: скаляр `$abc`, массив `@abc`, хеш `%abc`. В данной главе мы не будем рассматривать этот вид ссылок. Отметим только, что он лежит в основе механизма экспорта/импорта модулей.

(Работа с модулями обсуждается в главе 12.)

Тип ссылки можно определить при помощи встроенной функции `ref()`, которая рассматривает свой аргумент как ссылку и возвращает символическое обозначение ее типа. Если аргумент не является ссылкой, возвращается пустая строка. Встроенные типы обозначаются следующим образом:

REF	ссылка на ссылку;
SCALAR	ссылка на скаляр;
ARRAY	ссылка на массив;
HASH	ссылка на ассоциативный массив;
CODE	ссылка на подпрограмму;
GLOB	ссылка на переменную типа <code>typeglob</code> .

Ссылки в языке Perl бывают *жесткие* и *символические*. Понятия "жесткая ссылка" и "символическая ссылка" вместе с названиями проникли в Perl из мира UNIX, где они используются применительно к файловой системе. Разберем их применение в UNIX, чтобы лучше понимать, для чего они нужны в Perl.

Каждому файлу в UNIX соответствует *индексный дескриптор* — структура данных, имеющая определенный формат, расположенная в специально отведенной области диска и содержащая важнейшую информацию о файле: тип файла, его расположение на диске, права доступа и т. д. Каждый дескриптор имеет числовой номер, соответствующий его положению в таблице индексных дескрипторов. Этот номер и является внутренним именем файла для операционной системы. Для нее сущность файла заключается в его индексном дескрипторе, а не в его содержимом. Для пользователя, напротив, важно содержимое файла, а о существовании индексного дескриптора он может даже не подозревать. Кроме того, пользователю удобнее работать с именем файла, а не с числовым номером. Для удобства пользователя создается ссылка на файл, которая ставит в соответствие индексному дескриптору имя файла. Ссылка представляет собой запись в каталоге, который является тоже файлом, выполняющим специальную функцию регистрации других файлов. В простейшем случае эта запись содержит два поля: имя файла и номер индексного дескриптора. Можно создать несколько ссылок с разными именами в одном или нескольких каталогах, указывающих на один файл. Ссылка указывает на индексный дескриптор, но для краткости говорят о ссылке на файл. Следует подчеркнуть, что все ссылки равноправны. Ссылка, созданная первой, не имеет никакого преимущества перед ссылкой-

ми, созданными позднее. В индексном дескрипторе среди другой важной информации содержится счетчик ссылок. Удаление из каталога ссылки на файл уменьшает значение счетчика ссылок на единицу. Когда значение счетчика ссылок становится равным нулю, файл удаляется, а его индексный дескриптор освобождается для использования новым файлом.

Рассмотренные ссылки называются *жесткими* ссылками. Кроме них, существуют *символические* ссылки. Символическая ссылка является не просто записью в каталоге, а файлом особого типа, содержащим символьное имя другого файла. В качестве файла символьная ссылка имеет свой индексный дескриптор. Поскольку символическая ссылка является самостоятельным файлом, ее существование никак не отмечается в дескрипторе того файла, на который она указывает. В частности, если удалить все символические ссылки на файл, то это не будет означать его удаление.

Жесткие и символические ссылки в языке Perl напоминают одноименные понятия в файловой системе UNIX. Жесткая ссылка в Perl — это скалярная величина, которая содержит адрес некоторой области памяти, являющейся носителем данных. Сами данные будем называть *субъектом* ссылки. Символическая ссылка — это скалярная переменная, которая содержит имя другой скалярной переменной. "Истинной" ссылкой является жесткая ссылка. Она создается одним из способов, перечисленных в разделе 9.1. Внутренняя организация жестких ссылок такова, что для каждого субъекта ссылки поддерживается счетчик ссылок. Область памяти, занимаемая субъектом ссылки, освобождается, когда значение счетчика ссылок становится равным нулю. В большинстве случаев мы имеем дело с жесткими ссылками, а использование символических ссылок будем специально оговаривать.

Главным применением ссылок в языке Perl является создание сложных структур данных. Мы знаем, что основными типами данных в Perl являются скаляры, массивы и хеш-массивы. Многомерные массивы или более сложные структуры данных, аналогичные записям языка Pascal или структурам языка C, в языке Perl отсутствуют. В более ранних версиях языка отсутствовала и возможность создания сложных структур данных на основе имеющихся типов. Такая возможность появилась в версии Perl 5.0 вместе с появлением ссылок. В практике программирования часто встречаются данные, которые удобно представлять, например, в виде двумерных массивов, реже трехмерных массивов или других подобных структур. Двумерный массив можно рассматривать как одномерный массив, элементами которого являются также одномерные массивы. Возможность такого представления есть во многих языках программирования. В языке Perl невозможно создать массив с массивами в качестве элементов. То же самое относится и к хешам. Элементом массива или хеша может быть только скалярная величина. Поскольку ссылка является скалярной величиной, можно создать массив или хеш, элементами которого являются ссылки на массивы или хеши, и таким образом получить структуру, которую можно использовать как массив мас-

сиров (соответственно массив хешей, хеш массивов, хеш хешей). Благодаря ссылкам можно на основе массивов и хешей конструировать структуры данных произвольной сложности.

Помимо создания сложных структур данных, ссылки активно применяются для работы с объектами. Слово "объект" здесь обозначает основное понятие объектно-ориентированного подхода к программированию.

(Объекты рассматриваются в главе 13.)

В этой главе мы рассмотрим основное применение ссылок как средства для конструирования структур данных. Другие применения будут рассмотрены в соответствующих главах.

9.2. Создание ссылок

Существует несколько способов порождения ссылок. Рассмотрим их в порядке следования от чаще употребляемых синтаксических конструкций к более редким.

9.2.1. Операция ссылки "\"

Операция "\", примененная к единственному аргументу, создает ссылку на этот аргумент. В качестве последнего может выступать переменная любого типа или константа. Примеры:

```
$a=\5;
$scal_ref=$a;
$arr_ref=\@myarray;
$hash_ref=\%myhash;
$func_ref=\&myfunc;
```

В данном примере скалярной переменной `$a` присваивается значение ссылки на константу 5, т. е. адрес ячейки памяти, в которой хранится число 5. Адрес самой переменной `$a` хранится в переменной `$scal_ref`. Переменные `$arr_ref`, `$hash_ref`, `$func_ref` хранят адреса ячеек памяти, являющихся начальными точками размещения соответственно массива `@myarray`, хеш-массива `%myhash` и кода функции `myfunc`. К переменным, содержащим ссылки, можно применять все операции допустимые для скалярных величин. Их можно присваивать другим переменным, складывать, умножать, делить, выводить на экран и т. д. За исключением присваивания применение подобных операций к ссылкам, как правило, смысла не имеет. Например, вывод рассмотренных выше переменных

```
print $scal_ref, "\n", $arr_ref, "\n", $hash_ref, "\n", $func_ref, "\n";
```

будет состоять из строк, подобных следующим:


```
SCALAR (0x9b8994)
ARRAY (0x9b8a18)
HASH (0x9b8a60)
CODE (0x9b3d14)
```

Здесь каждая строка содержит слово, обозначающее тип ссылки и ее значение — адрес в виде шестнадцатеричного числа.

Операция, которую действительно имеет смысл применять к ссылкам, это операция *разыменования*, то есть операция получения того значения, на которое указывает ссылка. Синтаксические конструкции, используемые для разыменования ссылок, мы рассмотрим после того, как обсудим способы их создания.

9.2.2. Конструктор анонимного массива

В рассмотренном выше примере операция "`\`" применялась к переменным, обладающим именами. Perl позволяет создавать ссылки на анонимные массивы при помощи специальной конструкции, использующей квадратные скобки:

```
$arr_ref = [1,2,3];
```

В результате данной операции присваивания будет создан анонимный массив с элементами (1,2,3), а переменной `$arr_ref` будет присвоено значение ссылки на этот массив.

Компилятор различает случаи использования квадратных скобок для создания ссылки на анонимный массив и для обращения к отдельным элементам массива, как, например, в операции присваивания `$a = $myarray[2]`.

Замечание

Свободный синтаксис языка Perl допускает существование конструкций, смысл которых не очевиден. К рассматриваемой теме имеет отношение следующий пример. Формально выражение `\($a, $b, $c)` представляет собой анонимный массив из трех элементов (`$a`, `$b`, `$c`), к которому применяется операция ссылки "`\`". Означает ли это, что значением выражения является ссылка на анонимный массив? Нет, это просто сокращенная запись массива, состоящего из трех элементов-ссылок (`\$a`, `\$b`, `\$c`), а для создания ссылки на анонимный массив существует единственный способ, рассмотренный выше.

9.2.3. Конструктор анонимного ассоциативного массива

По аналогии с массивами можно создавать ссылки на анонимные ассоциативные массивы, используя фигурные скобки. Операция присваивания

```
%hash_ref = {
    'One'=>1,
    'Two'=>2,
    'Three'=>3
};
```

создаст анонимный хеш-массив ('One'=>1, 'Two'=>2, 'Three'=>3) и присвоит переменной %hash_ref значение ссылки на этот хеш.

Фигурные скобки используются во многих конструкциях, например, для обращения к индивидуальному элементу хеш-массива

```
$a = $myhash{"first"}
```

или для выделения блока операторов. Обычно такие случаи легко различимы, и их нельзя спутать с порождением ссылки на анонимный хеш. Но иногда возникают неоднозначные ситуации, требующие разрешения. Забегая вперед, приведем пример, связанный с определением функции пользователем.

(Желающие могут предварительно прочитать начало главы 11, в которой рассказывается о подпрограммах и функциях.)

Предположим, что необходимо определить функцию, которая создает анонимный хеш и возвращает ссылку на него. Возвращаемое значение можно задать при помощи встроенной функции return. Если конструкция return отсутствует, то в качестве возвращаемого значения по умолчанию принимается значение последнего выражения, вычисленного внутри функции. Таким образом, синтаксически допустимо следующее определение функции

```
sub get_hash_ref { { @_ } }
```

В данном примере внутренняя конструкция в фигурных скобках интерпретируется как блок. Для того чтобы она интерпретировалась как ссылка на анонимный хеш, необходимо использовать функцию return или поставить перед внутренней конструкцией знак "+":

```
sub get_hash_ref { return { @_ } }
sub get_hash_ref { +{ @_ } }
```

9.2.4. Другие способы

В предыдущих разделах рассмотрены основные способы создания ссылок:

- ☐ применение операции "\" к объекту ссылки;
- ☐ специальные конструкции [] и { }, создающие в определенном контексте ссылку соответственно на анонимный массив и анонимный ассоциативный массив.

Эти способы применяются наиболее часто в тех случаях, когда возникает необходимость в использовании ссылок. Но существуют и другие источники появления ссылок, о которых следует упомянуть для полноты изложения.

9.2.4.1. Конструктор анонимной подпрограммы

Мы уже использовали в примерах подпрограммы, не дожидаясь их систематического изучения. Поэтому можем рассмотреть в этой главе такой вид ссылки, как ссылка на анонимную подпрограмму.

Ссылка на анонимную подпрограмму может быть создана при помощи ключевого слова `sub`, за которым следует блок — последовательность операторов, заключенная в фигурные скобки:

```
$sub_ref = sub { print "Привет!\n"};
```

В результате операции присваивания в переменную `$sub_ref` заносится адрес, по которому размещается код анонимной подпрограммы. В данном примере подпрограмма состоит из единственного обращения к функции `print`, выводящей строку "Привет!".

Пример, иллюстрирующий данный вид ссылки, будет рассмотрен далее в этой главе.

9.2.4.2. Ссылка, создаваемая конструктором объекта

В версию 5.0 языка Perl была добавлена поддержка объектно-ориентированного программирования. Основой объектно-ориентированного подхода являются понятия *класс* и *объект*.

(Классы и объекты рассматриваются в главе 13.)

Понятие "объект" реализовано в языке Perl таким образом, что объект становится доступным в программе только через ссылку на него. Для создания объекта используется специальная подпрограмма — *конструктор*, которая, в свою очередь, применяет для этого встроенную функцию `bless()`. Конструктор возвращает ссылку на объект. Таким образом, это еще один способ порождения ссылок, без которого не обойтись тем, кто использует объектно-ориентированный подход в Perl.

9.2.4.3. Ссылки на данные типа `typeglob`

Компилятор Perl хранит имена всех переменных программы в *таблице символов*. Отдельная таблица символов существует для каждого *пакета*, образуя собственное пространство имен.

(О пакетах и таблицах символов рассказывается в главе 12.)

Каждый идентификатор, встречающийся в пакете, заносится в таблицу символов. Одинаковые идентификаторы, соответствующие переменным разных

типов, образуют гнездо, в котором каждому типу соответствует свой элемент, содержащий адрес переменной данного типа. Если, например, в программе имеются следующие строки

```
$a=5;
@a=(1,2,3,4,5);
%a=("one"=>1,"two"=>2,"three"=>3);
sub a {return "Hello, Mike!";};
```

то таблица символов содержит гнездо для идентификатора "a", состоящее из четырех элементов, хранящих адреса: скалярной переменной \$a, массива @a, ассоциативного массива %a и кода подпрограммы &a.

В языке Perl существует внутренний тип данных `typeglob`. Признаком этого типа является наличие префикса "*" в имени переменной. Тип `typeglob` служит для ссылки на все переменные разных типов с одинаковыми именами. Например, переменная *a обозначает ссылку на гнездо "a" в таблице символов. Используя специальную запись, можно при помощи переменной `typeglob` получить ссылки на отдельные элементы гнезда:

```
$scalarref = *a{SCALAR}; # эквивалентно $scalarref = \ $a;
$arrayref   = *a{ARRAY};  # эквивалентно $arrayref = \ @a;
$hashref    = *a{HASH};   # эквивалентно $hashref = \ %a;
$coderef    = *a{CODE};   # эквивалентно $coderef = \ &a;
$globref    = *a{GLOB};   # эквивалентно $globref = \ *a;
```

9.2.4.4. Неявное создание ссылок

В рассмотренных случаях осуществляется явное создание ссылки при помощи операции "\ " или специальных синтаксических конструкций. В них всегда явным образом определяется скалярная переменная, в которую и заносится значение ссылки. Ссылка может также создаваться неявно в случае, когда операция разыменования применяется к ссылке, ранее не созданной в программе явным образом, и в контексте выражения предполагается, что такая ссылка должна существовать. Возможно, последнее предложение звучит не совсем понятно. Его смысл станет ясным в следующем разделе.

9.3. Разыменование ссылок

Разыменованием ссылки называется получение объекта, на который указывает эта ссылка. Для разыменования, как и для создания ссылки, применяются различные синтаксические конструкции, подчас достаточно сложные для визуального восприятия. К ним нужно привыкнуть. Вид конструкции зависит от типа ссылки, к которой применяется разыменование. Рассмотрим их по степени возрастания сложности.

9.3.1. Разыменование простой скалярной переменной

Если ссылка на некоторый объект: скалярную переменную, массив, ассоциативный массив и т. д., является простой скалярной переменной без индексов, то для обращения к самому объекту применяется правило: вместо имени объекта подставить в выражение простую скалярную переменную, содержащую ссылку. Например:

```
1      $a = $$scal_ref;  
2      @b = @$arr_ref;  
3      %c = %$hash_ref;  
4      &f = &$code_ref;  
5      $$d[0] = 7;  
6      $h{"one"} = 1;
```

Здесь предполагается, что переменная `$scal_ref` содержит ссылку на скалярную величину, `$arr_ref` — ссылку на массив, `$hash_ref` — ссылку на ассоциативный массив, `$code_ref` — ссылку на подпрограмму.

Рассмотрим подробно пятую строку.

Во-первых, следует определить, что является ссылкой: скалярная переменная `$d`, указывающая на *анонимный* массив, или элемент `$d[0]` массива `@d`. Ответ содержится в сформулированном выше правиле разыменования. Поскольку в строке 5 применяется именно оно, то индексированная переменная `$d[0]` ссылкой быть не может. Ссылкой является простая скалярная переменная `$d`, которая используется в качестве имени. Из контекста видно, что на ее месте должно стоять имя массива, следовательно, `$d` является ссылкой на анонимный массив

Во-вторых, здесь мы имеем пример неявного создания ссылки, о котором говорилось в предыдущем разделе. Ссылка `$d` не была ранее создана явным образом, но ее существование предполагается в операции присваивания. Поэтому компилятор создаст ссылку `$d` на анонимный массив, поместит в нее адрес массива и по этому адресу сохранит значение первого элемента, равное 7.

Все сказанное можно отнести к шестой строке с единственным отличием: вместо ссылки на анонимный массив здесь фигурирует ссылка `$h` на анонимный хеш-массив.

9.3.2. Блоки в операциях разыменования ссылок

Если ссылка является не простой скалярной переменной, а, например, элементом массива или ассоциативного массива, то для ее разыменования нельзя применить правило предыдущего раздела. В этом случае следует за-

ключить ссылку в фигурные скобки и полученный блок использовать в качестве имени переменной в выражениях. Вообще, во всех случаях разыменования ссылок в качестве имени объекта можно использовать блок, результатом выполнения которого является ссылка соответствующего типа.

```
${$d[0]} = 7;  
${$h{"one"}} = 1;  
${&f()}[1] = 3;
```

Разберем первую строку. Начальный символ `$` является признаком скалярной переменной, за которым должно следовать ее имя. Вместо имени используется блок, следовательно, выражение внутри блока интерпретируется как ссылка. В данном случае осуществляется разыменование ссылки `$d[0]`, являющейся элементом массива `@d`. Аналогично, во второй строке осуществляется обращение к скалярной переменной, на которую указывает ссылка `$h{"one"}`, являющаяся элементом ассоциативного массива `%h`. В третьей строке блок, возвращающий ссылку, состоит из одного обращения к функции `f()`. Ее значение интерпретируется как ссылка на массив, и второму элементу этого массива присваивается значение 3.

9.3.3. Операция разыменования "->"

Применение правила разыменования предыдущего раздела может привести к появлению громоздких выражений, содержащих множество вложенных друг в друга блоков, и очень сложных для визуального восприятия. Непростыми являются уже вышеприведенные примеры. При построении же более сложных структур выражения становятся почти необозримыми. Даже достаточно простые конструкции требуют определенного усилия для того, чтобы понять, что они означают:

```
${$a[0]}[1] = 17;  
${$b[0]}{"one"} = 1;
```

В первой строке осуществляется обращение к отдельному элементу массива массивов, во второй — к отдельному элементу массива хеш-массивов.

Замечание

В действительности речь идет соответственно о массиве, элементами которого являются *ссылки* на анонимные массивы и о массиве, элементами которого являются *ссылки* на анонимные хеш-массивы. Но для краткости в подобных случаях мы будем употреблять сочетания "массив массивов", "массив хеш-массивов" и т. д.

Несколько упростить запись и улучшить наглядность можно, используя операцию "->" ("стрелка").

Аргумент слева от стрелки может быть любым выражением, возвращающим ссылку на массив или хеш-массив.

Если левосторонний аргумент является ссылкой на массив, то аргумент справа от стрелки — *индекс*, заключенный в квадратные скобки и определяющий элемент этого массива.

Если левосторонний аргумент является ссылкой на хеш-массив, то аргумент справа от стрелки — значение *ключа*, помещенное в фигурные скобки и определяющее элемент этого хеш-массива.

Результатом операции " \rightarrow " является соответственно *значение элемента* массива или хеш-массива. Предыдущий пример можно более компактно записать в виде

```
$a[0]→[1] = 17;  
$b[0]→{"one"} = 1;
```

Конструкция $\$a[0]→[1]$ обозначает второй элемент массива, определяемого ссылкой $\$a[0]$. Конструкция $\$b[0]→\{"one"\}$ обозначает элемент, соответствующий ключу "one" хеш-массива, задаваемого ссылкой $\$b[0]$.

Вообще, если $\$arr_ref$ — ссылка на массив, то $\$arr_ref→[\$i]$ обозначает i -й элемент этого массива. Если $\$hash_ref$ — ссылка на хеш-массив, то $\$hash_ref→\{"key"\}$ обозначает элемент этого хеш-массива, соответствующий ключу "key".

Если бы в последнем примере вместо именованных массивов @a и @b использовались ссылки на массив, например, $\$ref_a$ и $\$ref_b$, то соответствующие операции присваивания имели вид

```
$ref_a→[0]→[1] = 17;  
$ref_b→[0]→{"one"} = 1;
```

Здесь мы снова сталкиваемся с неявным созданием ссылок. По контексту элемент массива $\$ref_a→[0]$ должен быть ссылкой на массив, а $\$ref_b→[0]$ — ссылкой на хеш-массив. Обе ссылки ранее не были определены, но их существование предполагается в контексте выражения. Данные ссылки будут созданы автоматически.

Операция " \rightarrow " позволяет для обращения к отдельному элементу составного массива или хеш-массива использовать более простые выражения, например,

```
$a[$i]→[$j]→[$k] вместо ${${$a[$i]}[$j]}[$k],  
$b[$i]→{"key"}→[$j] вместо ${${$b[0]}{"key"}}[$j]
```

и т. д.

Дальнейшее упрощение связано с тем, что при обращении к элементам сложных структур, представляющих собой комбинации вложенных массивов

и хеш-массивов, можно опустить символы "->" между квадратными и/или фигурными скобками, содержащими индексы или ключи элементов. Предыдущие выражения примут еще более простой вид: `$a[$i][$j][$k]` и `$b{$i}{"key"}[$j]` соответственно.

9.4. Символические ссылки

Из предыдущего раздела мы знаем, что если ссылка не определена, но ее присутствие требуется контекстом, то она создается автоматически.

Если же определенная ранее скалярная величина не является ссылкой, но используется в качестве ссылки, то ее называют *символической* ссылкой. Значение символической ссылки интерпретируется как имя некоторой переменной. Над этой переменной будут выполняться все операции, применяемые к символической ссылке. Вспомним, что значением жесткой ссылки является адрес. В следующем примере переменная `$name_a` используется как символическая ссылка на переменную `$a`.

```

1  $name_a = "a";
2  $$name_a = 17;
3  @$name_a = (1,2,3);
4  $name_a->[3] = 4;
5  %$name_a = ("one"=>1, "two"=>2, "three"=>3);
6  &$name_a();

```

В строке 2 переменной `$a` присваивается значение 17. В строке 3 определяется и инициализируется массив `@a` с элементами (1,2,3). В строке 4 к массиву `@a` добавляется четвертый элемент со значением 4. В строке 5 инициализируется хеш-массив `%a`. В строке 6 осуществляется вызов функции `a()` (предположим, что такая функция существует).

Символическая ссылка может указывать только на переменную, имя которой содержится в таблице символов пакета.

(О пакетах и таблицах символов описано в главе 12.)

Лексические переменные, определяемые при помощи функции `my()`, в таблицу символов не входят, поэтому их имена невидимы для механизма, реализующего символические ссылки.

(О лексических переменных и применении функции `my()` рассказывается в главе 11.)

Для иллюстрации рассмотрим следующий пример:

```

$name_a="a";
{
    my $a="Hello!";

```



```
print $$name_a;  
};
```

Здесь переменная `$name_a` используется в качестве символической ссылки на переменную `$a`, и можно предположить, что результатом выполнения этой последовательности будет вывод строки "Hello!". В действительности переменная `$a` является невидимой для символической ссылки, поскольку она определена как лексическая переменная внутри блока {...}. Поэтому в результате выполнения данного фрагмента будет напечатана пустая строка.

Применение символических ссылок является потенциально опасным из-за возможности возникновения смысловых ошибок. Например, может показаться, что в результате выполнения следующей последовательности операторов

```
1      $a[0]="b";  
2      #.....  
3      $b[0]=2;  
4      $b[1]=2;  
5      #.....  
6      $a[0][0]=0;  
7      #.....  
8      $prod = $b[0]*$b[1];
```

переменная `$prod` получит значение 4. Но это не так. В строке 6 мы осуществляем присваивание, рассчитывая на то, что будет применен известный механизм неявного создания жесткой ссылки `$a[0]`. Мы "забыли" о том, что значение `$a[0]` уже использовалось в строке 1 и, следовательно, в строке 6 элемент массива `$a[0]` является символической ссылкой, указывающей на переменную с именем "b". Это имя будет подставлено вместо символической ссылки, в результате чего элемент массива `b[0]` получит новое значение 0. В итоге значение переменной `$prod` будет равно 0.

Во избежание подобных ошибок можно запретить использование символических ссылок в пределах текущего блока при помощи директивы

```
use strict 'refs';
```

Это ограничение, если требуется, можно отменить для внутреннего блока при помощи другой директивы

```
no strict 'refs';
```

(Директивы use, no рассматриваются в главе 12.)

Еще одно замечание, касающееся символических ссылок. В версии 5.001 появилась новая возможность: если переменную, являющуюся символической ссылкой, заключить в фигурные скобки, то такая конструкция интер-

претируется не как символическая ссылка, а как значение переменной, подобно тому, как аналогичная конструкция интерпретируется командной оболочкой shell операционной системы UNIX. В следующем фрагменте

```
1      use strict 'refs';
2      ${name};
3      ${"name"};
```

вторая строка представляет собой просто значение переменной `$name`, а третья строка интерпретируется как символическая ссылка, указывающая на переменную `$name` и вследствие применения директивы `use strict 'refs'` вызывает сообщение об ошибке вида

```
Can't use string ("name") as a SCALAR ref while "strict refs" in use
```

9.5. Использование ссылок

В данном разделе мы рассмотрим некоторые примеры, связанные с основным применением ссылок — конструированием структур данных.

В качестве первой структуры построим массив массивов или двумерный массив. Для примера рассмотрим массив `@calendar`, содержащий календарь, например, на 2000 год. Значением элемента `$calendar[$i][$j]` является название дня недели, приходящегося на $(j+1)$ -й день $(i+1)$ -го месяца, $i=(0..11)$, $j=(0..30)$ (рис. 9.1).

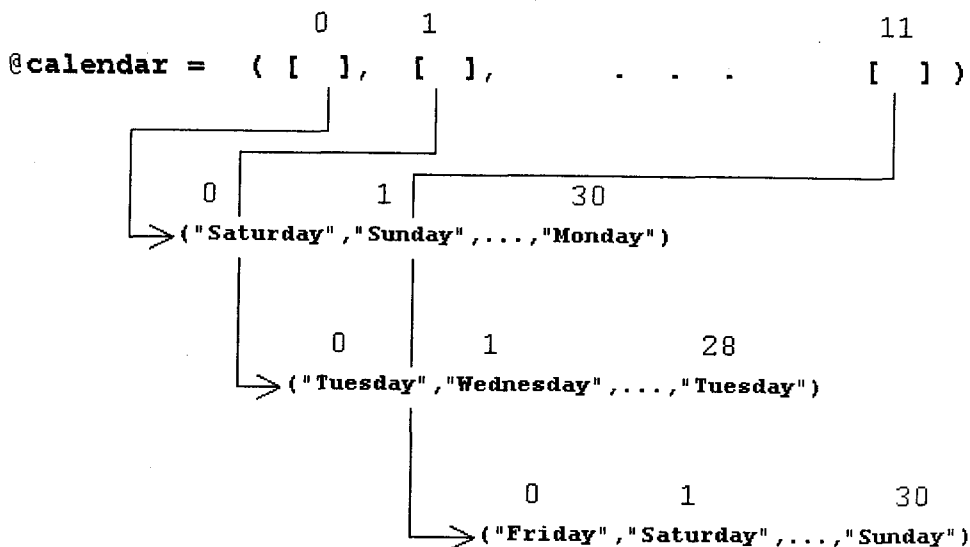


Рис. 9.1. Структура массива `@calendar`

9.5.1. Замыкания

Для заполнения массива @calendar нам потребуется функция, которая по заданному году, месяцу и дню месяца вычисляет соответствующий день недели. Читатель может сам написать свой вариант функции, может быть, более изящный. Мы предлагаем наш вариант (пример 9.1) с основной целью: рассказать об одном интересном свойстве анонимных подпрограмм. Кроме того, он правильно работает для любого года нашей эры.

Вычисление дня недели основано на том, что:

- 1 января 1 года нашей эры было понедельником;
- каждый год, номер которого делится на 4, является високосным, за исключением тех номеров, которые делятся на 100 и не делятся на 4.

(Вопросы создания функций пользователем рассмотрены в главе 11.)

Пример 9.1. Программа-календарь

```
sub GetDay {
    my $year = shift;
    my @days = (0,31,59,90,120,151,181,212,243,273,304,334);
    my @week = ("Monday","Tuesday","Wednesday","Thursday",
                "Friday","Saturday","Sunday");
    my $previous_years_days = ($year - 1 ) * 365 + int(($year-1)/4)
                             - int(($year-1)/100) + int(($year-1)/400);
    return sub { my ($month, $day)=@_;
                  my $n = $previous_years_days + $days[$month-1] + $day - 1;
                  $n++ if ($year%4 == 0 and $year%100 != 0 or
                           $year%400 == 0 and $month > 2);
                  return $week[$n%7];
                }
};
```

Аргументами функции GetDay() являются номер года, номер месяца и номер дня месяца. Внутри тела функции им соответствуют переменные \$year, \$month и \$day. Функция подсчитывает число дней \$n, прошедших с 1 января 1 года. Остаток от деления этого числа на 7 — \$n%7 — определяет день недели как элемент массива \$week[\$n%7].

Необходимые пояснения к тексту

Для передачи параметров в подпрограмму используется предопределенный массив @_. Встроенная функция shift() без параметров, вызванная внутри подпрограммы, возвращает первый элемент массива @_ и осуществляет

сдвиг всего массива влево, так, что первый элемент пропадает, второй становится первым и т.д. Элемент массива `$days[$i]` равен суммарному числу дней в первых `i` месяцах не високосного года, `i = (0..11)`. В переменной `$previous_years_days` запоминается вычисленное значение общего количества дней, прошедших с 1 января 1 года до начала заданного года.

Обратите внимание на то, что значением функции `GetDay()` является не название дня недели, а ссылка на анонимную функцию, которая возвращает название дня недели. Объясним, зачем это сделано.

Если бы функция `GetDay()` возвращала день недели, то для заполнения календаря на 2000 год, к ней необходимо было бы сделать 366 обращений, вычисляя каждый раз значение переменной `$previous_years_days`. Для каждого года это значение постоянно, поэтому его достаточно вычислить всего один, а не 366 раз.

На время вычисления функции формируется ее *вычислительное окружение*, включающее совокупность действующих переменных с их значениями. После завершения вычисления функции ее вычислительное окружение пропадает, и на него невозможно сослаться позже. Часто бывает полезным, чтобы функция для продолжения вычислений могла запомнить свое вычислительное окружение. В нашем примере полезно было бы запомнить значение переменной `$previous_years_days`, чтобы не вычислять его повторно. В языках программирования существует понятие *замыкание*, пришедшее из языка Lisp. Это понятие обозначает совокупность, состоящую из самой функции как описания процесса вычислений и ее вычислительного окружения в момент определения функции.

Анонимные процедуры в Perl обладают тем свойством, что по отношению к лексическим переменным, объявленным при помощи функции `my()`, выступают в роли замыканий. Иными словами, если определить анонимную функцию в некоторый момент времени при некоторых значениях лексических переменных, то в дальнейшем при вызове этой функции ей будут доступны значения этих лексических переменных, существовавшие на момент ее определения.

В нашем примере указанное свойство анонимных функций используем следующим образом. Чтобы анонимной функцией можно было воспользоваться в дальнейшем, присвоим ссылку на нее скалярной переменной:

```
$f = GetDay(2000,1,1);
```

Во время обращения к `GetDay()` было сформировано вычислительное окружение анонимной функции, на которую сейчас указывает переменная `$f`. Вычислительное окружение включает, в том числе, и переменную `$previous_years_days` с ее значением. Обратите внимание, что внутри анонимной функции значение этой переменной не вычисляется. В дальнейшем для заполнения календаря мы будем вызывать анонимную функцию через ссылку `$f`.

9.5.2. Массив массивов

Сформируем массив @calendar, используя результаты предыдущего раздела.

Пример 9.2. Формирование массива массивов

```
for $i (1,3..12) {  
    for $j (1..30) {  
        $calendar[$i-1][$j-1] = &$f($i, $j);  
    }  
};  
  
for $i (1,3,5,7,8,10,12) {  
    $calendar[$i-1][30] = &$f($i, 31);  
};  
  
for $j (1..28) {  
    $calendar[1][$j-1] = &$f(2, $j);  
};  
  
# Если год високосный, то добавляется еще один элемент массива  
$calendar[1][28] = &$f(2,29);
```

Массив @calendar состоит из 12 элементов по числу месяцев в году. Каждый элемент массива является ссылкой на другой массив, имеющий столько элементов, сколько дней в соответствующем месяце. Значениями элементов вложенных массивов являются английские названия соответствующих дней недели: "Monday", "Tuesday" и т. д.

Обращаем внимание на то, что при формировании массива @calendar осуществляется неявное создание ссылок \$calendar[\$i] и применяется компактная запись \$calendar[\$i][\$j] для обозначения индивидуального элемента двумерного массива, обсуждавшаяся в разделе 9.3.3.

Содержимое массива @calendar можно вывести для просмотра при помощи следующих операторов:

```
for $i (0..11) {  
    for $j (0..${$calendar[$i]}) {  
        print $j+1, ".", $i+1, " is $calendar[$i][$j]\n";  
    }  
};
```

Напомним, что запись \$#array обозначает верхнее значение индекса массива @array. В результате выполнения данного цикла будет выведена длинная последовательность строк вида

1.1 is Saturday

2.1 is Sunday

.....

9.5.3. Другие структуры данных

На основе массива `@calendar`, содержащего календарь на 2000 год, покажем, как можно строить более сложные структуры данных. Структура двумерного массива не очень удобна для представления содержащихся в ней данных в привычном виде настенного календаря. Перегруппируем данные, объединяя их в группы по дням недели. Для этого построим новую структуру, которую для краткости назовем "массив хешей массивов", отдавая себе отчет в том, что такое словосочетание не только далеко не изяшно, но и по существу неточно.

Новая структура представляет собой массив `@months`, состоящий из 12 элементов по числу месяцев в году. Каждый элемент содержит ссылку на анонимный хеш-массив. Каждый вложенный хеш-массив содержит набор ключей, имеющих имена, совпадающие с английскими названиями дней недели: "Monday", "Tuesday" и т. д. Каждому ключу соответствует значение, являющееся, в свою очередь, ссылкой на анонимный массив, содержащий все числа данного месяца, приходящиеся на день недели, соответствующий ключу: все понедельники, все вторники и т. д. Структура массива `@months` представлена на рис. 9.2.

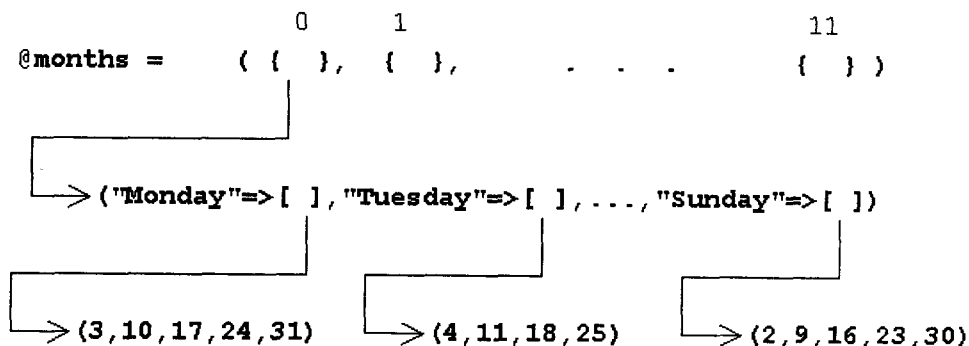


Рис. 9.2. Структура массива `@months`

Пример 9.3. Формирование массива хешей массивов

```

for $i (0..11) {
    for $j (0..${#{$calendar[$i]}}) {

```

```
    push @{$months[$i]{$calendar[$i][$j]}}, $j+1;
}
};
```

Замечание

Функция `push @array, list` помещает список `list` в конец массива `@array`.

Первым аргументом встроенной функции `push` является массив, в который попадают все дни ($i+1$)-го месяца, приходящиеся на один и тот же день недели: все понедельники, все вторники и т. д. На этот массив указывает ссылка `$months[$i]{"key"}`, где ключ `"key"` принимает значения `"Monday"`, `"Tuesday"` и т. д. Для обращения к самому массиву ссылку следует разыменовать, заключив в фигурные скобки: `@{$months[$i]{"key"}}`. Если вместо ключа `"key"` подставить нужное значение из `$calendar[$i][$j]`, то получим аргумент функции `push`.

Вновь сформированную структуру удобно использовать для вывода календаря в традиционном виде. Последовательность операторов

```
for $i (0..11) {
    print "month # ", $i+1, "\n";
    for $DayName (keys %{$months[$i]}) {
        print "    ${DayName}:          @{$months[$i]{$DayName}}\n";
    }
};
```

распечатает календарь в виде

month # 1	
Monday	3 10 17 24 31
Thursday	6 13 20 27
Wednesday	5 12 19 26
Sunday	2 9 16 23 30
Saturday	1 8 15 22 29
Friday	7 14 21 28
Tuesday	4 11 18 25
...	...

Встроенная функция `keys %hash` возвращает список всех ключей ассоциативного массива `%hash`. Вывод ключей осуществляется функцией `keys` в случайном порядке, поэтому дни недели расположены не в естественной последовательности, а случайным образом.

Для вывода ключей в порядке следования дней недели воспользуемся встроенной функцией сортировки

```
sort SUBNAME LIST
```

Замечание

Функция `sort()` сортирует список `LIST` и возвращает отсортированный список значений. По умолчанию используется обычный лексикографический (словарный) порядок сортировки. Его можно изменить при помощи аргумента `SUBNAME`, представляющего собой имя подпрограммы. Подпрограмма `SUBNAME` возвращает целое число, определяющее порядок следования элементов списка. Любая процедура сортировки состоит из последовательности сравнений двух величин. Для того чтобы правильно задать порядок сортировки, надо представить себе `SUBNAME` как функцию двух аргументов. В данном случае аргументы в подпрограмму `SUBNAME` передаются не общим для Perl способом – через массив `@_`, а через переменные `$a` и `$b`, обозначающие внутри подпрограммы соответственно первый и второй аргумент. Подпрограмму `SUBNAME` надо составить таким образом, чтобы она возвращала положительное целое, нуль, отрицательное целое, когда при сравнении аргумент `$a` назначается меньшим аргумента `$b`, равным аргументу `$b`, большим аргумента `$b` соответственно. Для этого внутри подпрограммы удобно использовать операции числового (`<=>`) и строкового (`cmp`) сравнения, возвращающие значения `-1`, `0`, `1`, если первый аргумент соответственно меньше второго, равен второму, больше второго.

Вместо имени подпрограммы в качестве аргумента `SUBNAME` может использоваться блок, определяющий порядок сортировки.

Зададим функцию `weekOrder`, определяющую порядок сортировки

```
sub WeekOrder {
    my %week=("Monday"=>0,
              "Tuesday"=>1,
              "Wednesday"=>2,
              "Thursday"=>3,
              "Friday"=>4,
              "Saturday"=>5,
              "Sunday"=>6);
    $week{$a}<=>$week{$b}
};
```

Используя функцию `sort()` с заданным порядком сортировки

```
for $i (0..11) {
    print "month # ", $i+1, "\n";
    for $DayName (sort WeekOrder keys %{$months[$i]}) {
        print "    $DayName @{$months[$i]{$DayName}}\n";
    }
};
```

получим структурированный вывод календаря в виде, упорядоченном по месяцам и дням недели:

month # 1

Monday	3 10 17 24 31
Tuesday	4 11 18 25
Wednesday	5 12 19 26
Thursday	6 13 20 27
Friday	7 14 21 28
Saturday	1 8 15 22 29
Sunday	2 9 16 23 30

.....

В качестве следующего примера построим на основе массива @months новую структуру, которую можно было бы назвать "хеш-массив хеш-массивов массивов", если бы такое название имело право на существование. В действительности, все просто. Речь идет о том, чтобы заменить в массиве @months числовые индексы ключами, совпадающими с названиями месяцев, и таким образом получить ассоциативный массив %months со сложной внутренней структурой (см. рис. 9.3).

```
%months = ("January"=>{ }, "February"=>{ }, ..., "December"=>{ })
```

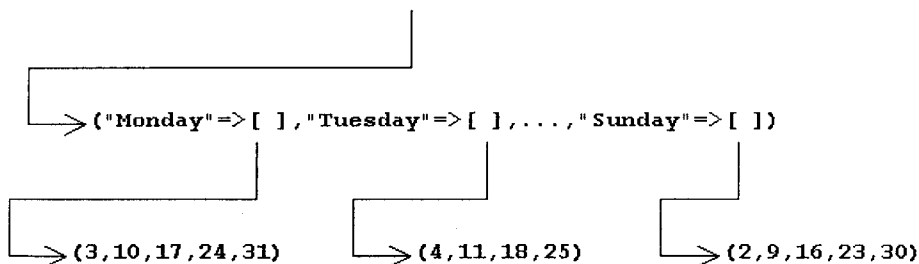


Рис. 9.3. Ассоциативный массив %months со сложной внутренней структурой

При построении хеш-массива %months воспользуемся вспомогательным хеш-массивом %OrderedMonths, который будем использовать для задания порядка сортировки:

Пример 9.4. Формирование хеш-массива %months

```
# вспомогательный массив %OrderedMonths
%OrderedMonths = ( "January"=>0,
                  "February"=>1,
                  "March"=>2,
                  "April"=>3,
```

```

    "May"=>4,
    "June"=>5,
    "July"=>6,
    "August"=>7,
    "September"=>8,
    "October"=>9,
    "November"=>10,
    "December"=>11 );

# формирование структуры
for $month (sort {$OrderedMonths{$a}<=>$OrderedMonths{$b}}
    keys %OrderedMonths) {
    $i = $OrderedMonths{$month};
    $months{$month}=$months{$i};
};

# Вывод элементов хеш-массива %months
for $month (sort {$OrderedMonths{$a}<=>$OrderedMonths{$b}}
    keys %OrderedMonths) {
    print "$month\n";
    $i = $OrderedMonths{$month};
    for $DayName (sort WeekOrder keys %{$months{$month}}) {
        print "    $DayName      @{$months{$i}{$DayName}}\n";
    }
};

```

В результате выполнения примера 9.3 будет распечатан календарь на 2000 год в виде:

```

January
Monday      3 10 17 24 31
Tuesday     4 11 18 25
Wednesday   5 12 19 26
Thursday    6 13 20 27
Friday      7 14 21 28
Saturday    1 8 15 22 29
Sunday      2 9 16 23 30
. . . . .

```

Рассмотренные примеры иллюстрируют подход, используемый в Perl для построения сложных структур данных. Читатель может сравнить возможности, предоставляемые языком Perl, с возможностями распространенных

языков программирования, таких как Pascal или C. Любая сложная структура в Perl на "верхнем" уровне представляет собой массив или ассоциативный массив, в который вложены ссылки на массивы или хеш-массивы следующего уровня и т. д. В этой иерархии ссылки на массивы и хеш-массивы могут чередоваться в произвольном порядке. При помощи такого подхода средствами Perl можно представить любую структуру C или запись языка Pascal. Perl позволяет с легкостью создавать структуры, которые в других языках создать трудно или невозможно, например, структуру, эквивалентную массиву, состоящему из элементов разных типов:

```
@array = (1, 2, 3, {"one"=>1, "two"=>2}, \&func, 4, 5);
```

Читатель может поупражняться в построении таких структур и открыть для себя новые нюансы применения этого гибкого и мощного подхода.

В заключение несколько слов о *фрагментах* массивов. Для доступа к элементам массива мы имеем специальную нотацию, состоящую из префикса \$, имени массива и индекса элемента в квадратных скобках, например, \$array[7]. Если здесь вместо индекса поместить список индексов, а префикс \$ заменить префиксом @, то такая запись будет обозначать фрагмент массива, состоящий из элементов с индексами из заданного списка. Подобную нотацию можно использовать в выражениях, например,

```
@subarray1 = @array[7..12];
```

```
@subarray2 = @array[3,5,7];
```

Массив @subarray1 является фрагментом массива @array, состоящим из элементов со значениями индекса от 7 до 12. Массив @subarray2 является фрагментом массива @array, состоящим из элементов со значениями индекса 3, 5 и 7. В первом случае список индексов задан при помощи операции "диапазон", во втором случае — перечислением.

Для многомерного массива понятие "фрагмент" обобщается и означает подмножество элементов, получающееся, если для некоторых индексов из диапазона их изменения выделить список допустимых значений. Для выделения одномерных фрагментов можно воспользоваться приведенной выше нотацией. Например, для выделения из массива @calendar фрагмента, содержащего календарь на первую неделю апреля, можно использовать запись

```
@april_first_week = @{$calendar[3]}[0..6];
```

Если выделяемый фрагмент является многомерным, то для его обозначения специальной нотации не существует. В этом случае следует сформировать новый массив, являющийся фрагментом исходного массива. Например, для выделения из массива @calendar календаря на первый квартал можно воспользоваться циклом

```
for $i (0..2) {  
    for $j (0..${#{$calendar[$i]}}) {
```

```

    $quarter1[$i][$j] = $calendar[$i][$j];
}
};

```

Вопросы для самоконтроля

1. Что такое ссылка?
2. Объясните разницу между жесткой и символической ссылкой.
3. Все ли корректно в следующем фрагменте


```

$href = \%hash;
$$href[0] = 17;

```
4. Каким будет значение переменной `$b` после выполнения следующих операторов:


```

$a = 1;
$b = ref $a;

```
5. Что обозначает каждое из выражений:


```

$$a[0];
${$a[0]};
$a->[0];
$a[0];

```
6. Приведите пример неявного создания ссылки.
7. `$arr_ref` — ссылка на анонимный массив. Как с ее помощью обратиться к третьему элементу этого массива? Напишите выражение.
8. Что такое "замыкание"?

Упражнения

1. Добавьте текст, содержащий последовательность операций, которые надо применить к переменной `$b`, чтобы получить значение переменной `$a`

```

$a = 7;
$b = \\ \\ $a;

```

В упражнениях 2-4 используйте результаты, полученные в примерах 9.1-9.3.

2. Вывести на экран все дни 2000 года, приходящиеся на воскресенья. Вывод должен содержать строку-заголовок, например, "All 2000' sundays are:", и по одной строке на каждый месяц года в виде: <название месяца> <дни месяца>.

3. Вывести на экран календарь на второй квартал года в виде

```
<название месяца>
<Monday>    <дни месяца>
. . .
<Sunday>    <дни месяца>
. . .
```

4. Вывести на экран календарь на первую неделю любого месяца. Вывод должен содержать строку-заголовок и по одной строке на каждый день недели в виде

```
<название месяца> <день месяца> <название дня недели>
```

5. Треугольником Паскаля называется следующая бесконечная таблица чисел:

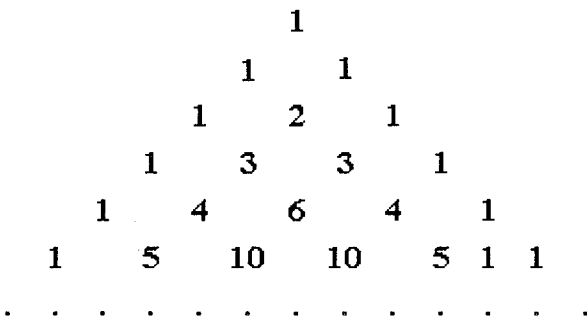


Рис. 9.4. Треугольник Паскаля

Каждое число в этой таблице равно сумме двух чисел, стоящих над ним слева и справа. Предложите структуру данных для хранения первых строк треугольника Паскаля. Напишите программу, заполняющую первые 32 строки и выводящую их на печать.



Работа со строками

Язык, созданный первоначально с главной целью — облегчить обработку большого количества отчетов, просто обязан располагать развитыми средствами для работы с текстом. Напомним, что в среде UNIX, из которой вышел язык Perl, средства для обработки текстовых строк имеются в различных утилитах: sed, awk, grep, cut. Командный интерпретатор shell, также обладающий некоторыми средствами для обработки строк, позволяет организовать совместную работу этих утилит, передавая выход одной программы на вход другой через механизм, называемый конвейером. Такой подход требует написания достаточно изощренных скриптов на языке shell в сочетании с обращением к внутренним командам утилит обработки текста sed или awk. Язык Perl, являясь средством создания программ-сценариев, в то же время один обладает всеми возможностями перечисленных утилит и даже их превосходит.

Типичная задача, возникающая при обработке текстового файла, заключается в том, чтобы найти в нем фрагмент, удовлетворяющий заданным условиям, и выполнить над ним некоторую операцию: удалить, заменить на другой фрагмент, извлечь для дальнейшего использования и т. д. Условия поиска можно достаточно просто выразить словами. Например: найти строку, содержащую слово Perl. Или: найти все фрагменты, находящиеся в конце строки и содержащие две цифры, за которыми следует произвольное количество прописных букв. Для формализованной записи подобных условий используются *регулярные выражения*, позволяющие описать *образец* или *шаблон* поиска при помощи специальных правил. Манипуляции с регулярными выражениями осуществляются при помощи соответствующих операций, которые мы также рассмотрим в этой главе.

10.1. Регулярные выражения

Регулярное выражение, по сути, представляет собой набор правил для описания текстовых строк. Сами правила записываются в виде последовательности обычных символов и метасимволов, которая затем в качестве образца используется в операциях поиска и замены текста. *Метасимволы* — это символы, имеющие в регулярном выражении специальное значение. Пользователи DOS/Windows хорошо знают метасимвол *, используемый для порождения имен файлов и обозначающий любую допустимую последователь-

ность. Регулярные выражения применяются многими программами UNIX, в том числе интерпретатором shell. Каждая из них использует свое множество метасимволов. В большинстве случаев они совпадают.

10.1.1. Метасимволы

В языке Perl к метасимволам относятся следующие символы:

"\", ".", "^", "\$", "|", "[", "(", ")", "*", "+", "?", "{".

Различные метасимволы выполняют в регулярном выражении различные функции, в частности, используются для обозначения одиночного символа или их группы, обозначают привязку к определенному месту строки, число возможных повторений отдельных элементов, возможность выбора из нескольких вариантов и т. д.

Регулярное выражение, подобно арифметическому выражению, строится с соблюдением определенных правил. В нем можно выделить операнды (элементы) и операции.

Простейшее регулярное выражение состоит из одного обычного символа. Обычный символ в регулярном выражении представляет самого себя. Соответственно, последовательность обычных символов представляет саму себя и не нуждается в дополнительной интерпретации. Для использования в операциях в качестве образца регулярное выражение заключается между двумя одинаковыми символами-ограничителями. Часто в качестве ограничителя используется символ "/" (косая черта). Например, образцу /Perl/ будут соответствовать все строки, содержащие слово Perl.

Если в регулярном выражении какой-либо метасимвол требуется использовать в буквальном, а не специальном значении, его нужно *экранировать* или *маскировать* при помощи другого метасимвола — "\". Например, образцу /\\"*/ соответствует фрагмент текста \". Здесь первый метасимвол "\" экранирует второй метасимвол "\", а третий метасимвол "\" экранирует метасимвол "*".

Метасимвол "." представляет любой одиночный символ, кроме символа новой строки. Так, образцу ./ будет соответствовать любая непустая строка. Если в операциях сопоставления с образцом установлен флаг s, то метасимвол "." соответствует также и символ новой строки.

Метасимвол "[" используется в конструкции [...] для представления любого одиночного символа из числа заключенных в скобки, т. е. он представляет *класс* символов. Два символа, соединенные знаком минус, задают диапазон значений, например, [A-Za-z] задает все прописные и строчные буквы английского алфавита. Если первым символом в скобках является символ "^", вся конструкция обозначает любой символ, не перечисленный в скобках. Например, [^0-9] обозначает все нецифровые символы. Ниже мы рассмотрим и другие способы представления классов символов.

Метасимволы "^" и "\$" используются для задания привязки к определенному месту строки. Метасимвол "^" в качестве первого символа регулярного выражения обозначает начало строки. Метасимвол "\$" в качестве последнего символа регулярного выражения обозначает конец строки. Например, следующим образцам соответствуют:

`/^$/` — пустая строка (начало и конец, между которыми пусто);

`/^Perl/` — слово Perl в начале строки;

`/Perl$/` — слово Perl в конце строки.

Метасимвол "|" можно рассматривать как символ операции, задающей выбор из нескольких вариантов (подобно логической операции ИЛИ). Например, образцу `/a|b|c/` соответствует фрагмент текста, содержащий любой из символов a, b, c. Если вариантами выбора являются одиночные символы, как в данном примере, то лучше использовать конструкцию, определяющую класс символов, в данном случае `[abc]`. Но в отличие от конструкции [...] операция "|" применима и тогда, когда вариантами выбора являются последовательности символов. Например, образцу `/Word|Excel|Windows/` соответствует фрагмент текста, содержащий любое из слов Word, Excel, Windows.

Следующая группа метасимволов служит в качестве коэффициентов или *множителей*, определяющих количество возможных повторений отдельных *атомарных* элементов регулярного выражения.

- ☐ r^* — нуль и более повторений r ;
- ☐ r^+ — одно и более повторений r ;
- ☐ $r^?$ — нуль или одно повторение r ;
- ☐ $r\{n\}$ — ровно n повторений r ;
- ☐ $r\{n,\}$ — n и более повторений r ;
- ☐ $r\{n,m\}$ — минимум n , максимум m повторений r .

Атомарные элементы, или *атомы* — это простейшие элементы, из которых строится регулярное выражение. Это не обязательно одиночный символ.

Пример 10.1. Множители

Образец	Соответствие
<code>/.*/</code>	любая строка;
<code>/.+/</code>	любая непустая строка;
<code>/[0-9]{3}/</code>	любая последовательность из трех цифр;
<code>/\[0-9]/</code>	последовательность, состоящая из любого числа символов [

В первых двух примерах атомом является метасимвол ".". В третьем образце в качестве атома выступает конструкция `[0-9]`, определяющая класс цифровых

вых символов. В четвертом образце атом — это пара символов "\[", включающая метасимвол "\", отменяющий специальное значение следующего за ним метасимвола "[". Полный список атомов мы приведем после изучения всех необходимых синтаксических конструкций.

Алгоритм, применяемый в операциях поиска и замены (см. ниже) для обработки регулярных выражений, содержащих множители, является "жадным": он пытается найти для образца, снабженного множителем, максимальный сопоставимый фрагмент текста. Рассмотрим, например, что происходит при поиске в строке

"Скроен колпак не по-колпаковски, надо колпак переколпаковать"

фрагмента, удовлетворяющего образцу /. *колпак/.

Алгоритм найдет максимальный фрагмент, удовлетворяющий выражению . * (вся строка без завершающего символа новой строки), затем начнет двигаться назад, отбрасывая в найденном фрагменте по одному символу, до тех пор, пока не будет достигнуто соответствие с образцом. Найденный фрагмент будет иметь вид "Скроен колпак не по-колпаковски, надо колпак переколпак".

Можно заставить алгоритм работать иначе, снабдив множитель "*" модификатором "?". В этом случае алгоритм из "жадного" превращается в "ленивый" и будет для образца, снабженного множителем, искать минимальный соответствующий фрагмент. "Ленивый" алгоритм для множителя "*" начнет поиск в строке с пустого фрагмента "", добавляя к нему по одному символу из строки до тех пор, пока не достигнет соответствия с образцом. В этом случае найденный фрагмент будет иметь вид "Скроен колпак". Все сказанное справедливо и для других множителей. Например, в строке "1234567" будет найден:

- для образца /\d*/ — максимальный фрагмент "1234567";
- для образца /\d+/ — максимальный фрагмент "1234567";
- для образца /\d?/ — максимальный фрагмент "1";
- для образца /\d{2,5}/ — максимальный фрагмент "12345";
- для образца /\d*?/ — минимальный фрагмент "";
- для образца /\d+?/ — минимальный фрагмент "1";
- для образца /\d??/ — минимальный фрагмент "";
- для образца /\d{2,5}?/ — минимальный фрагмент "12".

10.1.2. Метапоследовательности

Символ "\", непосредственно предшествующий одному из метасимволов, отменяет специальное значение последнего. Если же "\" непосредственно предшествует обычному символу, то, напротив, такая последовательность во многих случаях приобретает специальное значение. Подобного рода после-

довательности будем называть *метапоследовательностями*. Метапоследовательности в регулярном выражении служат, в основном, для представления отдельных символов, их классов или определенного места в строке, дополняя и иногда дублируя функции метасимволов. Рассмотрим существующие метапоследовательности.

- ❑ `\nnn` — представляет символ, восьмеричный код которого равен *nnn*. Например, последовательность `\120\145\162\154` представляет слово Perl (`\120` — восьмеричный код буквы P, `\145` — буквы e, `\162` — буквы r, `\154` — буквы l);
- ❑ `\xnn` — представляет символ, шестнадцатеричный код которого равен *nn*. Слово Perl, например, представляется последовательностью `\x50\x65\x72\x6C`;
- ❑ `\cn` — представляет управляющий символ, который генерируется при нажатии комбинации клавиш `<Ctrl>+<N>`, где N — символ, например, `\cD` соответствует `<Ctrl>+<D>`;
- ❑ `\$` — символ "\$";
- ❑ `\@` — символ "@";
- ❑ `\%` — символ "%";
- ❑ `\a` — представляет символ с десятичным ASCII-кодом 7 (звонок). При выводе производит звуковой сигнал;
- ❑ `\e` — символ Esc, десятичный ASCII-код 27;
- ❑ `\f` — символ перевода страницы, десятичный ASCII-код 12;
- ❑ `\n` — символ новой строки, десятичный ASCII-код 10;
- ❑ `\r` — символ "возврат каретки", десятичный ASCII-код 13;
- ❑ `\t` — символ горизонтальной табуляции, десятичный ASCII-код 9;
- ❑ `\v` — символ вертикальной табуляции, десятичный ASCII-код 11;
- ❑ `\s` — представляет класс *пробельных* символов. К пробельным символам относятся: пробел, символ табуляции, возврат каретки, символ новой строки и символ перевода страницы. То же самое, что и `[\t,\r,\n,\f]`;
- ❑ `\S` — представляет класс *непробельных* символов, то же самое, что и класс `[^\t,\r,\n,\f]`;
- ❑ `\d` — класс цифровых символов, то же, что и `[0-9]`;
- ❑ `\D` — класс нецифровых символов, то же, что и `[^0-9]`;
- ❑ `\w` — представляет класс *буквенно-цифровых* символов, состоящий из букв, цифр и символа подчеркивания "_". То же самое, что и `{a-zA-Z_0-9}`. Обратите внимание, что в этот класс входят только буквы английского алфавита;

- \w — представляет класс небуквенно-цифровых символов. То же, что и `[^a-zA-Z_0-9]`;
- \A — обозначает начало строки;
- \Z — обозначает конец строки;

Замечание

Последовательность \A эквивалентна метасимволу ^ в начале регулярного выражения, а последовательность \Z — метасимволу \$ в конце регулярного выражения, за исключением одного случая. Назовем строку, содержащую внутри себя символы новой строки (ASCII 10), *мульти-строкой*. Фактически мульти-строка состоит из отдельных строк, разделенных ограничителями — символами новой строки. При выводе мульти-строка отображается в виде нескольких строк. Если к мульти-строке применяется операция поиска или замены с опцией /m (см. ниже), то последовательности \A и \Z обозначают соответственно начало и конец всей мульти-строки, а метасимволам ^ и \$ соответствуют еще и границы внутренних строк, образующих мульти-строку.

- \b — обозначает границы слова. Под словом понимается последовательность символов из класса \w. Граница слова определяется как точка между символами из класса \w и символами из класса \W;
- \B — обозначает не-границы слова, т. е. класс символов `[^\b]`;
- \l — обозначает, что следующий символ регулярного выражения преобразуется к нижнему регистру. Например, запись `/\lp/` означает, что символ `p` будет преобразован к нижнему регистру, после чего новый образец `/p/` может быть использован в соответствующей операции поиска или замены;
- \u — обозначает, что следующий символ регулярного выражения преобразуется к верхнему регистру;
- \L... \E — обозначает, что все символы в регулярном выражении между \L и \E преобразуются к нижнему регистру;
- \U... \E — обозначает, что все символы в регулярном выражении между \U и \E преобразуются к верхнему регистру;
- \Q... \E — обозначает, что все метасимволы в регулярном выражении между \Q и \E экранируются при помощи символа "\". Например, запись `/\Q^*+ \E/` эквивалентна записи `/\^*\+/?/`;
- \G — обозначает точку, в которой закончился предыдущий поиск `m//g` (см. описание операции поиска `m//g`).

10.1.3. Атомы

Из всех метасимволов, перечисленных в начале раздела, нам осталось рассмотреть "(" и ")". Эти метасимволы служат для группирования ряда элементов, входящих в состав образца, в один элемент. Например, образцу `/(abc)+/` со-

ответствует строка, состоящая из одного или более повторений последовательности `abc`, в то время, как образцу `/abc+/` — строка, состоящая из начальных символов `ab`, за которыми следует один или более символов `c`.

Теперь мы можем перечислить "атомы", из которых строится регулярное выражение.

- ☐ Регулярное выражение в скобках, представляющее несколько элементов, сгруппированных в один.
- ☐ Любой обычный символ (не метасимвол).
- ☐ Символ `"."`, представляющий любой одиночный символ, кроме символа новой строки.
- ☐ Конструкция `[...]`, представляющая класс символов, перечисленных в квадратных скобках.
- ☐ Метаследовательность, представляющая символ или класс символов (см. раздел 10.1.2): `\a, \n, \r, \t, \f, \e, \d, \D, \w, \W, \s, \S`.
- ☐ Метаследовательность вида `\nnn`, определяющая символ при помощи его восьмеричного ASCII-кода `nnn` (см. раздел 10.1.2).
- ☐ Метаследовательность вида `\xnn`, определяющая символ при помощи его шестнадцатеричного ASCII-кода `nn` (см. раздел 10.1.2).
- ☐ Метаследовательность вида `\cn`, представляющая управляющий символ `Ctrl-n` (см. раздел 10.1.2).
- ☐ Конструкция вида `\number`, представляющая обратную ссылку (см. раздел 10.1.4).
- ☐ Любая конструкция вида `\character`, не имеющая специального значения, а представляющая собственно символ `character`, например: `*, \y, \h`.

Напомним, что в регулярном выражении множители `*`, `+`, `?`, `{n,m}` применяются именно к атому, расположенному непосредственно слева.

10.1.4. Обратные ссылки

Ранее мы установили, что группу элементов регулярного выражения можно заключить в скобки и рассматривать как один элемент. Заключение группы элементов в скобки имеет дополнительный и весьма полезный эффект. Если в результате поиска будет найден фрагмент текста, соответствующий образцу, заключенному в скобки, то этот фрагмент сохранится в специальной переменной. Внутри регулярного выражения к нему можно будет обратиться, используя запись `\number`, где `number` — номер конструкции `()` в исходном регулярном выражении. Запись `\number`, указывающую на найденный по образцу фрагмент текста, будем называть *обратной ссылкой*. Можно задать любое количество конструкций вида `()` и ссылаться на соответствующие найденные фрагменты текста, как на `\1`, `\2` и т. д. Например, образцу `/(.+) - \1/` со-

ответствуют слова "ха-ха", "хи-хи", "ку-ку" и т. п., а образцу `/(.)(.)?.\2\1/` — все палиндромы из четырех или пяти букв. (Палиндром — слово или предложение, которое одинаково читается слева направо и справа налево.)

Внутри образца конструкция `\n` ($n=1, \dots, 9$) всегда обозначает обратную ссылку. Запись вида `\nn` также интерпретируется как обратная ссылка, но только в том случае, если в исходном выражении задано не менее, чем `nn` скобочных конструкций вида `()`. Иначе запись `\nn` обозначает символ с восьмеричным кодом `nn`.

Для ссылки на найденный фрагмент текста за пределами регулярного выражения, например, при задании замещающего текста в операции замены, вместо записи `\number` используется запись `$number`. Например, операция замены (см. раздел 10.2)

```
$str=~s/(\S+)\s+(\S+)/$2 $1/
```

меняет местами первые два слова в строке `$str`.

Область действия переменных `$1`, `$2` и т. д. распространяется до наступления одного из следующих событий: конец текущего блока; конец строки, являющейся аргументом функции `eval`; следующее совпадение с образцом. Аналогичным образом определяется область действия и для следующих предопределенных переменных, используемых в операциях сопоставления с образцом.

- ❑ `$&` — часть строки, найденная при последней операции сопоставления с образцом.
- ❑ `$`` — часть строки, стоящая перед совпавшей частью при последней успешной операции сопоставления с образцом.
- ❑ `$'` — часть строки, стоящая после совпавшей части при последней успешной операции сопоставления с образцом.

Например, в результате выполнения операции поиска (см. раздел 10.2)

```
$str=~m/two/
```

в строке `$str="one two three"` образца `/two/` переменным будут присвоены следующие значения:

```
$& - "two";
```

```
$` - "one";
```

```
$' - "three".
```

Эти значения будут сохраняться до наступления одного из перечисленных выше событий, и их можно использовать, например, для формирования строки с обратным порядком следования слов: `$rstr=$'.$&.$``. Строка `$rstr` будет иметь вид `"three two one"`.

Следует отметить, что, если обращение к одной из переменных `$&`, `$``, `$'` встречается *где-либо* в программе, то интерпретатор perl будет вычислять и запоминать их для *каждой* операции сопоставления с образцом, что, в свою очередь, замедляет выполнение всей программы. Поэтому не следует использовать данные переменные без особой необходимости.

10.1.5. Расширенный синтаксис регулярных выражений

Выше мы использовали скобки для группирования нескольких элементов регулярного выражения в один элемент. Побочным эффектом данной операции является запоминание найденного фрагмента текста, соответствующего образцу, заключенному в скобки, в специальной переменной. Если скобки используются только для группирования элементов регулярного выражения, то найденный фрагмент текста можно не запоминать. Для этого после открывающей скобки "(" следует поместить конструкцию "?:", например, в случае задания альтернативы `/(?:Perl|perl)/`.

`?:pattern`

Конструкция относится к классу конструкций общего вида `(?...)`, добавляющих новые возможности для задания образцов за счет расширения синтаксиса регулярного выражения, а не за счет введения новых метасимволов или метапоследовательностей. Символ, следующий за символом "?", определяет функцию, выполняемую данной синтаксической конструкцией. В настоящее время определены около десяти расширенных конструкций регулярного выражения, большая часть которых рассмотрена в данном разделе. Оставшиеся конструкции, на наш взгляд, не являются необходимыми для первоначального знакомства с языком.

`(?#text)`

Текст после символа `#` и до закрывающей скобки `)`, комментарий, игнорируется интерпретатором и используется для добавления непосредственно в регулярное выражение.

`(?:pattern)`

`(?imsx-imsx:pattern)`

Использовать скобки только для группирования элементов без создания обратных ссылок. Символы `imsx-imsx` между вопросительным знаком и двоеточием интерпретируются как флаги, модифицирующие функцию данного выражения (см. ниже).

`(?=pattern)`

Следующий фрагмент в тексте должен соответствовать образцу `pattern`. Обычно образец для операций поиска или замены задается при помощи

регулярного выражения. Результатом операции поиска является фрагмент, соответствующий образцу, который сохраняется в специальной переменной `$&`. Конструкция `(?=pattern)` в составе регулярного выражения позволяет задать условие поиска, не включая найденный фрагмент, соответствующий образцу `pattern`, в результат, сохраняемый в переменной `$&`. Конструкция `(?=pattern)` в регулярном выражении задает условие, что *следующий* фрагмент текста должен удовлетворять образцу `pattern`. Обращаем внимание на слово *следующий*. Данная конструкция неприменима для задания условия, что *предыдущий* фрагмент текста должен соответствовать заданному образцу. Например, образцу `/b+(?=c+)/` соответствует часть строки, состоящая из одной или более литер `b`, за которыми следуют одна или более литер `c`, причем найденный фрагмент текста будет содержать только последовательность литер `b` без последовательности литер `c`.

Пример 10.2. Расширенный синтаксис регулярных выражений

Рассмотрим строку

```
$str = "aaabbbcccd";
```

В результате операции поиска

```
$str =~ m/b+(?=c+)/;
```

будут сохранены следующие значения в специальных переменных:

```
$` - aaa
```

```
$& - bbb
```

```
$' - cccddd
```

Если в операции поиска указать образец `/b+c+/,` то значения специальных переменных будут следующими:

```
$` - aaa
```

```
$& - bbbccc
```

```
$' - ddd
```

В свою очередь, операция поиска по образцу `/(?=b+)c+/,` в нашем примере не даст результата. Данный образец задает условие, что следующий фрагмент текста должен содержать непустую последовательность литер `b`. В нашей строке такой фрагмент будет найден (это фрагмент `bbb`), но не будет включен в результат поиска. Следующий фрагмент, в соответствии с образцом, должен представлять непустую последовательность литер `c`, но в нашем случае этого соответствия не будет, так как мы остановились перед фрагментом `bbb`, не включив его в результат, и поэтому следующим фрагментом будет `bbb`, а не `ccc`.

Конструкцию `(?=pattern)` будем называть *регулярным выражением с положительным постусловием*.

(?!pattern)

Конструкция (?!pattern) в регулярном выражении задает условие, что *следующий* фрагмент текста *не* должен удовлетворять образцу pattern. Найденный фрагмент не запоминается в переменной \$&. Например, результат операции поиска

```
$str =~ m/b+(?!c+)/;
```

в рассмотренной выше строке \$str будет зафиксирован в следующих значениях специальных переменных:

```
$` - aaa
```

```
$& - bb
```

```
$' - bccccddd
```

Найденная подстрока соответствует образцу: она состоит из двух литер bb, за которыми не следует последовательность литер c.

По аналогии с предыдущей данную конструкцию назовем *регулярным выражением с отрицательным постусловием*.

(?<=pattern)

Конструкция (?<=pattern) в регулярном выражении задает условие, что *предыдущий* фрагмент текста должен удовлетворять образцу pattern. Найденный фрагмент не запоминается в переменной \$&. Образец pattern должен иметь фиксированную длину, т. е. не содержать множителей.

В нашем примере в результате операции поиска

```
$str =~ m/(?<=b)b+/;
```

значения специальных переменных будут распределены следующим образом:

```
$` - aaab
```

```
$& - bb
```

```
$' - cccccddd
```

Данную конструкцию назовем *регулярным выражением с положительным преусловием*.

(?<!pattern)

Конструкция (?<!pattern) в регулярном выражении задает условие, что *предыдущий* фрагмент текста *не* должен удовлетворять образцу pattern. Найденный фрагмент не запоминается в переменной \$&. Образец pattern должен иметь фиксированную длину, т. е. не содержать множителей.

В нашем примере в результате операции поиска

```
$str =~ m/(?<!b)c+/;
```

специальные переменные получают следующие значения:

\$` — aaabbbbc

\$& — cc

\$' — ddd

Данную конструкцию будем называть *регулярным выражением с отрицательным предусловием*.

(?imsx-imsx)

Задание флагов операции сопоставления с образцом осуществляется в самом образце. Флаги модифицируют выполнение операции и обычно являются частью синтаксиса самой операции. Расширенная конструкция (?imsx-imsx) позволяет задать флаги операции внутри самого образца. Эта возможность может быть полезной, например, в таблицах, когда разные элементы таблицы требуются по-разному сопоставлять с заданным образцом, например, некоторые элементы — с учетом регистра, другие — без учета. Допустимыми являются следующие флаги.

□ i — поиск без учета регистра;

□ m — строка трактуется как мульти-строка, состоящая из нескольких строк, разделенных символом новой строки;

□ s — строка трактуется как одна строка, в этом случае метасимволу "." соответствует любой одиночный символ, включая символ новой строки;

□ x — разрешается использовать в образцах пробелы и комментарии. При использовании флага x пробелы в образцах игнорируются. Признаком комментария является символ #, как и в основном тексте Perl-программы. Пробелы позволяют сделать образец более читаемым.

Одна из литер i, m, s, x после знака "-" обозначает отмену соответствующего флага.

При помощи данной расширенной конструкции можно задать, например, следующий образец

```
/(?ix) perl # игнорирование регистра при поиске/
```

Флаг i предписывает не учитывать регистр в операциях сопоставления с образцом, так что образцу будет соответствовать и слово perl, и слово Perl. Флаг x позволяет выделить слово "perl" пробелами и использовать в образце комментариев. И пробелы, и комментариев не будут учитываться в операции сопоставления с образцом.

10.1.6. Сводка результатов

Изложенное в данном разделе можно суммировать в виде набора правил, которыми следует руководствоваться при работе с регулярными выражениями.

□ Любой одиночный символ, не являющийся метасимволом, представляет самого себя.

- В заключение раздела приведем в табл. 10.1 и 10.2 сводку метасимволов и метапоследовательностей, рассмотренных в данной главе.

Таблица 10.1. Символы, имеющие специальное значение
в регулярных выражениях Perl

Метасимвол	Интерпретация
\	Отменяет (экранирует) специальное значение следующего за ним метасимвола
.	Любой одиночный символ, кроме символа новой строки
s	Любой одиночный символ, включая символ новой строки, если в операции сопоставления с образцом задан флаг s
^	Обозначает начало строки, если является первым символом образца
\$	Обозначает конец строки, если является последним символом образца
	Разделяет альтернативные варианты
[...]	Любой одиночный символ из тех, которые перечислены в квадратных скобках. Пара символов, разделенных знаком минус, задает диапазон символов. Например, [A-Za-z] задает все прописные и строчные буквы английского алфавита. Если первым символом в скобках является символ "^", то вся конструкция обозначает любой символ, не перечисленный в скобках. Внутри скобок символы ".", "*", "[", "]", "\", "}" и "?" теряют свое специальное значение

Таблица 10.1 (окончание)

Метасимвол	Интерпретация
(...)	Группирование элементов образца в один элемент
*	Нуль и более повторений регулярного выражения, стоящего непосредственно перед *
+	Одно или более повторений регулярного выражения, стоящего непосредственно перед +
?	Одно или ни одного повторения регулярного выражения, стоящего непосредственно перед ?
{ n, m }	Минимальное n и максимальное m число повторений регулярного выражения, стоящего перед {n, m}. Конструкция {n} означает ровно n повторений, {n, } — минимум n повторений

Таблица 10.2. Метапоследовательности в регулярных выражениях Perl

Метапоследовательность	Значение
\0nn	Символ, восьмеричный код которого равен nn
\a	При выводе производит звуковой сигнал
\A	Обозначает начало строки
\b	Обозначает границы слова. Под словом понимается последовательность символов из класса \w (см. ниже). Граница слова определяется как точка между символами из класса \w и символами из класса \W (см. ниже)
\B	Обозначает не-границы слова
\cn	Управляющий символ, который генерируется при нажатии комбинации клавиш <Ctrl>+<N>
\d	Любой цифровой символ, то же, что и [0-9]
\D	Любой нецифровой символ, то же, что и [^0-9]
\e	Символ Esc, ASCII 27
\E	Ограничитель последовательностей \L, \U, \Q
\f	Символ перевода страницы, ASCII 12
\G	Обозначает точку, в которой закончился предыдущий поиск m//g
\l	Преобразует следующий символ регулярного выражения к нижнему регистру

Таблица 10.2 (окончание)

Метапоследовательность	Значение
<code>\L</code>	Преобразует все последующие символы в регулярном выражении к нижнему регистру до тех пор, пока не встретится последовательность <code>\E</code>
<code>\n</code>	Символ новой строки, ASCII 10
<code>\Q</code>	Эквивалентно экранированию всех последующих метасимволов в регулярном выражении при помощи символа <code>\</code> до тех пор, пока не встретится последовательность <code>\E</code>
<code>\r</code>	Символ "возврат каретки", ASCII 13
<code>\s</code>	Класс пробельных символов: пробел (space), символ табуляции (tab), возврат каретки (carriage return), символ перевода строки (line feed) и символ перевода страницы (form feed); эквивалентно <code>[\t, \r, \n, \f]</code>
<code>\S</code>	Класс непробельных символов
<code>\t</code>	Символ табуляции, ASCII 9
<code>\u</code>	Преобразует следующий символ к верхнему регистру
<code>\U</code>	Преобразует все последующие символы в регулярном выражении к верхнему регистру до тех пор, пока не встретится последовательность <code>\E</code>
<code>\v</code>	Символ вертикальной табуляции, ASCII 11
<code>\w</code>	Любая буква, цифра или символ подчеркивания
<code>\W</code>	Любой символ, не являющийся буквой, цифрой или символом подчеркивания
<code>\xnn</code>	Символ, шестнадцатеричный код которого равен nn
<code>\Z</code>	Обозначает конец строки

10.2. Операции с регулярными выражениями

В данной главе неоднократно упоминались операции с регулярными выражениями, такие как поиск по образцу. Основными среди них являются: операция поиска `m//`, операция замены `s///` и операция транслитерации `tr///`.

10.2.1. Операция поиска

`m/PATTERN/cgimosx`

Операция поиска `m/PATTERN/` осуществляет поиск образца `PATTERN`. Результатом операции является значение `1` (ИСТИНА) или пустая строка ""

(ЛОЖЬ). По умолчанию поиск осуществляется в строке, содержащейся в специальной переменной `$_`. Можно назначить другую строку для поиска в ней заданного образца при помощи операций связывания `=~` или `!~`:

```
$var =~ m/PATTERN/cgimosx
```

В результате последней операции поиск образца `PATTERN` будет осуществляться в строке, задаваемой переменной `$var`. Если в правой части операции связывания стоит операция поиска `m//`, то в левой части может находиться не обязательно переменная, а любое строковое выражение.

Операция `!~` отличается от операции `=~` тем, что возвращает противоположное логическое значение. Например, в результате поиска в строке `"aaabbbcccc"` образца `/b+/:`

```
$s="aaabbbcccc" =~ m/b+/;
```

```
$s="aaabbbcccc" !~ m/b+/;
```

в обоих случаях будет найден фрагмент `bbb`. Но в первом случае возвращаемое значение, сохраненное в переменной `$s`, будет равно `1` (ИСТИНА), а во втором случае — пустой строке `"` (ЛОЖЬ).

Образец `PATTERN` может содержать переменные, значения которых подставляются при каждом выполнении поиска по данному образцу.

Флаги `cgimosx` модифицируют выполнение операции поиска. Флаги `imsx` имеют тот же смысл, что и в рассмотренном выше случае конструкции расширенного регулярного выражения `(?imsx-imsx)`.

- `i` — поиск без учета регистра;
- `m` — трактуется как мульти-строка, состоящая из нескольких строк, разделенных символом новой;
- `s` — строка трактуется как одна строка, в этом случае метасимволу `".` соответствует любой одиночный символ, включая символ новой строки;
- `x` — разрешается использовать в образцах пробелы и комментарии.

Остальные флаги выполняют следующие функции.

- `g` — Задает глобальный поиск образца в заданной строке. Это означает, что будут найдены все фрагменты текста, удовлетворяющие образцу, а не только первый из них, как имеет место по умолчанию. Возвращаемое значение зависит от контекста. Если в составе образца имеются подобразцы, заключенные в скобки `()`, то в контексте массива для каждого заключенного в скобки подобразца возвращается список всех найденных фрагментов. Если в составе образца нет подобразцов, заключенных в скобки, то в контексте массива возвращается список всех найденных фрагментов, удовлетворяющих образцу. В скалярном контексте каждая операция `m//g` осуществляет поиск следующего фрагмента, удовлетво-

ряющего образцу, возвращая значение 1 (ИСТИНА), если он найден, и пустую строку "", если не найден. Позиция строки, в которой завершился последний поиск образца при установленном флаге `g`, может быть получена при помощи встроенной функции `pos()` (см. ниже). Обычно при неудачном поиске начальная позиция поиска восстанавливается в начало строки. Флаг `c` отменяет восстановление начальной позиции поиска при неудачном поиске образца.

Рассмотрим следующий скрипт.

Пример 10.3. Операция поиска `m//`

```
$str="abaabbaaabbbbaaaabbbb";
# контекст массива, нет подобразцов в скобках
@result=$str =~m/a+b+/g;
print "контекст массива, нет конструкций в скобках:\n";
print "\@result=@result\n";
# контекст массива, есть конструкции в скобках, задающие обратные ссылки
@result=$str =~m/(a+)(b+)/g;
print "контекст массива, есть конструкции в скобках:\n";
print "\@result=@result\n";
# скалярный контекст
print "скалярный контекст:\n";
while ($result=$str =~m/(a+)(b+)/g) {
    print "result=$result, current match is $&, position=",pos($str),"n";
}
```

Результатом его выполнения будет вывод:

```
контекст массива, нет конструкций в скобках:
@result=ab aabb aaabbb aaaabbbb
контекст массива, есть конструкции в скобках:
@result=a b aa bb aaa bbb aaaa bbbb
скалярный контекст:
result=1, current match is ab, position=2
result=1, current match is aabb, position=6
result=1, current match is aaabbb, position=12
result=1, current match is aaaabbbb, position=20
```

□ `c` — Используется совместно с флагом `g`. Отменяет восстановление начальной позиции поиска при неудачном поиске образца.

Рассмотрим скрипт

Пример 10.4. Флаг /с в операции m//

```
$str="abaabbaaabbbbaaaabbbb";
while ($result=$str =~m/(a+)(b+)/g) {
    print "result=$result, current match is $&, position=",pos($str),"\\n";
}
print "last position=", pos($str), "\\n";
```

Здесь поиск образца /a+b+/ в строке \$str осуществляется в цикле до первой неудачи. При последнем (неудачном) поиске начальная позиция поиска по умолчанию устанавливается в начало строки, в этом случае вывод имеет вид:

```
result=1, current match is ab, position=2
result=1, current match is aabb, position=6
result=1, current match is aaabbb, position=12
result=1, current match is aaaabbbb, position=20
last position=
```

Если глобальный поиск осуществлять при установленном флаге c:

```
while ($result=$str =~m/(a+)(b+)/gc) {
    . . .
```

то при последнем неудачном поиске начальная позиция поиска не переустанавливается. Вывод имеет вид:

```
result=1, current match is ab, position=2
result=1, current match is aabb, position=6
result=1, current match is aaabbb, position=12
result=1, current match is aaaabbbb, position=20
last position=20
```

При задании образца для глобального поиска m//g можно использовать метасимвол \G, представляющую точку, в которой закончился последний поиск m//g. Например, в результате выполнения скрипта

```
$str="1) abc 2) aabbcc 3) aaabbbccc 4) aaaabbbbcccc";
$str =~m/3\\)\s+/g;
$str =~m/\Ga+/;
```

сначала по образцу будет найден фрагмент "3)", а затем фрагмент, удовлетворяющий образцу /a+/ и расположенный сразу за точкой, в которой завершился последний поиск. Этим фрагментом является "aaa".

- ○ — Значения переменных, входящих в состав образца `PATTERN`, подставляются только один раз, а не при каждом поиске по данному образцу. Рассмотрим, например, следующий скрипт:

Пример 10.5. Флаг `o` в операции `m//`

```
@pattnlist=("a+", "b+", "c+", "d+");
foreach $pattn (@pattnlist) {
    $line = <STDIN>;
    $line =~ m/$pattn/o;
    print "pattn=$pattn \${&= \$&\n";
}
```

Массив `@pattnlist` содержит список образцов "a+", "b+", "c+" и "d+". В цикле по элементам этого списка в переменную `$line` считывается очередная строка из стандартного ввода. В ней осуществляется поиск по образцу, совпадающему с текущим элементом списка. Поскольку использован флаг `o`, подстановка значений в образце `/ $pattn/` будет осуществлена один раз за время жизни данной Perl-программы, т. е. в качестве образца на каждом шаге цикла будет использовано выражение "a+". Если операцию поиска осуществлять без флага `o`:

```
$line =~ m/$pattn/,
```

то в качестве образца будут последовательно использованы все элементы списка "a+", "b+", "c+" и "d+".

В качестве символов-ограничителей для выделения образца можно использовать любую пару символов, не являющихся цифрой, буквой или пробельным символом. Если в качестве ограничителя используется символ `/`, то литеру `m` в обозначении операции можно опустить и использовать упрощенную форму `/PATTERN/` вместо `m/PATTERN/`.

Если в качестве ограничителя используется одинарная кавычка `'`, то подстановка значений переменных внутри образца не производится.

Если в качестве ограничителя используется символ `"?: ?PATTERN?`, то при применении операции поиска находится только одно соответствие. Например, в результате выполнения скрипта

```
$str="abaabbbaabbbaaaabbbb";
while ($result = $str =~ m?a+b?g) {
    print "result=$result, current match is ${&, position=", pos($str), "\n";
}
```

будет найдено только первое соответствие образцу:

```
result=1, current match is ab, position=2
```


10.2.2. Операция замены

`s/PATTERN/REPLACEMENT/egimosx`

Операция замены `s/PATTERN/REPLACEMENT/` осуществляет поиск образца `PATTERN` и, в случае успеха, замену найденного фрагмента текстом `REPLACEMENT`. Возвращаемым значением является число сделанных замен или пустая строка (ЛОЖЬ), если замен не было. По умолчанию поиск и замена осуществляются в специальной переменной `$_`. Ее можно заменить другой строкой при помощи операций связывания `=~` или `!~` :

```
$var =~ s/PATTERN/REPLACEMENT/egimosx
```

Флаг `g` задает глобальную замену всех фрагментов, удовлетворяющих образцу `PATTERN`, текстом `REPLACEMENT`.

Флаг `e` означает, что заменяющий текст `REPLACEMENT` следует рассматривать как Perl-выражение, которое надо предварительно вычислить. Например, в результате выполнения скрипта

Пример 10.6. Операция замены `s///`

```
$str="abaabbbaabbbbaaabbbb";  
$result=$str =~s[(a+b+)]<length($1)>ge;  
print "result=$result new str=$str\n";
```

будет выведено число сделанных замен `$result` и новое значение строки `$str`, в которой каждый найденный фрагмент, соответствующий образцу `[a+b+]`, заменен числом, равным его длине:

```
result=4 new str=2468
```

Флаги `imosx` имеют тот же смысл, что для операции поиска `m///`.

Так же, как и в операции замены, в качестве ограничителей для выделения образца можно использовать любую пару символов, не являющихся цифрой, буквой или пробельным символом. Можно использовать различные ограничители для выделения образца и замещающего текста, например, `s(pattern)<replacement>`.

10.2.3. Операция транслитерации

`tr/SEARCHLIST/REPLACEMENTLIST/cds`

Преобразует каждый символ из списка поиска `SEARCHLIST` в соответствующий символ из списка замены `REPLACEMENTLIST` и возвращает число преобразованных символов. По умолчанию преобразования осуществляются в строке, задаваемой переменной `$_`. Как и в рассмотренных выше операциях

поиска и замены, при помощи операций связывания `=~` и `!~` можно задать для преобразования строку, отличную от принятой по умолчанию

```
$str =~ tr/SEARCHLIST/REPLACEMENTLIST/cds
```

Списки `SEARCHLIST` и `REPLACEMENTLIST` задаются перечислением символов, могут содержать диапазоны — два символа, соединенных знаком "-", и иметь собственные символы-ограничители, например, `tr(a-j)/0-9/`. Операция `tr///` имеет синонимичную форму, используемую в потоковом редакторе `sed`:

```
y/SEARCHLIST/REPLACEMENTLIST/cds
```

Флаги `cds` имеют следующий смысл.

- ❑ `c` — вместо списка поиска `SEARCHLIST` использовать его дополнение до основного множества символов (обычно расширенное множество ASCII).
- ❑ `d` — удалить все символы, входящие в список поиска `SEARCHLIST`, для которых нет соответствия в списке замены `REPLACEMENTLIST`. Если флаг `d` не установлен и список замены `REPLACEMENTLIST` короче, чем список поиска `SEARCHLIST`, то вместо недостающих символов в списке замены используется последний символ этого списка. Если список замены пуст, то символы из списка поиска `SEARCHLIST` преобразуются сами в себя, что удобно использовать для подсчета числа символов в некотором классе.
- ❑ `s` — все последовательности символов, которые были преобразованы в один и тот же символ, заменяются одним экземпляром этого символа.

Пример 10.7. Операция `tr//`

```
$str =~ tr/A-Z/a-z;           # преобразование прописных букв в строчные
$count=$str =~ tr/\000//c;    # подсчет числа символов в строке $str
$str =~ tr/\200-\377/ /cs;    # любая последовательность символов
                              # с ASCII-кодами от 128 до 255 преобразуется
                              # в единственный пробел
```

Следующий скрипт преобразует русский текст в DOS-кодировке 866, содержащийся в файле "866.txt", в русский текст в Windows-кодировке 1251, и записывает преобразованный текст в файл "1251.txt".

```
open(IN866, "866.txt");
open(OUT1251, ">1251.txt");
while ($line=<IN866>) {
    $line =~ tr/\200-\257\340-\361\300-\377\250\270/;
    print OUT1251 $line;
}
```

```
close(IN866);
close(OUT1251);
```

10.2.4. Операция заключения в кавычки *qr//*

```
qr/STRING/imosx
```

Операция *qr//* по своему синтаксису похожа на другие операции заключения в кавычки, такие как *q//*, *qq//*, *qx//*, *qw//*. Она обсуждается в данном разделе, так как имеет непосредственное отношение к регулярным выражениям. Регулярное выражение, содержащее переменные, метасимволы, метапоследовательности, расширенный синтаксис, перед использованием должно быть обработано компилятором. Операция *qr//* осуществляет предварительную компиляцию регулярного выражения *STRING*, преобразуя его в некоторое внутреннее представление, с тем, чтобы сохранить скомпилированное регулярное выражение в переменной, которую затем можно использовать без повторной компиляции самостоятельно или в составе других регулярных выражений.

Преимущества от применения операции *qr//* проявляются, например, в следующей ситуации. Допустим, что мы собираемся многократно использовать в качестве образца достаточно сложное регулярное выражение, например, */^([]*) *([]*)/*. Его можно использовать непосредственно в операции сопоставления с образцом

```
if ($line =~ /^([ ]*) *([ ]*)/) {...},
```

или сохранить в переменной *\$pattern="^([]*) *([]*)"* и обращаться к переменной:

```
if ($line =~ /$pattern/) {...},
```

В обоих случаях регулярное выражение при каждом обращении обрабатывается компилятором, что при многократном использовании увеличивает время выполнения. Если сохранить образец при помощи операции *qr//*:

```
$pattn = qr/^([ ]*) *([ ]*)/,
```

то переменная *\$pattn* будет содержать откомпилированное регулярное выражение, которое можно неоднократно использовать без дополнительной компиляции.

Флаги *imosx* имеют тот же смысл, что и в операции замены *m//*. Например, в следующем тексте операция *qr//* применяется с флагами *ox*:

Пример 10.8. Операция *qr//*

```
$s="aA1Bb2cC3Dd45Ee";
@pattns=("\\d+ # последовательность цифр",
```

```

" [A-Z]+ # последовательность прописных букв",
" [a-z]+ # последовательность строчных букв");
foreach $pattn (@pattns) {
    my $pattern=qr/$pattn/ox;
    while ($s=~/$pattern/g) {
        $p=$p.$&;
    }
}
print "s=$s p=$p\n";

```

В данном примере определен массив `@pattns`, состоящий из регулярных выражений. В цикле по элементам массива проверяется наличие в заданной строке `$s` фрагмента, соответствующего текущему образцу. Найденный фрагмент добавляется в конец строки `$p`. Флаг `x` в операции `qr//` позволяет использовать образцы в том виде, в каком они хранятся в массиве — с пробелами и комментариями. Если в операции `qr//` флаг `o` не установлен, то в результате выполнения скрипта строка `$p` будет состоять из символов строки `$s`, расположенных в следующем порядке: сначала все цифры, затем все прописные буквы, затем все строчные буквы. Если, как в данном тексте, флаг `o` установлен, то в операции `$pattern=qr/$pattn/ox` подстановка переменной `$pattn` произойдет только один раз, и строка `$s` будет три раза проверяться на наличие фрагмента, удовлетворяющего первому образцу `$pattns[1]`. В результате строка `$p` будет состоять только из цифр, входящих в строку `$s`, повторенных три раза.

10.3. Функции для работы со строками

В данном разделе мы рассмотрим некоторые встроенные функции языка Perl, предназначенные для работы со строками текста. Часть из них использует рассмотренное выше понятие регулярного выражения.

Функция `chop()`

```
chop [list]
```

удаляет последний символ из всех элементов списка `list`, возвращает последний удаленный символ. Список может состоять из одной строки. Если аргумент отсутствует, операция удаления последнего символа применяется к встроенной переменной `$_`. Обычно применяется для удаления завершающего символа перевода строки, остающегося при считывании строки из входного файла.

Функция `length()`

```
length EXPR
```

возвращает длину скалярной величины `EXPR` в байтах.

Пример 10.9. Функции `chop()` и `length()`

```
#!/usr/bin/perl
$input = <STDIN>;
$Len = length($input);
print "Строка до удаления последнего символа: $input\n";
print "Длина строки до удаления последнего символа:  $Len\n";
$Chopped = chop($input);
$Len = length($input);
print "Строка после удаления последнего символа: $input\n";
print "Длина строки после удаления последнего символа:  $Len\n";
print "Удаленный символ: <$Chopped>\n";
```

Если после запуска данного скрипта ввести строку "qwerty", то вывод будет иметь вид:

```
qwerty
Строка до удаления последнего символа: qwerty
Длина строки до удаления последнего символа: 7
Строка после удаления последнего символа: qwerty
Длина строки после удаления последнего символа: 6
Удаленный символ: <
>
```

Последним символом, удаленным функцией `chop()`, является символ новой строки, сохраненный в переменной `$Chopped`. При выводе он вызывает переход на следующую строку, поэтому в данном выводе третья строка — пустая. В последней операции `print` вывод осуществляется в две строки, так как переменная `$Chopped` содержит символ новой строки.

Функции `lc()`, `uc()`, `lcfirst()`, `ucfirst()`

предназначены для преобразования строчных букв в прописные и наоборот.

Функция `lc EXPR`

возвращает выражение, полученное из выражения `EXPR` преобразованием всех символов в строчные.

Функция `uc EXPR`

возвращает выражение, полученное из выражения `EXPR` преобразованием всех символов в прописные.

Функция `lcfirst EXPR`

возвращает выражение, полученное из выражения `EXPR` преобразованием первого символа в строчный.

Функция `ucfirst EXPR`

возвращает выражение, полученное из выражения `EXPR` преобразованием первого символа в прописной.

Пример 10.10. Функции `uc()`, `lc()`, `ucfirst()`, `lcfirst()`

```
#!/usr/bin/perl
print "\nФ-ция uc() преобразует ", $s="upper case", " в ", uc $s;
print "\nФ-ция ucfirst() преобразует ", $s="uPPER CASE", " в ", ucfirst $s;
print "\nФ-ция lc() преобразует ", $s="LOWER CASE", " в ", lc $s;
print "\nФ-ция lcfirst() преобразует ", $s="Lower case", " в ", lcfirst $s;
```

В результате выполнения данного скрипта будут напечатаны строки:

```
Ф-ция uc() преобразует upper case в UPPER CASE
Ф-ция ucfirst() преобразует uPPER CASE в UPPER CASE
Ф-ция lc() преобразует LOWER CASE в lower case
Ф-ция lcfirst() преобразует Lower case в lower case
```

Функция `join()`

```
join EXPR, LIST
```

объединяет отдельные строки списка `LIST` в одну, используя в качестве разделителя строк значение выражения `EXPR`, и возвращает эту строку.

Функция `split()`

```
split [/PATTERN/[ , EXPR[ , LIMIT]]]
```

разбивает строку `EXPR` на отдельные строки, используя в качестве разделителя образец, задаваемый регулярным выражением `PATTERN`. В списковом контексте возвращает массив полученных строк, в скалярном контексте — их число. Если функция `split()` вызывается в скалярном контексте, выделяемые строки помещаются в предопределенный массив `@_`. Об этом не следует забывать, так как массив `@_` обычно используется для передачи параметров в подпрограмму, и обращение к функции `split()` неявно в скалярном контексте эти параметры уничтожит.

Если присутствует параметр `LIMIT`, то он задает максимальное количество строк, на которое может быть разбита исходная строка. Отрицательное значение параметра `LIMIT` трактуется как произвольно большое положительное число.

Если параметр `EXPR` опущен, разбивается строка `$_`. Если отсутствует также и параметр `PATTERN`, то в качестве разделителей полей используются пробельные символы после пропуска всех начальных пробельных символов (что

соответствует заданию образца в виде `/\s+/\`). К пробельным символам относятся пробел (`space`), символ табуляции (`tab`), возврат каретки (`carriage return`), символ перевода строки (`line feed`) и символ перевода страницы (`form feed`).

Замечание

Предопределенная глобальная переменная `$_` служит для обозначения используемой по умолчанию области ввода и поиска по образцу. Обычно мы осуществляем ввод при помощи операции `"<>"` ("ромб"). Внутри угловых скобок `<>` может стоять дескриптор файла ввода, например, `<STDIN>`. Если дескриптор файла отсутствует, то в качестве файлов ввода используются файлы, переданные программе Perl в качестве аргументов командной строки. Пусть, например, программа содержится в файле `script.pl`.

```
#!/usr/bin/perl
while (<>) {
    print;
};
```

Программа вызвана следующим образом

```
script.pl file1 file2 file3
```

Тогда операция `<>` будет считывать строки сначала из файла `file1`, затем из файла `file2` и, наконец, из файла `file3`. Если в командной строке файлы не указаны, то в качестве файла ввода будет использован стандартный ввод.

Только в случае, когда условное выражение оператора `while` состоит из единственной операции "ромб", вводимое значение автоматически присваивается предопределенной переменной `$_`. Вот что означают слова о том, что переменная `$_` применяется для обозначения используемой по умолчанию области ввода. Аналогично обстоит дело с поиском по образцу.

Пример 10.11. Функции `split()` и `join()`

```
#!/usr/bin/perl
while (<>) {
    chop;
    print "Число полей во входной строке '$_' равно ", $n=split;
    print "\nВходная строка разбита на строки:\n";
    foreach $i (@_) {
        print $i . "\n";
    }
    print "Объединение списка строк в одну строку через '+':\n";
    $joined = join "+", @_;
```

```
print "$joined\n";  
}
```

В результате применения операции ввода <> внутри условного выражения оператора `while` вводимая строка будет присвоена переменной `$_`. Функция `chop()` без параметров применяется к переменной `$_`. В операции `print` вторым операндом является выражение `$n=split`, в котором функция `split` вызывается в скалярном контексте и без параметров. Поэтому она применяется по умолчанию к переменной `$_`. В качестве разделителей полей по умолчанию используется множество пробельных символов, а результат помещается в массив `@_`. Затем к массиву `@_` применяется функция `join()`, объединяющая строки-элементы массива в одну строку.

Если ввести строку "one two three", то вывод будет иметь вид:

```
one two three
```

Число полей во входной строке 'one two three' равно 3

Входная строка разбита на строки:

```
one
```

```
two
```

```
three
```

Объединение списка строк в одну строку через '+':

```
one+two+three
```

Функция `index()`

```
index STR, SUBSTR[, POSITION]
```

находит первое, начиная с позиции `POSITION`, вхождение подстроки `SUBSTR` в строку `STR`, и возвращает найденную позицию. Если параметр `POSITION` не задан, по умолчанию принимается значение `POSITION = $[`. Если подстрока `SUBSTR` не найдена, возвращается значение `$[- 1`.

Замечание

Предопределенная переменная `$[` содержит индекс первого элемента в массиве и первого элемента в строке. По умолчанию ее значение равно 0. В принципе его можно изменить, но делать это не рекомендуется. Таким образом, по умолчанию значение параметра `POSITION` полагается равным 0, а функция `index` возвращает -1, если не найдена подстрока `SUBSTR`.

Функция `rindex()`

```
rindex STR, SUBSTR, POSITION
```

находит последнее, ограниченное справа позицией `POSITION`, вхождение подстроки `SUBSTR` в строку `STR`, и возвращает найденную позицию. Если подстрока `SUBSTR` не найдена, возвращается значение `$[- 1`.

Пример 10.12. Функции `index()` и `rindex()`

```
#!/bin/perl
$STR = "Этот безумный, безумный, безумный, безумный мир!";
$SUBSTR = "безумный";
$POS = 7;

print "Индекс первого символа строки по умолчанию равен ${\n";
print "Позиция первого вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",index($STR, $SUBSTR), "\n";
print "Позиция первого после позиции $POS вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",index($STR, $SUBSTR, $POS), "\n";
print "Позиция последнего вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",rindex($STR, $SUBSTR), "\n";
print "Позиция последнего перед позицией $POS вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",rindex($STR, $SUBSTR, $POS), "\n";

$|=2;

print "\nИндекс первого символа строки по умолчанию изменен на ${\n";
print "Позиция первого вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",index($STR, $SUBSTR), "\n";
print "Позиция первого после позиции $POS вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",index($STR, $SUBSTR, $POS), "\n";

print "Позиция последнего вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",rindex($STR, $SUBSTR), "\n";
print "Позиция последнего перед позицией $POS вхождения подстроки '$SUBSTR'
      в строку '$STR' = ",rindex($STR, $SUBSTR, $POS), "\n";
```

В результате выполнения скрипта будут выведены следующие строки:

Индекс первого символа строки по умолчанию равен 0

Позиция первого вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 5

Позиция первого после позиции 7 вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 15

Позиция последнего вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 35

Позиция последнего перед позицией 7 вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 5

Индекс первого символа строки по умолчанию изменен на 2

Позиция первого вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 7

Позиция первого после позиции 7 вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 7

Позиция последнего вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 37

Позиция последнего перед позицией 7 вхождения подстроки 'безумный'

в строку 'Этот безумный, безумный, безумный, безумный мир!' = 7

Функция `substr()`

`substr EXPR, OFFSET [,LENGTH [,REPLACEMENT]]`

извлекает из выражения `EXPR` подстроку и возвращает ее. Возвращаемая подстрока состоит из `LENGTH` символов, расположенных справа от позиции `OFFSET`. Если параметр `LENGTH` опущен, возвращается вся оставшаяся часть выражения `EXPR`. Если параметр `LENGTH` отрицательный, его абсолютное значение задает количество символов от конца строки, не включаемых в результирующую подстроку. Если параметр `OFFSET` имеет отрицательное значение, смещение отсчитывается с конца строки. Функция `substr()` может стоять в левой части операции присваивания. Например, в результате выполнения операторов

```
$Str = "Язык Pascal";
```

```
substr($Str, 5,6) = "Perl";
```

переменная `$Str` получит значение "Язык Perl". Тот же результат будет достигнут, если указать параметр `REPLACEMENT`, значение которого будет подставлено в `EXPR` вместо выделенной подстроки. Сама подстрока в этом случае возвращается в качестве значения функции `substr()`.

Пример 10.13. Функция `substr()`

```
#!/bin/perl
```

```
# Исходная строка
```

```
$Str = "Карл у Клары украл кораллы";
```

```
$Offset = 13;
```

```
print "Исходная строка:$Str\n";
```

```
# Смещение 13, длина подстроки не задана
```

```
$Substr = substr $Str, $Offset;
```

```
print "Смещение $Offset, длина подстроки не задана, результат:\n";
```

```
print "$Substr\n";
```

```
# Смещение 13, длина подстроки +5
$Substr = substr $Str, $Offset, 5;
print "Смещение $Offset, длина подстроки +5, результат:\n";
print "$Substr\n";

# Смещение 13, длина подстроки -1
$Substr = substr $Str, $Offset, -1;
print "Смещение $Offset, длина подстроки -1, результат:\n";
print "$Substr\n";

# Отрицательное смещение -7, длина подстроки +7
$Offset = -7;
$Substr = substr $Str, $Offset, 7;
print "Отрицательное смещение $Offset, длина подстроки +7, результат:\n";
print "$Substr\n";

# Отрицательное смещение -7, длина подстроки -1
$Substr = substr $Str, $Offset, -1;
print "Отрицательное смещение $Offset, длина подстроки -1, результат:\n";
print "$Substr\n";

# Замена подстроки
$Repl = "бокалы";
$Substr = substr $Str, $Offset, 7, $Repl;
print "В строке '$Str' слово '$Repl' заменяет слово '$Substr'\n";
```

Вывод выглядит следующим образом:

Исходная строка: 'Карл у Клары украл кораллы'

Смещение 13, длина подстроки не задана, результат:

украл кораллы

Смещение 13, длина подстроки +5, результат:

украл

Смещение 13, длина подстроки -1, результат:

украл коралл

Отрицательное смещение -7, длина подстроки +7, результат:

кораллы

Отрицательное смещение -7, длина подстроки -1, результат:

коралл

В строке 'Карл у Клары украл бокалы' слово 'бокалы' заменяет слово 'кораллы'

Функция eval()

eval EXPR

рассматривает параметр EXPR как текст Perl-программы, компилирует его и, если не обнаруживает ошибок, выполняет в текущем вычислительном окружении. Если параметр EXPR отсутствует, вместо него по умолчанию используется глобальная переменная `$_`. Компиляция программного кода EXPR осуществляется при каждом вызове функции eval() во время выполнения основной программы. Если выполнение мини-программы EXPR завершается успешно, функция eval() возвращает значение последнего выражения, вычисленного внутри EXPR. Если код EXPR содержит синтаксические ошибки, или обращение к функции die(), или возникла ошибка во время выполнения EXPR, то в специальную переменную `$@` помещается сообщение об ошибке, а функция eval() возвращает *неопределенное значение*.

Замечание

1. Если скалярной переменной не присвоено никакое допустимое значение (число, строка или ссылка), то говорят, что она имеет неопределенное значение. Неопределенные значения возникают в различных случаях, например, при попытке чтения данных после достижения конца файла или в результате системных ошибок. Неопределенное значение представляется пустой строкой "", но его следует отличать от определенного значения, равного "". Например, в результате выполнения операторов

```
$e = eval '$a = 1/0'; # деление на 0
$b = "";
print "a= $a\n" if defined $a;
print "e= $e\n" if defined $e;
print "b= $b\n" if defined $b;
```

переменные `$a` и `$e` будут иметь неопределенное значение, а переменная `$b` — определенное значение "".

Для того чтобы определить, имеет выражение EXPR определенное значение или нет, существует функция defined EXPR, возвращающая соответствующее булевское значение. В данном примере результатом выполнения будет вывод единственной строки

b=

2. Специальная переменная `$@` служит для запоминания сообщения об ошибке, возникшей при последнем обращении к функции eval().

3. Существует вторая форма функции eval()

eval BLOCK,

где BLOCK представляет собой блок — последовательность операторов, заключенную в фигурные скобки. Вторая форма отличается от первой тем, что

синтаксический анализ параметра `BLOCK` осуществляется всего один раз — во время компиляции основной программы, содержащей обращение к функции `eval()`. Таким образом, если параметр `BLOCK` содержит синтаксические ошибки, то они обнаружатся на этапе компиляции основной программы. При использовании первой формы синтаксические ошибки обнаружатся только во время выполнения.

Основным применением функции `eval()` является перехватывание исключений. *Исключением* мы называем ошибку, возникающую при выполнении программы, когда нормальная последовательность выполнения прерывается (например, при делении на ноль). Обычной реакцией на исключение является аварийное завершение программы. Язык Perl предоставляет возможность перехватывать исключения без аварийного выхода. Если исключение возникло в основной программе, то программа завершается. Если ошибка возникла внутри мини-программы функции `eval()`, то аварийно завершается только функция `eval()`, а основная программа продолжает выполняться и может реагировать на возникшую ошибку, сообщение о которой ей доступно через переменную `$@`.

В следующем примере функция `eval()` применяется для перехватывания ошибки, связанной с делением на 0 при вычислении функции `ctg(x)`. Используются встроенные функции `sin`, `cos` и `warn`. Последняя функция осуществляет вывод сообщения, задаваемого ее аргументом, на стандартное устройство вывода ошибок `STDERR`.

Пример 10.14. Функция `eval()`

```
#!/bin/perl
$fi = 0.314159265358979;
$f = '$ctg = cos($x)/sin($x)';
for $i (0..10) {
    $x = $i*$fi;
    eval $f;
    print "x= $x      ctg(x) = $ctg\n" if defined $ctg;
    warn "x= $x ", $@ if not defined $ctg;
};
```

Вывод программы выглядит следующим образом

```
x= 0 Illegal division by zero at (eval 1) line 1.
x= 0.314159265358979      ctg(x) = 3.07768353717526
x= 0.628318530717958      ctg(x) = 1.37638192047118
. . .
```

Замечание

Иногда бывает полезно искусственно вызвать исключительную ситуацию. Для этого можно воспользоваться функцией `die()` LIST. Назначение функции `die()` — генерировать исключения. Если функция `die()` вызывается в основной программе вне функции `eval()`, то она осуществляет аварийное завершение основной программы и выводит сообщение об ошибке LIST на стандартное устройство вывода ошибок `STDERR`. Если она вызывается внутри функции `eval()`, то осуществляет аварийное завершение `eval()` и помещает сообщение об ошибке в специальную переменную `$@`.

Функция `pos()`

```
pos [$SCALAR]
```

возвращает позицию, в которой завершился последний глобальный поиск `$SCALAR =~ m/.../g`, осуществленный в строке, задаваемой переменной `$SCALAR`. Возвращаемое значение равно числу `length($`) + length($&)`. Следующий глобальный поиск `m/.../g` в данной строке начнется именно с этой позиции.

Если аргумент `$SCALAR` отсутствует, возвращается позиция завершения последнего глобального поиска, осуществленного в строке `$_`.

Пример 10.15. Функция `pos()`

```
$words = "one two three four";
while ($words =~ m/\w+/g) {
    print "pos=", pos($words), " length(\`)=", length($`),
          " length(\`&)=", length($&), "\n";
}
```

В результате выполнения данного скрипта будут выведены номера позиций, соответствующих окончаниям слов в строке `$words`:

```
pos=3   length($`) = 0   length($&) = 3
pos=7   length($`) = 4   length($&) = 3
pos=13  length($`) = 8   length($&) = 5
pos=18  length($`) = 14  length($&) = 4
```

Функцию `pos()` можно использовать в левой части операции присваивания для изменения начальной позиции следующего поиска:

```
# изменение начальной позиции для последующего поиска
$words = "one two three four";
pos $words = 4;
while ($words =~ m/\w+/g) {
```

```
print pos $words, "\n";
```

```
}
```

В последнем случае поиск слов начнется со второго слова, и будут выведены номера позиций 7, 13 и 18.

Функция `quotemeta()`

```
quotemeta [EXPR]
```

возвращает строку `EXPR`, в которой все символы, кроме алфавитно-цифровых символов и символа подчеркивания `"_"`, экранированы символом `"\"`. Например, в результате выполнения

```
print quotemeta "*****", "\n";
```

будет выведена строка

```
\*\*\*\*\*
```

Если аргумент `EXPR` отсутствует, вместо него используется переменная `$_`.

Вопросы для самоконтроля

1. Что такое регулярное выражение?
2. Какие символы имеют в регулярном выражении Perl специальное значение?
3. Что такое метапоследовательность, как она образуется?
4. Что такое обратная ссылка?
5. Какая переменная используется в операции подстановки по умолчанию?
6. Какой смысл имеет символ `"$"` в следующих регулярных выражениях:

```
/abc*$/
```

```
/[abc*$]/
```

```
/$abc/
```

7. Какой смысл имеет символ `"^"` в следующих регулярных выражениях:

```
/^abc/
```

```
/[^abc]/
```

```
/abc^/
```

8. Объясните, какие множества строк соответствуют следующим образцам. Приведите пример.

```
/a.out/
```

```
/a\.out/
```

```
/\d{2,3}-\d{2}-\d{2}/
```

```
/(.)(.)\2\1/  
/(.)(.)\02\01/
```

9. Напишите образец, задающий палиндром из шести букв.

10. Напишите команду замены, которая:

- заменяет все символы новой строки пробелами;
- выделяет из полного маршрутного имени файла имя файла;
- выделяет из полного маршрутного имени файла имя каталога.

11. Каково значение следующих выражений, если значение переменной \$var равно "123qwerty"?

```
$var =~ /./  
$var =~ /[A-Z]*/  
$var =~ /\w{4-9}/  
$var =~ /(\d)2(\1)/  
$var =~ /qwerty$/  
$var =~ /123?/
```

11. Какое значение будет иметь переменная \$var после следующих операций подстановки, если ее начальное значение равно "qwerty123qwerty"?

```
$var =~ s/qwerty/XYZ/;  
$var =~ s/[a-z]/X/g;  
$var =~ s/B/W/i;  
$var =~ s/(.)\d.*\1/d/;  
$var =~ s/(\d+)/$1*2/e;
```

12. Начальное значение переменной \$var равно "qwerty123qwerty". Каким оно будет после выполнения операций транслитерации?

```
$var =~ tr/a-z/A-Z/;  
$var =~ tr/a-z/0-9/;  
$var =~ tr/a-z/0-9/d;  
$var =~ tr/231/564/;  
$var =~ tr/123/ /s;  
$var =~ tr/123//cd;
```

13. Переменная \$var имеет значение "qwertyqwerty". Каково значение, возвращаемое функцией?

```
substr ($var, 0, 3);  
substr ($var, 4);  
substr ($var, -2, 2);  
substr ($var, 2, 0);
```



```

index ($var, "rt");
index ($var, "rtyu");
index ($var, "er", 1);
index ($var, "er", 7);
rindex ($var, "er");

```

Упражнения

1. Напишите программу, которая читает стандартный ввод, умножает каждое встретившееся число на 2 и выводит результирующую строку.
2. Напишите программу, которая читает стандартный ввод, удваивает каждую букву и выводит результирующую строку.
3. Напишите программу, подсчитывающую, сколько раз каждый алфавитно-цифровой символ встретился во входном файле.
4. Напишите программу, которая считывает строку из стандартного файла ввода, меняет в ней порядок следования символов на обратный и выводит результат.
5. Напишите программу, которая выполняет преобразование русского текста из одной системы кодировки в другую:

```

{Dos 866, Windows 1251, UNIX KOI8} <=> {Dos 866, Windows 1251,
UNIX KOI8}

```

Для выполнения задания можно воспользоваться табл. 10.3, содержащей шестнадцатеричные коды символов русского алфавита.

Таблица 10.3. Таблицы кодов русского алфавита

Символ	866	1251	KOI8	Символ	866	1251	KOI8
А	80	C0	E1	а	A0	E0	C1
Б	81	C1	E2	б	A1	E1	C2
В	82	C2	F7	в	A2	E2	D7
Г	83	C3	E7	г	A3	E3	C7
Д	84	C4	E4	д	A4	E4	C4
Е	85	C5	E5	е	A5	E5	C5
Ё	F0	A8	B3	ё	F1	B8	A3
Ж	86	C6	F6	ж	A6	E6	D6
З	87	C7	FA	з	A7	E7	DA
И	88	C8	E9	и	A8	E8	C9

Таблица 10.3 (окончание)

Символ	866	1251	KOI8	Символ	866	1251	KOI8
Й	89	C9	EA	й	A9	E9	CA
К	8A	CA	EB	к	AA	EA	CB
Л	8B	CB	EC	л	AB	EB	CC
М	8C	CC	ED	м	AC	EC	CD
Н	8D	CD	EE	н	AD	ED	CE
О	8E	CE	EF	о	AE	EE	CF
П	8F	CF	F0	п	AF	EF	D0
Р	90	D0	F2	р	E0	F0	D2
С	91	D1	F3	с	E1	F1	D3
Т	92	D2	F4	т	E2	F2	D4
У	93	D3	F5	у	E3	F3	D5
Ф	94	D4	E6	ф	E4	F4	C6
Х	95	D5	E8	х	E5	F5	C8
Ц	96	D6	E3	ц	E6	F6	C3
Ч	97	D7	FE	ч	E7	F7	DE
Ш	98	D8	FB	ш	E8	F8	DB
Щ	99	D9	FD	щ	E9	F9	DD
Ъ	9A	DA	FF	ъ	EA	FA	DF
Ы	9B	DB	F9	ы	EB	FB	D9
Ь	9C	DC	F8	ь	EC	FC	D8
Э	9D	DD	FC	э	ED	FD	DC
Ю	9E	DE	E0	ю	EE	FE	C0
Я	9F	DF	F1	я	EF	FF	D1



Подпрограммы и функции

Подпрограммы в языке Perl играют ту же роль, что и функции в языке C, или процедуры и функции в языке Pascal. Они выполняют две основные задачи:

- ❑ позволяют разбить одну большую программу на несколько небольших частей, делая ее более ясной для понимания;
- ❑ объединяют операторы в одну группу для повторного использования.

В языке Perl не различаются понятия "подпрограмма" и "функция", эти слова являются синонимами.

11.1. Определение подпрограммы

Подпрограмма может быть определена в любом месте основной программы при помощи описания

```
sub name [(proto)] [{block}];
```

Здесь

name имя подпрограммы;

(proto) прототип, конструкция, используемая для описания передаваемых подпрограмме параметров;

{block} блок операторов, являющийся определением подпрограммы и выполняющийся при каждом ее вызове.

Форма

```
sub name [(proto)];
```

представляет собой предварительное объявление подпрограммы без ее определения. Пользователь, предпочитающий помещать описания всех подпрограмм в конце основной программы, должен при вызове еще не определенной функции использовать специальный синтаксис `&name` или `name()` (см. раздел 11.2). Если же некоторое имя предварительно объявить в качестве имени функции, то сразу после объявления к этой функции можно обращаться просто по имени без применения специального синтаксиса.

Пример 11.1. Определение подпрограммы

```
#!/usr/bin/perl

sub max {
    my $maximum = shift @$_;
    my $x;
    foreach $x (@_) {
        $maximum=$x if ($x > $maximum);
    }
    return $maximum
}

print "Наибольший аргумент=", max(3,5,17,9), "\n";
```

В данном примере функция `max()` возвращает наибольший из своих аргументов. Об использовании функции `my()` и массива `@_` будет рассказано ниже.

Данный способ определения подпрограмм не является единственным. Существуют и другие варианты:

- ☐ текст подпрограммы может храниться в отдельном файле и загружаться в основную программу при помощи ключевых слов `do`, `require`, `use`;
- ☐ строка, содержащая текст подпрограммы, может быть передана в качестве аргумента функции `eval()` (см. *раздел 10.3*); в этом случае компиляция кода подпрограммы осуществляется при каждом вызове функции `eval()`;
- ☐ анонимную подпрограмму можно определить при помощи ссылки на нее (см. *раздел 9.2.4.1*).

Применение функции `eval()` и ссылки на анонимную подпрограмму были рассмотрены ранее.

Конструкция `do filename` вызывает выполнение Perl-программы, содержащейся в файле `filename`. Если файл `filename` недоступен для чтения, функция `do` возвращает неопределенное значение и присваивает соответствующее значение специальной переменной `$!`. Если файл `filename` может быть прочитан, но возникают ошибки при его компиляции или выполнении, то функция `do` возвращает неопределенное значение и помещает в переменную `$@` сообщение с указанием строки, содержащей ошибку. Если компиляция прошла успешно, функция `do` возвращает значение последнего выражения, вычисленного в файле `filename`.

Замечание

Специальная переменная `$!` служит для хранения сообщения о последней системной ошибке. Такая ошибка возникает при обращении к операционной

системе с запросом на предоставление некоторой услуги, как, например, создание файла, чтение или запись в него.

Специальная переменная `$@` используется для хранения сообщения, генерируемого при последнем обращении к функциям `eval()` или `do filename`.

Пример 11.2. Применение конструкции `do filename`

```
# файл "1.pl":
#!/usr/bin/perl
do "2.pl";
print "ошибка: $@\n" if $@;
do "3.pl";
print "системная ошибка: $!\n" if $!;

# файл "2.pl":
$x=1;
$y=0;
$z=$x/$y;
print "z= $z\n";
```

Perl-программа "1.pl", используя конструкцию `do filename`, пытается выполнить сценарии, содержащиеся в файлах "2.pl" и "3.pl". Первый из них содержит в третьей строке операцию деления на 0, вызывающую появление ошибки во время выполнения программы, а второй вообще не существует. В результате выполнения файла "1.pl" будут выведены следующие сообщения:

ошибка: Illegal division by zero at 2.pl line 3.

системная ошибка: No such file or directory

Ключевые слова `use` и `require` используются для включения в текущую программу подпрограмм из других модулей.

(Директивы компилятора `use` и `require` рассмотрены в главе 12.)

11.2. Вызов подпрограммы

Мы знаем, что принадлежность к тому или иному типу определяется префиксом имени: `$x` — скалярная переменная, `@x` — массив, `%x` — ассоциативный массив. Префиксом функции является символ "&". К любой подпрограмме можно обратиться, указав ее имя с префиксом `&`:

```
&name args;
&name(args);
&name;
```

Здесь `args` обозначает список аргументов подпрограммы. Если список аргументов отсутствует, вместо него используется специальный массив `@_`.

Если после имени подпрограммы следуют скобки, префикс `&` можно опустить:

```
name(args);
name();
```

Если до обращения к ней подпрограмма была объявлена или импортирована, то скобки также можно опустить:

```
sub name { . . . };
. . .
name args;
. . .
name;
```

Если подпрограмма вызывается через ссылку на нее, префикс является обязательным:

```
$subref = sub { . . . };
. . .
&subref(args);
. . .
&subref;
```

Подпрограмма может быть использована в выражении как функция, возвращающая значение. По умолчанию значением подпрограммы является последнее вычисленное в ней выражение. Его можно изменить, указав явно в качестве аргумента функцию `return()` в любой точке подпрограммы. Возвращаемое значение может быть скалярной величиной или массивом.

11.3. Локальные переменные в подпрограммах

Областью видимости или *областью действия* переменной мы будем называть часть программы, где данная переменная может быть использована. В языке Perl, как мы знаем, нет обязательного явного описания переменных. Точкой определения переменной является место, где она впервые встречается в программе. Область действия большинства переменных ограничена *пакетом*. Исключение составляют некоторые специальные предопределенные глобальные переменные интерпретатора perl. Пакет — это механизм, позволяющий создать свое пространство имен для некоторого отрезка программы (этот отрезок может включать всю программу). Каждый фрагмент кода Perl-программы относится к соответствующему пакету.

(Пакеты рассматриваются в главе 12, а специальные переменные — в главе 14.)

Таким образом, переменная, впервые встретившаяся в некоторой подпрограмме, становится доступной во всем пакете, к которому эта подпрограмма принадлежит. Любая переменная в Perl по умолчанию считается глобальной, но эта глобальность ограничена рамками пакета. Иногда бывает необходимо ограничить область действия переменной рамками подпрограммы или блока, в которых она определена. Такие переменные называются локальными. В языке Perl существуют два способа описания локальных переменных: при помощи функций `my()` и `local()`.

11.3.1. Функция *my()*

Функция `my()` используется для объявления одной или нескольких переменных локальными:

```
my EXPR
```

и ограничивает их область действия:

- подпрограммой;
- заключенным в фигурные скобки блоком операторов;
- выражением, переданным на выполнение функции `eval()` (см. раздел 10.3);
- файлом, в зависимости от того, в каком месте вызвана для объявления переменных сама функция `my()`.

Если выражение `EXPR` содержит список переменных, то он должен быть заключен в скобки:

```
my ($myvar, @mylist, %myhash);
```

Одновременно с объявлением переменные могут быть инициализированы:

```
my $pi = 3.14159;
```

```
my ($pi, $exp) = (3.14159, 2.71828);
```

Переменные, объявленные при помощи функции `my()`, доступны в своей области действия *только* для подпрограмм, определенных в этой области. Для подпрограмм, определенных за ее пределами, они недоступны. Такие переменные называют *лексическими*, а саму область видимости — *лексической* или *статической областью видимости*.

11.3.2. Функция *local()*

Функция `local()` также используется для объявления и инициализации переменных:

```
local EXPR;
```

```
local ($myvar, @mylist, %myhash);
```

```
local $pi = 3.14159;
```

```
local ($pi, $exp) = (3.14159, 2.71828);
```

но, в отличие от функции `my()` она создает не локальные переменные, а временные значения для глобальных переменных внутри:

- подпрограммы;
- заключенного в фигурные скобки блока операторов;
- выражения, переданного на выполнение функции `eval()`;
- файла;

в зависимости от того, в каком месте вызвана для объявления переменных сама функция `local()`. Если функция `local()` применяется для описания нескольких переменных, они должны быть заключены в скобки. Если глобальная переменная, объявленная при помощи этой функции, ранее встречалась до объявления и имела некоторое значение, то это значение сохраняется в скрытом стеке и восстанавливается после выхода соответственно из подпрограммы, блока, функции `eval()` или файла. Переменная, объявленная при помощи функции `local()`, или, точнее, ее временное значение, доступна для *любой* функции, вызванной внутри подпрограммы, блока, функции `eval()` или файла, в которых сделано объявление. Такую переменную называют *динамической*, а ее область видимости — *динамической областью видимости*. В названии отражается тот факт, что область видимости переменной динамически изменяется с каждым вызовом функции, получающей доступ к этой переменной.

Функция `my()` является относительно новой, она появилась в версии Perl 5. Для создания действительно локальных переменных рекомендуется использовать именно функцию `my()`, а не функцию `local()`. Впрочем, есть несколько исключений. О них мы расскажем ниже.

В следующем примере показано, чем отличаются переменные, объявленные при помощи функций `my()` и `local()`.

Пример 11.3. Описание переменных при помощи функций `my` и `local`

```
sub f1{
    local ($x) = "aaaa";
    my ($y)     = "bbbb";
    print("f1: x = $x\n");
    print("f1: y = $y\n\n");
    f2();
    print("f1: x = $x\n");
    print("f1: y = $y\n\n");
}

sub f2{
    print("f2: x = $x\n");
    print("f2: y = $y\n\n");
```



```
$x = "cccc";  
$y = "dddd";  
print("f2: x = $x\n");  
print("f2: y = $y\n\n");  
}  
f1;
```

Результатом выполнения данного примера будет следующий вывод:

```
f1: x = aaaa  
f1: y = bbbb  
  
f2: x = aaaa  
f2: y =  
  
f2: x = cccc  
f2: y = dddd  
  
f1: x = cccc  
f1: y = bbbb
```

Как видно из приведенного результата, функция `f2()` не имеет доступа к переменной `$y`, объявленной при помощи функции `my()` внутри функции `f1()`, и, напротив, имеет доступ к переменной `$x`, объявленной внутри `f1()` при помощи функции `local()`.

11.4. Передача параметров

Информация в подпрограмму и обратно передается через параметры (аргументы). Для передачи параметров в подпрограмму используется специальный массив `@_`. Все параметры запоминаются в элементах массива `$_[0]`, `$_[1]` и т. д. Такой механизм позволяет передавать в подпрограмму произвольное количество параметров.

Массив `@_` является локальным для данной подпрограммы, но его элементы — это псевдонимы действительных скалярных параметров. Изменение элемента массива `@_` вызывает изменение соответствующего действительного параметра.

В языках программирования различают передачу параметров *по ссылке* и *по значению*. При передаче параметров по значению подпрограмма получает копию переменной. Изменение копии внутри подпрограммы не влияет на ее оригинал. При передаче параметров по ссылке подпрограмма получает доступ к самой переменной и может ее изменять.

Передача параметров через специальный массив `@_` фактически является передачей параметров по ссылке. В языке Perl можно реализовать передачу

параметров по значению, если внутри подпрограммы при помощи функции `my()` объявить локальные переменные и присвоить им значения фактических параметров из массива `@_`, как это сделано в следующем примере.

Пример 11.4. Использование функции `my` для передачи параметров по значению

```
#!/usr/bin/perl
# Передача в подпрограмму параметров по значению
sub f {
    my($x, $y) = @_;
    return (++$x * --$y);
}
$val = f(9,11);
print "Значение (9+1) * (11-1) равно $val.\n";
$x = 9;
$y = 11;
$val = f($x,$y);
print "Значение ($x+1) * ($y-1) равно $val.\n";
print "Значение \$x остается равным $x, а \$y равным $y.\n";
```

Результат выполнения:

Значение (9+1) * (11-1) равно 100.

Значение (9+1) * (11-1) равно 100.

Значение \$x остается равным 9, а \$y равным 11.

11.4.1. Передача по ссылке параметров-массивов

Итак, подпрограмма получает и возвращает параметры через специальный массив `@_`. Если параметр является массивом или хеш-массивом, его элементы также сохраняются в массиве параметров `@_`. При передаче в подпрограмму нескольких параметров-массивов или хеш-массивов они утрачивают свою целостность. Иными словами, после записи параметров-массивов (хеш-массивов) в массив `@_` из него невозможно выделить отдельный параметр-массив (хеш-массив): все параметры в массиве `@_` хранятся единой "кучей". Для сохранения при передаче в подпрограмму целостности массива или хеш-массива существуют два основных подхода.

11.4.1.1. Использование типа *typeglob*

Первый подход, более старый, заключается в использовании внутреннего типа данных, называемого `typeglob`. Принадлежность к типу `typeglob` обо-

значается префиксом `"*"`. Префикс `"*"` можно рассматривать как метасимвол, вместо которого может стоять любой из префиксов `"$"`, `"@"`, `"%"`, `"&"`, обозначающих тип данных "скаляр", "массив", "хеш-массив", "функция" соответственно. Интерпретатор преобразует переменную типа `typeglob`, например, `*abc`, в скалярную величину. Эта величина является ссылкой на гнездо в таблице символов, содержащее элементы разных типов с одинаковым именем `abc`, и представляет любой из этих элементов. Например, запись `*abc` обозначает всю совокупность, а также любую из следующих переменных: скаляр `$abc`, массив `@abc`, хеш `%abc`, функция `&abc`.

(Таблицы символов обсуждаются в главе 12.)

Передача в подпрограмму вместо параметра-массива или хеш-массива соответствующей переменной типа `typeglob` является имитацией передачи параметра-массива (хеш-массива) по ссылке с сохранением его целостности. Рассмотрим следующий пример.

Пример 11.5. Использование типа `typeglob` для передачи параметров-массивов и хеш-массивов

```
sub doublargs {
    local(*mylist, *myhash) = @_;
    foreach $item (@mylist) {
        $item *= 2;
    }
    foreach $key (keys %myhash) {
        $myhash{$key} *= 2;
    }
}

@somelist=(1,2,3);
%somehash=("one"=>5, "two"=>15, "three"=>20);
print "начальные значения:\n"@somelist=@somelist\n";
foreach $key (keys %somehash) {
    print "\$somehash{$key}=$somehash{$key} ";
}
print "\n";
doublargs(*somelist,*somehash);
print "итоговые значения:\n"@somelist=@somelist\n";
foreach $key (keys %somehash) {
    print "\$somehash{$key}=$somehash{$key} ";
}
print "\n";
```

Подпрограмма `doublargs` принимает на вход массив и хеш-массив и изменяет их элементы, умножая на 2. Вместо массива и хеш-массива в подпрограмму передаются соответствующие переменные типа `typeglob`, которые легко выделить из массива `@_`, так как фактически они являются скалярами. Обратите внимание на применение функции `local`. Использовать вместо нее функцию `my` здесь нельзя. Переменная типа `typeglob` не может быть локальной, она представляет несколько одноименных переменных разных типов из таблицы символов. Далее возникает вопрос, каким образом изменение в подпрограмме массива `@mylist` влияет на изменение фактического параметра `@someslist`. Дело в том, что операция присваивания вида `*x = *y` создает синоним `*x` для гнезда таблицы символов `*y`, так что осуществление операции над `$x`, `@x`, `%x` эквивалентно осуществлению этой операции над `$y`, `@y`, `%y`. В результате присваивания

```
local(*mylist, *myhash) = @_;
```

создается псевдоним `*mylist` для `*someslist`, поэтому все изменения элементов массива `@mylist` внутри подпрограммы эквивалентны изменениям элементов массива `@someslist`. Все сказанное справедливо и для хеш-массивов `%myhash` и `%somehash`. Результат подтверждает корректность передачи массива и хеш-массива по ссылке:

начальные значения:

```
@someslist=1 2 3
```

```
$somehash{one}=5 $somehash{three}=20 $somehash{two}=15
```

итоговые значения:

```
@someslist=2 4 6
```

```
$somehash{one}=10 $somehash{three}=40 $somehash{two}=30
```

11.4.1.2. Использование ссылок

Второй, более новый способ передачи массивов в подпрограмму заключается в том, чтобы вместо собственно массивов или хеш-массивов передавать ссылки на них. Ссылка является скалярной величиной и ее легко выделить в массиве параметров `@_`. Внутри подпрограммы остается только применить к ссылке операцию разыменования для того, чтобы получить доступ к фактическому параметру. Поскольку ссылки появились только в версии Perl 5, то этот способ является относительно новым. При помощи ссылок предыдущий пример можно записать в следующем виде.

Пример 11.6. Использование ссылок для передачи параметров-массивов и хеш-массивов

```
sub doublparms {
my ($listref, $hashref) = @_;
```

```

foreach $item (@$listref) {
    $item *= 2;
}
foreach $key (keys %$hashref) {
    $$hashref{$key} *= 2;
}
}

@somelist=(1,2,3);
$somehash=("one"=>5, "two"=>15, "three"=>20);
print "начальные значения:\n@somelist=@somelist\n";
foreach $key (keys %somehash) {
    print "\$somehash{$key}=$somehash{$key} ";
}
print "\n";
doublparms(\@somelist,\%somehash);
print "итоговые значения:\n\n@somelist=@somelist\n";
foreach $key (keys %somehash) {
    print "\$somehash{$key}=$somehash{$key} ";
}
print "\n";

```

Здесь для описания локальных переменных использована функция `my`. Как мы выяснили ранее в этой главе, применение функции `my` в подобном случае реализует передачу параметров по значению. Другими словами, их изменение внутри подпрограммы не влияет на фактические параметры. Каким же образом в данном случае осуществляется передача массива и хеш-массива по ссылке? Дело в том, что по значению передаются только *ссылки*, указывающие на фактические параметры: массив `@somelist` и хеш-массив `%somehash`. Используя операции разыменования внутри подпрограммы, мы получаем доступ непосредственно к массиву `@somelist` и хеш-массиву `%somehash`, и изменяем их элементы. В результате выполнения данного сценария будет выведено:

начальные значения:

```
@somelist=1 2 3
```

```
$somehash{one}=5 $somehash{three}=20 $somehash{two}=15
```

итоговые значения:

```
@somelist=2 4 6
```

```
$somehash{one}=10 $somehash{three}=40 $somehash{two}=30
```

11.5. В каких случаях функцию *local* нельзя заменить функцией *my*

В следующих случаях функция `local()` является незаменимой.

❑ Присваивание временного значения глобальной переменной.

В первую очередь это относится к некоторым предопределенным глобальным переменным, таким как `$ARGV`, `$_` и т. д. Рассмотрим пример.

Пример 11.7. Создание временного значения глобальной переменной

```
#!/usr/bin/perl
$/ = undef;
@ARGV = ("a");
$_ = <>;
print "Первое значение области ввода \$_ = ", split, "\n";
{
    local @ARGV = ("aa");
    local $_ = <>;
    print "Второе значение области ввода \$_ = ", split, "\n";
}
{
    local @ARGV = ("aaa");
    local $_ = <>;
    @fields = split;
    print "Третье значение области ввода \$_ = ", split, "\n";
}
print "Восстановленное значение области ввода \$_ = ", split, "\n";
```

Пусть имеются три файла

"a":	"aa":	"aaa":
1111 1111 1111	2222 2222 2222	3333 3333 3333
aaaa bbbb cccc	dddd eeee ffff	gggg hhhh iiii

В приведенной программе используются специальные глобальные переменные `$/`, `$_` и `@ARGV`.

Специальная переменная `$/` содержит значение разделителя входных записей, которым по умолчанию является символ новой строки. Присваивание этой переменной неопределенного значения позволяет при помощи одной операции ввода `<>` считать весь файл, а не только первую строку.

Специальная переменная `$_` используется по умолчанию для сохранения вводимых данных, если в операции ввода соответствующий параметр не указан. Она также используется по умолчанию в качестве аргумента функции `split()`, если в ней не задана строка, подлежащая разбиению на отдельные строки.

Массив `@ARGV` содержит аргументы командной строки самой программы. Если при вызове программы ей будет передано имя файла, то оно будет сохранено в массиве `@ARGV`. Операция ввода `<>` применяется к файлам, переданным в программу в качестве аргументов командной строки, т. е. к файлам, имена которых хранятся в массиве `@ARGV`. В нашем примере программа вызывается без аргументов, поэтому имя входного файла "a" задается внутри программы прямой записью в массив `@ARGV`. Первая операция ввода `<>`, следовательно, осуществляется из файла "a". Далее следуют два блока операторов, заключенных в фигурные скобки. В каждом из них при помощи функции `local()` создаются временные значения для глобальных переменных `@ARGV` и `$_`. В первом блоке данные считываются из файла "aa" и сохраняются в качестве временного значения глобальной переменной `$_`, во втором — из файла "aaa" и также сохраняются в качестве следующего временного значения переменной `$_`. По выходе из второго блока глобальная переменная `$_` восстанавливает свое первоначальное значение. В результате выполнения данной программы будет напечатано:

Первое значение области ввода `$_ = 111111111111aaaabbbbcccc`

Второе значение области ввода `$_ = 222222222222ddddeeeeffff`

Третье значение области ввода `$_ = 333333333333ggggghhhhiiii`

Восстановленное значение области ввода `$_ = 111111111111aaaabbbbcccc`

❑ Создание локального дескриптора файла, каталога или локального псевдонима для функции.

В следующем примере функция `local()` используется для создания локального дескриптора файла внутри блока операторов.

Пример 11.8. Создание локального дескриптора файла

```
#!/usr/bin/perl
open(FILEHANDLE, ">b");
print FILEHANDLE "Новая строка в файл 'b'\n";
{
    local *FILEHANDLE;
    open(FILEHANDLE, ">bb");
    print FILEHANDLE "Новая строка в файл 'bb'\n";
    close FILEHANDLE;
}
```

```
{
    local *FILEHANDLE;
    open(FILEHANDLE, ">bbb");
    print FILEHANDLE "Новая строка в файл 'bbb'\n";
    close FILEHANDLE;
}

print FILEHANDLE "Еще одна строка в файл 'b'\n";
close FILEHANDLE;
```

В результате выполнения данного сценария в текущем каталоге будут созданы файлы:

```
"b":
Новая строка в файл 'b'
Еще одна строка в файл 'b'
"bb":
Новая строка в файл 'bb'
"bbb":
Новая строка в файл 'bbb'
```

Заметьте, что во время выполнения операций с файлами "bb" и "bbb" файл "b" остается открытым.

Аналогичным образом может быть определено локальное имя для функции.

Пример 11.9. Создание локального псевдонима функции

```
#!/usr/bin/perl

# функция NumberOfArgs() возвращает число своих параметров
sub NumberOfArgs {
    return $_[0] + 1;
}

print "NumberOfArgs: число параметров=", NumberOfArgs(1,2,3,4), "\n";

{
    local *Numbers = *NumberOfArgs;
    print "Numbers: число параметров=", Numbers(1,2,3), "\n";
}

{
    local *N = \&NumberOfArgs;
    print "N: число параметров=", N(1,2), "\n";
}
```


Результат выполнения:

NumberOfArgs: число параметров=4
 Numbers: число параметров=3
 N: число параметров=2

❑ Временное изменение элемента массива или хеш-массива.

В следующем примере внутри блока операторов временно изменяется значение одного элемента глобального хеш-массива %ENV, содержащего значение переменной \$PATH, входящей в состав среды интерпретатора UNIX shell.

Пример 11.10. Временное изменение элемента хеш-массива

```
#!/usr/bin/perl
print "значение переменной среды $PATH:\n$ENV{PATH}\n";
{
  local $ENV{PATH} = "/home/mike/bin";
  print "временное значение переменной среды $PATH: $ENV{PATH}\n";
}
print "прежнее значение переменной среды $PATH:\n$ENV{PATH}\n";
```

Результат будет выведен в следующем виде:

```
значение переменной среды $PATH:
/sbin:/usr/sbin:/usr/bin:/bin:/usr/X11R6/bin:/usr/local/bin:/opt/bin
временное значение переменной среды $PATH: /home/mike/bin
прежнее значение переменной среды $PATH:
/sbin:/usr/sbin:/usr/bin:/bin:/usr/X11R6/bin:/usr/local/bin:/opt/bin
```

11.6. Прототипы

Встроенные функции Perl имеют определенный синтаксис: имя, число и тип параметров. Прототипы позволяют накладывать ограничения на синтаксис функции, объявляемой пользователем. Прототип представляет собой запись, которая состоит из заключенного в скобки списка символов, определяющих количество и тип параметров подпрограммы. Например, объявление

```
sub func ($$) {
    ...
}
```

определяет функцию func() с двумя скалярными аргументами. Символы для обозначения типа аргумента приведены в табл. 11.1.

Таблица 11.1. Символы, используемые в прототипах для задания типа аргумента

Символ	Тип данных
\$	Скаляр
@	Массив
%	Ассоциативный массив
&	Анонимная подпрограмма
*	Тип <code>typeglob</code>

Запись вида `\char`, где `char` — один из символов табл. 11.1, обозначает, что при вызове подпрограммы имя фактического параметра должно обязательно начинаться с символа `char`. В этом случае в подпрограмму через массив параметров `@_` передается ссылка на фактический параметр, указанный при ее вызове. Обязательные параметры в прототипе отделяются от необязательных точкой с запятой.

В табл. 11.2 в качестве примера приведены объявления пользовательских функций `mybuiltin()`, синтаксис которых соответствует синтаксису встроенных функций `builtin()`.

Таблица 11.2. Примеры прототипов

Объявление	Обращение к функции
<code>sub mylink (\$\$)</code>	<code>mylink \$old, \$new</code>
<code>sub myvec (\$\$\$)</code>	<code>myvec \$var, \$offset, 1</code>
<code>sub myindex (\$\$;\$)</code>	<code>myindex &getstring, "substr"</code>
<code>sub mysyswrite (\$\$\$;\$)</code>	<code>mysyswrite \$buf, 0, length(\$buf) - \$off, \$off</code>
<code>sub myreverse (@)</code>	<code>myreverse \$a, \$b, \$c</code>
<code>sub myjoin (\$@)</code>	<code>myjoin ":", \$a, \$b, \$c</code>
<code>sub mypop (\@)</code>	<code>mypop @array</code>
<code>sub mysplICE (\@\$\$\$@)</code>	<code>mysplICE @array, @array, 0, @pushme</code>
<code>sub mykeys (\%)</code>	<code>mykeys %{\$hashref}</code>
<code>sub myopen (*;\$)</code>	<code>myopen HANDLE, \$name</code>
<code>sub mypipe (**)</code>	<code>mypipe READER, WRITER</code>
<code>sub mygrep (&@)</code>	<code>mygrep { /pattern/ } \$a, \$b, \$c</code>
<code>sub myrand (\$)</code>	<code>myrand 42</code>
<code>sub mytime ()</code>	<code>mytime</code>

Следует иметь в виду, что проверка синтаксиса, задаваемого при помощи прототипа, не осуществляется, если подпрограмма вызвана с использованием префикса `&: &subname`.

11.7. Рекурсивные подпрограммы

Язык Perl допускает, чтобы подпрограмма вызывала саму себя. Такая подпрограмма называется *рекурсивной*. При написании рекурсивных подпрограмм следует иметь в виду, что все переменные, значения которых изменяются внутри подпрограммы, должны быть локальными, т. е. объявленными при помощи функций `my()` или `local()`. В этом случае при каждом вызове подпрограммы создается новая копия переменной. Это позволяет избежать неопределенности и замещения текущего значения переменной ее значением из следующего вызова подпрограммы.

Рекурсивные подпрограммы следует применять осторожно. Многие алгоритмы, являющиеся по сути итеративными, можно реализовать при помощи рекурсивной подпрограммы. Однако такая подпрограмма окажется неэффективной по времени выполнения и потребляемым ресурсам памяти. Вместе с тем, существуют задачи, решить которые можно только при помощи рекурсивных алгоритмов. В этом случае применение рекурсивных подпрограмм является не только вполне оправданным, но и необходимым. Одной из таких задач, которую операционная система решает постоянно, является рекурсивный просмотр дерева каталогов. Рассмотрим пример рекурсивной Perl-подпрограммы `tree()`, которая делает то же самое: просматривает дерево каталогов, начиная с каталога, заданного параметром подпрограммы, и выводит список файлов, содержащихся в каждом подкаталоге.

Пример 11.11. Рекурсивный просмотр дерева каталогов

```
sub tree {
    local (*ROOT);
    my ($root)=$_[0];
    opendir ROOT, $root;
    my (@filelist) = readdir ROOT;
    closedir ROOT;
    for $x (@filelist) {
        if ($x ne "." and $x ne "..") {
            $x=$root."/".$x;
            print "  $x\n" if (-f $x);
            if (-d $x) {
                print "$x:\n";
            }
        }
    }
}
```

```

        tree($x);
    }
}
}
}

```

Здесь использованы встроенные подпрограммы Perl `opendir()`, `closedir()`, `readdir()`, применяемые соответственно для открытия каталога, его закрытия и чтения содержимого. Подпрограмма `tree()` рекурсивно просматривает каталог, переданный ей в качестве параметра, и выводит имена вложенных подкаталогов и содержащихся в них файлов в следующем виде:

```

/home/httpd/cgi-bin:
/home/httpd/html:
    /home/httpd/html/index.html
/home/httpd/html/manual:
    /home/httpd/html/manual/LICENSE
    /home/httpd/html/manual/bind.html
    /home/httpd/html/manual/cgi_path.html
. . .

```

Вопросы для самоконтроля

1. Какие формы обращения к подпрограмме вы знаете?
2. Что такое область видимости переменной?
3. Как ограничить область видимости переменной?
4. Чем отличаются переменные, объявленные при помощи функции `my()` от переменных, объявленных при помощи функции `local()`?
5. Каким образом данные передаются в подпрограмму и из подпрограммы?
6. Что такое передача параметров по ссылке и по значению?
7. Какой тип данных называется `typeglob`?
8. Как осуществить передачу по ссылке параметра-массива?
9. В каких случаях функция `local()` не может быть заменена функцией `my()`?
10. Что такое прототип?
11. Какие значения будут иметь переменные `$x`, `@list1`, `@list2` после выполнения программы

```

#!/usr/bin/perl
$x = 0;
@list1 = (1, 2, 3);

```

```
@list2 = func();  
sub func {  
    local ($x);  
    $x = 1;  
    @list1 = (4, 5, 6);  
}
```

Упражнения

1. Напишите подпрограмму, которая выводит пронумерованный список своих аргументов.
2. Напишите подпрограмму, которая выводит пронумерованный список своих аргументов в обратном порядке.
3. Напишите подпрограмму, которая подсчитывает число символов из стандартного ввода и выводит результат.
4. Напишите подпрограмму, которая выводит свои параметры-массивы в обратном порядке по элементам.
5. Напишите подпрограмму, которая для двух своих параметров-массивов осуществляет взаимный обмен элементов с одинаковыми индексами.
6. Одной из известных задач, для решения которых применяется рекурсия, является задача о Ханойских башнях.
7. На плоскости установлены три стержня: *a*, *b*, *c* (рис. 11.1).

На стержень **a** нанизаны **n** дисков, расположенных по возрастанию диаметра. Необходимо переместить диски со стержня **a** на стержень **c**, используя стержень **b** и соблюдая следующие ограничения: можно перемещать только один диск одновременно, диск большего диаметра никогда не может находиться на диске меньшего диаметра.

Напишите подпрограмму, которая описывает последовательность переноса дисков в ходе решения задачи, выводя сообщения вида:

Перенос диска со стержня *a* на стержень *c*.

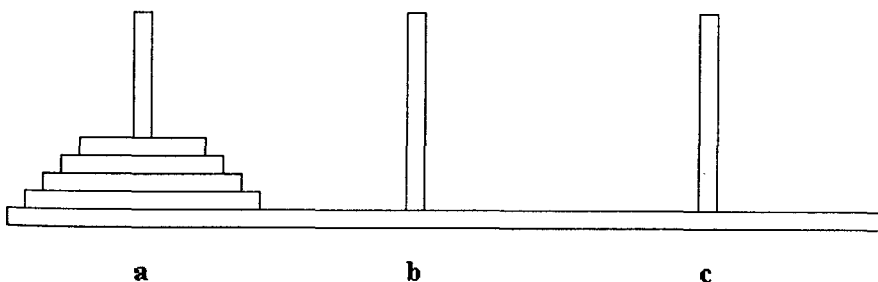


Рис. 11.1. Задача о Ханойских башнях

Пакеты, библиотеки, модули



12.1. Пакеты

В главе 11 мы упомянули о том, что область действия переменных ограничена пакетом. Рассмотрим этот вопрос более подробно.

Итак, *пакет* — это способ создания собственного изолированного пространства имен для отдельного отрезка программы. Каждый фрагмент кода Perl-программы относится к некоторому пакету. Объявление

```
package NAMESPACE;
```

определяет пакет `NAMESPACE`. Ключевое слово `package` является именем встроенной функции, в результате обращения к которой компилятору предписывается использовать новое пространство имен. Область действия объявления пакета определяется аналогично области видимости локальных переменных, объявленных при помощи функций `my()` или `local()`. Она распространяется либо до следующего объявления пакета, либо до конца одной из таких единиц программы:

- ☐ подпрограммы;
- ☐ блока операторов, заключенного в фигурные скобки;
- ☐ строки, переданной на выполнение функции `eval()`;
- ☐ файла.

Область действия зависит от того, в каком месте вызвана для объявления пакета функция `package()`. Все идентификаторы, встретившиеся внутри этой области, принадлежат к пространству имен текущего пакета. Исключение составляют идентификаторы лексических переменных, созданных при помощи функции `my()`.

Объявление пакета может быть сделано несколько раз в разных точках программы. Каждое объявление означает переключение на новое пространство имен. По умолчанию предполагается, что основная программа всегда начинается с объявления пакета

```
package main;
```

Таким образом, те переменные, которые мы называем в языке Perl глобальными, в действительности представляют собой переменные, чьи идентификаторы по умолчанию принадлежат пакету `main`.

К переменной из другого пакета можно обратиться, указав перед ее именем префикс, состоящий из имени этого пакета, за которым следуют два двоето-

чия: `$PackageName::name`. Такие имена условимся называть *квалифицированными* именами. Если имя пакета отсутствует, предполагается имя `main`, т. е. записи `$::var` и `$main::var` обозначают одну и ту же переменную.

Специальная лексема `__PACKAGE__` служит для обозначения имени текущего пакета.

Пример 12.1. Объявление пакета и переключение пространства имен

```
#!/usr/bin/perl
$x=__PACKAGE__;
print "package $x:\n";
print "\$x= $x\n";
print "\$two::x= $two::x\n";
print "\$three::x= $three::x\n";

eval 'package two;
    $x=__PACKAGE__;
    print "package $x:\n";
    print "\$x= $x\n";
    print "\$main::x= $main::x\n";
    print "\$three::x= $three::x\n";';

print "package $x:\n";
print "\$x= $x\n";

package three;
$x=__PACKAGE__;
print "package $x:\n";
print "\$x= $x\n";
print "\$main::x= $main::x\n";
print "\$two::x= $two::x\n";

package main;
print "package $x:\n";
print "\$x= $x\n";
print "\$two::x= $two::x\n";
print "\$three::x= $three::x\n";
```

В результате выполнения будут выведены следующие значения:

```
package main:
$x= main
```

```
$two::x=
$three::x=
    package two:
$х= two
$main::x= main
$three::x=
    package main:
$х= main
    package three:
$х= three
$main::x= main
$two::x= two
    package main:
$х= main
$two::x= two
$three::x= three
```

В данном примере используются три пакета, каждый со своим пространством имен: `main`, `two`, `three`. В каждом пакете определена переменная `$х`, значение которой совпадает с именем пакета. С пакетом `main` связаны следующие отрезки программы:

- ☐ от начала программы до вызова функции `eval()`;
- ☐ после вызова функции `eval()` до объявления пакета `package three`;
- ☐ после явного объявления пакета `package main` до конца файла, содержащего данную программу.

Для выражения, выполняемого функцией `eval()`, определено собственное пространство имен `two`. Оно действует только в пределах этого выражения. Если бы внутри функции `eval()` не был определен собственный пакет `two`, все действия внутри нее были связаны с пакетом, в котором функция `eval()` была скомпилирована, т. е. с пакетом `main`.

С пакетом `three` связана часть программы от объявления `package three` до объявления `package main`.

В каждом пакете происходит обращение к переменным из двух других пакетов при помощи указания соответствующего префикса имени переменной.

Компилятор создает для каждого пакета отдельное пространство имен. Переменным `$х` из разных пакетов присваиваются их значения по мере выполнения соответствующего кода программы. Вот почему при первом обращении из пакета `main` к переменным `two::$х` и `$three::x` их значения еще не определены.

12.1.1. Таблицы символов

С каждым пакетом связана *таблица символов*. Она представляет собой хеш-массив, имя которого образовано из имени пакета, за которым следуют два двоеточия. Например, таблица символов пакета `main` хранится в хеш-массиве `%main::`. Ключами этого хеш-массива являются идентификаторы переменных, определенных в пакете, значениями — значения типа `typeglob`, указывающие на гнездо, состоящее из одноименных переменных разных типов: скаляр, массив, хеш-массив, функция, дескриптор файла или каталога.

Тип `typeglob`, с которым мы уже сталкивались в главе 11 — это внутренний тип данных языка Perl, который используется для того, чтобы при помощи одной переменной типа `typeglob` сослаться на все одноименные переменные разных типов. Признаком типа `typeglob` является символ `"*"`. Если переменной типа `typeglob` присвоить значение другой переменной типа `typeglob`:

```
*y = *x;
```

то для всех переменных с именем `x`: `$x`, `@x`, `%x`, `&x`, будут созданы псевдонимы `$y`, `@y`, `%y`, `&y` соответственно. Можно создать псевдоним только для переменной определенного типа, например, для скалярной:

```
*y = \ $x;
```

Ключами в хеш-массиве таблицы символов являются все идентификаторы, определенные в пакете. Поэтому можно получить данные о переменных всех типов, определенных в пакете, проверяя значения элементов этого хеш-массива. Например, чтобы вывести имена всех переменных, определенных в пакете `main`, можно использовать следующий код.

Пример 12.2. Вывод таблицы символов пакета `main`

```
# !/usr/bin/perl
my ($key, $item);
print "Таблица символов пакета main:\n";
for $key (sort keys %main::) {
    local *myglob = %main::{$key};
    print "определен скаляр \ $key    = $myglob\n" if defined $myglob;
    if (defined @myglob) {
        print "определен массив \@ $key    : \n";
        for $item (0..$#myglob) {
            print " \ $key [$item]    = $myglob[$item]\n";
        }
    }
}
```

```

if (defined %myglob) {
    print "определен хеш-массив \%$key : \n";
    for $item (sort keys %myglob) {
        print "\$$key {$item} = %myglob{$item} \n";
    }
}

print "определена функция $key() \n" if defined &myglob;
}

```

При помощи типа `typeglob` можно создавать скалярные псевдоконстанты. Например, после присваивания

```
*PI = \3.14159265358979;
```

выражение `$PI` обозначает операцию разыменования ссылки на константу. Его значением является значение самой константы 3.14159265358979. Значение `$PI` нельзя изменить, так как это означало бы попытку изменить константу.

В таблицу символов пакета, отличного от `main`, входят только идентификаторы, начинающиеся с буквы или символа подчеркивания. Все остальные идентификаторы относятся к пакету `main`. Кроме того, к нему относятся следующие начинающиеся с буквы идентификаторы: `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INC`, `SIG`. Например, при обращении внутри некоторого пакета `pack` к хеш-массиву `%ENV` подразумевается специальный хеш-массив `%ENV` основного пакета `main`, даже если имя `main` не используется в качестве префикса для обозначения принадлежности идентификатора `ENV`.

12.1.2. Конструктор и деструктор пакета *BEGIN* и *END*

Конструктором в объектно-ориентированном программировании называется специальная подпрограмма, предназначенная для создания объекта. *Деструктором* называется подпрограмма, вызываемая для выполнения завершающих действий, связанных с ликвидацией объекта: закрытие файлов, вывод сообщений и т. д.

Для создания пакета, как мы знаем, требуется только его объявление (в том числе, предполагаемое по умолчанию объявление `package main`). Вместе с тем, существуют специальные подпрограммы, выполняющие функции инициализации и завершения пакета. По аналогии их можно назвать конструкторами и деструкторами пакета, хотя никаких пакетов они не создают и не удаляют. Это подпрограммы `BEGIN` и `END`. При описании этих подпрограмм ключевое слово `sub`, необходимое при объявлении обычной подпрограмм-

мы, можно опустить. Таким образом, синтаксис подпрограмм BEGIN, END имеет вид:

```
BEGIN {block}  
END {block}
```

Подпрограмма BEGIN выполняется сразу после своего определения до завершения компиляции оставшейся части программы. Попробуйте запустить интерпретатор perl в интерактивном режиме. Если ему передать строку

```
print "Привет!";
```

то он напечатает ее только после того, как обнаружит во входном потоке признак конца файла (например, комбинацию <Ctrl>+<D>). Если же в интерактивном режиме определить конструктор пакета

```
BEGIN {print "Привет!"};
```

то вывод строки "Привет!" будет осуществлен немедленно. Это свойство конструктора можно использовать, чтобы в начале пакета определять или импортировать имена из других пакетов (см. раздел 12.3). Затем эти имена будут влиять на процесс компиляции оставшейся части пакета.

Можно определить несколько блоков BEGIN внутри файла, они будут выполняться один за другим в порядке определения.

Подпрограмма END выполняется настолько поздно, насколько это возможно, т. е. при завершении работы интерпретатора. Можно указать несколько блоков END, при этом они будут выполняться в порядке, обратном определению.

Пример 12.3. Конструктор и деструктор пакета BEGIN и END

```
END {  
    print "Завершаем работу, до свидания\n";  
}  
BEGIN {  
    print "Привет, начинаем работу\n";  
}  
print "Это тело программы\n";  
BEGIN {  
    print "Еще один блок BEGIN после блока END\n";  
}
```

Здесь сознательно выбран не совсем естественный порядок следования конструкторов и деструкторов BEGIN и END в тексте программы, чтобы подчеркнуть, в каком порядке они будут вызываться. Вывод выглядит так:

Привет, начинаем работу
Еще один блок BEGIN после блока END
Это тело программы
Завершаем работу, до свидания

12.1.3. Автозагрузка

При попытке обратиться к функции из некоторого пакета, которая в нем не определена, интерпретатор завершает работу с выдачей сообщения об ошибке. Если же в этом пакете определить функцию с именем `AUTOLOAD`, то при вызове из пакета несуществующей функции вместо нее будет вызвана функция `AUTOLOAD` с параметрами, переданными при вызове несуществующей функции. При этом интерпретатор `perl` продолжает выполнение программы. Полное имя несуществующей функции с указанием имени пакета сохраняется в переменной `$AUTOLOAD` из того же пакета, что и функция `AUTOLOAD`. Например, для основного пакета `main` можно определить функцию `AUTOLOAD`, как в следующем примере.

Пример 12.4. Функция `AUTOLOAD`

```
#!/usr/bin/perl
sub AUTOLOAD {
    print "Функция $AUTOLOAD не определена\n";
}
print "Начало работы\n";
f();
print "Конец работы\n";
```

Функция `f()`, в отличие от функции `AUTOLOAD`, не определена в пакете `main`, поэтому в результате выполнения данной программы будут выведены сообщения:

```
Начало работы
Функция main::f не определена
Конец работы
```

Этот пример достаточно тривиальный, но он дает представление об использовании функции `AUTOLOAD`. В состав дистрибутива Perl входят стандартные модули, многие из которых содержат собственные, достаточно сложные определения функции `AUTOLOAD`, которые можно рассмотреть в качестве более сложного примера.

12.2. Библиотеки

Подпрограммы, как мы отметили в начале главы, служат для объединения группы операторов с целью их повторного использования. Следующим шагом является объединение группы подпрограмм и их сохранение в отдельном файле для последующего использования другими программами. Для реализации этой задачи в языке Perl имеются два механизма: библиотеки и модули.

Исторически первым средством являются библиотеки, появившиеся в версии Perl 4. Библиотека представляет собой пакет, областью действия которого является отдельный файл. Иными словами, библиотека — это файл, содержащий в качестве первой строки объявление пакета.

```
package package_name;
```

Имя файла библиотеки обычно имеет расширение `pl`.

Замечание

После появления модулей (см. раздел 12.3) термин "библиотека" часто используют в широком смысле для обозначения всего множества модулей в составе Perl, содержащего разнообразные коллекции подпрограмм. Если не оговорено противное, мы будем использовать слово "библиотека" для обозначения файла библиотеки в соответствии с версией Perl 4.

Для использования библиотеки в основной программе, ее следует подключить к последней при помощи директивы компилятора `require`.

Ключевое слово `require` служит для обозначения встроенной функции Perl. Фактически обращение к функции `require()` используется в качестве директивы компилятора. Поэтому дальше мы будем использовать применительно к ключевому слову `require` оба термина: "функция" и "директива компилятора". Выясним, какой смысл имеет эта директива.

12.2.1. Функция *require()*

```
require [EXPR]
```

загружает внешние функции из библиотеки Perl во время выполнения. Она используется для того, чтобы сделать библиотеку подпрограмм доступной для любой Perl-программы.

Если параметр `EXPR` отсутствует, вместо него используется специальная переменная `$_`.

Если параметр является числом, это соответствует требованию, что для выполнения данного сценария необходим интерпретатор `perl` с номером вер-

сии, не меньшим, чем значение параметра. Таким образом, сценарий, который требует Perl версии 5.005, может иметь в качестве первой строки:

```
require 5.005;
```

Более ранние версии Perl вызовут немедленное завершение интерпретатора с выдачей сообщения об ошибке.

Если параметр является строкой, функция `require` включает в основную программу библиотечный файл, задаваемый параметром `EXPR`. Логика работы функции `require` соответствует следующему коду:

```
sub require {
    my($filename) = @_ ;
    return 1 if $INC{$filename};
    my($realfilename,$result);
    ITER: {
        foreach $prefix (@INC) {
            $realfilename = "$prefix/$filename";
            if (-f $realfilename) {
                $result = do $realfilename;
                last ITER;
            }
        }
        die "Can't find $filename in \@INC";
    }
    die "$@" if $@;
    die "$filename did not return true value" unless $result;
    $INC{$filename} = $realfilename;
    return $result;
}
```

Замечание

Специальный встроенный массив `@INC` содержит имена каталогов, в которых следует искать сценарии Perl, подлежащие выполнению в конструкциях `do filename`, `require` или `use`. Первоначально содержит:

- имена каталогов, переданные при запуске интерпретатору perl в качестве параметра ключа `-I`;
- имена библиотечных каталогов по умолчанию (зависят от операционной системы);
- символическое обозначение текущего каталога `"."`.

Специальный встроенный хеш-массив `%INC` содержит по одному элементу для каждого файла, включенного при помощи `do` или `require`. Ключом является

имя файла в том виде, как оно указано в качестве аргумента функций `do` или `require`, а значением — его полное маршрутное имя.

Встретив директиву `require myfile`, интерпретатор `perl` просматривает специальный хеш-массив `%INC`, проверяя, не был ли файл `myfile` уже включен ранее при помощи функций `require` или `do`. Если да, то выполнение функции `require` завершается. Таким образом файл под одним именем может быть включен только один раз. В противном случае интерпретатор просматривает каталоги, имена которых содержатся в специальном массиве `@INC`, в поисках файла `myfile`. Если файл найден, он выполняется, иначе директива `require` завершает выполнение с выдачей сообщения об ошибке:

```
Can't find myfile in @INC
```

Замечание об использовании имени файла в директиве `require EXPR`

Обычно имена библиотечных файлов имеют суффикс `".pl"`, например, `myfile.pl`. Интерпретатор `perl` воспринимает точку `"."` в качестве знака операции конкатенации двух строк `"myfile"` и `".pl"` и пытается найти файл `myfilepl`. Во избежание подобных ошибок имя файла в директиве `require` следует заключать в кавычки:

```
require "myfile.pl";
```

Если аргумент `EXPR` является словом без суффиксов, не заключенным в кавычки, то директива `require` предполагает, что оно имеет суффикс `".pm"`, чтобы облегчить загрузку стандартных модулей, имеющих расширение `".pm"`.

12.2.2. Создание и подключение библиотечного файла

Для создания собственной библиотеки следует выполнить следующие шаги.

- Создать каталог для хранения библиотечных файлов.
- Сохранить наборы подпрограмм в отдельных файлах-библиотеках. Переместить библиотечные файлы в специально созданный для них каталог.
- В конец каждого библиотечного файла поместить строку `"1;".` Смысл этого действия заключается в следующем. Как видно из приведенного текста, включение библиотечного файла в основную программу осуществляется через его выполнение функцией `do`:

```
$result = do $realfilename;
```

Значение `$result`, возвращаемое функцией `require`, должно быть ненулевым, что является признаком успешного выполнения кода инициализации. Простейший способ получить ненулевое значение — добавление в конец каждого библиотечного файла строки `"1;".`

- ❑ В основной программе использовать директиву `require`, указав в качестве ее аргументов имена требуемых библиотечных файлов.
- ❑ Добавить в массив `@INC` имя каталога, содержащего библиотечные файлы, либо при запуске основной программы передать это имя интерпретатору `perl` при помощи ключа `-I`.

Пример 12.5. Создание и подключение библиотечного файла

Создадим библиотечный файл `mylib.pl` и поместим его в каталог `/usr/temp/perl/lib`. Файл `mylib.pl` содержит единственную подпрограмму `NumOfArgs()`, которая выводит число аргументов, переданных ей при вызове.

библиотечный файл `/usr/temp/perl/lib/mylib.pl`

```
sub NumOfArgs {
    return $#_ + 1;
}
1;
```

Создадим файл основной программы `mytmain.pl`:

```
#!/usr/bin/perl
unshift (@INC, "/usr/temp/perl/lib");
require "mylib.pl";
print "Число аргументов=", NumOfArgs(1,2,3,4), "\n";
```

В результате выполнения файла `mytmain.pl` будет выведена строка

Число аргументов=4

Обратите внимание на выполнение всех шагов, необходимых для создания и подключения библиотеки.

12.3. Модули

Дальнейшим развитием понятия библиотеки явилось понятие модуля, возникшее в версии Perl 5. Модуль представляет собой библиотеку подпрограмм, обладающую дополнительными свойствами по сравнению с библиотеками Perl 4. Он позволяет управлять экспортом своих имен в другие программы, объявляя, какие из них экспортируются по умолчанию, а какие должны быть явно указаны в соответствующем операторе вызывающей программы.

Под *экспортом* мы здесь понимаем предоставление возможности другим модулям импортировать символы из пространства имен данного модуля. Соответственно под *импортом* мы понимаем включение в собственное пространство имен символов, экспортируемых другим модулем.

Для целей управления экспортом каждый модуль должен располагать методом `import()` и определить специальные массивы `@EXPORT` и `@EXPORT_OK`.

(Понятие "метод" используется в объектно-ориентированном программировании, которое обсуждается в главе 13.)

Вызывающая программа обращается для импорта символов к методу `import()` экспортирующего модуля.

Специальный массив `@EXPORT` содержит идентификаторы, экспортируемые по умолчанию.

Специальный массив `@EXPORT_OK` содержит идентификаторы, которые будут экспортироваться только в том случае, если они явно указаны в списке импорта вызывающей программы.

С появлением модулей появилась новая директива для их подключения к основной программе. Эта директива реализуется функцией `use()`.

12.3.1. Функция *use()*

Относительно ключевого слова `use` можно сказать то же самое, что и относительно ключевого слова `require`. Оно служит для обозначения встроенной функции Perl. Фактически же обращение к функции `use()` используется в качестве директивы компилятора, поэтому мы также будем использовать применительно к ключевому слову `use` оба термина: "функция" и "директива компилятора".

Функция `use()`

```
use Module [LIST]
```

```
use VERSION
```

служит для загрузки модуля во время компиляции.

Директива `use` автоматически экспортирует имена функций и переменных в основное пространство имен текущего пакета. Для этого она вызывает метод `import()` импортируемого модуля. Механизм экспорта имен устроен таким образом, что каждый экспортирующий модуль должен иметь свой метод `import()`, который используется программой, импортирующей имена. Метод `import()` должен быть определен в самом экспортирующем модуле или наследован у модуля `Exporter`. Большинство модулей не имеют своих собственных методов `import()`, вместо этого они экспортируют его из модуля `Exporter`.

Логику работы директивы `use` можно описать одной строкой:

```
BEGIN { require Module; import Module LIST; }
```

Здесь значением параметра `Module` должно быть слово без суффиксов, не заключенное в кавычки.

Если первый аргумент директивы `use` является числом, он обозначает номер версии интерпретатора `perl`. Если номер версии текущего интерпретатора `perl` меньше, чем значение `VERSION`, интерпретатор выводит сообщение об ошибке и завершает работу.

Конструктор пакета `BEGIN` вызывает немедленное выполнение подпрограммы `require()` и метода `import()` до завершения компиляции оставшейся части файла.

Выше мы рассмотрели логику работы функции `require()`. Она загружает в память файл `Module.pm`, выполняя его при помощи функции `do()`. Затем метод `import()` модуля `Module.pm` импортирует в вызывающую программу имена, определенные в `Module.pm`, в соответствии со списком `LIST`.

Если список импорта `LIST` отсутствует, из `Module` будут импортированы те имена, которые перечислены в специальном массиве `@EXPORT`, определенном в самом `Module`.

Если список импорта задан, то в вызывающую программу из модуля `Module` будут импортированы только имена, содержащиеся в списке `LIST`.

12.3.2. Создание и подключение модуля

Для создания модуля `MyModule` следует создать пакет и сохранить его в файле `MyModule.pm`. Расширение `.pm` является признаком того, что данный файл является модулем `Perl`.

В следующем примере мы создадим собственный модуль `MyModule`, содержащий одну функцию `MyArgs()`, одну скалярную переменную `$MyArgs`, один массив `@MyArgs` и один хеш-массив `%MyArgs`. Затем создадим файл основной программы `MyMain.pl`, экспортирующий модуль `MyModule`, используя директиву `use`.

Пример 12.6. Создание и подключение модуля

Файл модуля `MyModule.pm`:

```
package MyModule;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(MyArgs);
@EXPORT_OK = qw($MyArgs @MyArgs %MyArgs);
sub MyArgs {
    my ($x, $i);
    @MyArgs = @_;
    $MyArgs = $#MyArgs + 1;
```

```
foreach $x (@MyArgs) {  
    $MyArgs{$x}=++$i;  
}  
}
```

Файл основной вызывающей программы `MyMain.pl`:

```
#!/usr/bin/perl  
  
use MyModule qw(:DEFAULT $MyArgs @MyArgs %MyArgs);  
MyArgs one, two, three, four;  
print "число аргументов=$MyArgs\n";  
print "массив аргументов: @MyArgs\n";  
print "хеш-массив аргументов:\n";  
foreach $k (keys %MyArgs) {  
    print "\$MyArgs{$k}=$MyArgs{$k} ";  
}
```

Первые пять строк файла `MyModule.pm` являются стандартными для определения модуля Perl. Их можно использовать в качестве образца при создании собственных модулей.

Первая строка служит определением пакета.

Вторая строка осуществляет включение встроенного модуля `Exporter`. Так предоставляется возможность наследовать метод `import`, реализованный в этом модуле, и использовать стандартные соглашения для задания списка импорта в вызывающей программе.

Третья строка определяет массив `@ISA`, состоящий из одного элемента, содержащего название пакета `Exporter`. С каждым пакетом ассоциируется свой массив `@ISA`, включающий имена других пакетов, представляющих классы. Иногда интерпретатор встречает обращение к методу, не определенному в текущем пакете. Он ищет этот метод, просматривая пакеты, определенные в массиве `@ISA` текущего пакета. Таким образом в языке Perl реализован механизм наследования.

В четвертой и пятой строках определяются имена, которые будут экспортироваться за пределы модуля. Специальный массив `@EXPORT` содержит имена, экспортируемые по умолчанию. В четвертой строке указывается, что из данного модуля по умолчанию будет экспортировано имя функции `MyArgs`.

В пятой строке специальный массив `@EXPORT_OK` содержит имена, которые будут экспортироваться только в том случае, если они явно указаны в списке импорта вызывающей программы. В примере это имена переменной `$MyArgs`, массива `@MyArgs` и ассоциативного массива `%MyArgs`.

Функция `MyArgs` подсчитывает число своих аргументов и запоминает его в переменной `$MyArgs`. Затем она помещает аргументы в массив `@MyArgs` и

формирует ассоциативный массив `%MyArgs`, в котором ключами являются имена аргументов функции `MyArgs`, а значениями — их порядковые номера.

К основной программе `MyMain.pl` модуль `MyModule` подключается при помощи директивы `use`. Директива `use` содержит список импорта

```
qw(:DEFAULT $MyArgs @MyArgs %MyArgs)
```

Обычно список импорта включает в себя имена переменных и функций. Кроме того, он может содержать некоторые управляющие им спецификации. Спецификация `:DEFAULT` означает включение в список импорта *всех* элементов специального массива `@EXPORT`. В нашем случае это значит, что в список импорта будет добавлено имя функции `MyArgs`, содержащееся в списке `@EXPORT`. Кроме того, список импорта явно содержит имена `$MyArgs`, `@MyArgs` и `%MyArgs`. Экспорт этих имен по явному запросу вызывающей программы разрешен модулем `MyModule` путем их включения в список `@EXPORT_OK`.

В результате выполнения основной программы `MyMain.pl` будет получен вывод:

```
число аргументов=4
```

```
массив аргументов: one two three four
```

```
хеш-массив аргументов:
```

```
$MyArgs{one}=1 $MyArgs{three}=3 $MyArgs{two}=2 $MyArgs{four}=4
```

12.3.3. Функция `no()`

Существует функция `no()`, противоположная по своему назначению функции `use()` и также выполняющая роль директивы компилятора

```
no Module LIST
```

Директива `no` отменяет действия, связанные с импортом, осуществленные ранее директивой `use`, вызывая метод `unimport()` `Module LIST`.

12.3.4. Стандартные модули Perl

В данном разделе мы рассмотрели вопросы, связанные с созданием модулей. В состав дистрибутива Perl входит большой набор стандартных модулей, предназначенных для выполнения определенных функций. Помимо них, существует огромная коллекция модулей, разработанных программистами всего мира, известная как коллекция модулей CPAN (Comprehensive Perl Archive Network). Ее копия поддерживается в сети Internet на многих анонимных ftp-серверах. В качестве отправной точки можно обратиться по адресу <http://www.perl.com/CPAN/modules/>. Список стандартных модулей и категорий модулей CPAN приведен в приложении 2. Здесь же мы в заклю-

чение рассмотрим специальный вид стандартных модулей — прагма-библиотеки.

12.3.5. Прагма-библиотеки

Многие языки программирования позволяют управлять процессом компиляции посредством директив компилятора. В языке Perl эта возможность реализована при помощи так называемых прагма-библиотек. В современной терминологии, связанной с программированием, слово "pragma" используется для обозначения понятия, смысл которого в русском языке выражается сочетанием "директива компилятора". В языке Perl термин "pragma" обозначает модуль, содержащий коллекцию подпрограмм, используемых на этапе компиляции. Его назначение — передать компилятору информацию о том, как модифицировать процесс компиляции. Поскольку сочетание "библиотека директив компилятора" звучит несколько тяжеловато, мы используем для обозначения таких модулей название "прагма-библиотека".

Как и остальные модули, прагма-библиотека подключается к основной программе при помощи директивы `use` и выполняет функцию директивы компилятора. Область действия большинства таких директив ограничена, как правило, блоком, в котором они встречаются. Для отмены соответствующей директивы используется функция `no`.

Например, для ускорения выполнения некоторых отрезков программы можно заставить компилятор использовать целочисленную арифметику вместо принятой по умолчанию арифметики с плавающей точкой, а затем снова вернуться к последней.

Пример 12.7. Использование прагма-библиотеки в качестве директивы компилятора

```
#!/usr/bin/perl
print "Арифметика с плавающей точкой: 2/3= ", 2/3, "\n";
use integer;
print "Целочисленная арифметика: 2/3= ", 2/3, "\n";
no integer;
print "Возврат к арифметике с плавающей точкой: 2/3= ", 2/3, "\n";
```

В результате выполнения данного примера будет получен вывод

Арифметика с плавающей точкой: 2/3= 0.6666666666666667

Целочисленная арифметика: 2/3= 0

Возврат к арифметике с плавающей точкой: 2/3= 0.6666666666666667

В дистрибутивный комплект Perl входит стандартный набор прагма-библиотек. Некоторые из них представлены в табл. 12.1.

Таблица 12.1. Некоторые прагма-библиотеки

Прагма-библиотека	Назначение
diagnostics	Включает режим диагностики с выдачей подробных сообщений
integer	Применение целочисленной арифметики вместо арифметики с плавающей точкой
lib	Позволяет добавлять элементы в специальный массив @INC во время компиляции
overload	<p>Режим переопределения операций Perl, например, директива</p> <pre>package Number; use overload "+" => \&add;</pre> <p>определяет функцию Number::add() в качестве операции сложения</p>
sigtrap	Директива, позволяющая управлять обработкой сигналов в UNIX
strict	<p>Режим ограниченного использования "опасных" конструкций Perl</p> <pre>use strict "refs";</pre> <p>генерирует ошибку выполнения при использовании символических ссылок</p> <pre>use strict "vars";</pre> <p>генерирует ошибку компиляции при попытке обращения к переменной, которая не была объявлена при помощи директивы use vars, локализована при помощи функции my() или не является квалифицированным именем</p> <pre>use strict "subs";</pre> <p>генерирует ошибку компиляции при попытке использовать идентификатор, который не заключен в кавычки, не имеет префикса типа и не является именем подпрограммы, за исключением тех случаев, когда он заключен в фигурные скобки, или стоит слева от символа ==></p> <pre>use strict;</pre> <p>эквивалентно заданию всех трех рассмотренных выше ограничений</p>

Таблица 12.1 (окончание)

Прагма-библиотека	Назначение
subs	Служит для предварительного объявления подпрограмм, указанных в списке: use subs qw(sub1 sub2 sub3);
vars	Служит для предварительного объявления переменных, указанных в списке use vars qw(\$scal @list %hash); после чего их можно использовать при включенной директиве use strict, не опасаясь возникновения ошибки компиляции

Вопросы для самоконтроля

1. Что такое пакет?
2. Верно ли, что пакет должен всегда занимать отдельный файл?
3. Что такое таблица символов?
4. Сколько таблиц символов могут быть связаны с одним файлом, функцией, блоком операторов, заключенным в фигурные скобки?
5. Какие функции выполняют конструктор и деструктор пакета BEGIN и END?
6. Как определить имя текущего пакета?
7. Для чего нужна функция AUTOLOAD?
8. Что такое библиотека?
9. Назовите действия, необходимые для создания библиотечного файла.
10. Что такое модуль? В чем разница между модулем и библиотекой?
11. Объясните назначение массивов @EXPORT и @EXPORT_OK.
12. Чем похожи и чем отличаются функции use() и require()?
13. Объясните, как создать модуль и подключить его к вызывающей программе.
14. Объясните назначение функции no().
15. Что такое прагма-библиотека?

Упражнения

1. Напишите программу, которая выводит таблицу символов для пакета, заданного ее аргументом.
2. Создайте два модуля. Модуль `mod1` должен содержать подпрограмму `reverselist()`, которая переставляет в обратном порядке элементы в массиве, переданном ей в качестве параметра. Модуль `mod2` должен содержать массив `@list`. Используйте оба модуля в основной программе, которая при помощи подпрограммы `reverselist()` переставляет элементы в массиве `@list`.



Объектно-ориентированное программирование в языке Perl

Эта глава не предназначена для того, чтобы изучать по ней основы объектно-ориентированного программирования (ООП). Мы лишь хотим дать представление о том, как основные идеи ООП реализованы в языке Perl. Начнем с краткого обзора этих идей. В основе ООП лежат понятия *класса* и *объекта*. Эти понятия тесно связаны друг с другом.

Класс представляет собой сочетание структуры данных и тех действий, которые можно выполнить над этими данными. Данные называют *свойствами*, а действия — *методами*. Совмещение в классе структуры данных и действий над ними называют *инкапсуляцией*.

Объект является экземпляром класса. Свойства объекта обусловлены его принадлежностью к определенному классу. Понятия "объект" и "класс" отражают два различных вида иерархий, которые можно обнаружить в любой достаточно сложной системе. Рассмотрим пример.

Персональный компьютер является сложной системой. Жесткий диск — составная часть этой системы. Другими ее частями являются центральный процессор, память и т. д. Можно сказать, что жесткий диск — это часть *структурной* иерархии под названием "персональный компьютер". С другой стороны, жесткий диск является абстракцией, обобщением свойств, присущих всем жестким дискам, и этим он отличается, например, от *гибкого* диска, рассматриваемого как абстракция, обобщающая свойства всех гибких дисков. В *типовой* иерархии жесткие диски Quantum — это особый тип жестких дисков, жесткие диски Quantum Fireball EL — особый тип жестких дисков Quantum, жесткие диски Quantum Fireball EL объемом 5,1 Гбайт — особый тип жестких дисков Quantum Fireball EL и т. д.

В данной аналогии абстрактный жесткий диск является *базовым* классом, жесткие диски фирмы Quantum образуют *подкласс* класса жестких дисков, жесткие диски Quantum Fireball EL — подкласс класса жестких дисков фирмы Quantum и т. д. Подкласс называют также *производным* классом или классом-потомком. Для него класс, расположенный выше в иерархии, является базовым, *надклассом* или родительским классом. В нашем примере *объект* (конкретный жесткий диск) является составной частью *структуры* под названием "персональный компьютер", и обладает своими свойствами в силу принадлежности к определенному *типу* дисков Quantum Fireball EL.

5,1 Гбайт, отличающемуся по своим характеристикам от других типов жестких дисков.

Действие, которое можно выполнить над данными, определяет *метод*. Он представляет собой подпрограмму. Различают *методы класса* и *методы объекта*. Последние могут применяться к любому объекту класса. Методы класса, называемые также *статическими* методами, не зависят от отдельного экземпляра, они применяются к целому классу как к отдельному объекту.

Для создания объектов применяются специальные статические методы, называемые *конструкторами*.

Для корректного удаления объектов используются специальные методы, называемые *деструкторами*.

Термин *наследование* в ООП обозначает способность классов наследовать свойства и методы у своих родительских классов.

Термин *полиморфизм* в ООП обозначает свойство класса-потомка переопределять методы родительского класса.

Инкапсуляция, наследование, полиморфизм — базовые понятия, лежащие в основе объектно-ориентированного программирования. Объектно-ориентированный подход возник в результате непрекращающихся попыток человека преодолеть сложность окружающего мира, проявляющуюся в любых областях деятельности. Он предлагает более естественный способ разделения сложной задачи на ряд более простых, используя понятие объекта, сочетающего данные и действия над ними. Он также позволяет экономить усилия, связанные с написанием программного кода, так как однажды созданные методы не нужно переписывать — они просто наследуются производными классами. Идея повторного использования кода в своем развитии проходит ряд этапов: подпрограмма, библиотека, модуль, класс.

В этой главе мы рассмотрим, каким образом основные концепции объектно-ориентированного программирования реализованы в языке Perl.

13.1. Классы и объекты

В языке Perl нет специального синтаксиса для описания классов, объектов или методов. Для их реализации используются уже знакомые нам синтаксические конструкции. Класс в Perl представляет собой пакет, объект является ссылкой, а метод — обычной подпрограммой.

В качестве пакета каждый класс имеет собственное изолированное пространство имен и таблицу символов, реализованную в виде хеш-массива. К переменным класса можно обращаться, используя их квалифицированные имена, содержащие в качестве префикса имя класса, за которым следуют два двоеточия, например, `$CLASSNAME::var`. Для того чтобы пакет стал клас-

сом, в нем нужно определить специальную подпрограмму — конструктор, которая используется для создания отдельных экземпляров класса — объектов (см. раздел 13.2.1).

Объект в Perl представляет собой просто ссылку, но не любую, а связанную с определенным классом. Тип ссылки можно определить при помощи функции `ref EXPR`, которая, рассматривая свой аргумент как ссылку, возвращает символическое обозначение ее типа. Для встроенных типов Perl используются следующие символические обозначения:

- ☐ `REF` — ссылка на ссылку;
- ☐ `SCALAR` — ссылка на скаляр;
- ☐ `ARRAY` — ссылка на массив;
- ☐ `HASH` — ссылка на ассоциативный массив;
- ☐ `CODE` — ссылка на подпрограмму;
- ☐ `GLOB` — ссылка на переменную типа `typeglob`.

Для ссылки-объекта функция `ref()` возвращает имя класса, к которому этот объект принадлежит. Обычная ссылка становится объектом после ее "посвящения" в члены класса при помощи функции

```
bless REF [, CLASSNAME]
```

Слово "bless" в английском языке имеет значение "освящать, благословлять". В данном контексте его можно перевести как "посвящать" или "санкционировать". Функция `bless REF` санкционирует принадлежность субъекта ссылки `REF` к классу `CLASSNAME`. Она возвращает ссылку на этот субъект, но уже другого типа — `CLASSNAME` (напомним, что в главе 9 субъектом ссылки мы условились называть то, на что она указывает, т. е. собственно структуру данных некоторого типа). Она связывает обычную ссылку с именем класса. Если имя класса не задано, то используется имя текущего класса. После выполнения функции `bless()` к созданному ей объекту можно обращаться, используя его квалифицированное имя `$CLASSNAME::REF`. Сказанное иллюстрируется следующим примером.

Пример 13.1. Функция `bless()` изменяет тип своего аргумента

```
$h = { };  
print("тип переменной \$h      - ". ref($h), "\n");  
bless($h, "MyClass");  
print("тип переменной \$h      - ". ref($h), "\n");
```

В результате будет выведен тип переменной `$ref` до и после вызова функции `bless()`:

тип переменной \$h	— HASH
тип переменной \$h	— MyClass

Наследование в Perl отличается от наследования в других объектно-ориентированных языках программирования тем, что наследуются только методы. Наследование данных реализуется программистом самостоятельно.

Наследование методов реализовано следующим образом. С каждым пакетом ассоциирован свой специальный массив @ISA, в котором хранится список базовых классов данного пакета. Таким образом, подкласс располагает информацией о своих базовых классах. Если внутри текущего класса встречается обращение к методу, не определенному в самом классе, то интерпретатор в поисках отсутствующего метода просматривает классы в том порядке, в котором они встречаются в массиве @ISA. Затем просматривается предопределенный класс UNIVERSAL. В нем изначально нет никаких явных определений, но автоматически содержатся некоторые общие методы, которые неявно наследуются всеми классами. В этом смысле класс UNIVERSAL можно считать базовым классом всех остальных классов.

Если метод не найден ни в одном из просмотренных классов, они вновь просматриваются в том же порядке в поисках подпрограммы AUTOLOAD (см. раздел 12.1.3). Если таковая обнаружена, она выполняется вместо вызванного несуществующего метода с теми же параметрами. Квалифицированное имя несуществующего метода при этом доступно через переменную \$AUTOLOAD.

13.2. Методы

Методы в Perl являются обычными подпрограммами. Начнем их изучение с методов, которые обязательно должны быть определены в каждом классе. Такими методами являются конструкторы. Знакомство с ними позволит лучше понять, каким способом в языке Perl представляются объекты.

13.2.1. Конструкторы

Конструктор — это просто подпрограмма, возвращающая ссылку. Обычно (но не обязательно) конструктор имеет имя new. Выше мы сказали, что объект является ссылкой. Конструктор, создающий объект, то есть ссылку, тоже возвращает ссылку. О каких ссылках идет речь, на что они указывают? Рассмотрим простой пример.

```
package MyClass;

sub new {
    my $class = shift;
    my $self = {};
```

```
    bless $self, $class;
    return $self;
}
```

В операторе `my $self = {}` создается ссылка на анонимный хеш-массив, первоначально пустой, которая сохраняется в локальной переменной `$self`. Функция `bless()` "сообщает" субъекту ссылки `$self`, то есть анонимному хеш-массиву, что он отныне является объектом класса `MyClass`, и возвращает ссылку на этот объект. Затем ссылка на новый объект класса `MyClass` возвращается в качестве значения конструктора `new()`. Обратите внимание на следующие обстоятельства.

Во-первых, объект класса `MyClass` представлен анонимным хеш-массивом. Это обычный, хотя и не обязательный, способ представления объекта. Преимущество использования для этой цели хеш-массива заключается в том, что он может содержать произвольное число элементов, к которым можно обращаться по произвольно заданному ключу. Таким образом, все данные объекта сохраняются в указанном хеш-массиве. В некоторых случаях для представления объекта может использоваться другой тип данных, например массив, скаляр.

Во-вторых, обязательным для конструктора является обращение к функции `bless()`. Внутри подпрограммы-конструктора функцию `bless()` следует применять с двумя аргументами, явно указывая имя класса. В этом случае конструктор может быть наследован классом-потомком. Конструктор определяет, объект какого именно класса он создает, используя значение своего первого аргумента. Первым аргументом конструктора должно быть имя класса.

В-третьих, конструктору, создающему объект, в качестве первого аргумента передается имя класса этого объекта.

Пример 13.2. Конструктор класса

```
# Модуль класса Staff.pm:
package Staff;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(new);
sub new {
    my ($class,@items) = shift;
    my $self = {};
    bless $self, $class;
    return $self;
}
```

```
# Основная программа:
#!/usr/bin/perl
use Staff;
$someone=new(Staff);
${$someone}{"имя"}="Александр";
${$someone}{"фамилия"}="Александров";
${$someone}{"возраст"}="37";
for $i (sort keys %{$someone}) {
    print "$i=>${$someone}{$i}\n";
}
```

В данном примере класс `Staff` служит для представления анкетных данных. В качестве внутренней структуры для представления объекта наилучшим образом подходит хеш-массив, так как в него при необходимости можно добавлять новые элементы с произвольно заданными ключами, например, "имя", "фамилия", "образование" и т. д. Класс оформлен в виде отдельного модуля, способного управлять экспортом своих методов. Чтобы конструктор `new()` можно было вызвать в основной программе, он включен в файл экспорта `@EXPORT`. В результате вызова конструктора возвращается ссылка на объект класса. Значения элементов хеш-массива выводятся:

```
возраст => 37
имя     => Александр
фамилия => Александров
```

13.2.2. Методы класса и методы объекта

Различают *методы класса* и *методы объекта*, которые называют также *статическими* и *виртуальными*, соответственно. Первым аргументом метода должна быть ссылка, представляющая объект, или имя пакета. То, каким образом каждый метод обрабатывает свой первый аргумент, определяет, является ли он методом класса или методом объекта.

Статические методы применяются к целому классу, а не к отдельным его объектам. В качестве первого аргумента статическому методу передается имя класса. Типичным примером статических методов являются конструкторы. В Perl методы выполняются в пространстве имен того пакета, в котором они были определены, а не в пространстве имен пакета, в котором они вызваны. Поэтому статические методы часто свой первый аргумент игнорируют, так как и без него известно, к какому пакету метод принадлежит. Последнее не относится к конструкторам, передающим свой первый аргумент функции `bless()` в качестве имени класса.

Методы объекта применяются к отдельному объекту. Их первым аргументом должна быть ссылка, указывающая на объект, к которому применяется данный метод. Методы объекта могут выполнять разные действия, но наиболее типичными являются действия, связанные с отображением и изменением данных, инкапсулированных в объекте. В примере 13.2 мы присвоили значения, а затем их распечатали, обращаясь к данным объекта напрямую. Таким способом мы просто продемонстрировали, что конструктор действительно возвращает ссылку, представляющую объект. В действительности это плохой способ, и применять его не рекомендуется. В некоторых объектно-ориентированных языках программирования прямой доступ к данным объекта вообще невозможен. Объект следует рассматривать как "черный ящик", содержимое которого можно получить или изменить, только используя методы объекта. Такое ограничение помогает сохранить целостность и скрыть некоторые внутренние данные объекта. Отредактируем текст примера 13.2, дополнив его методами, которые используются для изменения и просмотра данных.

Пример 13.3. Методы объекта

```
# Модуль класса Staff.pm;

package Staff;

require Exporter;

@ISA = qw(Exporter);

@EXPORT = qw(new showdata setdata);

sub new {
    my ($class, $data) = @_;
    my $self = $data;
    bless $self, $class;
    return $self;
}

sub showdata {
    my $self = shift;
    my @keys = @_ ? @_ : sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
    return $self;
}

sub setdata {
    my ($self, $data) = @_;
    for $i (keys %$data) {
```

```

$self->{$i}=$data->{$i};
}
return $self;
}

```

В данном примере по сравнению с предыдущим изменен конструктор `new()`. Теперь второй параметр, представленный локальной переменной `$data`, содержит ссылку. Эту ссылку функция `bless()` свяжет с классом `Staff`, превратив в его объект. Таким образом, при помощи этого параметра можно управлять типом внутренней структуры данных, которая и представляет объект. Это может быть ссылка на хеш-массив, массив, скаляр и т. д. Параметры, передаваемые конструктору, называют *переменными объекта*. Они используются для того, чтобы установить начальные значения данных каждого вновь создаваемого объекта.

Если обратиться к основной программе:

```

#!/usr/bin/perl
use Staff;
$someone=new(Staff, {"имя">"","фамилия">""});
setdata($someone,{"имя">"Максим","фамилия">"Исаев",
                  "возраст">42,"занятия спортом">"теннис"});
showdata($someone);

```

то будут выведены следующие данные:

```

возраст => 42
занятия спортом => теннис
имя => Максим
фамилия => Исаев

```

В разных ситуациях один и тот же метод может выступать как метод класса или как метод объекта. Для этого он должен "уметь" определить тип своего первого аргумента: если аргумент является ссылкой, то метод действует как метод объекта, если именем пакета, то есть строкой, то как метод класса. Подобную информацию можно получить при помощи функции `ref()`. Она возвращает значение ЛОЖЬ (пустая строка), если ее аргумент не является ссылкой, то есть объектом. В противном случае функция `ref()` возвращает имя пакета, принадлежность к которому была для данного объекта санкционирована функцией `bless()`.

13.2.3. Вызов метода

Существуют две синтаксические формы вызова как методов класса, так и методов объекта.

Первая форма имеет вид:

```
method class_or_object, parameters
```

например,

```
$somebody = new Staff, {"имя"=>"Анна"};           # метод класса
showdata $somebody, "имя", "фамилия";             # метод объекта
showdata {"имя"=>"Мария", "возраст"=>18};          # метод объекта
showdata new Staff "возраст";                     # метод объекта
showdata setdata new Staff, {"имя"=>"Глеб"}, "имя"; # метод объекта
```

Данная форма представляет собой обычный вызов функции, который может быть вложенным в другой вызов. Первым аргументом функции является ссылка (для методов объекта) или имя пакета (для методов класса).

В приведенном примере первая строка содержит вызов конструктора `new`, в котором первым (и единственным) аргументом является имя пакета.

Вторая строка содержит вызов метода объекта, в котором первым аргументом является объект-ссылка.

В третьей строке первый аргумент задается при помощи блока `{}`, возвращающего ссылку на анонимный хеш-массив. Данный хеш-массив не будет объектом, так как он не объявлен объектом класса `Staff` при помощи функции `bless()`, но синтаксически такая конструкция возможна.

В четвертой строке метод объекта вызывается с двумя аргументами. Первым аргументом является ссылка, возвращаемая конструктором `new()`, вторым — строка `"возраст"`.

В пятой строке конструктор `new` создает объект, который передается в качестве первого аргумента методу `setdata`. Вторым аргументом метода `setdata` является ссылка на анонимный хеш-массив `{"имя"=>"Глеб"}`. Метод `showdata` в качестве первого аргумента использует ссылку, возвращаемую методом `setdata`, а в качестве второго аргумента — строку `"имя"`.

Вторая форма обращения к методу имеет вид

```
class_or_object ->method(parameters)
```

Например, предыдущие вызовы могут быть записаны также в виде:

```
$somebody = Staff->new({"имя"=>"Анна"});
$somebody->showdata("имя", "фамилия");
new Staff->showdata("возраст");
new Staff->setdata({"имя"=>"Глеб"})->showdata("имя");
```

Вторая форма требует обязательного заключения аргументов в скобки.

Замечание

Как видим, обе формы записи могут быть достаточно сложными. В разных случаях любая из них может оказаться предпочтительной в смысле читаемости текста программы. Если нет серьезных оснований против, то рекомендуем использовать вторую форму, исходя из следующих соображений.

При использовании первой формы на том месте, где должен стоять объект или имя класса, синтаксис позволяет использовать либо имя класса, либо скалярную переменную, содержащую ссылку, либо блок {...}, возвращающий значение ссылки. Вторая форма для представления объекта позволяет использовать более сложные конструкции, например, элемент хеш-массива:

```
$obj->{keyname}->method().
```

При употреблении первой формы могут возникнуть трудноопределимые ошибки. Например, если происходит обращение к методу в области видимости одноименной функции, то при использовании первой формы вызова компилятор может вместо метода вызвать функцию. Вторая форма подобную ошибку исключает.

Для того чтобы вызвать метод определенного класса, следует перед именем метода указать имя пакета, как при вызове обычной подпрограммы. Например, чтобы вызвать метод, определенный в пакете `Staff`, следует использовать запись вида:

```
$someone = new Staff;
Staff::showdata($someone, "имя");
```

В данном случае просто вызывается метод `showdata` из пакета `Staff`. Ему передается в качестве аргумента объект `$someone` и прочие аргументы. Если вызвать метод следующим образом:

```
$someone=new Staff;
$something->Staff::showdata("имя");
```

то это будет означать, что для объекта `$someone` следует сначала найти метод `showdata`, начав поиск с пакета `Staff`, а затем вызвать найденный метод с объектом `$someone` в качестве первого аргумента. Напомним, что с каждым пакетом ассоциируется свой массив `@ISA`, содержащий имена других пакетов, представляющих классы. Если интерпретатор встречает обращение к методу, не определенному в текущем пакете, он ищет этот метод, рекурсивно (то есть включая производные классы) просматривая пакеты, определенные в массиве `@ISA` текущего пакета. Если подобный вызов делается внутри пакета, являющегося классом, то для того, чтобы указать в качестве начала поиска базовый класс, не указывая его имя явно, можно использовать имя псевдокласса `SUPER`:

```
$someone->SUPER::showdata("имя");
```

Такая запись имеет смысл только внутри пакета, являющегося классом.

13.2.4. Деструкторы

В главе 9 было сказано, что для каждого субъекта ссылки поддерживается счетчик ссылок. Область памяти, занимаемая субъектом ссылки, освобождается, когда значение счетчика ссылок становится равным нулю. Объект, как мы знаем, является просто ссылкой, поэтому с ним происходит то же самое: когда значение счетчика ссылок становится равным нулю, внутренняя структура данных, представляющая объект (обычно хеш-массив), освобождает память. Интерпретатор сам отслеживает значение счетчика ссылок и автоматически удаляет объект. Пользователь может определить собственные действия, завершающие работу объекта, при помощи специального метода — деструктора. Деструктор нужен для того, чтобы корректно завершить жизненный цикл объекта, например, закрыть открытые объектом файлы или просто вывести нужное сообщение. В соответствующее время деструктор будет автоматически вызван интерпретатором.

Деструктор должен быть определен внутри своего класса. Он всегда имеет имя `DESTROY`, а в качестве единственного аргумента — ссылку на объект, подлежащий удалению. Создавая подпрограмму-деструктор, следует обратить внимание на то, чтобы значение ссылки на удаляемый объект, передаваемое в качестве первого элемента массива параметров `$_[0]`, не изменялось внутри подпрограммы.

Подчеркнем, что создавать деструктор не обязательно. Он лишь предоставляет возможность выполнить некоторые дополнительные завершающие действия. Основная работа по удалению объекта выполняется автоматически. Все объекты-ссылки, содержащиеся внутри удаляемого объекта как его данные, также удаляются автоматически.

Метод `DESTROY` не вызывает другие деструкторы автоматически. Рассмотрим следующую ситуацию. Конструктор класса, вызывая конструктор своего базового класса, создает объект базового класса. Затем при помощи функции `bless()` делает последний объектом собственного класса. Оба класса, текущий и базовый, имеют собственные деструкторы. Поскольку конструктор базового класса, вызванный конструктором текущего класса, создал собственный объект, то при его удалении должен вызываться деструктор базового класса. Но этот объект уже перестал быть объектом базового класса, так как одновременно объект может принадлежать только одному классу. Поэтому при его удалении будет вызван только деструктор текущего класса. При необходимости деструктор текущего класса должен вызвать деструктор своего базового класса самостоятельно.

13.3. Обобщающий пример

В заключение рассмотрим небольшой пример, поясняющий некоторые вопросы, рассмотренные в этой главе.

Пример 13.4. Обобщающий пример

```
#!/usr/bin/perl

package Staff;

sub new {
    my ($class, $data) = @_;
    my $self = $data;
    bless $self, $class;
    return $self;
}

sub setdata {
    my ($self, $data) = @_;
    for $i (keys %$data) {
        $self->{$i}=$data->{$i};
    }
    return $self;
}

sub showdata {
    my $self = shift;
    my @keys = @_ ? @_ : sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
    return $self;
}

sub AUTOLOAD {
    print "пакет Staff: отсутствует функция $AUTOLOAD\n";
}

sub DESTROY {
    print "Удаляется объект класса Staff\n";
}

#####

package Graduate;

@ISA = (Staff);

sub new {
    my ($class, $data) = @_;
    # наследование переменной объекта
    my $self = Staff->new($data);
```

```

$self->{"образование"}="высшее";
bless $self, $class;
return $self;
}

sub showdata {
    my $self = shift;
    return $self if ($self->{"образование"} ne "высшее");
    my @keys = sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
    return $self;
}

sub DESTROY {
    my $self= shift;
    $self->SUPER::DESTROY;
    print "Удаляется объект класса Graduate\n";
}

#####

package main;

$someone=Graduate->new({"фамилия"=>"Кузнецов", "имя"=>"Николай"});
$somebody=Staff->new({"фамилия"=>"Петрова", "имя"=>"Анна"});
$someone->showdata;
$somebody->Graduate::showdata;
$someone->getdata;

```

Для простоты все классы расположены в одном файле. Если класс занимает отдельный модуль, необходимо позаботиться об управлении экспортом имен при помощи списков @EXPORT и @EXPORT_OK, а также о подключении соответствующих модулей к вызывающей программе (см. раздел 12.3).

В данном примере определен пакет main и два класса: Staff и Graduate. Staff является *базовым* классом Graduate.

Наследование методов задается при помощи массива @ISA, ассоциированного с *производным* классом Graduate. Помимо наследования методов, которое обеспечивает Perl, организовано *наследование переменных объекта* через вызов в конструкторе класса Graduate конструктора базового класса Staff. В результате объект класса Graduate получает при создании *переменную объекта* Staff (параметр, переданный конструктору new), которую изменяет,

добавляя в соответствующий хеш-массив новый элемент с ключом "образование" и значением "высшее".

Ситуация, когда конструктор производного класса вызывает конструктор базового, также обсуждалась в разделе 13.2.4. В подобном случае при удалении объекта автоматически вызывается только деструктор производного класса, хотя при вызове конструктора последовательно создавались объекты двух классов. В примере деструктор DESTROY класса Graduate вызывает деструктор базового класса. В данном случае это совершенно не обязательно, так как оба деструктора просто выводят сообщения об удалении своих объектов, но ведь это только модель. В иной ситуации такое решение может быть необходимым.

Следующая тема связана с поиском несуществующего метода, к которому происходит обращение внутри класса, и применением в этом случае функции AUTOLOAD. В пакете main вызывается метод getdata. Для объекта \$someone. \$someone является объектом класса Graduate, в котором метод getdata не определен. Обратите внимание на форму вызова метода getdata. Если бы он был вызван в виде getdata(\$someone), это означало, что вызывается функция getdata из пакета main с параметром \$someone. Поскольку в пакете main такая функция не определена, выполнение программы было бы завершено с выдачей сообщения вида:

```
Undefined subroutine &main::getdata called at . . .
```

Форма обращения \$someone->getdata однозначно определяет, что вызывается не функция, а метод объекта. Следовательно, его надо искать сначала в собственном классе Graduate, а затем в классе Staff, который определен в качестве базового для Graduate. В классе Staff метод getdata также не определен, но определена функция AUTOLOAD, которая и вызывается вместо этого метода.

Полиморфизм, т. е. переопределение методов базового объекта, показан на примере метода showdata. Класс Staff служит для порождения анкетных форм учета персонала. Его подкласс Graduate описывает подмножество таких форм только для лиц с высшим образованием. Конструктор класса Graduate автоматически добавляет в форму запись о наличии высшего образования. Метод showdata, наследованный у базового класса, изменен таким образом, чтобы игнорировать анкеты без такой записи. Поэтому информация об объекте \$somebody, принадлежащем к базовому классу, напечатана не будет.

В результате выполнения примера будут выведены строки, соответствующие действиям, заданным в программе:

```
имя => Николай  
образование => высшее  
фамилия => Кузнецов
```

пакет Staff: отсутствует функция Graduate::getdata

Удаляется объект класса Staff

Удаляется объект класса Graduate

Удаляется объект класса Staff

Вопросы для самоконтроля

1. Как связаны между собой понятия "объект" и "класс"?
2. Что такое инкапсуляция, наследование, полиморфизм в объектно-ориентированном программировании?
3. Как в языке Perl реализовано понятие "класс"?
4. Что представляет собой объект в языке Perl?
5. Для чего предназначена функция `bless()`?
6. Как, по вашему мнению, связаны между собой понятия "класс", "пакет", "модуль"?
7. Как в Perl осуществляется наследование методов? Данных?
8. Какой синтаксис используется для объявления метода?
9. Чем конструкторы в Perl отличаются от других методов?
10. В чем разница между методами класса и методами объекта?
11. Какие формы вызова объекта вы знаете? В чем, по-вашему, заключаются их преимущества и недостатки?
12. Какие имена может иметь подпрограмма-деструктор?
13. Как можно реализовать переопределение метода базового класса (полиморфизм)?
14. Как осуществить наследование переменных объекта?

Упражнение

Создайте класс, объект которого представляет дерево файловой системы для заданного каталога и позволяет вывести это дерево с указанием всех вложенных каталогов и содержащихся в них файлов.



Запуск интерпретатора и режим отладки

При запуске интерпретатора perl из командной строки можно задать разнообразные режимы его работы. Это достигается передачей ему специальных *опций*, называемых еще *переключателями* или просто *ключами*, включающих или выключающих разные режимы работы интерпретатора. Знание всех возможностей, предоставляемых опциями, позволяет более эффективно использовать интерпретатор для решения возникающих задач. Например, опция `-e` позволяет задать строку кода Perl непосредственно в командной строке. Эта возможность удобна для быстрого выполнения небольшой задачи, не прибегая к созданию файла с исходным текстом программы. Другие опции могут сэкономить время системного администратора, позволив решить некоторые задачи непосредственным вызовом perl из командной строки без написания кода достаточно большого объема.

Внутренний отладчик, которым снабжен интерпретатор perl, неоценим при отладке сложных сценариев Perl, особенно при разработке собственных модулей. Конечно, по общему признанию апологетов Perl, возможности его отладчика не так изощренны, как в C, C++ или Visual Basic, однако они могут оказаться мощным средством выявления ошибок, не обнаруживаемых самим компилятором даже в режиме отображения предупреждений.

В этой главе мы познакомимся с некоторыми опциями командной строки интерпретатора и их использованием для решения повседневных задач системного администрирования. Узнаем, как можно воспользоваться встроенным отладчиком в системе UNIX, а также изучим необходимый набор команд отладчика для выполнения основных действий в процессе обнаружения ошибок программ Perl.

14.1. Опции командной строки

Наиболее общая форма синтаксиса строки запуска интерпретатора perl имеет следующий вид:

```
perl [опции] [--] [файл_программы] [параметры_программы]
```

Опции perl задаются сразу же после имени интерпретатора и представляют собой двухсимвольные последовательности, начинающиеся с дефиса `--`:

```
perl -a -p prog.pl file1 file2
```


Некоторым из них может потребоваться передать параметры, которые задаются непосредственно после соответствующих опций:

```
perl -w -I/usr/include -0055 prog.pl file1 file2
```

Опции без параметров можно группировать, задавая целую группу с одним дефисом. Следующие вызовы perl эквивалентны:

```
perl -wd prog.pl file1 file2
perl -w -d prog.pl file1 file2
```

Опции с параметрами таким способом задавать нельзя. Можно, однако, одну опцию с параметром добавить в конец группы опций без параметров:

```
perl -wdI/usr/include -0055 prog.pl file1 file2
```

Передать опции для установки соответствующего режима работы интерпретатора можно непосредственно и из самой программы Perl. Первая строка сценария со специальным комментарием `#!` предназначена именно для этой цели:

```
#!/usr/bin/perl -w -d -I/usr/include
```

Теперь настало время узнать, какие опции существуют и как они влияют на режим работы интерпретатора. Мы решили сначала перечислить все возможные опции perl, а потом объяснить использование некоторых из них для выполнения задач системного администрирования UNIX. В табл. 14.1 представлены все опции с их кратким описанием. Советуем читателю внимательно ознакомиться с содержанием этой таблицы, так как для некоторых опций приведенное краткое описание в то же время является и исчерпывающим для понимания существа данной опции.

Таблица 14.1. Опции командной строки

Опция	Назначение
-0[nnn]	Задаёт разделитель записей файла ввода, устанавливая значение специальной переменной <code>\$/</code> . Необязательный параметр <code>nnn</code> — восьмеричный код символа; если отсутствует, принимается равным 0 (код символа <code>\0</code>). Эта опция удобна, если при каждом запуске сценария Perl в него необходимо передавать файлы с разными символами завершения записи
-a	Включает режим авторазбиения строки ввода при совместном использовании с опцией <code>-n</code> или <code>-p</code> (строка передается функции <code>split()</code> , а результат ее выполнения помещается в специальный массив <code>@F</code>). Умалчиваемый разделитель — пробел; опция <code>-F</code> позволяет задать регулярное выражение для иного разделителя

Таблица 14.1 (продолжение)

Опция	Назначение
-c	Проверка синтаксиса программы без ее выполнения (блоки BEGIN, END и use выполняются, так как это необходимо для процесса компиляции)
-d[:модуль]	Запускает сценарий в режиме отладки под управлением модуля отладки или трассировки, установленного как Devel::модуль
-D[число/список]	Устанавливает флаги отладки
-e 'строка_кода'	Выполнение строки кода Perl, заданной параметром этой опции, а не файла сценария файл_программы, указанного в командной строке вызова интерпретатора. В строке кода можно задать несколько операторов, разделенных символом ';'. Допускается задание нескольких опций -e. (В Windows строка кода задается в двойных кавычках и для использования самих кавычек в коде Perl необходимо применять управляющую последовательность '\'.)
-F/рег_выраж/	Задаёт регулярное выражение для разделителя при автоматической разбивке строки файла ввода в случае задания опции -a. Например, -F/:+/ определяет в качестве разделителя одно или более двоеточий. По умолчанию используется пробел / /. Символы "/" при задании рег_выраж не обязательны
-i[расширение]	<p>Задаёт режим редактирования по месту файла, переданного как параметр в сценарий Perl. Используется совместно с опциями -n или -p. Редактирование выполняется по следующей схеме: файл переименовывается (если задано расширение), открывается файл вывода с именем исходного файла и все операции print() сценария осуществляют вывод в этот новый файл.</p> <p>Параметр расширение используется для задания имени копии файла по следующей схеме: если в нем нет символов "*", то оно добавляется в конец имени исходного файла; каждый символ "*" этого параметра заменяется именем исходного файла</p>
-Iкаталог	Задаёт каталог поиска сценариев Perl, выполняемых операциями do, require и use (сохраняется как элемент специального массива @INC). Также задаёт каталоги размещения файлов, включаемых в сценарий Perl директивой #include препроцессора C. Можно использовать несколько опций для задания необходимого количества каталогов поиска

Таблица 14.1 (продолжение)

Опция	Назначение
-l [nnn]	Включает автоматическую обработку концов строк ввода. Работа в этом режиме преследует две цели: <ol style="list-style-type: none"> 1. Автоматическое удаление завершающего символа записи, хранящегося в специальной переменной \$/, при включенном режиме -n или -p; 2. Присваивание специальной переменной \$\\, в которой хранится символ завершения записи при выводе операторами print, восьмеричного значения nnn, представляющего код символа завершения; если этот параметр не задан, то переменной \$\\ присваивается текущее значение \$/
-m[-]модуль	Выполняет оператор use module () (опция -m) или use module (опция -M) перед выполнением сценария Perl. Если параметр модуль задан с дефисом, то use заменяется на no. Третья форма этих опций позволяет передать параметры в модуль при его загрузке
-M[-]модуль	
-[mM] [-] модуль=пар1[, пар2] ..	
-n	<p>Заданный сценарий Perl выполняется в цикле</p> <pre>while (<>) { сценарий Perl }</pre> <p>Это позволяет обработать программой Perl в цикле каждую строку всех файлов, имена которых переданы в качестве параметров сценария Perl. Эта опция эмулирует функциональность sed с ключом -n и awk</p>
-p	<p>Аналогичен опции -n, но добавляется печать каждой обрабатываемой строки, как в редакторе sed:</p> <pre>while (<>) { сценарий Perl } continue { print or die "-p destination: \$!\n"; }</pre>
-P	Перед компиляцией сценарий Perl обрабатывается препроцессором C, что позволяет использовать в нем команды препроцессора #define, #include и все команды условной компиляции C (#if, #else и т. д.). В опции -I можно задать каталоги расположения файлов, включаемых командой #include

Таблица 14.1 (окончание)

Опция	Назначение
-s	Включает режим синтаксического анализа строки вызова сценария Perl на наличие после имени файла сценария, но до параметров имен файлов "пользовательских опций" (параметров, начинающихся с дефиса). Такие параметры извлекаются из массива @ARGV и в программе объявляются и определяются переменные с именами введенных опций. Следующий сценарий напечатает "Задана опция opt1" тогда и только тогда, когда он будет вызван с опцией -opt1: <pre>#!/usr/bin/perl -s if (\$opt1) { print " Задана опция opt1\n"; }</pre>
-S	Поиск файла программы Perl файл_программы осуществляется с использованием значения переменной среды PATH
-T	Включает режим проверки на безопасность полученных извне данных в операциях с файловой системой. Полезен при реализации CGI-сценариев. Эта опция должна быть первой в списке опций интерпретатора perl, если она применяется к сценарию
-u	Создание дампа ядра после компиляции сценария Perl. Последующее использование программы undump позволяет создать двоичный выполняемый файл сценария
-U	Разрешает выполнение небезопасных операций. При использовании этой опции в UNIX программа Perl выполняется в незащищенном режиме и ей предоставляется полный доступ к файловой системе
-v	Отображает номер версии, а также другую важную информацию об интерпретаторе perl
-V[:переменная]	Отображает информацию конфигурации интерпретатора perl (расположение необходимых библиотек, значения переменных среды и т. д.). Если задано имя переменной среды, отображает ее установленное значение
-w	Включает режим отображения предупреждений во время компиляции программы Perl. Рекомендуется <i>всегда</i> применять эту опцию
-x[каталог]	Извлекает и выполняет сценарий Perl, расположенный в текстовом файле. Эта возможность удобна, если сценарий прислан по электронной почте. Интерпретатор сканирует файл, пока не найдет строку, начинающуюся с символов "#!" и содержащую слово "perl". После этого выполняется сценарий до лексемы <code>END</code> . Если задан параметр каталог, то перед выполнением сценария он становится текущим

Как видно из табл. 14.1, интерпретатор perl располагает достаточно большим набором опций, разработанных специально в помощь системному администратору UNIX. Эти опции позволяют решать задачи, даже не прибегая к созданию текстового файла сценария Perl.

Одной из наиболее часто встречающихся задач системного администрирования является задача обработки содержимого большого числа файлов и данных (например, изменение значений некоторых переменных среды в конфигурационных файлах). Можно написать сценарий Perl, в цикле обрабатывающий строки переданных ему файлов, а можно воспользоваться опциями `-n` или `-p`, которые организуют цикл по файлам, и задать необходимые действия для каждой строки файла в опции `-e`. Таким образом, быстрого решения задачи гарантирована!

Итак, мы хотим в конфигурационных файлах `config1`, `config2` и `config3`, содержащих строки вида `ключ = ЗНАЧЕНИЕ`, изменить значение ключа `"key1"` на величину 5. Эта задача решается следующим вызовом perl из командной строки:

```
perl -p -i.bak -e "m/(\w+)\s*=\s*(.+)/i;  
if($1 eq \"key1\"){$_ = \"$1 = 5\\n\"};" config1 config2 config3
```

Несколько комментариев к строке вызова интерпретатора perl. В ней использованы опции `-p` и `-i` для обработки содержимого конфигурационных файлов по месту (опция `-i`, причем задано расширение `.bak` для копий исходных файлов) в неявном цикле с печатью (опция `-p`). Код модификации содержимого файлов задается с помощью опции `-e`. Он достаточно прост. Строка файла, прочитанная в специальную переменную Perl `$_`, проверяется на содержание подстроки вида `ключ = ЗНАЧЕНИЕ`. В регулярном выражении определены две группы: первая соответствует ключу, а вторая — его значению. Таким способом мы сохраняем имя ключа и его значение в специальных переменных `$1` и `$2`, соответственно. Второй оператор кода, в случае совпадения имени ключа с заданным, заносит в переменную `$_` строку с новым значением ключа (`key1 = 5`).

Как выполняется этот сценарий? Создается копия исходного конфигурационного файла `config1` (файл `config1.bak`) и новый пустой файл с именем `config1`. Последовательно в переменную `$_` читаются строки из файла-копии, а в переменную `$1` заносится имя ключа, если таковой обнаружен оператором `m/.../`. Если имя ключа равно `"key1"`, то в переменной `$_` формируется строка задания нового значения ключа. Перед чтением следующей строки файла `config1.bak` в файл `config1` записывается содержимое переменной `$_`. Эта процедура выполняется со всеми заданными конфигурационными файлами. В результате выполнения этой программы формируются конфигурационные файлы, в которых значение ключа `key1` изменено на 5.

Опция `-p` достаточно полезна для анализа содержимого файлов. Например, следующая команда отобразит на экране содержимое файла `prog.pl`:

```
perl -p -e"1" prog.pl
```

Замечание

Задание кода программы Perl при работе с интерпретатором `perl` является необходимым действием. Его можно задать, указав имя файла программы после всех опций, либо непосредственно в командной строке в опции `-e`. При печати содержимого файла мы задали в командной строке программу из одного оператора, который, по существу, ничего не делает.

При работе с интерпретатором из командной строки очень часто используются регулярные выражения для отображения какой-либо части содержимого файла. Следующий вызов `perl` отображает на экране монитора последний столбец данных из файла `test.dat` (элементы столбцов в строке файла разделены пробелами):

```
perl -p -e "s/\s*.\s*(.)\s*/$1\n/;" test.dat
```

Замечание

При использовании опции `-p` следует помнить, что она печатает содержимое специальной переменной `$_`. Поэтому информацию, подлежащую отображению, следует помещать именно в эту переменную, как в последнем примере.

Можно было бы привести еще массу полезных примеров вызова интерпретатора `perl` с различными опциями для решения разнообразных задач, но мы ограничимся приведенными примерами. При написании этой главы мы ставили целью всего лишь показать, на что способен интерпретатор `perl`. Думайте, размышляйте, и вы сможете решать многие задачи, не прибегая к написанию больших файлов с кодом Perl.

14.2. Отладчик Perl

Написав программу и устранив ошибки компиляции, любой программист тайне надеется, что она будет работать именно так, как он и задумывал. Практика, однако, в большинстве случаев совершенно иная: при первом запуске программа работает, но выполняет не совсем, а иногда и совсем не то, что надо; при одних исходных данных она работает идеально, а при других "зависает". И это случается не только с начинающим, но и с опытным программистом. В таких случаях на помощь приходит *отладчик* — специальная программа, которая позволяет программисту, обычно в интерактивном режиме, предпринять определенные действия в случае возникновения необычного поведения программы.

Для вызова встроенного отладчика интерпретатора perl его необходимо запустить с опцией `-d` (задав ее либо в командной строке, либо в строке специального комментария `#!` самой программы). Отладчик инициализируется, когда начинается выполнение сценарий Perl. При этом отображается версия отладчика, первая выполняемая строка кода программы и его приглашение `DB<1>` на ввод команд пользователя:

```
Loading DB routines from perl5db.pl version 1.0402
```

```
Emacs support available.
```

```
Enter h or `h h' for help.
```

```
main:.(example1.pl:3):  print "\000\001\002\003\004\005\006\007\n";
DB<1>
```

После чего можно вводить команды отладчика для выполнения определенных действий: установить точки останова, определить действия при выполнении определенной строки кода программы, посмотреть значения переменных, выполнить часть кода сценария и т. д. Эта глава и посвящена краткому описанию команд, задающих необходимые действия при отладке разработанного сценария Perl. Мы здесь не учим стратегии поиска ошибок, а только лишь показываем возможности отладчика при ее реализации.

Замечание

Все, что здесь говорится о встроенном отладчике, относится к пользователям системы UNIX. Тот, кто работает в системах семейства Windows и пользуется интерпретатором perl фирмы ActiveState, может совсем не читать этот раздел, так как интерпретатор этой фирмы снабжен отладчиком, работа с которым осуществляется через графический интерфейс пользователя, а не с помощью командной строки. Хотя ничто не мешает таким пользователям воспользоваться встроенным отладчиком, аналогичным отладчику интерпретатора perl для UNIX, и работать с ним из командной строки. Для этого следует воспользоваться командным файлом `cmdddb.bat`, расположенным в каталоге `\perl520\debugger`.

14.2.1. Просмотр текста программы

При загрузке отладчика отображается первая строка кода программы Perl. Но для выполнения определенных действий (установки точки останова, задания определенных действий при выполнении конкретного оператора Perl и т. д.) нам необходимо видеть текст программы. Как это осуществить? В отладчике для подобных действий предусмотрен ряд команд.

Команда `l` отображает на экране монитора 10 строк текста программы, расположенных непосредственно за последней отображенной строкой. Последовательное выполнение этой команды позволяет быстро пролистать текст

программы. Вызов команды `l` с указанием номера строки отобразит ее содержимое. Несколько последовательных строк программы можно увидеть, задав в команде `l` через дефис номер начальной и конечной строки. Например, команда `l 1-5` отобразит пять строк программы:

```
DB<5> l 1-5
```

```
1      #! perl -w
2==>   open FILE, "books" or die $!;
3:     open REPORT, ">report";
4
5:     select REPORT;
```

Обратите внимание, что отладчик отображает не только содержимое строк, но и их номера, причем после номера строки с первым выполняемым оператором следует стрелка, а номер каждой последующей выполняемой строки завершается символом двоеточия ":". В угловых скобках приглашения отладчика `DB<5>` отображается порядковый номер выполненной команды от начала сеанса отладки.

Команда `w` отобразит блок, или окно строк — три строки до текущей, текущая строка и шесть после текущей. Текущей в отладчике считается строка программы после последней отображенной. Если команде `w` передать номер строки, то отобразится окно строк относительно заданной строки текста — три строки до заданной, сама заданная строка и шесть после нее.

Последняя команда отображения текста программы — команда `-` (дефис), которая отображает 10 строк текста, предшествующих текущей строке.

14.2.2. Выполнение кода

Команда `s` предназначена для последовательного пошагового выполнения программы: каждый ее вызов выполняет следующую строку кода. Эта строка должна выполняться в программе без изменения потока выполнения операторов. После выполнения очередной строки кода отображается строка, которая должна быть выполнена следующей:

```
DB<1> s
```

```
main::(example2.pl:3): @s = split;
```

При следующем выполнении команды `s` будет выполнена отображенная строка программы: `@s = split;`.

Если следующая выполняемая строка кода расположена в модуле, код которого доступен (например, вызов подпрограммы), то осуществляется переход в него, соответствующая строка кода выполняется и отладчик приостанавливает выполнение программы до ввода следующей программы. Последующие команды `s` будут построчно выполнять код подпрограммы, пока не вы-

полнится последняя строка ее кода. Таким образом, команда `s` позволяет выполнять подпрограмму с заходом в нее.

Команда `n` работает аналогично команде `s`, последовательно выполняя строки кода программы, за одним исключением — она выполняет код подпрограммы без захода в него, т. е. код подпрограммы выполняется полностью, и при следующем вызове команды `n` выполняется оператор строки кода, непосредственно следующей за строкой с вызовом подпрограммы. Таким образом, команда `n` позволяет "обойти" пошаговое выполнение операторов подпрограммы.

Совет

Если мы осуществляем пошаговое выполнение программы (командами `s` или `n`), то нажатие клавиши `<Enter>` эквивалентно вызову последней команды `s` или `n`.

Если мы в процессе отладки попали в тело подпрограммы (после, например, выполнения очередной команды `s`), то командой `r` можно немедленно завершить ее выполнение. Выполнение программы приостанавливается на первом после вызова подпрограммы операторе в ожидании очередной команды пользователя.

Пошаговое выполнение программы командами `s` и `n` может оказаться утомительным делом при отладке кода большого объема. В отладчике Perl предусмотрена команда `c`, которая выполняет программу от текущего оператора до первой встретившейся точки останова или до завершения программы, если точки останова на соответствующем участке кода не определены.

Эта же команда позволяет выполнить программу от текущей строки до определенной строки кода программы. Для этого необходимо в вызове команды `c` указать номер строки, до которой должна выполняться программа, если только не встретится точка останова.

14.2.3. Просмотр значений переменных

В процессе отладки можно посмотреть значение любой переменной программы в любой момент времени. Команда

```
V [пакет [переменная]]
```

отображает значение заданной переменной указанного пакета. Выполненная без параметров, она отображает значения всех переменных программы из всех пакетов.

При работе с этой командой следует иметь в виду, что при задании переменной, значение которой необходимо посмотреть, не надо задавать никакого префикса, а только идентификатор переменной. Отладчик отобразит значения всех переменных указанного пакета с заданным идентификатором. Например, если в программе определена скалярная переменная `$ref`, мас-

сив скаляров `@ref` и хеш-массив `%ref`, то выполнение команды `V main ref` приведет к следующему результату:

```
DB<1> V main ref
$ref = 24
@ref = (
  0  1
  1  2
)
%ref = (
  'One' => 1
  'Two' => 2
)
```

Команда `x` аналогична команде `v`, но она отображает значения переменных текущего пакета. Ее параметром является идентификатор переменной, имя пакета указывать не надо. Вызванная без параметров, она отображает значения всех переменных текущего пакета. Например, команда `x ref` отобразит значения переменной `$ref`, массива `@ref` и хеш-массива `%ref` текущего пакета, внутри которого приостановлено выполнение программы. По умолчанию программа выполняется в пакете `main`.

Команда `t` работает как переключатель, включая режим отображения строк выполняемого кода (режим трассировки) или выключая его:

```
DB<11> t
Trace = on
DB<11> c 5
main::(example2.pl:3): @ref = (1,2,3,4);
main::(example2.pl:4): %ref = ("One",1, "Two",2);
main::(example2.pl:5): $_ = "  qwerty \t\tqwerty";
DB<12> t
Trace = off
DB<12> c 7
main::(example2.pl:7): mySub();
DB<13>
```

Обратите внимание, что при включенном режиме трассировки отображаются все строки выполняемого кода, тогда как после его выключения, отображается только следующий, подлежащий выполнению оператор.

14.2.4. Точки останова и действия

В процессе отладки программы возникает необходимость приостановить ее выполнение в определенных подозрительных местах, посмотреть значения

переменных и предпринять дальнейшие действия по отладке кода. Нам уже известна команда `c` отладчика, которая непрерывно выполняет код программы до первой встретившейся точки останова, но как задать ее?

Для этих целей служит команда `b` (сокращение от английского глагола *break* — прервать) отладчика. Ее параметром является номер строки кода, в которой устанавливается точка останова: отладчик приостановит выполнение программы Perl перед заданной строкой. Если команда `b` вызывается без параметра, то точка останова определяется в текущей строке.

Можно определить точку останова в первой строке кода подпрограммы. Для этого команде `b` необходимо передать в качестве параметра имя подпрограммы. Например, следующая команда

```
DB<11> b mySub
```

устанавливает точку останова в первой строке кода подпрограммы `mySub`.

Иногда необходимо, чтобы выполнение программы приостанавливалось в некоторой точке программы только при выполнении каких-либо условий (например, равенства заданному числу значения какой-нибудь переменной, или совпадения значений двух других переменных и т. п.). Команда `b` позволяет задавать подобные условные точки останова. Для этого ей можно передать в качестве второго параметра условие, при истинности которого точка останова будет восприниматься отладчиком как действительная точка останова. Если условие перед выполнением строки кода не будет истинно, то останова программы в этой точке не произойдет. Например, следующая команда

```
DB<1> b 4 $r==1
```

определяет условную точку останова в строке 4. Отладчик приостановит выполнение программы перед этой строкой по команде `c` только, если значение переменной `$r` будет равно 1.

Команда `L` отображает список всех установленных точек останова, как безусловных, так и условных:

```
DB<1> b 4 $r==1
```

```
DB<2> b 6
```

```
DB<3> L
```

```
example2.pl:
```

```
4:      %ref = ("One",1, "Two",2);
```

```
      break if ($r==1)
```

```
6:      @s = split;
```

```
      break if (1)
```

Отображаемая информация о точке останова представляет номер строки и код Perl, а также условие, при котором действует точка останова (`break`

if (УСЛОВИЕ)). Для безусловной точки останова условие всегда истинно и равно 1.

Для удаления точки останова достаточно выполнить команду `d` с параметром, равным номеру строки, в которой определена точка останова. Команда `D` удаляет все точки останова, определенные в сеансе работы с отладчиком.

Полезно при отладке программы задать действия, которые будут предприняты перед выполнением операторов определенной строки. Например, напечатать значения каких-либо переменных или изменить их при выполнении некоторого цикла. Подобное поведение программы можно реализовать, задав действия командой `a`. Два ее параметра определяют строку кода и сами действия (обычный оператор Perl) перед началом выполнения операторов заданной строки:

```
a 75 print "*** $ref\n";
```

Можно задать несколько операторов для выполняемых действий, однако следует учитывать, что это может привести к смещению отображаемой на экране монитора информации из самой программы и установленных действий.

Любое действие можно выполнить немедленно, набрав код в строке приглашения отладчика `DB<>`. Подобные действия не изменяют текст программы (операторы действий не записываются в ее файл), но позволяют создавать новые переменные и использовать их в вычислениях. Правда, по завершении сеанса отладки подобная информация пропадает.

Мы познакомили читателя лишь с основными командами отладчика, наиболее важными и полезными, с нашей точки зрения, для процесса поиска ошибок. Их полный набор с краткими описаниями представлен в табл. 14.2. Более подробную информацию можно всегда найти в документации, с которой распространяется Perl, или из различных ресурсов Internet.

Таблица 14.2. Основные команды отладчика

Команда	Описание
T	Отображается содержимое стека вызванных подпрограмм
s	Пошаговое выполнение программы (с заходом в подпрограммы)
n	Пошаговое выполнение программы (без захода в подпрограммы)
<Enter>	Повтор последней команды s или n
r	Завершение текущей подпрограммы и возврат из нее

Таблица 14.2 (продолжение)

Команда	Описание
c [строка]	Непрерывное выполнение кода программы до первой точки останова или указанной строки, или подпрограммы
c [подпрогр]	
l строка+число	Отображает число плюс одну строку кода, начиная с заданной строки
l строка1-строка2	Отображает диапазон строк: от строки с номером строка1 до строки с номером строка2
l строка	Отображает заданную строку
l подпрогр	Отображает первый блок строк кода подпрограммы
l	Отображает следующий блок из 10 строк
-	Отображает предыдущий блок из 10 строк
w [строка]	Отображает блок строк вокруг заданной строки
.	Возврат к выполненной строке
f файл	Переключение на просмотр файла. Файл должен быть загружен
/образец/	Поиск строки по образцу; направление вперед от текущей строки. Завершающая косая черта не обязательна
?образец?	Поиск строки по образцу; направление назад от текущей строки. Завершающий символ "?" не обязателен
L	Отображение всех установленных точек останова
S [[!]образец]	Отображение имен подпрограмм, [не] соответствующих образцу
t	Включение/выключение режима трассировки
b [строка] [условие]	Установка точки останова в заданной строке и условия ее действия
b подпрогр [условие]	Установка точки останова в первой строке подпрограммы и условия ее действия
b load файл	Установка точки останова на операторе require файл
b postpone подпрогр [условие]	Установка точки останова в первой строке подпрограммы после ее компилирования
b compile подпрогр	Остановка после компилирования подпрограммы

Таблица 14.2 (продолжение)

Команда	Описание
d [строка]	Удаление точки останова из строки
D	Удаление всех точек останова
a [строка] команда	Установка действий, выполняемых перед выполнением операторов строки
A	Удаление всех действий
W выражение	Добавление глобального наблюдаемого выражения
W	Удаление всех глобальных наблюдаемых выражений
V [пакет [переменная]]	Отображение переменной пакета
X [переменная]	Отображение переменной текущего пакета
x выражение	Вычисляет выражение в списковом контексте и отображает его значение
m выражение	Вычисляет выражение в списковом контексте, отображает методы, вызывавшиеся при вычислении первого элемента результата
m класс	Отображает вызываемые методы заданного класса
O [опция [=знач]] [опция "знач"] [опция?] ...	Устанавливает или запрашивает значения опций отладчика. Эти опции влияют на поведение самого отладчика и режимы работы некоторых команд отладчика, например, V, X, и x. Их достаточно много, и мы рекомендуем обратиться к странице perldebug.html из документации Perl
< выражение	Определение команды Perl, которая должна выполняться перед каждым приглашением отладчика ввести команду
<< выражение	Добавление команды в список команд Perl, которые должны выполняться перед каждым приглашением отладчика ввести команду
> выражение	Определение команды Perl, которая должна выполняться после каждого приглашения отладчика ввести команду
>> выражение	Добавление команды в список команд Perl, которые должны выполняться после каждого приглашения отладчика ввести команду
{ команда	Определение команды отладчика, которая должна выполняться перед каждым его приглашением ввести команду

Таблица 14.2 (окончание)

Команда	Описание
{ { команда	Добавление команды в список команд отладчика, которые должны выполняться перед каждым его приглашением ввести команду
! номер	Выполнение предыдущей команды отладчика; номер задает отсчет от первой выполненной команды
! -номер	Выполнение предыдущей команды отладчика; номер задает отсчет от последней выполненной команды
! образец	Выполнение последней команды, которая начиналась с указанного образца
!! команда	Выполнение команды в подпроцессе (читает из DB::IN, записывает в DB::OUT)
Н -число	Отображение последних выполненных команд (без параметра отображаются все)
р выражение	Эквивалентна "print {DB::OUT} выражение" в текущем пакете
команда	Выполнение команды отладчика, направляя DB::OUT на текущий пейджер
команда	Аналогична предыдущей команде, но только DB::OUT временно становится текущим
= [псевдоним значение]	Определяет псевдоним команды или отображает список текущих псевдонимов, если выполняется без параметров
команда	Выполнение оператора Perl в текущем пакете
v	Отображает версии загруженных модулей
R	Перезагрузка отладчика (некоторые его установки могут быть потеряны). Сохраняются следующие установки: история, точки останова и действия, установки отладчика опцией O и опции команд -w, -I, -e
h [команда]	Получение справки, если в виде h, то выводится полностранично
h h	Отображение всех команд отладчика
q или <Ctrl>+<D>	Выход

* * *

В этой, можно сказать, завершающей главе мы познакомились с возможностями интерпретатора perl для решения некоторых задач системного адми-

нистрирования в UNIX. Установка некоторых опций интерпретатора при его запуске из командной строки меняет режим работы, позволяя практически без написания кода изменять, проверять, копировать и отображать содержимое файлов.

Для удобства и ускорения отладки больших программ в интерпретаторе perl предусмотрен встроенный отладчик. Его команды позволяют приостанавливать выполнение сценария Perl в подозрительных точках, задавать определенные действия при выполнении кода программы, просматривать стек вызова подпрограмм, менять в цикле значения переменных программы и многие другие полезные при поиске ошибок действия. Использование отладчика ускоряет процесс разработки программ Perl.

Вопросы для самоконтроля

1. Какую функцию выполняют опции интерпретатора perl?
2. Какие существуют способы задания опций интерпретатора?
3. Перечислите наиболее полезные опции для выполнения задач, связанных с системным администрированием в UNIX.
4. Зачем нужен отладчик и как его инициировать?
5. Перечислите основные действия, которые позволяет выполнять отладчик в процессе отладки программы Perl.



Язык Perl и CGI-программирование

15.1. Основные понятия

Основу "всемирной паутины" WWW составляют *Web-узлы*. Это компьютеры, на которых выполняется специальная программа — *Web-сервер*, ожидающая запроса со стороны клиента на выдачу документа. Документы сохраняются на *Web-узле*, как правило, в формате HTML. Клиентом *Web-сервера* является программа-браузер, выполняющаяся на удаленном компьютере, которая осуществляет запрос к *Web-серверу*, принимает запрошенный документ и отображает его на экране.

Аббревиатура CGI (*Common Gateway Interface*) обозначает часть *Web-сервера*, которая может взаимодействовать с другими программами, выполняющимися на этом же *Web-узле*. В этом смысле она является шлюзом (*gateway* — шлюз) для передачи данных, полученных от клиента, программам обработки — таким, как СУБД, электронные таблицы и др. CGI включает общую среду (набор переменных) и протоколы для взаимодействия с этими программами.

Общая схема работы CGI состоит из следующих элементов.

1. Получение *Web-сервером* информации от клиента-браузера. Для передачи данных *Web-серверу* в языке HTML имеется средство, называемое *форма*. Форма задается в HTML-документе при помощи тэгов `<FORM>...</FORM>` и состоит из набора *полей ввода*, отображаемых браузером в виде графических элементов управления: селекторных кнопок, опций, строк ввода текста, управляющих кнопок и т. д. (рис. 15.1).
2. Анализ и обработка полученной информации. Данные, извлеченные из HTML-формы, передаются для обработки CGI-программе. Они не всегда могут быть обработаны CGI-программой самостоятельно. Например, они могут содержать запрос к некоторой базе данных, которую CGI-программа читать "не умеет". В этом случае CGI-программа на основании полученной информации формирует запрос к компетентной программе, выполняющейся на том же компьютере. CGI-программа может быть написана на любом языке программирования, имеющем средства обмена данными между программами. В среде UNIX для этой цели наиболее часто используется язык Perl. Так как UNIX является наиболее популяр-

ной операционной системой для Web-серверов, то можно считать Perl наиболее популярным языком CGI-программирования. Программа на языке Perl представляет собой последовательность операторов, которые *интерпретатор* языка выполняет при каждом запуске без преобразования исходного текста программы в выполняемый двоичный код. По этой причине CGI-программы называют также *CGI-сценариями* или *CGI-скриптами*.

- Создание нового HTML-документа и пересылка его браузеру. После обработки полученной информации CGI-программа создает динамический или, как говорят, виртуальный HTML-документ, или формирует ссылку на уже существующий документ и передает результат браузеру.

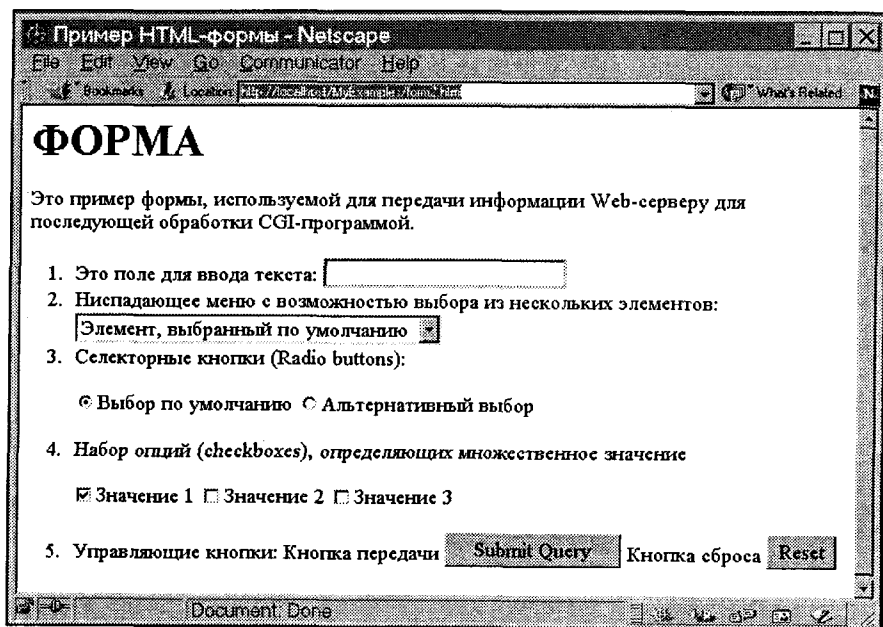


Рис. 15.1. Пример отображения HTML-формы браузером

15.2. HTML-формы

HTML-формы предназначены для пересылки данных от удаленного пользователя к Web-серверу. С их помощью можно организовать простейший диалог между пользователем и сервером (например, регистрацию пользователя на сервере или выбор нужного документа из представленного списка). Формы поддерживаются всеми популярными браузерами.

(Различные аспекты передачи данных Web-серверу будут рассмотрены в разделе 15.3.)

Мы предполагаем, что читатель знаком с основами языка HTML и структурой HTML-документа. В этом разделе рассмотрены средства HTML, используемые для создания форм. В описании тэгов приведены только наиболее употребительные атрибуты и опущены атрибуты, специфические для отдельных браузеров.

15.2.1. Тэг <FORM>

```
<FORM ACTION="URL" METHOD=метод_передачи ENCTYPE=MIME-тип>
  содержание_формы
</FORM>
```

В HTML-документе для задания формы используются тэги <FORM>...</FORM>, отмечающие, соответственно, начало и конец формы. Документ может содержать несколько форм, но они не могут быть вложены одна в другую. Тэг <FORM> имеет атрибуты ACTION, METHOD и ENCTYPE. Отдельные браузеры (Netscape, Internet Explorer) поддерживают дополнительные атрибуты помимо стандартных, например, CLASS, NAME, STYLE и др.

Атрибут ACTION — единственный обязательный. Его значением является адрес (URL) CGI-программы, которая будет обрабатывать информацию, извлеченную из данной формы.

Атрибут METHOD определяет метод пересылки данных, содержащихся в форме, от браузера к Web-серверу. Он может принимать два значения: GET (по умолчанию) и POST.

Замечание

Взаимодействие между клиентом-браузером и Web-сервером осуществляется по правилам, заданным протоколом HTTP, и состоит из запросов клиента и ответов сервера. Запрос разбивается на три части. В первой строке запроса содержится HTTP-команда, называемая *методом*, URL запрашиваемого файла и номер версии протокола HTTP. Вторая часть — заголовок запроса. Третья часть — тело запроса, собственно данные, посылаемые серверу. *Метод* сообщает серверу о цели запроса. В протоколе HTTP определены несколько методов. Для передачи данных формы в CGI-программу используются два метода: GET и POST.

При использовании метода GET данные формы пересылаются в составе URL запроса, к которому присоединяются после символа "?" в виде совокупности пар *переменная=значение*, разделенных символом "&". В этом случае первая строка запроса может иметь следующий вид:

```
GET      /cgi-bin/cgi-program.pl?name=Mike&surname=Ivanoff      HTTP/1.1
         └────────────────────────────────────────┘
         URL запроса                                     Данные формы
```

После выделения данных из URL сервер присваивает их переменной среды QUERY_STRING, которая может быть использована CGI-программой.

При использовании метода POST данные формы пересылаются Web-серверу в теле запроса, после чего передаются сервером в CGI-программу через стандартный ввод.

Значением атрибута ENCTYPE является *медиа-тип*, определяющий формат кодирования данных при передаче их от браузера к серверу. Браузер кодирует данные, чтобы избежать их искажения в процессе передачи. Возможны два значения этого атрибута: *application/x-www-form-urlencoded*, используемое по умолчанию, и *multipart/form-data*.

Замечание

Одним из первых применений Internet была электронная почта, ориентированная на пересылку текстовых сообщений. Часто возникает необходимость вместе с текстом переслать данные в нетекстовом формате, например, упакованный zip-файл, рисунок в формате GIF, JPEG и т. д. Для того чтобы пересылать средствами электронной почты такие файлы без искажения, они кодируются в соответствии с некоторым стандартом. Стандарт MIME (*Multipurpose Internet Mail Extensions*, многоцелевые расширения электронной почты для Internet) определяет набор *MIME-типов*, соответствующих различным типам данных, и правила их пересылки по электронной почте. Для обозначения MIME-типа используется запись вида *тип/подтип*. *Тип* определяет общий тип данных, например, *text*, *image*, *application* (тип *application* обозначает специфический внутренний формат данных, используемый некоторой программой), а *подтип* — конкретный формат внутри типа данных, например, *application/zip*, *image/gif*, *text/html*. MIME-типы нашли применение в Web, где они называются также *медиа-типами*, для идентификации формата документов, передаваемых по протоколу HTTP. В HTML-форме атрибут ENCTYPE определяет медиа-тип, который используется для кодирования и пересылки специального типа данных — содержимого формы.

Как видно из примера на рис. 15.1, форма отображается в окне браузера в виде набора стандартных элементов управления, используемых для заполнения полей формы значениями, которые затем передаются Web-серверу. Значение вводится в поле ввода пользователем или назначается по умолчанию. Для создания полей средствами языка HTML существуют специальные тэги: <INPUT>, <SELECT>, <TEXTAREA>, которые могут употребляться только внутри тэга <FORM>.

15.2.2. Тэг <INPUT>

```
<INPUT TYPE=тип_поля_ввода NAME=имя_поля_ввода другие_атрибуты>
```

Наиболее употребительный тэг, с помощью которого можно генерировать внутри формы поля для ввода строки текста, пароля, имени файла, различные кнопки. Он имеет два обязательных атрибута: TYPE и NAME. Атрибут TYPE

определяет тип поля: селекторная кнопка, кнопка передачи и др. Атрибут NAME определяет имя, присваиваемое полю. Оно не отображается браузером, а используется в качестве идентификатора значения, передаваемого Web-серверу. Остальные атрибуты меняются в зависимости от типа поля. Ниже приведено описание типов полей, создаваемых при помощи тэга <INPUT>, и порождаемых ими элементов ввода.

☐ TYPE=TEXT

Создает элемент для ввода строки текста. Дополнительные атрибуты:

☐ MAXLENGTH=n

Задаёт максимальное количество символов, разрешенных в текстовом поле. По умолчанию не ограничено.

- SIZE=n

Максимальное количество отображаемых символов.

- VALUE=начальное_значение

Первоначальное значение текстового поля.

☐ TYPE=PASSWORD

Создает элемент ввода строки текста, отличающийся от предыдущего только тем, что все вводимые символы представляются в виде символа *.

Замечание

Поле PASSWORD не обеспечивает безопасности введенного текста, так как на сервер он передается в незашифрованном виде.

☐ TYPE=FILE

Создает поле для ввода имени локального файла, сопровождаемое кнопкой **Browse**. Выбранный файл присоединяется к содержимому формы при пересылке на сервер. Имя файла можно ввести непосредственно, или, воспользовавшись кнопкой **Browse**, выбрать его из диалогового окна, отображающего список локальных файлов. Для корректной передачи присоединенного файла следует установить значения атрибутов формы равными ENCTYPE="multipart/form-data" и METHOD=POST. В противном случае будет передана введенная строка, то есть маршрутное имя файла, а не его содержимое. Дополнительные атрибуты MAXLENGTH и SIZE имеют тот же смысл, что и для элементов типа TEXT и PASSWORD.

☐ TYPE=CHECKBOX

Создает элемент-переключатель, принимающий всего два значения (on/off, вкл./выкл., истина/ложь) и отображаемый в виде квадратной кнопки. Элементы-переключатели CHECKBOX можно объединить в группу, установив одинаковое значение атрибута NAME для всех ее элементов. Дополнительные атрибуты:

- `VALUE=строка`

Значение, которое будет передано серверу, если данная кнопка выбрана. Если кнопка не выбрана, значение не передается. Обязательный атрибут.

- `CHECKED`

Если указан атрибут `CHECKED`, элемент является выбранным по умолчанию.

Если переключатели образуют группу, то передаваемым значением является строка разделенных запятыми значений атрибута `VALUE` всех выбранных элементов.

☐ `TYPE=RADIO`

Создает элемент "радиокнопка", существующий только в составе группы подобных элементов, из которых может быть выбран только один. Все элементы группы должны иметь одинаковое значение атрибута `NAME`. Отображается в виде круглой кнопки. Дополнительные атрибуты:

- `VALUE=строка`

Обязательный атрибут, значение которого передается серверу при выборе данной кнопки. Должен иметь уникальное значение для каждого члена группы.

- `CHECKED`

Устанавливает элемент выбранным по умолчанию. Один и только один элемент в группе должен иметь этот атрибут.

☐ `TYPE=SUBMIT`

Создает кнопку передачи, нажатие которой вызывает пересылку на сервер всего содержимого формы. По умолчанию отображается в виде прямоугольной кнопки с надписью **Submit**. Дополнительный атрибут

`VALUE=название_кнопки`

позволяет изменить надпись на кнопке. Атрибут `NAME` для данного элемента может быть опущен. В этом случае значение кнопки не включается в список параметров формы и не передается на сервер. Если атрибуты `NAME` и `VALUE` присутствуют, например,

```
<INPUT TYPE=SUBMIT NAME="submit_button" VALUE="OK">
```

то в список параметров формы, передаваемых на сервер, включается параметр `submit_button="OK"`. Внутри формы могут существовать несколько кнопок передачи.

☐ `TYPE=RESET`

Создает кнопку сброса, нажатие которой отменяет все сделанные изменения, восстанавливая значения полей формы на тот момент, когда она

была загружена. По умолчанию отображается в виде прямоугольной кнопки с надписью **Reset**. Надпись можно изменить при помощи дополнительного атрибута

VALUE=название_кнопки

Значение кнопки **Reset** никогда не пересылается на сервер, поэтому у нее отсутствует атрибут NAME.

☐ TYPE=IMAGE

Создает элемент в виде графического изображения, действующий аналогично кнопке **Submit**. Дополнительные атрибуты:

- SRC=url_изображения

Задает ссылку (url) на файл с графическим изображением элемента.

- ALIGN=тип_выравнивания

Задает тип выравнивания изображения относительно текущей строки текста точно так же, как одноименный атрибут тэга .

Если на изображении элемента щелкнуть мышью, то координаты указателя мыши в виде NAME.x=n&NAME.y=m включаются браузером в список параметров формы, посылаемых на сервер.

☐ TYPE=HIDDEN

Создает скрытый элемент, не отображаемый пользователю. Информация, хранящаяся в скрытом поле, всегда пересылается на сервер и не может быть изменена ни пользователем, ни браузером. Скрытое поле можно использовать, например, в следующем случае. Пользователь заполняет форму и отправляет ее серверу. Сервер посылает пользователю для заполнения вторую форму, которая частично использует информацию, содержащуюся в первой форме. Сервер не хранит историю диалога с пользователем. Он обрабатывает каждый запрос независимо и при получении второй формы не будет знать, как она связана с первой. Чтобы повторно не вводить уже введенную информацию, можно заставить CGI-программу, обрабатывающую первую форму, переносить необходимые данные в скрытые поля второй формы. Они не будут видимы пользователем и, в то же время, доступны серверу. Значение скрытого поля определяется атрибутом VALUE.

15.2.3. Тэг <SELECT>

```
<SELECT NAME=имя_поля SIZE=n MULTIPLE>
```

элементы OPTION

```
</SELECT>
```

Тэг <SELECT> предназначен для того, чтобы организовать внутри формы выбор из нескольких вариантов без применения элементов ввода типа CHECKBOX

и RADIO. Дело в том, что если элементов выбора много, то представление их в виде переключателей и радиокнопок увеличивает размеры формы, делая ее труднообозримой. С помощью тэга `<SELECT>` варианты выбора более компактно представляются в окне браузера в виде элементов ниспадающего меню или списка прокрутки. Тэг имеет следующие атрибуты.

☐ `NAME=строка`

Обязательный атрибут. При выборе одного или нескольких элементов формируется список выбранных значений, который передается на сервер под именем NAME.

☐ `SIZE=n`

Устанавливает число одновременно видимых элементов выбора. Если $n=1$, то отображается ниспадающее меню, если $n>1$, то список прокрутки с n одновременно видимыми элементами.

☐ `MULTIPLE`

Означает, что из меню или списка можно выбрать одновременно несколько элементов. Если этот атрибут задан, то список выбора ведет себя как группа переключателей CHECKBOX, если не задан — как группа радиокнопок RADIO.

Элементы меню задаются внутри тэга `<SELECT>` при помощи тэга `<OPTION>`:

```
<OPTION SELECTED VALUE=строка>содержимое_тэга</OPTION>
```

Закрывающий тэг `</OPTION>` не используется. Атрибут VALUE содержит значение, которое пересылается серверу, если данный элемент выбран из меню или списка. Если значение этого атрибута не задано, то по умолчанию оно устанавливается равным содержимому тэга `<OPTION>`. Например, элементы

```
<OPTION VALUE=Red>Red
```

```
<OPTION>Red
```

имеют одно значение Red. В первом случае оно установлено явно при помощи атрибута VALUE, во втором — по умолчанию. Атрибут SELECTED изначально отображает элемент как выбранный.

15.2.4. Тэг `<TEXTAREA>`

```
<TEXTAREA NAME=имя ROWS=m COLS=n>
```

```
текст
```

```
</TEXTAREA>
```

Создает внутри формы поле для ввода многострочного текста, отображаемое в окне браузера в виде прямоугольной области с горизонтальной и вертикальной полосами прокрутки. Для пересылки на сервер каждая введенная строка дополняется символами `%0D%0A` (ASCII-символы "Возврат каретки" и

"Перевод строки" с предшествующим символом %), полученные строки объединяются в одну строку, которая и отправляется на сервер под именем, задаваемым атрибутом NAME. Атрибуты:

☐ NAME

Необходимый атрибут, используемый для идентификации данных при пересылке на сервер.

☐ COLS=n

Задаёт число столбцов видимого текста.

☐ ROWS=n

Задаёт число строк видимого текста.

Между тэгами <textarea> и </textarea> можно поместить текст, который будет отображаться по умолчанию.

15.2.5. Пример формы

Ниже представлен пример формы, включающей набор характерных полей (рис. 15.2), и HTML-код, использованный для ее создания.

The screenshot shows a Netscape browser window with the title "Регистрационная страница Клуба любителей фантастики". The address bar shows "http://www.klf.ru/welcome.html". The form contains the following elements:

- Text input: "Введите регистрационное имя:" with the value "bob".
- Text input: "Введите пароль:" with masked characters "*****".
- Text input: "Подтвердите пароль:" with masked characters "*****".
- Text: "Ваш возраст:" followed by radio buttons for "До 20", "20-30", "30-50", and "старше 50".
- Text: "На каких языках читаете:" followed by radio buttons for "русский", "английский", "французский", and "немецкий".
- Text: "Какой формат данных является для Вас предпочтительным" followed by radio buttons for "HTML" (selected) and "Plain text".
- Text input: "Ваши любимые авторы:" with the value "Желязны".
- Buttons: "OK" and "Отменить".

Рис. 15.2. Пример заполнения формы

Пример 15.1. HTML-код формы, представленной на рис. 15.2

```

<html><head><title>Пример формы</title></head>
<body>
<h2>Регистрационная страница Клуба любителей фантастики</h2>
Заполнив анкету, вы сможете пользоваться нашей электронной библиотекой.
<br>
<form method="get" action="/cgi-bin/registrar.cgi">
<pre>
Введите регистрационное имя: <input type="text" name="regname">
Введите пароль:           <input type="password" name="password1" max-
length=8>
Подтвердите пароль:      <input type="password" name="password2" max-
length=8>
</pre>
Ваш возраст:
<input type="radio" name="age" value="lt20" checked >До 20
<input type="radio" name="age" value="20_30">20-30
<input type="radio" name="age" value="30_50">30-50
<input type="radio" name="age" value="gt50">старше 50
<br><br>
На каких языках читаете:
<input type="checkbox" name="language" value="russian" checked>русский
<input type="checkbox" name="language" value="english">английский
<input type="checkbox" name="language" value="french">французский
<input type="checkbox" name="language" value="german">немецкий
<br><br>
Какой формат данных является для Вас предпочтительным
<br>
<select name="format" size=2 >
    <option selected value="HTML">HTML
    <option value="Plain text">Plain text
    <option value="PostScript">PostScript
    <option value="PDF">PDF
</select>
<br><br>
Ваши любимые авторы:
<br>
<textarea name="wish" cols=40 rows=3>

```

```
</textarea>  
<br><br>  
<input type="submit" value="OK"> <input type="reset" value="Отменить">  
</form>  
</body>  
</html>
```

Данная форма содержит:

- ☐ текстовое поле для ввода регистрационного имени пользователя;
- ☐ текстовое поле для ввода пароля, отображаемого в окне символами *;
- ☐ текстовое поле для подтверждения пароля, также отображаемого символами *;
- ☐ группу радиокнопок для указания возраста пользователя (единственный выбор);
- ☐ группу переключателей для указания языков, которыми владеет пользователь (множественный выбор);
- ☐ список прокрутки для указания предпочтительного формата данных (выбор из ограниченного списка);
- ☐ блок ввода многострочного текста для перечисления любимых авторов (неизвестное заранее количество строк);
- ☐ кнопку передачи с меткой **OK** (у этого элемента отсутствует атрибут NAME, он не нужен, так как в данном примере всего одна кнопка передачи, а, значит, CGI-программе нет необходимости определять, от какой именно кнопки поступила команда передачи данных);
- ☐ кнопку сброса с меткой **Отменить**.

Итак, пользователь заполнил форму и щелкнул кнопку передачи **Submit**. Дальнейшее прохождение данных выглядит следующим образом.

1. Информация кодируется и пересылается на Web-сервер, который передает ее для обработки CGI-программе.
2. CGI-программа обрабатывает полученные данные, возможно, обращаясь за помощью к другим программам, выполняющимся на том же компьютере, и генерирует новый "виртуальный" HTML-документ, либо определяет ссылку на уже имеющийся.
3. Новый HTML-документ или ссылка передаются CGI-программой Web-серверу для возврата клиенту.

Рассмотрим эти шаги более подробно.

15.3. Передача информации CGI-программе

Как мы уже знаем, существуют два метода кодирования информации, содержащейся в форме: стандартный метод *application/x-www-form-urlencoded*, используемый по умолчанию, и дополнительный *multipart/form-data*. Второй метод нужен только в том случае, если к содержимому формы присоединяется локальный файл, выбранный при помощи элемента формы `<INPUT TYPE=FILE>`. В остальных случаях следует использовать метод кодирования по умолчанию.

Схема кодирования *application/x-www-form-urlencoded* одинакова для обоих методов пересылки GET и POST и заключается в следующем.

Для каждого элемента формы, имеющего имя, заданное атрибутом NAME, формируется пара *"name=value"*, где *value* — значение элемента, введенное пользователем или назначенное по умолчанию. Если значение отсутствует, соответствующая пара имеет вид *"name="*. Для радиокнопок и переключателей используются значения только выбранных элементов. Если элемент выбран, а значение атрибута VALUE не определено, по умолчанию используется значение "ON".

Все пары объединяются в строку, в качестве разделителя служит символ &. Так как имена и значения представляют собой обычный текст, то они могут содержать символы, недопустимые в составе URL (метод GET пересылает данные как часть URL). Такие символы заменяются последовательностью, состоящей из символа % и их шестнадцатеричного ASCII-кода. Символ пробела может заменяться не только кодом %20, но и знаком + (плюс). Признак конца строки, встречающийся в поле TEXTAREA, заменяется кодом %0D%0A. Такое кодирование называется URL-кодированием.

Закодированная информация пересылается серверу одним из методов GET или POST. Основное отличие заключается в том, как метод передает информацию CGI-программе.

При использовании метода GET данные формы пересылаются серверу в составе URL запроса, к которому добавляются после символа ? (вспомним, что запрос — это формализованный способ обращения браузера к Web-серверу). Тело запроса в этом случае является пустым. Для формы из примера 15.1 запрос выглядит следующим образом:

```
GET /cgi-bin/registrar.cgi? regname=bob&password1=
rumata&password2=rumata&age=1t20&language=russian&format=
HTML&wish=%F6%C5%CC%D1%DA%CE%D9 HTTP/1.0
```

(заголовки запроса, сообщающие серверу информацию о клиенте)

<пусто> (тело запроса)

Часть URL после символа "?" называется *строкой запроса*. Web-сервер, получив запрос, присвоит переменной среды QUERY_STRING значение строки запроса и вызовет CGI-программу, обозначенную в первой части URL до символа "?": /cgi-bin/registrar.cgi. CGI-программа registrar.cgi сможет затем обратиться к переменной QUERY_STRING для обработки закодированных в ней данных.

Внимание

Обратите внимание на то, что данные, введенные в поле типа PASSWORD, передаются открытым текстом без шифрования. При передаче данных методом GET они в составе URL помещаются в файл регистрации доступа access.log, обычно открытый для чтения всем пользователям. Таким образом "секретные" данные, введенные в поле типа PASSWORD, оказываются доступными посторонним.

Замечание

Метод GET позволяет передавать данные CGI-программе вообще без использования форм. Информацию, содержащуюся в приведенном выше URL, можно передать при помощи следующей гиперссылки, помещенной в HTML-документ:

```
<A HREF="http://www.domain/cgi-bin/registrar.cgi?regname=
&bob&password1=rumata&password2=rumata&age=1t20&language=russian&
format=HTML&wish=%F6%C5%CC%D1%DA%CE%D9 ">CGI-программа</A>
```

заменяв в этом фрагменте символ "&" его символьным примитивом `&` или `&` для правильной интерпретации браузером.

К сожалению, эта информация является статической. Форма же позволяет менять данные.

Строка запроса — не единственный способ передачи данных через URL. Другим способом является *дополнительная информация о пути* (*extra path information*), представляющая собой часть URL, расположенную после имени CGI-программы. Сервер выделяет эту часть и сохраняет ее в переменной среды PATH_INFO. CGI-программа может затем использовать эту переменную для извлечения данных. Например, URL

```
http://www.domain/cgi-bin/registrar.cgi/
&regname=bob&password1=rumata&password2=rumata&age=1t20&language=
&russian&format=HTML&wish=%F6%C5%CC%D1%DA%CE%D9
```

содержит уже знакомые нам данные, но не в виде строки запроса, а в виде *дополнительной информации о пути*. При получении запроса с таким URL сервер сохранит данные в переменной среды

```
PATH_INFO= /regname=bob&password1=rumata&password2=rumata&age=
&1t20&language=russian&format=HTML&wish=%F6%C5%CC%D1%DA%CE%D9"
```

Название объясняется тем, что обычно этим способом передается информация о местоположении какого-либо файла (*extra path information*). Например, URL

```
http://www.domain/cgi-bin/registrar.cgi/texts/jdk_doc.txt
```

содержит дополнительную информацию `PATH_INFO=/texts/jdk_doc.txt` о местонахождении файла `jdk_doc.txt` относительно корневого каталога дерева документов. Другая переменная среды `PATH_TRANSLATED` содержит информацию об абсолютном местоположении файла в файловой системе, например,

```
PATH_TRANSLATED="/home/httpd/docs/texts/jdk_doc.txt"
```

а переменная `DOCUMENT_ROOT` содержит путь к корневому каталогу дерева документов, в нашем случае `DOCUMENT_ROOT="/home/httpd/docs/"`.

При использовании метода `POST` данные формы пересылаются серверу в теле запроса. Если в примере 15.1 вместо метода `GET` использовать метод `POST`

```
<form method="post" action="/cgi-bin/registrar.cgi">
```

то запрос клиента будет иметь следующий вид:

```
POST /cgi-bin/registrar.cgi HTTP/1.1
```

```
.
```

```
    (заголовки запроса, сообщающие серверу информацию о клиенте)
```

```
.
```

```
Content-length: 126
```

```
regname=bob&password1=rumata&password2=rumata&age=1t20&language=russian&  
ϕformat=HTML&wish=%F6%C5%CC%D1%DA%CE%D9
```

В этом фрагменте среди прочих заголовков выделен заголовок `Content-length`, сообщающий серверу количество байт, переданных в теле запроса. Это значение сервер присваивает переменной среды `CONTENT_LENGTH`, а данные посылает в стандартный ввод CGI-программы.

Методы `GET` и `POST` имеют свои достоинства и недостатки. Метод `GET` обеспечивает лучшую производительность при пересылке форм, состоящих из небольшого набора коротких полей. При пересылке большого объема данных следует использовать метод `POST`, так как браузер или сервер могут накладывать ограничения на размер данных, передаваемых в составе URL, и отбрасывать часть данных, выходящую за границу. Метод `POST`, к тому же, является более надежным при пересылке конфиденциальной информации.

15.4. CGI-сценарии

Назначение CGI-программы — создать новый HTML-документ, используя данные, содержащиеся в запросе, и передать его обратно клиенту. Если такой документ уже существует, то передать ссылку на него. Какой язык мож-

но использовать для написания CGI-программ? Сам интерфейс CGI не накладывает ограничений на выбор языка программирования. Зная, какую задачу решает CGI-программа и каким образом она получает входную информацию, мы можем назвать свойства, которыми должен обладать язык CGI-программирования.

- ☐ Средства обработки текста. Необходимы для декодирования входной информации, поступающей в виде строки, состоящей из отдельных полей, разделенных символами-ограничителями.
- ☐ Средства доступа к переменным среды. Необходимы, так как с помощью переменных среды данные передаются на вход CGI-программы.
- ☐ Возможность взаимодействовать с другими программами. Необходима для обращения к СУБД, программам обработки графики и другим специальным программам.

Выбор языка зависит и от операционной системы Web-сервера. Большая часть имеющихся серверов предназначена для работы под управлением операционной системы UNIX. Учитывая эти соображения, мы можем заключить, что язык Perl, обладающий развитыми средствами обработки текста и создания сценариев, первоначально созданный для работы в ОС UNIX и перенесенный на множество других платформ, является наиболее подходящим средством создания сценариев CGI. Кроме того, CGI-программирование на языке Perl имеет поддержку в виде готовых модулей CPAN, свободно доступных в сети Internet.

CGI-сценарий на языке Perl — это программа, имеющая свою специфику. Она, как правило, генерирует HTML-документ, посылаемый клиенту в виде *ответа сервера*. Ответ сервера, так же, как и запрос клиента, имеет определенную структуру. Он состоит из следующих трех частей:

1. Строка состояния, содержащая три поля: номер версии протокола HTTP, код состояния и краткое описание состояния, например:

```
HTTP/1.0 200 OK      # запрос клиента обработан успешно
HTTP/1.0 404 Not Found  # Документ по указанному адресу
                        не существует
```

2. Заголовки ответа, содержащие информацию о сервере и о возвращаемом HTML-документе, например:

```
Date: Mon, 26 Jul 1999 18:37:07 GMT # Текущая дата и время
Server: Apache/1.3.6                # Имя и номер версии сервера
Content-type: text/html             # Описывает медиа-тип содержимого
```

3. Содержимое ответа — HTML-документ, являющийся результатом выполнения CGI-программы.

CGI-программа передает результат своей работы (HTML-документ) серверу, который возвращает его клиенту. При этом сервер не анализирует и не изменяет полученные данные, он может только дополнять их некоторыми за-

головками, содержащими общую информацию (например, текущая дата и время) и информацию о самом себе (например, имя и версия сервера). Информация о содержимом ответа формируется CGI-программой и должна содержать как минимум один заголовок, сообщающий браузеру формат возвращаемых данных:

```
Content-type: text/html
```

Замечание

Информацию о заголовках можно найти в спецификации протокола HTTP. Мы же ограничимся еще одним примером. Если в качестве ответа клиенту посылается статический документ, например, подтверждение о получении заполненной формы, то неэффективно каждый раз создавать его заново. Лучше создать один раз и сохранить в файле. В этом случае CGI-сценарий вместо заголовка `Content-type: media-type`, описывающего формат данных, формирует заголовок `Location: URL`, указывающий серверу местонахождение документа, который следует передать клиенту.

Заголовки отделяются от содержимого документа пустой строкой.

Напишем простейший CGI-сценарий, посылающий пользователю HTML-страницу с приветствием

Пример 15.2. CGI-сценарий `hello.cgi`

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
print "<html><head><title>HELLO</title></head>\n";
print "<body>\n";
print "<h2>Вас приветствует издательство БХВ — Санкт-Петербург</h2>\n";
print "</body></html>\n";
```

Если поместить файл `hello.cgi` в каталог CGI-программ Web-сервера, а затем обратиться к нему из браузера, то браузер отобразит HTML-документ, созданный программой `hello.cgi` (рис. 15.3).

Замечание

Большинство Web-серверов по умолчанию предполагают, что файлы CGI-сценариев находятся в специальном каталоге, обычно называемом `cgi-bin`. Можно настроить сервер таким образом, чтобы все файлы, находящиеся в определенном каталоге, он воспринимал не как обычные документы, а как выполняемые сценарии. Можно также указать серверу, что все файлы с определенным расширением (например, `.cgi`) должны рассматриваться как CGI-сценарии. Когда пользователь открывает URL, ассоциированный с CGI-программой, клиент посылает запрос серверу, запрашивая файл. Сервер распознает, что запрошенный адрес является адресом CGI-программы, и пытается

выполнить эту программу. Подробности конфигурирования Web-серверов можно найти в соответствующей литературе и документации на конкретный сервер.

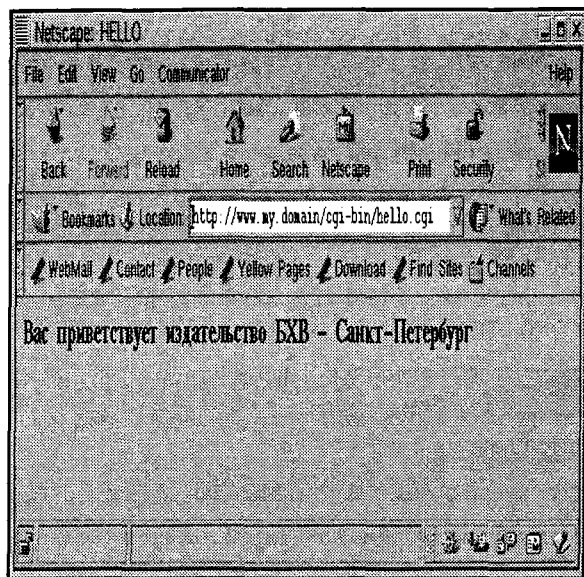


Рис. 15.3. Web-страница, сформированная программой hello.cgi

15.4.1. Переменные среды CGI

В зависимости от метода данные формы передаются в CGI-программу или через стандартный ввод (POST), или через переменную среды QUERY_STRING (GET). Помимо этих данных CGI-программе доступна и другая информация, поступившая от клиента в заголовках запроса или предоставленная Web-сервером. Эта информация сохраняется в переменных среды UNIX. С некоторыми из них мы уже познакомились ранее. В табл. 15.1 перечислены переменные, обычно используемые в CGI.

Таблица 15.1. Переменные среды CGI

Переменная среды	Описание
GATEWAY_INTERFACE	Версия CGI, которую использует сервер
SERVER_NAME	Доменное имя или IP-адрес сервера
SERVER_SOFTWARE	Имя и версия программы-сервера, отвечающей на запрос клиента (например, Apache 1.3)
SERVER_PROTOCOL	Имя и версия информационного протокола, который был использован для запроса (например, HTTP 1.0)

Таблица 15.1 (окончание)

Переменная среды	Описание
SERVER_PORT	Номер порта компьютера, на котором работает сервер (по умолчанию 80)
REQUEST_METHOD	Метод, использованный для выдачи запроса (GET, POST)
PATH_INFO	Дополнительная информация о пути
PATH_TRANSLATED	Та же информация, что и в переменной PATH_INFO с префиксом, задающим путь к корневому каталогу дерева Web-документов
SCRIPT_NAME	Относительное маршрутное имя CGI-сценария (например, /cgi-bin/program.pl)
DOCUMENT_ROOT	Корневой каталог дерева Web-документов
QUERY_STRING	Строка запроса — информация, переданная в составе URL запроса после символа "?"
REMOTE_HOST	Имя удаленной машины, с которой сделан запрос
REMOTE_ADDR	IP-адрес удаленной машины, с которой сделан запрос
REMOTE_USER	Идентификационное имя пользователя, посылающего запрос
CONTENT_TYPE	Медиа-тип данных запроса, например, "text/html".
CONTENT_LENGTH	Количество байт в теле запроса, переданных в CGI-программу через стандартный ввод
HTTP_HOST	Хост-имя компьютера, на котором работает сервер
HTTP_FROM	Адрес электронной почты пользователя, направившего запрос
HTTP_ACCEPT	Список медиа-типов, которые может принимать клиент
HTTP_USER_AGENT	Браузер, которым клиент пользуется для выдачи запроса
HTTP_REFERER	URL документа, на который клиент указывал перед обращением к CGI-программе

Внимание

Имена переменных среды CGI на разных Web-серверах могут различаться. Следует обратиться к документации на соответствующий сервер.

CGI-программа на языке Perl имеет доступ к переменным среды через специальный предопределенный хеш-массив %ENV, к элементам которого можно обратиться по ключу, совпадающему с именем переменной среды. Ниже

приведены пример CGI-сценария, формирующего HTML-документ с информацией о всех установленных переменных среды, и отображение этого документа в окне браузера.

Пример 15.3. Получение информации о переменных среды CGI

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
print "<html>\n";
print "<head><title>Переменные среды</title></head>\n";
print "<body><h2>Переменные среды</h2>\n";
print "<hr><pre>\n";
foreach $name (sort(keys %ENV))
{
    print "$name:    $ENV{$name}\n";
}
print "<hr></pre>\n";
print "</body></html>\n";
```

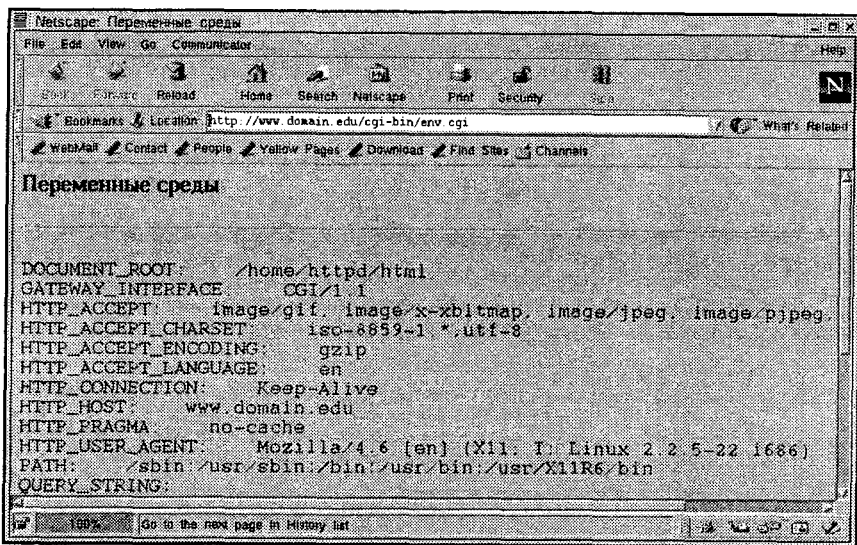


Рис. 15.4. Информация о переменных среды CGI

15.4.2. Обработка данных формы

Данные формы поступают в CGI-программу в закодированном виде, поэтому в качестве первого шага CGI-сценарий должен выполнить декодирование

ние полученной информации. При пересылке данных методом GET данные формы присваиваются переменной среды QUERY_STRING, при передаче методом POST — передаются в программу через стандартный ввод и тоже могут быть присвоены некоторой внутренней переменной. Таким образом, декодирование данных сводится к следующей последовательности манипуляций со строкой:

- ☐ замена каждой группы %hh, состоящей из шестнадцатеричного ASCII-кода hh с префиксом %, на соответствующий ASCII-символ;
- ☐ замена символов "+" пробелами;
- ☐ выделение отдельных пар *имя=значение*, разделенных ограничителем &;
- ☐ выделение из каждой пары *имя=значение* имени и значения соответствующего поля формы.

Программа декодирования HTML-формы может выглядеть, например, так:

Пример 15.4. Декодирование HTML-формы

```
#!/usr/bin/perl
# Декодирование данных формы, переданных методом GET
$form_data = $ENV{'QUERY_STRING'};
# преобразование цепочек %hh в соответствующие символы
$form_data =~ s/%(..)/pack("C", hex($1))/eg;
# преобразование плюсов в пробелы
$form_data =~ tr/+//;
# разбиение на пары имя=значение
@pairs = split (/&/, $form_data);
# выделение из каждой пары имени и значения поля формы и сохранение
# их в ассоциативном массиве $form_fields
foreach $pair (@pairs)
{
    ($name, $value)=split(=/,$pair);
    $form_fields{$name}=$value;
}
```

Если данные формы переданы методом POST, то в приведенном тексте следует заменить оператор присваивания

```
$form_data = $ENV{'QUERY_STRING'};
```

оператором

```
read(STDIN,$form_data,$ENV{'CONTENT_LENGTH'});
```

считывающим из стандартного ввода программы `CONTENT_LENGTH` байтов, составляющих содержимое запроса клиента, в переменную `$form_data`.

В приведенном примере используются две новые функции: `pack()` и `hex()`. Поясним их назначение прежде, чем перейти к обсуждению текста программы.

Функция

`pack template, list`

упаковывает список значений `list` в двоичную структуру по заданному шаблону `template`. Аргумент `template` представляет собой последовательность символов, определяющих формат представления пакуемых данных:

a/A	Текстовая строка, заполненная нулями/пробелами
b/B	Двоичная строка, значения расположены в порядке возрастания/убывания
c/C	Обычное символьное значение/ Символьное значение без знака
f/d	Значение в формате с плавающей точкой одинарной/двойной точности
h/H	Шестнадцатеричная строка, младший/старший полубайт первый
i/I	Целое со знаком/ без знака
l/L	Значение типа <code>long</code> со знаком/без знака
n/N	Значение типа <code>short/long</code> с "сетевым" порядком байтов ("старший в старшем")
p/u	Указатель на строку/Uu-кодированная строка
s/S	Значение типа <code>short</code> со знаком/без знака
v/V	Значение типа <code>short/long</code> с VAX-порядком байтов ("старший в младшем")
x/X	Нулевой байт/резервная копия байта
@	Заполнение нулевыми байтами (до абсолютной позиции)

За каждым символом может следовать число, обозначающее счетчик применений данного символа в качестве формата. Символ `*` в качестве счетчика означает применение данного формата для оставшейся части списка.

Пример 15.5. Функция `pack()`

```
$x = pack "cccc", 80, 101, 114, 108;
$x = pack "c4", 80, 101, 114, 108;
```

```
$x = pack "B32", "01010000011001010111001001101100";
$x = pack "H8", "5065726C";
$x = pack "H*", "5065726C";
$x = pack "cB8H2c", 80, "01100101", 72, 108;
```

Значение переменной `$x` во всех случаях равно `"Perl"`.

Функция

`hex expr`

Интерпретирует аргумент `expr` как шестнадцатеричную строку и возвращает ее десятичное значение.

В тексте программы примера 15.4 все представляется очевидным. Разберем только наиболее насыщенную строку

```
$form_data =~ s/%(..)/pack ("C", hex ($1))/eg;
```

Образец для поиска задан в виде регулярного выражения `%(..)`. Этому образцу удовлетворяет произвольная последовательность вида `%xy`, где `x`, `y` — любые символы. В результате кодирования данных в качестве `x`, `y` могут появиться только шестнадцатеричные цифры, поэтому можно не задавать более точный, но менее компактный шаблон `%([0-9A-Fa-f][0-9A-Fa-f])`. Часть выражения заключена в скобки `(..)`. При нахождении подходящего фрагмента `%hh` его часть, содержащая шестнадцатеричное число `hh`, сохраняется в переменной, которая затем будет использована в качестве аргумента функции `hex($1)` для преобразования в десятичное значение. Функция `pack` упакует это десятичное значение в двоичную структуру, которая в соответствии с шаблоном `"c"` будет интерпретироваться как символ. Этот символ заменяет в тексте найденную цепочку `%hh`.

После выделения и декодирования данных можно приступить к их обработке. Попробуем написать CGI-сценарий, обрабатывающий данные формы из примера 15.1.

15.4.3. Пример создания собственного CGI-сценария

Программа должна декодировать полученные данные, проверять заполнение обязательных полей формы и правильность подтверждения пароля, в зависимости от результатов проверки формировать документ для отсылки клиенту. Сохраним сценарий в файле `/cgi-bin/regarstrar.cgi`. Полный маршрут к данному файлу определяется параметрами конфигурации Web-сервера. Местоположение каталога `cgi-bin` обычно указывается относительно корня дерева документов Web-сервера, а не корневого каталога файловой системы. Например, если корнем является каталог `/home/httpd/html/`, то файл сцена-

рия будет иметь маршрутное имя /home/httpd/html/cgi-bin/regarstrar.cgi, которое в запросе клиента будет указано как /cgi-bin/regarstrar.cgi. В первом приближении текст сценария может выглядеть следующим образом.

Пример 15.6. Первый вариант сценария

```
#!/usr/bin/perl
print "Content-type:text/html\n\n";
$method = $ENV{'REQUEST_METHOD'};
if ($method eq "GET") {
    $form_data = $ENV{'QUERY_STRING'};
}
else {
    read (STDIN, $form_data, $ENV{'CONTENT_LENGTH'});
}
$form_data =~ s/%(..)/pack ("C", hex ($1))/eg;
$form_data =~ tr/+//;
@pairs = split (/&/, $form_data);
foreach $pair (@pairs) {
    ($name, $value)=split(/=/, $pair);
    $FORM{$name}=$value;
}
#Проверка заполнения обязательных полей
if (!$FORM{'regname'} || !$FORM{'password1'}) {
print<<goback
    <html>
    <head><title>Неполные данные</title></head>
    <body><h2>Извините, Вы пропустили обязательные данные</h2>
    <br>
    <a href="http://www.klf.ru/welcome.html">Попробуйте еще раз,
    пожалуйста</a>
    </body>
    </html>
goback
; }
#Проверка правильности ввода пароля
elsif ($FORM{'password1'} eq $FORM{'password2'}) {
print<<confirmation
<html>
```

```

<head><title>Поздравляем!</title></head>
<body><h2>Поздравляем!</h2><br>
Ваша регистрация прошла успешно. Вы можете пользоваться нашей
библиотекой. Спасибо за внимание.
</body>
</html>
confirmation
;}
else {
print<<new_form
    <html><head><title>Ошибка при вводе пароля</title></head>
    <body><h3>Введенные Вами значения пароля не совпадают
    <br><form method="get" action="/cgi-bin/registrar.cgi">
    <pre>
Введите пароль:      <input type="password" name="password1">
Подтвердите пароль: <input type="password" name="password2">
    </pre>
new_form
;
foreach $key ( keys %FORM) {
    if ($key ne "password1" && $key ne "password2") {
        print "<input type=\"hidden\" name=$key value=$FORM{$key}>\n";
    }
}
print<<EndOfHTML
    <br><br>
    <input type="submit" value="OK"> <input type="reset"
value="Отменить">
    </form></body></html>
EndOfHTML
;}
```

После вывода строки заголовка осуществляется считывание переданной серверу информации в переменную `$form_data`. В зависимости от метода передачи, эта информация считывается из переменной среды `QUERY_STRING` (метод GET) или из стандартного ввода программы (метод POST).

Считанная информация декодируется и помещается в ассоциативный массив `%FORM`.

Отсутствие обязательных данных — регистрационного имени и пароля — проверяется с помощью условия `if (!$FORM{'regname'} || !$FORM{'password1'})`.

В случае отсутствия необходимых данных формируется виртуальный HTML-документ, предлагающий повторить попытку, который и посылается клиенту (рис. 15.5).

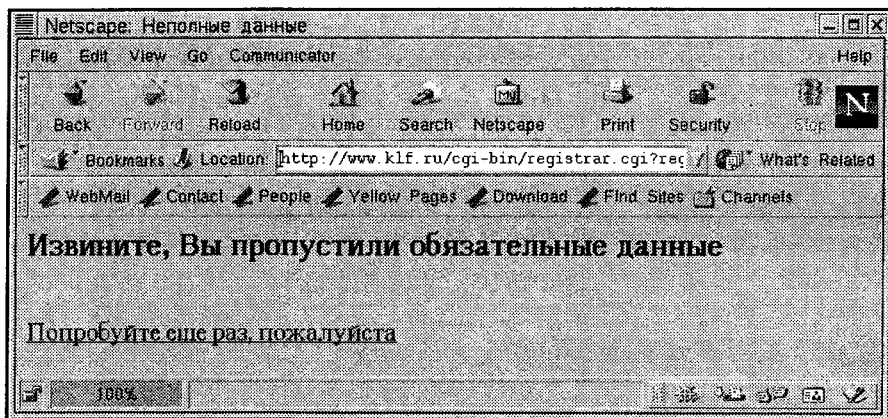


Рис. 15.5. Ответ сервера в случае отсутствия обязательной информации

При выводе этого документа в операции `print` использована конструкция "документ здесь". Она позволяет использовать внутри себя символы, которые при заключении в обычные двойные кавычки необходимо маскировать символом `"\"`, например, сами двойные кавычки `"`, символы `"@"`, `"$"`, `"%"`.

Условие `elsif ($FORM{'password1'} eq $FORM{'password2'})` предназначено для проверки совпадения двух копий введенного пользователем пароля. Если значения совпадают, то пользователю посылается сообщение, подтверждающее успешную регистрацию (рис. 15.6).

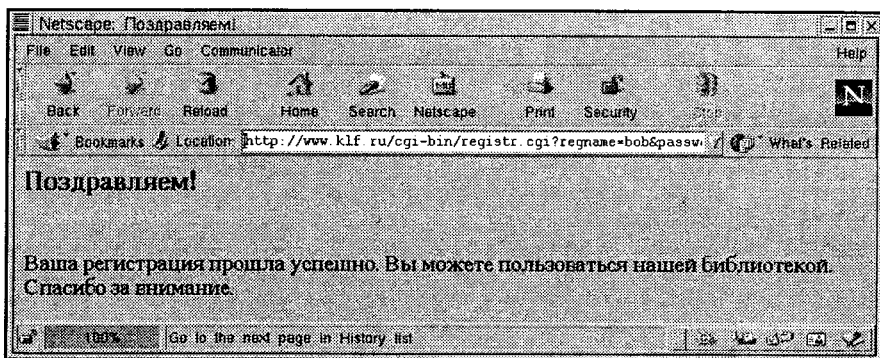


Рис. 15.6. Подтверждение регистрации

В противном случае формируется HTML-документ, предлагающий ввести пароль повторно (рис. 15.7). Этот новый документ содержит форму, в состав

которой входят два видимых поля типа "password" — для ввода и подтверждения пароля, и скрытые поля типа "hidden" — для сохранения остальных данных, введенных при заполнении исходной формы. Каждое скрытое поле новой формы наследует у соответствующего поля исходной формы атрибуты name и value. Если эти данные не сохранить, то их придется вводить заново, принуждая пользователя повторно выполнять уже сделанную работу. Информация, сохраненная в скрытых полях, невидима пользователю и недоступна для изменения.

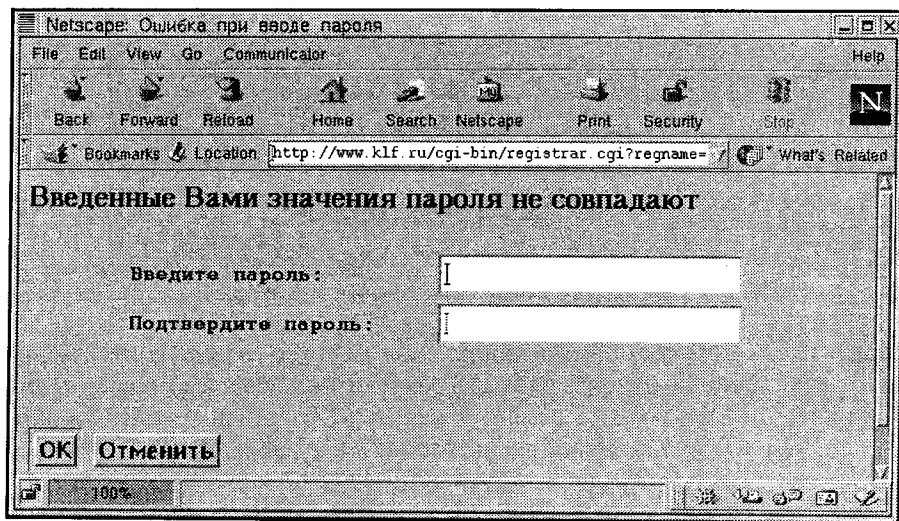


Рис. 15.7. Повторное приглашение для ввода пароля

Культура Perl допускает различные уровни владения языком. В рассмотренном варианте использован минимальный набор средств. Очевидно, что часть кода, например, декодирование, требуется при обработке не только данной, но и любой другой формы. Естественным шагом в развитии исходного варианта сценария является выделение этой части в отдельную подпрограмму и предоставление доступа к ней другим сценариям. Для этого преобразуем исходный текст в соответствии с планом, изложенным в примере 15.7.

Пример 15.7. Структурированный вариант сценария

1. Часть исходного кода может быть использована другими CGI-программами. Преобразуем ее в отдельный модуль, сохраняемый в файле CGI_UTILS.pm.

```
package CGI_UTILS;  
require Exporter;
```

```

@ISA = qw(Exporter);
@EXPORT = qw(print_header process_input);
# Подпрограмма вывода заголовка ответа
sub print_header {
    print "Content-type: text/html\n\n";
}
# Подпрограмма декодирования данных формы
sub process_input {
    my ($form_ref)=@_;
    my ($form_data,@pairs);
    my ($temp)="";
    if ($ENV{'REQUEST_METHOD'} eq 'POST') {
        read(STDIN,$form_data,$ENV{'CONTENT_LENGTH'});
    }
    else {
        $form_data=$ENV{'QUERY_STRING'};
    }
    $form_data=~s/%(..)/pack("c",hex($1))/ge;
    $form_data=~tr/+// ;
    $form_data=~s/\\n/\\0/g;
    @pairs=split(/&/,$form_data);
    foreach $item(@pairs) {
        ($name,$value)=split (/=/,$item);
        if (!defined($form_ref->{$name})) {
            $form_ref->{$name}=$value;
        }
        else {
            $form_ref->{$name} .= "\0$value";
        }
    }
    foreach $item (sort keys %$form_ref) {
        $temp.=$item."=".$form_ref->{$item}."&";
    }
    return($temp);
}
1;

```

2. Текст основного сценария обработки формы registrar.cgi преобразуем следующим образом:

```

#!/usr/bin/perl
use cgi_utils;

```

```

my %FORM, $file_rec;

$file_rec=&process_input(\%FORM);

#Проверка заполнения обязательных полей
#if ($FORM{'regname'} eq "" || $FORM{'password1'} eq "") {
if (!$FORM{'regname'} || !$FORM{'password1'}) {
    print "Location: /goback.html\n\n";
}
}

#Проверка правильности ввода пароля
elseif ($FORM{'password1'} eq $FORM{'password2'}){
    print "Location: /confirmation.html\n\n";
    open (OUTF, ">>users");
    print OUTF $file_rec, "\n";
    close OUTF
}

else {
&print_header;
print<<new_form
    <html>
    <head><title>Ошибка при вводе пароля</title></head>
    <body><h3>Введенные Вами значения пароля не совпадают
    <br><form method="get" action="/cgi-bin/registrar.cgi">
    <pre>
Введите пароль:      <input type="password" name="password1">
Подтвердите пароль:  <input type="password" name="password2">
    </pre>

new_form
;
foreach $key ( keys %FORM) {
    if ($key ne "password1" && $key ne "password2") {
        print "<input type=\"hidden\" name=$key value=$FORM{$key}>\n";
    }
}

print<<EndOfHTML
    <br><br>
    <input type="submit" value="OK">
    <input type="reset" value="Отменить">
    </form>
    </body>

```

```
</html>
```

```
EndOfHTML
```

```
;
```

```
}
```

```
exit
```

3. В исходном варианте сценария в качестве ответов сервера при получении неполных данных и для подтверждения регистрации пользователя формируются виртуальные HTML-документы. В этом нет необходимости, так как они содержат только статическую информацию. Соответствующие фрагменты сценария преобразуем в HTML-код готовых документов, которые сохраним в отдельных файлах. В основном сценарии в качестве ответа сервера возвращаются ссылки на эти документы.

Файл `confirmation.html` содержит документ, посылаемый клиенту в качестве сообщения об успешной регистрации:

```
<html>
```

```
<head><title>Поздравляем!</title></head>
```

```
<body><h2>Поздравляем!</h2><br>
```

Ваша регистрация прошла успешно. Вы можете пользоваться нашей библиотекой.

```
<br>
```

Спасибо за внимание.

```
</body>
```

```
</html>
```

Файл `goback.html` содержит документ, посылаемый клиенту при получении неполных данных:

```
<html>
```

```
<head><title>Неполные данные</title></head>
```

```
<body><h2>Извините, Вы пропустили обязательные данные</h2>
```

```
<br>
```

```
<a href="http://www.klf.ru/welcome.html">Попробуйте еще раз,
```

```
☞ пожалуйста</a>
```

```
</body>
```

```
</html>
```

В приведенном тексте появились некоторые новые элементы, которые необходимо пояснить.

Подпрограмма `process_input` модуля `cgi_utils.pm` передает декодированные данные через вызываемый по ссылке параметр — ассоциативный массив. Кроме того, она возвращает при помощи функции `return()` те же данные,

но в виде строки, состоящей из пар имя=значение, разделенных символом "&". Обратите внимание на то, как подпрограмма вызывается в основной программе:

```
$file_rec=&process_input(\%FORM);
```

В качестве аргумента ей передается ссылка на ассоциативный массив. В тексте подпрограммы появилась проверка наличия полей формы с совпадающими именами и разными значениями:

```
if (!defined($form_ref->{$name})) {  
    $form_ref->{$name}=$value;  
}  
else {  
}}
```

Этот фрагмент необходим для того, чтобы правильно обработать следующую ситуацию из нашего примера. Выбраны несколько переключателей, определяющих языки, которыми владеет пользователь: русский, английский, французский. Так как соответствующие элементы формы имеют одинаковые имена `name=language`, то без проверки в ассоциативный массив `%form_ref`, куда помещаются обработанные данные, попадет только информация от последнего обработанного элемента `name=language value=french`. В подобном случае обычное присваивание заменяется операцией присваивания с конкатенацией

```
$form_ref->{$name} .= "\0$value",
```

которая к переменной `$form_ref->{$name}` добавляет нулевой символ и значение `$value`.

В основной программе `registrar.cgi` обратим внимание на то, как передается ссылка на готовый HTML-документ. Для этого вместо заголовка `Content-type:text/html` выводится заголовок `Location: URL`, сообщающий серверу адрес документа.

Еще один новый элемент в основной программе — сохранение данных в файле с именем `users`.

15.4.4. Модуль CGI.pm

Пример, рассмотренный выше, демонстрирует наивный подход, когда кажется, что все необходимые программы надо писать самостоятельно с самого начала. Но программирование CGI — это такая область, в которой Perl давно и активно применяется, и многое из того, что может потребоваться, уже давно кем-то написано. Надо только найти и использовать. В данном разделе мы сделаем краткий обзор одного из таких готовых средств, предназначенных для поддержки разработки CGI-приложений.

Модуль CGI.pm, созданный Линкольном Штейном, входит в состав дистрибутивного комплекта Perl, начиная с версии 5.004, и его даже не нужно специально устанавливать.

Этот модуль содержит большой набор функций для создания и обработки HTML-форм. Мы посвятили значительную часть предыдущего раздела изучению многочисленных тэгов, чтобы затем написать HTML-код для создания формы в примере 15.1. Модуль CGI позволяет сделать то же самое, но без использования HTML. С его помощью можно описать форму на языке Perl, используя вместо тэгов обращения к функциям модуля. В результате получится не документ HTML, а сценарий на языке Perl, который при вызове будет "на лету" генерировать HTML-форму и передавать серверу для отправки клиенту.

Модуль CGI является не просто модулем, а классом, что позволяет использовать преимущества объектно-ориентированного подхода. Модуль предоставляет пользователю на выбор два вида интерфейса с самим собой: процедурно-ориентированный и объектно-ориентированный.

При использовании процедурно-ориентированного способа работы с модулем CGI функции модуля нужно явным образом импортировать в пространство имен вызывающей программы, а затем обращаться к ним как обычно. В этом случае в вызывающей программе должны быть строки, аналогичные следующим:

```
#!/usr/bin/perl
use CGI qw/:standard/;
print header(),
      start_html('Пример формы'),
      h1('Пример формы'),
      . . .
```

Директива `use` импортирует в пространство имен вызывающей программы некоторый стандартный набор функций. Помимо него, существуют другие наборы функций модуля CGI. Их можно импортировать, указав имя соответствующего набора в списке импорта директивы `use`. Имена всех наборов можно просмотреть в файле CGI.pm, где они содержатся в хеш-массиве `%EXPORT_TAGS`.

Функции `header()`, `start_html()`, `h1()` являются функциями модуля CGI. Они будут рассмотрены ниже.

При использовании объектно-ориентированного интерфейса в директиве `use` вызывающей программы не нужно указывать список импортируемых имен функций. В этом случае взаимодействие с модулем CGI осуществляется через объект класса CGI, который нужно создать в вызывающей программе при помощи конструктора `new()`. Объектно-ориентированный вариант приведенного выше фрагмента выглядит следующим образом:

```
#!/usr/bin/perl
use CGI;
$query = new CGI;
    print $query->header(),
        $query->start_html('Пример формы'),
        $query->h1('Пример формы'),
    . . .
```

Замечание

Функции модуля CGI.pm являются методами класса CGI. Для того чтобы их можно было вызывать и как функции, и как методы, синтаксис не требует в качестве обязательного первого параметра указывать объект класса CGI. Поэтому в качестве функций к ним можно обращаться обычным образом, а как к объектам — только используя форму `$object->method()`.

Модуль CGI, как мы отметили выше, содержит большой набор методов, и в наши планы не входит их подробное изучение. Документация, входящая в состав самого модуля, достаточно подробно описывает его компоненты. Чтобы получить представление о работе модуля CGI, создадим с его помощью небольшой сценарий. Для этого вернемся к рассмотрению формы из примера 15.1.

Будем для определенности использовать традиционный процедурно-ориентированный интерфейс. Рассмотрим следующий сценарий.

Пример 15.8. CGI-сценарий для создания HTML-формы

```
#!/usr/bin/perl
use CGI qw(:standard);
print header;
print start_html('Пример формы'),
    h2('Регистрационная страница Клуба любителей фантастики'),
    'Заполнив анкету, вы сможете пользоваться нашей электронной
    библиотекой.',
    br,
    start_form,
    "Введите регистрационное имя:", textfield('regname'),
    p,
    "Введите пароль: ", password_field(-name=>'password1',
        -maxlength=>'8'),
    p,
    "Подтвердите пароль: ", password_field(-name=>'password2',
        -maxlength=>'8'),
```



```

p,
"Ваш возраст",
p,
radio_group(-name=>'age',
            -value=>{'lt20','20_30','30_50','gt50'},
            -default=>'lt20',
            -labels=>{'lt20'=>'до 20','20_30'=>'20-30',
                      '30_50'=>'30-50','gt50'=>'старше 50'}),
br,br,
"На каких языках читаете:",
checkbox_group(-name=>'language',
            -values=>
['русский','английский','французский','немецкий'],
            -defaults=>['русский']),
br,br,
"Какой формат данных является для Вас предпочтительным ",
br,
popup_menu(-name=>'type',
            -values=>['Plain text','PostScript','PDF']),
br,br,
"Ваши любимые авторы:",
br,
textarea(-name=>'wish', -cols=>40, -rows=>3),
br,
submit(-name=>'OK'), reset(-name=>'Отменить'),
end_form,
hr;

if (param()) {
print
    "Ваше имя: ",em(param('regname')),
    p,
    "Ваш возраст: ", em(param('age')),
    p,
    "Вы читаете на языках: ",em(join(", ",param('language'))),
    p,
    "Предпочтительный формат данных для Вас: ",em(param('type')),
    p,

```

```

    "Ваши любимые авторы: ", em(join(", ", param('wish'))),
    hr;
}
print end_html;

```

Обсудим приведенный текст. Директива `use`, как мы отметили выше, осуществляет импорт стандартного набора функций модуля `CGI.pm` в пространство имен вызывающего пакета. В самом сценарии на месте тэгов исходного HTML-кода стоят обращения к функциям модуля: каждому тэгу соответствует вызов функции. Вызов функции модуля `CGI` можно осуществлять двумя способами:

с использованием позиционных параметров

```
print textfield('regname', 'начальное значение', 50, 80);
```

с использованием именованных параметров

```
print textfield(-name=>'regname',
               -default=>'начальное значение',
               -size=>50,
               -maxlength=>80);
```

Обработка позиционного параметра внутри функции зависит от его места в списке параметров. Обработка именованного параметра не зависит от его места в списке параметров. Функции модуля `CGI` могут иметь большое число параметров, порядок следования которых трудно запомнить, поэтому в этом модуле была реализована возможность вызова функций с именованными параметрами. Кроме того, применение именованных параметров делает текст программы более понятным. В тексте примера функции вызываются с именованными параметрами, если параметров больше одного.

Познакомимся с функциями, использованными в примере.

- ❑ Функция `header()` без параметров создает для виртуального ответа сервера стандартный HTTP-заголовок

```
Content-Type: text/html
```

и вставляет после него необходимую пустую строку. Параметры позволяют задать дополнительную информацию для заголовка, например, указать другой медиа-тип содержимого или код ответа, посылаемый браузеру:

```
print header(-type=>'image/gif',
            -status=>'404 Not Found');
```

- ❑ Функция `start_html()` создает HTML-заголовок и начальную часть документа, включая открывающий тэг `<BODY>`. При помощи параметров функции можно задать дополнительные тэги внутри тэга `<HEAD>`, а также значения атрибутов. Все параметры являются необязательными. В примере

функция `start_html()` вызвана с одним позиционным параметром, определяющим название документа.

Модуль CGI содержит методы (функции) для поддержки многих тэгов HTML2, HTML3, HTML4 и расширений, используемых в браузерах Netscape. Тэгам соответствуют одноименные методы модуля CGI.pm, имена которых записываются при помощи символов нижнего регистра. Если при этом возникают конфликты имен, в названия методов следует вводить символы верхнего регистра, как, например, в следующих случаях.

1. Название тэга `<TR>` совпадает с именем встроенной функции `tr()`. Имя соответствующего метода записывать в виде `TR()` или `Tr()`.
2. Название тэга `<PARAM>` совпадает с именем собственного метода модуля CGI `param()`. Для обозначения метода, соответствующего тэгу, использовать имя `PARAM()`.
3. Название тэга `<SELECT>` совпадает с именем встроенной функции `select()`. Для обозначения метода использовать имя `Select()`.
4. Название тэга `<SUB>` совпадает с именем ключевого слова объявления функции `sub`. Для обозначения метода использовать имя `Sub()`.

Тэгам, имеющим атрибуты, соответствуют методы, имеющие в качестве первого аргумента ссылку на анонимный хеш-массив. Ключами этого хеш-массива являются имена атрибутов тэга, а значениями — значения атрибутов.

Методы, соответствующие тэгам, и методы, предназначенные для генерирования других элементов HTML-документа, возвращают строки, содержащие соответствующие элементы. Чтобы эти строки попали в создаваемый документ, их нужно вывести, как это делается в примере при помощи функции `print`.

В примере использованы следующие методы, соответствующие тэгам HTML.

- ❑ Функция `h2` соответствует тэгу `<H2>`. Она определяет, что ее аргумент является в документе заголовком второго уровня.
- ❑ Функция `br` соответствует тэгу `
` и обозначает, что последующий текст размещается с начала новой строки.
- ❑ Функция `p` соответствует тэгу `<P>` и обозначает начало абзаца.
- ❑ Функция `hr` соответствует тэгу `<HR>` и обозначает горизонтальную линию, разделяющую документ на части.
- ❑ Функция `em` соответствует тэгу `` и обозначает, что ее аргумент в документе должен быть выделен курсивом.

Следующие функции используются для создания формы и ее элементов.

- ❑ Функция `start_form` соответствует тэгу `<FORM>`. Она может иметь три параметра

```
start_form(-method=>$method,
           -action=>$action,
           -encoding=>$encoding);
```

при помощи которых можно задать метод передачи формы Web-серверу (-method), программу, предназначенную для обработки формы (-action), и способ кодирования данных (-encoding). Все параметры являются необязательными. По умолчанию используются значения

```
method: POST;
action: данный сценарий;
encoding: application/x-www-form-urlencoded.
```

❑ Функция `end_form` создает закрывающий тэг `</FORM>`.

❑ Функция `textfield` соответствует тэгу `<INPUTE TYPE=TEXT>`. Она имеет следующий синтаксис

```
textfield(-name=>'field_name',
          -default=>'starting value',
          -size=>50,
          -maxlength=>80);
```

Параметры соответствуют атрибутам тэга. Обязательным является первый параметр.

❑ Функция `password_field` соответствует тэгу `<INPUTE TYPE=PASSWORD>`. Ее синтаксис:

```
password_field(-name=>'secret',
               -value=>'starting value',
               -size=>8,
               -maxlength=>12);
```

Параметры имеют тот же смысл, что и одноименные атрибуты соответствующего тэга. Обязательным является первый параметр.

❑ Функция `radio_group` служит для создания группы "радиокнопок" — элементов, задаваемых тэгом `<INPUTE TYPE=RADIO>`. Ее синтаксис имеет следующую форму

```
radio_group(-name=>'group_name',
            -values=>['bim', 'bam', 'bom'],
            -default=>'bom',
            -linebreak=>'true',
            -labels=>\%labels);
```

Первый аргумент является обязательным, соответствует одноименному атрибуту тэга. Второй аргумент тоже обязательный и задает значения

элементов. Эти значения отображаются в качестве названий кнопок. Он должен быть ссылкой на массив. Остальные аргументы являются необязательными. Третий аргумент задает кнопку, которая выбрана по умолчанию. Если значение четвертого аргумента 'true', каждая следующая кнопка группы размещается в начале новой строки. Пятым аргументом является ссылка на хеш-массив, который связывает значения, присвоенные кнопкам, с метками, которые отображаются в виде названий кнопок. Если аргумент не задан, то в качестве названий отображаются сами значения.

- Функция `checkbox_group` служит для создания группы элементов-переключателей, задаваемых тэгом `<INPUT TYPE= CHECKBOX>`.

```
checkbox_group(-name=>'group_name',  
            -values=>['bim', 'bam', 'bom'],  
            -default=>['bim', 'bom'],  
            -linebreak=>'true',  
            -labels=>\%labels);
```

Аргументы имеют тот же смысл, что и одноименные аргументы функции `radio_group`. Поскольку в группе переключателей можно одновременно выбрать несколько элементов, третий аргумент может быть или единственным элементом, или ссылкой на массив, содержащий список значений, выбранных по умолчанию. Обязательными являются первый и второй аргументы.

- Функция `popup_menu` служит для создания меню, задаваемого при помощи тэга `<SELECT>`. Имеет следующий синтаксис:

```
popup_menu(-name=>'menu_name',  
            -values=>['bim', 'bam', 'bom'],  
            -default=>'bom',  
            -labels=>\%labels);
```

Первый аргумент задает имя меню. Вторым аргументом является ссылка на массив, содержащий список значений, присвоенных элементам меню. Первый и второй аргументы обязательны, остальные — нет. Третий аргумент задает элемент меню, выбранный по умолчанию. Четвертый аргумент является ссылкой на хеш-массив. Хеш-массив значению каждого элемента меню ставит в соответствие строку, которая будет отображаться в меню для этого элемента. Если четвертый аргумент отсутствует, то для каждого элемента меню отображается его значение, заданное вторым аргументом.

- Функция `textarea` соответствует тэгу `<TEXTAREA>`, задающему в документе текстовое поле для ввода многострочного текста. Имеет следующий синтаксис

```
textarea(-name=>'region',
         -default=>'starting value',
         -rows=>10,
         -columns=>50);
```

Первый параметр, задающий имя элемента формы `<TEXTAREA>`, является обязательным, остальные — нет. Второй параметр задает строку, отображаемую по умолчанию. Третий и четвертый параметры задают соответственно число строк и столбцов, отображаемых в текстовом поле.

- ❑ Функция `submit` соответствует тэгу `<INPUT TYPE=SUBMIT>`, задающему кнопку передачи. Ее синтаксис:

```
print $query->submit(-name=>'button_name',
                   -value=>'value');
```

Первый параметр является необязательным. Он задает имя кнопки, которое отображается в качестве ее названия. Нужен только для переопределения названия **Submit** и в тех случаях, когда надо различать несколько имеющихся кнопок передачи. Второй параметр тоже необязательный. Он задает значение, которое посылается в строке запроса при щелчке на этой кнопке.

- ❑ Функция `reset` соответствует тэгу `<INPUT TYPE=RESET>`, задающему кнопку сброса. Может иметь параметр, переопределяющий название **Reset**, отображаемое по умолчанию.
- ❑ Функция `end_html` завершает HTML-документ, добавляя тэги `</BODY>` `</HTML>`.

Пример 15.8 содержит также код, который не связан с созданием формы. Он состоит из одного условного оператора, в котором в качестве условия используется значение, возвращаемое функцией `param()`. Эта функция используется также внутри блока условного оператора. Разберем, для чего она применяется. При помощи функции `param()` модуля `CGI` можно выполнить следующие действия.

- ❑ Получение списка имен параметров, переданных сценарию.

Если сценарию переданы параметры в виде списка пар "имя=значение", функция `param()` без аргументов возвращает список имен параметров сценария:

```
@names = param;
```

- ❑ Получение значений именованных параметров.

Функция `param()` с единственным аргументом — именем параметра, возвращает значение этого параметра. Если параметру соответствует несколько значений, функция `param()` возвращает список этих значений:

```
@values = param('language');
```

в противном случае — одно значение:

```
$value = param('regname');
```

❑ Задание значений параметров.

```
param(-name=>'language',-values=>['russian','english','french']);
```

Можно задавать значения параметров, используя вызов функции `param()` в форме с позиционными параметрами, но тогда нужно знать порядок следования этих параметров:

```
param('language','russian','english','french');
```

При помощи функции `param()` можно устанавливать начальные значения элементов формы или изменять ранее установленные значения.

Часть сценария, предшествующая условному оператору, предназначена для создания формы из примера 15.1. Заключительная часть, состоящая из условного оператора, обрабатывает заполненную и отправленную Web-серверу форму. Это происходит потому, что по умолчанию приложением, обрабатывающим форму, является данный сценарий (см. описание `start_form`). Таким образом, в одном сценарии содержится код, и создающий форму, и ее обрабатывающий.

Сохраним код, приведенный в примере 15.8, в файле `welcome.cgi`. Этот файл можно поместить на Web-сервере в стандартный каталог `cgi-bin`, предназначенный для хранения CGI-сценариев. Предположим, что Web-сервер имеет Internet-адрес www.klf.ru. Если из удаленного браузера послать запрос по адресу <http://www.klf.ru/cgi-bin/welcome.cgi>, то Web-сервер, получив запрос, выполнит сценарий `welcome.cgi`. Сценарий "на лету" создаст HTML-документ, содержащий форму, и передаст его Web-серверу, который отправит документ браузеру. Браузер, получив документ, отобразит его, как это показано на рис. 15.8.

После заполнения формы и нажатия кнопки **ОК** данные формы будут вновь отправлены Web-серверу, который передаст их для обработки все тому же сценарию `welcome.cgi`. Сценарий "на лету" создаст новый HTML-документ с учетом полученных данных и через сервер направит его браузеру. Браузер отобразит новый документ, как показано на рис. 15.9.

Сценарий `welcome.cgi` можно передать для выполнения интерпретатору `perl`, а результат вывести в файл, чтобы посмотреть, как вызовы функций модуля CGI преобразуются в тэги HTML-документа. Документ HTML, созданный сценарием `welcome.cgi`, имеет следующий вид.

Пример 15.9. Текст HTML-документа, созданного сценарием `welcome.cgi`

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Пример формы</TITLE>
</HEAD><BODY>
```

```
<H2>Регистрационная страница Клуба любителей фантастики</H2>
Заполнив анкету, вы сможете пользоваться нашей электронной библиотекой.
<BR>
<FORM METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">
Введите регистрационное имя:<INPUT TYPE="text" NAME="regname" ><P>
Введите пароль: <INPUT TYPE="password" NAME="password1" MAXLENGTH=8><P>
Подтвердите пароль: <INPUT TYPE="password" NAME="password2" MAXLENGTH=8>
<P>Ваш возраст<P>
<INPUT TYPE="radio" NAME="age" VALUE="lt20" CHECKED>до 20
<INPUT TYPE="radio" NAME="age" VALUE="20_30">20-30
<INPUT TYPE="radio" NAME="age" VALUE="30_50">30-50
<INPUT TYPE="radio" NAME="age" VALUE="gt50">старше 50
<BR><BR>На каких языках читаете:
<INPUT TYPE="checkbox" NAME="language" VALUE="русский" CHECKED>русский
<INPUT TYPE="checkbox" NAME="language" VALUE="английский">английский
<INPUT TYPE="checkbox" NAME="language" VALUE="французский">французский
<INPUT TYPE="checkbox" NAME="language" VALUE="немецкий">немецкий
<BR><BR>
Какой формат данных является для Вас предпочтительным
<BR><SELECT NAME="type">
<OPTION VALUE="Plain text">Plain text
<OPTION VALUE="PostScript">PostScript
<OPTION VALUE="PDF">PDF
</SELECT>
<BR><BR>
Ваши любимые авторы:
<BR><TEXTAREA NAME="wish" ROWS=3 COLS=40></TEXTAREA>
<BR>
<INPUT TYPE="submit" NAME="OK" VALUE="OK">
<INPUT TYPE="reset" VALUE="Отменить">
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="language">
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="age">
</FORM>
<HR></BODY></HTML>
```

В действительности документ, созданный сценарием `welcome.cgi`, состоит из небольшого количества длинных строк, что связано с тем, как они формируются методами модуля CGI. Поэтому реально сформированный текст для удобства представлен в более структурированном виде. Но это единственное изменение, не влияющее на смысл автоматически созданного документа.

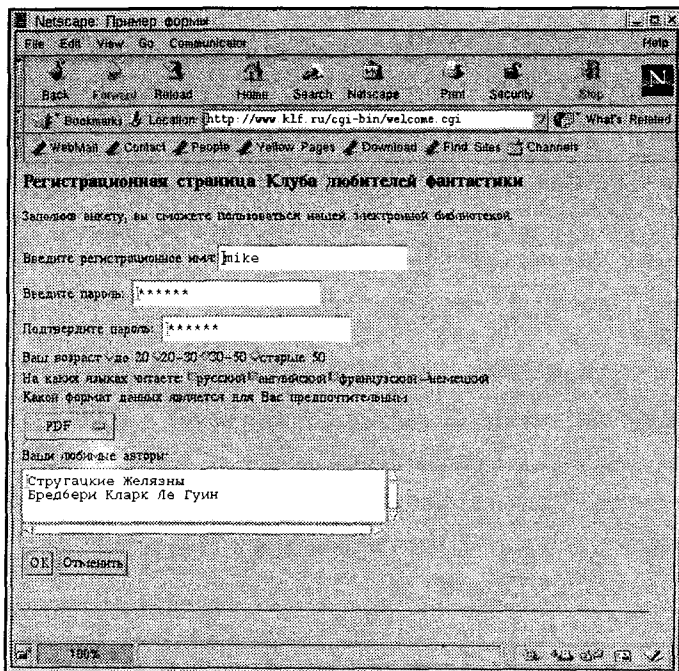


Рис. 15.8. Документ, созданный CGI-сценарием из примера 15.8

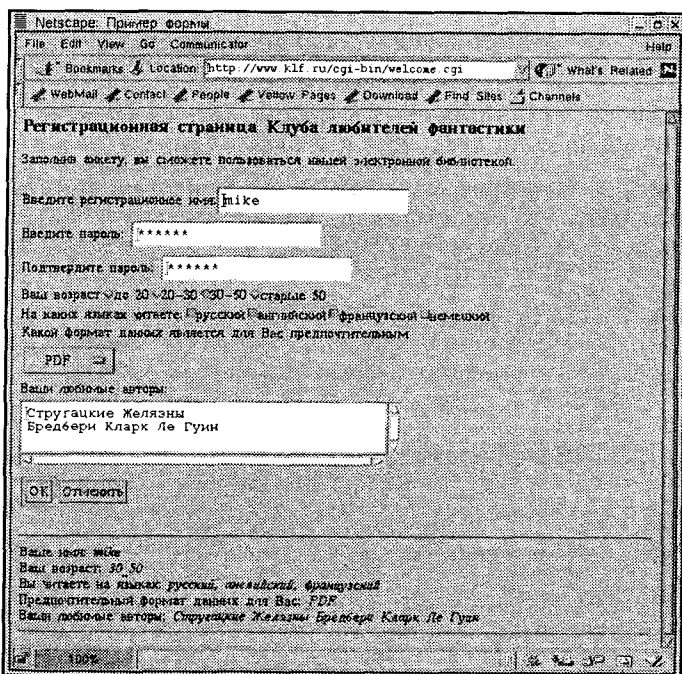


Рис. 15.9. Документ, созданный после обработки данных формы

Вопросы для самоконтроля

1. Что обозначает аббревиатура CGI?
2. Что такое Web-сервер?
3. Что такое клиент Web-сервера?
4. Для чего нужна HTML-форма?
5. Какую первую строку должна выводить CGI-программа?
6. Какие методы передачи данных формы Web-серверу вы знаете? В чем их особенности?
7. Перечислите элементы управления, которые может содержать форма.
8. Как кодируется информация, передаваемая CGI-программе?
9. В чем заключается специфика CGI-сценариев?
10. Каким образом CGI-программа имеет доступ к переменным среды CGI?
11. Какая информация содержится в переменной среды `QUERY_STRING`?
12. Для чего предназначен модуль CGI.pm?

Упражнения

1. Запишите текст примера 15.8 в обозначениях, использующих объектно-ориентированный интерфейс с модулем CGI.
2. Используя модуль CGI, дополните пример 15.8 кодом, осуществляющим проверку введенных данных, как это сделано в примере 15.76.
3. Ниже приведены тексты трех примеров, входящих в отдельный дистрибутив модуля CGI.pm. Разберите, что делают эти сценарии.

a) `clickable_image.cgi`

```
#!/usr/bin/perl
use CGI;
$query = new CGI;
print $query->header;
print $query->start_html("A Clickable Image");
print <<END;
<H1>A Clickable Image</H1>
</A>
END
print "Sorry, this isn't very exciting!\n";
print $query->startform;
```

```

print $query->image_button('picture','./wilogo.gif');
print "Give me a: ", $query->popup_menu('letter',
    ['A','B','C','D','E','W']), "\n";
print "<P>Magnification: ", $query->radio_group('magnification',
    ['1X','2X','4X','20X']), "\n";

print "<HR>\n";
if ($query->param) {
    print "<P>Magnification, <EM>", $query->param('magnification'),
        "</EM>\n";

    print "<P>Selected Letter, <EM>", $query->param('letter'), "</EM>\n";
    ($x,$y) = ($query->param('picture.x'), $query->param('picture.y'));
    print "<P>Selected Position <EM>($x,$y)</EM>\n";
}
print $query->end_html;

```

6) quadraphobia.cgi

```

#!/usr/bin/perl
use CGI qw/:standard/;

print header,
    start_html('QuadraPhobia'),
    h1('QuadraPhobia'),
    start_form(),
    image_button(-name=>'square',
        -src=>'red_square.gif',
        -width=>200,
        -height=>200,
        -align=>MIDDLE),
    end_form();

if (param()) {
    ($x,$y) = (param('square.x'), param('square.y'));
    $pos = 'top-left' if $x < 100 && $y < 100;
    $pos = 'top-right' if $x >= 100 && $y < 100;
    $pos = 'bottom-left' if $x < 100 && $y >= 100;
    $pos = 'bottom-right' if $x >= 100 && $y >= 100;
    print b("You clicked on the $pos part of the square.");
}

print p,a({href=>'../source.html'}, "Code examples");
print end_html();

```

B) popup.cgi

```
#!/usr/local/bin/perl
```

```
use CGI;
```

```
$query = new CGI;
```

```
print $query->header;
```

```
print $query->start_html('Popup Window');
```

```
if (!$query->param) {
```

```
    print "<H1>Ask your Question</H1>\n";
```

```
    print $query->startform(-target=>'_new');
```

```
    print "What's your name? ", $query->textfield('name');
```

```
    print "<P>What's the combination?<P>",
```

```
    $query->checkbox_group(-name=>'words',
```

```
                        -values=>['eenie', 'meenie', 'minie', 'moe'],
```

```
                        -defaults=>['eenie', 'moe']));
```

```
    print "<P>What's your favorite color? ",
```

```
    $query->popup_menu(-name=>'color',
```

```
                    -values=>['red', 'green', 'blue', 'chartreuse']),
```

```
    "<P>";
```

```
    print $query->submit;
```

```
    print $query->endform;
```

```
} else {
```

```
    print "<H1>And the Answer is...</H1>\n";
```

```
    print "Your name is <EM>", $query->param(name), "</EM>\n";
```

```
    print "<P>The keywords are: <EM>",
```

```
        join(" ", $query->param(words)), "</EM>\n";
```

```
    print "<P>Your favorite color is <EM>",
```

```
        $query->param(color), "</EM>\n";
```

```
}
```

```
print qq{<P><A HREF="cgi_docs.html">Go to the documentation</A>};
```

```
print $query->end_html;
```



Ресурсы Perl

Язык Perl распространяется и развивается в глобальной сети Internet. Самые последние новости из мира Perl можно найти на многочисленных Web-узлах и просто обычных страницах HTML, авторы которых являются апологетами языка Perl и входят в так называемое сообщество Perl — свободное объединение людей, использующих и развивающих язык Perl там, где требуется что-нибудь и как-нибудь автоматизировать. Задав в любой поисковой машине (например, AltaVista, Yahoo) поиск по ключевому слову "Perl", вы получите тысячи ссылок на различные ресурсы Internet, так или иначе связанные с языком Perl и программированием на этом языке. Чтобы читатель не запутался и не растерялся в этом обилии информации, мы решили в последней главе нашего самоучителя отметить основные ресурсы Internet — конференции, Web-узлы и FTP-серверы, с которых можно и стоит начинать вхождение в многообразный и интересный мир Perl.

16.1. Конференции

Конференции, или группы новостей Usenet, — это такие ресурсы Internet, куда люди могут посылать статьи со своими мыслями или вопросами на темы, обсуждаемые в конференции. Если поднятый вопрос заинтересует кого-либо из участников конференции, то могут разгореться жаркие дебаты.

Не советуем сразу же посылать в конференции Perl свои вопросы. Прежде всего ознакомьтесь с их содержанием, так как вполне возможно, что подобные (или близкие) темы в них уже обсуждались, и ваш аналогичный вопрос может вызвать раздражение участников.

В Internet существуют пять известных конференций, в которых обсуждаются вопросы Perl-программирования:

- ☐ **news://comp.lang.perl** — основная группа новостей, с которой рекомендуется начинать при поиске необходимой информации;
- ☐ **news://comp.lang.perl.announce** — публикуется информация о новых модулях Perl;
- ☐ **news://comp.lang.perl.modules** — дискуссии об использовании модулей, именно здесь можно найти информацию о новых версиях модулей;
- ☐ **news://comp.lang.perl.misc** — здесь обсуждается информация, не вошедшая в другие группы новостей;

- news://comp.lang.perl.tk — обсуждаются проблемы использования модуля Tk для создания интерфейса пользователя.

Регулярное чтение указанных конференций может дать вам больше знаний, чем чтение книг по языку Perl (мы не имеем в виду получение базовых знаний о нем), так как обсуждаемые в конференциях вопросы возникают в результате *повседневной* практической деятельности большинства их участников. В книгах обычно отражается накопленный к определенному моменту времени практический опыт автора, но новые задачи-то продолжают возникать.

Здесь же мы хотим дать один небольшой, но полезный совет: прежде чем начинать поиск ответа на возникший вопрос в конференциях, обратитесь к файлу, содержащему часто задаваемые вопросы, или, как его еще называют, файл Perl FAQ (*Frequently Asked Questions*). Его можно найти по адресу:

<http://www.perl.com/perl/faq/>

16.2. Специализированные Web-узлы Perl

Группы новостей — интересные и полезные места обмена информацией, но доступ к ним зависит от вашего поставщика услуг Internet. Не все они предоставляют доступ к конференциям с помощью своего сервера новостей, а если и предоставляют, то может оказаться, что именно той конференции, о которой вы мечтали, в списке доступных нет. В таких случаях следует обратить свое внимание на *Web-узлы*, полностью или частично посвященные языку Perl. И первым в этом ряду следует назвать Web-узел, расположенный по адресу

<http://www.perl.com>

Этот узел является, если можно так сказать, официальным "представителем" информации по языку Perl. Здесь всегда можно узнать последние новости из мира Perl: какая версия интерпретатора считается устойчивой, а какая находится в разработке, где найти и как можно загрузить их на свой компьютер. Узел содержит обширную информацию о реализациях языка на различных платформах (UNIX, Linux, Win32, Mac, OS2 и др.), документацию по последней версии языка и многое другое. Здесь же имеются ссылки на интересные статьи и разработки.

На других многочисленных Web-узлах можно найти учебные пособия и книги по языку Perl, материалы конференций, разнообразные советы по программированию на языке Perl и многое другое. Мы включили в приводимый ниже список только небольшую часть узлов Internet, имеющих отношение к Perl и наиболее интересных с нашей точки зрения:

- <http://www.perl.oreilly.com> — узел издательства O'Reilly, на котором можно найти и приобрести книги по языку Perl, ссылки на другие узлы, а также последние новости;

- ❑ <http://conference.perl.com/pace/conf/> — здесь расположены материалы третьей конференции по языку Perl, проходившей с 21 по 24 августа 1999 года;
- ❑ <http://perl.lco.net/> — дискуссионный форум по программированию на языке Perl;
- ❑ <http://www.perl.org> — узел некоммерческой организации, поддерживающей сообщество Perl;
- ❑ <http://www.perlmonks.org> — можно найти разнообразные советы по программированию, электронные учебники как для начинающих, так и для тех, кто желает углубить свои знания по языку Perl;
- ❑ <http://www.perlfoo.org> — здесь можно найти реализацию с помощью Perl интересных задач;
- ❑ <http://feenix.metronet.com/1h/perlinfo/> — архив статей о Perl, начиная с 1996 года;
- ❑ <http://www.perl.com/pub/language/versus/index.html> — статьи, посвященные сравнению языка Perl с языками программирования Tcl, Python, Java, C, csh и др.
- ❑ <http://www.perlfaq.com> — интерактивная база данных по часто задаваемым вопросам, позволяет осуществлять поиск на основе запроса пользователя;
- ❑ <http://www.cre.canon.co.uk/~neilb//perl/VHLL/slide01.html> — здесь можно найти набор слайдов Ларри Уолла по объяснению языка Perl 5;
- ❑ <http://www.stars.com> — интересный узел, посвященный проблемам создания Интернет-приложений, в том числе и на базе языка Perl;
- ❑ <http://www.activestate.com> — узел фирмы ActiveState Tool Co., на котором имеются свободно распространяемые двоичные версии интерпретаторов для различных платформ: UNIX, Linux, Win32 и Mac.

Мы перечислили только очень небольшую часть адресов Web-узлов, на которых можно найти информацию по Perl. На узлах многих университетов существует описание курсов, посвященных изучению языка Perl и его применению для решения задач Web-программирования и системного администрирования.

16.3. Архив CPAN

CPAN (Comprehensive Perl Archive Network) представляет гигантское хранилище практически всех документов и исходных текстов Perl, а также информацию и исходные тексты огромного количества модулей, расширяющих возможности языка Perl и упрощающих программирование на нем. Если у вас возникла проблема, требующая решения, загляните на CPAN и внима-

тельно посмотрите среди предлагаемых там модулей — может быть, в каком-нибудь из них эта или схожая задачи уже решены. Все модули на CPAN свободно распространяемые, и вы можете бесплатно загрузить и установить на собственном компьютере любой из них. Попасть в архив CPAN можно по адресу:

<http://www.cpan.com>

На домашней странице этого Web-узла можно найти ссылки на документацию в разных форматах, архив конференции **news:comp.lang.perl.announce**, начиная с 1995 года, ссылки на страницу модулей и программы Perl для работы с аудиофайлами, администрирования, обработки почты, организации поиска, автоматизации в Web, организации сетевого взаимодействия и написания CGI-сценариев. Здесь же находится ссылка на двоичные дистрибутивы (порты) Perl для разных платформ и на файл с часто задаваемыми вопросами.

В сети Internet, кроме Web-узлов, существуют ресурсы, на которых хранятся файлы с разнообразной информацией. Это так называемые FTP-серверы. Многие из них предоставляют возможность свободной загрузки на компьютер пользователя хранящихся на них файлов по протоколу FTP. Все, что находится на основном узле архива CPAN, доступно через многочисленные FTP-серверы, которые являются обычными копиями (узел-зеркало) содержимого основного узла. На всех узлах-зеркалах содержится совершенно идентичная информация. Организация FTP-сервера архива CPAN представлена в табл. 16.1.

Таблица 16.1. Организация архива CPAN

Файл или каталог	Описание
README.html	Основной документ
CPAN.html	Описание организации архива CPAN, ссылок на все модули и внесения в CPAN собственного модуля
ENDINGS	Описание содержимого файлов в соответствии с их расширением
SITES.html	Список FTP-серверов и Web-узлов архивов CPAN
RECENT.html	Последние поступления
ROADMAP.html	Организация архива CPAN
authors/	Каталог модулей и расширений Perl, классифицированных по авторам
clpa/	Каталог архивов групп новостей comp.lang.perl.announce

Таблица 16.1 (окончание)

Файл или каталог	Описание
doc/	Каталог, содержащий документацию Perl
indices/	Каталог, содержащий файлы с указателями на ресурсы CPAN
misc/	Каталог, содержащий разнообразные ресурсы Perl, не вошедшие ни в один из перечисленных разделов
modules/	Каталог модулей и расширений Perl
ports/	Каталог двоичных дистрибутивов Perl для разных платформ
scripts/	Каталог разных скриптов Perl на разные случаи жизни (просмотрите содержимое этого каталога, если не хотите изобретать велосипед!)
src/	Каталог исходных текстов и исправлений самого Perl и вспомогательных утилит

Попасть на один из FTP-серверов архива CPAN можно по ссылке с домашней страницы архива (www.cpan.org) или по одному из следующих адресов, взятых из большого списка FTP-серверов, содержащихся в файле SITES.html:

<ftp://ftp.funet.fi/pub/languages/perl/CPAN/> (Финляндия)

<ftp://ftp.rz.ruhr-uni-bochum.de/pub/CPAN/> (Германия)

<ftp://sunsite.auc.dk> (Дания)

<ftp://ftp.chg.ru/pub/lang/perl/CPAN/> (Россия)

<ftp://ftp.sai.msu.su/pub/lang/perl/CPAN/> (Россия)

* * *

На этом наше первоначальное знакомство с языком Perl заканчивается. Мы надеемся, что читатель, вдумчиво анализировавший все приводимые в самоучителе примеры, добросовестно выполнявший упражнения и отвечавший на предлагавшиеся в конце каждой главы вопросы, сможет теперь самостоятельно продолжить углубление своих знаний в этом интересном и полезном языке программирования. Приведенные в данной главе ссылки на ресурсы Perl в Internet помогут ему в этой благородной задаче.



Стандартные функции Perl

В этом приложении представлено краткое описание всех функций Perl версии 5.005 в соответствии с категориями их применения. Если в тексте самоучителя функция рассматривалась более подробно, то дается ссылка на соответствующий раздел.

Все функции этого приложения в выражениях могут служить терминами. По существу, они являются операциями языка Perl (их синтаксис позволяет задавать параметры в круглых скобках, что внешне похоже на вызов функции) и распадаются на две большие категории: списковые операции и именованные унарные операции. Принадлежность функции к соответствующей категории влияет на ее старшинство по отношению к операции "запятая". У списковых операций может быть несколько параметров, тогда как именованная унарная операция работает исключительно с одним параметром. Таким образом, операция "запятая" после унарной операции завершает "список" ее параметров, тогда как в списковой операции просто является разделителем ее параметров. Параметр унарной операции обычно вычисляется в скалярном контексте, а списковая операция может предоставлять и скалярный, и списковый контекст для своих параметров. Если она предоставляет оба контекста, то сначала задаются скалярные параметры, после которых следует списковый параметр операции.

В приводимом в таблицах синтаксисе функций, если она является списковой операцией и требует списка в качестве своего параметра, предоставляя тем самым списковый контекст для всех элементов этого списка, то такой параметр обозначается список. Такой список может представлять собой комбинацию скалярных значений и списковых значений. Последние включают в список все свои элементы, начиная с той точки списка, где они заданы, увеличивая тем самым протяженность списка-параметра. Все элементы параметра список должны быть отделены запятыми.

Параметры всех функций можно задавать и в круглых скобках, и без них. При использовании скобок следует всегда помнить о следующем правиле: если операция выглядит *как* функция, то она и *есть* функция, а поэтому в выражении она имеет наивысший приоритет. При задании параметров без скобок функция рассматривается либо как списковая, либо как именованная унарная операция, и, следовательно, следует учитывать ее старшинство при вычислении выражения, в которое она входит.

В скалярном контексте в случае неудачного выполнения функции обычно возвращают неопределенное значение `undef`, в списковом контексте — пустой список.

В языке Perl не регламентировано, что следует считать значением списка, который может возвращать функция, в скалярном контексте. Здесь можно встретиться с различными ситуациями. Каждая операция и функция сама решает, какое значение будет наиболее приемлемым при вычислениях в скалярном контексте: некоторые операции возвращают количество элементов списка, другие — значение первого или последнего элемента списка, третьи — количество успешно выполненных операций.

При использовании всех, приведенных в таблицах этого приложения, функций следует помнить обо всех нюансах работы со списковыми и именованными унарными операциями языка Perl.

Таблица П1.1. Функции обработки строк и скаляров

Функция	Назначение и синтаксис	Ссылка
<code>chomp</code>	Удаляет из каждого строкового элемента списка замыкающий символ завершения записи, соответствующий значению переменной <code>\$/</code> (по умолчанию — символ новой строки <code>"\n"</code>). Возвращает общее количество удаленных символов. Список может состоять из одной переменной. <code>chomp</code> СПИСОК <code>chomp</code> (эквивалентно <code>chomp \$_</code>)	
<code>chop</code>	Удаляет из каждого строкового элемента списка последний символ. Возвращаемое значение — удаленный символ из последнего элемента списка. Список может состоять из одной переменной. <code>chop</code> СПИСОК <code>chop</code> (эквивалентно <code>chop \$_</code>)	10.3
<code>chr</code>	Возвращает символ, код которого представлен числовым параметром. <code>chr</code> ЧИСЛО <code>chr</code> (эквивалентно <code>chr \$_</code>)	
<code>crypt</code>	Шифрует ТЕКСТ с использованием заданного в параметре шифра. Обратной функции дешифровки не существует. <code>crypt</code> ТЕКСТ, ШИФР	
<code>hex</code>	Интерпретирует строковое ВЫРАЖЕНИЕ как шестнадцатеричное число и вычисляет его десятичный эквивалент. <code>hex</code> ВЫРАЖЕНИЕ <code>hex</code> (эквивалентно <code>hex \$_</code>)	16.4.2

Таблица П1.1 (продолжение)

Функция	Назначение и синтаксис	Ссылка
index	Возвращает позицию первого вхождения указанной подстроки в заданную строку или -1, если подстрока не найдена. Если задан параметр ПОЗИЦИЯ, то поиск подстроки осуществляется, начиная с заданной позиции в строке (0 – начало строки). index СТРОКА, ПОДСТРОКА[, ПОЗИЦИЯ]	10.3
lc	Преобразует все прописные буквы строкового параметра ВЫРАЖЕНИЕ в строчные и возвращает полученную строку. Использует текущие установки локализации, если используется use local. lc ВЫРАЖЕНИЕ lc (эквивалентно lc \$_)	10.3
lcfirst	Преобразует первый символ строкового параметра ВЫРАЖЕНИЕ в нижний регистр и возвращает полученную строку. Использует текущие установки локализации, если используется use local. lcfirst ВЫРАЖЕНИЕ lcfirst (эквивалентно lcfirst \$_)	10.3
length	Возвращает количество байтов в строке, являющейся значением параметра ВЫРАЖЕНИЕ. length ВЫРАЖЕНИЕ length (эквивалентно length \$_)	10.3
oct	Интерпретирует строковое ВЫРАЖЕНИЕ как восьмеричное число и вычисляет его десятичный эквивалент. Если строка начинается с символов "0x", то ее содержимое интерпретируется как шестнадцатеричное число. oct ВЫРАЖЕНИЕ oct (эквивалентно oct \$_)	
ord	Возвращает числовой ASCII-код первого символа строки, являющейся значением параметра ВЫРАЖЕНИЕ. ord ВЫРАЖЕНИЕ ord (эквивалентно ord \$_)	
pack	Упаковывает массив или список значений в двоичную структуру в соответствии с заданным шаблоном, представляющим собой последовательность символов, которые задают порядок и тип значений. Возвращает строку, содержащую полученную структуру. pack ШАБЛОН, СПИСОК	16.4.2

Таблица П1.1 (окончание)

Функция	Назначение и синтаксис	Ссылка
reverse	В списковом контексте возвращает список значений, состоящий из элементов заданного параметром СПИСОК списка; в скалярном контексте соединяет все элементы списка в одну строку и возвращает строку, состоящую из символов полученной строки, но в обратном порядке. reverse СПИСОК	
rindex	Возвращает позицию последнего вхождения указанной подстроки в заданную строку или -1, если подстрока не найдена. Если задан параметр ПОЗИЦИЯ, то поиск подстроки осуществляется, до заданной позиции в строке (включая символ в этой позиции). rindex СТРОКА, ПОДСТРОКА[, ПОЗИЦИЯ]	10.3
sprintf	Возвращает строку, представляющую форматный вывод списка значений, определяемого параметром СПИСОК, в соответствии с заданной первым параметром ФОРМАТ строкой формата. Символы форматирования соответствуют аналогичной функции в языке C. sprintf ФОРМАТ, СПИСОК	
substr	Извлекает из строки, заданной параметром СТРОКА, подстроку длиной равной значению параметра ДЛИНА, начиная с символа, заданного параметром СМЕЩЕНИЕ. Если СМЕЩЕНИЕ отрицательно, то извлечение начинается с последнего символа строки. Если значение параметра ДЛИНА отрицательно, то от конца строки отсекается количество символов, равное абсолютному значению этого параметра. Если задана строка ЗАМЕЩЕНИЕ, то выделенная подстрока замещается ею в параметре СТРОКА, который в этом случае должен быть lvalue. substr СТРОКА, СМЕЩЕНИЕ[, ДЛИНА[, ЗАМЕЩЕНИЕ]]	10.3
uc	Преобразует все строчные буквы строкового параметра ВЫРАЖЕНИЕ в прописные и возвращает полученную строку. Использует текущие установки локализации, если используется use local. uc ВЫРАЖЕНИЕ uc (эквивалентно uc \$_)	10.3
ucfirst	Преобразует первый символ строкового параметра ВЫРАЖЕНИЕ в верхний регистр и возвращает полученную строку. Использует текущие установки локализации, если используется use local. ucfirst ВЫРАЖЕНИЕ ucfirst (эквивалентно ucfirst \$_)	10.3

Таблица П1.2. Функции, связанные с регулярными выражениями

Функция	Назначение и синтаксис	Ссылка
pos	<p>Возвращает значение указателя позиции в строке после последней выполненной для нее операции поиска по образцу <code>m//g</code>. Если используется в левой части операции присваивания, то изменяет значение указателя, влияя на <code>\G</code> в регулярном выражении.</p> <p>pos СТРОКА</p> <p>pos (эквивалентно pos \$_)</p>	10.3
quotemeta	<p>Возвращает строку, в которой перед каждым не алфавитно-цифровым символом поставлена обратная косая черта. Значение параметра ВЫРАЖЕНИЕ интерпретируется как строка. (Внутренняя функция реализации управляющего символа <code>\Q</code> в строках в двойных кавычках.)</p> <p>quotemeta ВЫРАЖЕНИЕ</p> <p>quotemeta (эквивалентно quotemeta \$_)</p>	10.3
split	<p>Разбивает строку, являющуюся значением параметра ВЫРАЖЕНИЕ, на массив строк с использованием разделителя, определяемого регулярным выражением в параметре ОБРАЗЕЦ. В списковом контексте возвращает полученный массив, в скалярном контексте — количество найденных строк разбиения. Если задан параметр ПРЕДЕЛ, разбивает на не более, чем заданное этим параметром число строк. Если ВЫРАЖЕНИЕ не задано, то используется <code>\$_</code>, если не задан и образец, то разбивает по пробельным символам.</p> <p>split /ОБРАЗЕЦ/[, ВЫРАЖЕНИЕ[, ПРЕДЕЛ]]</p> <p>split (эквивалентно split /\s+/, \$_)</p>	10.3
study	<p>Оптимизирует строковые данные параметра СТРОКА для дальнейшего использования в повторных операциях поиска по образцу. В цикле с несколькими операциями поиска может сэкономить время выполнения.</p> <p>study ВЫРАЖЕНИЕ</p> <p>study (эквивалентно study \$_)</p>	

Таблица П1.3. Числовые функции

Функция	Назначение и синтаксис	Ссылка
abs	<p>Вычисление абсолютного значения выражения.</p> <p>abs ВЫРАЖЕНИЕ</p> <p>abs (эквивалентно abs \$_)</p>	

Таблица П1.3 (продолжение)

Функция	Назначение и синтаксис	Ссылка
atan2	Вычисление $\arctg(X/Y)$. atan2 X, Y	
cos	Вычисление функции \cos (ВЫРАЖЕНИЕ), выражение в радианах. cos ВЫРАЖЕНИЕ cos (эквивалентно cos \$ _)	
exp	Вычисление значения экспоненциальной функции $e^{\text{ВЫРАЖЕНИЕ}}$. exp ВЫРАЖЕНИЕ exp (эквивалентно exp \$ _)	
hex	Интерпретирует строковое ВЫРАЖЕНИЕ как шестнадцатеричное число и вычисляет его десятичный эквивалент. hex ВЫРАЖЕНИЕ hex (эквивалентно hex \$ _)	16.4.2
int	Вычисление целой части числа (отбрасывается дробная часть). int ВЫРАЖЕНИЕ int (эквивалентно int \$ _)	
log	Вычисляет натуральный логарифм числа (по основанию e). log ВЫРАЖЕНИЕ log (эквивалентно log \$ _)	
oct	Интерпретирует строковое ВЫРАЖЕНИЕ как восьмеричное число и вычисляет его десятичный эквивалент. Если строка начинается с символов "0x", то ее содержимое интерпретируется как шестнадцатеричное число. oct ВЫРАЖЕНИЕ oct (эквивалентно oct \$ _)	
rand	Возвращает случайное дробное число в интервале от 0 до значения параметра ВЫРАЖЕНИЕ. rand ВЫРАЖЕНИЕ rand (эквивалентно rand 1)	
sin	Вычисление функции \sin (ВЫРАЖЕНИЕ), выражение в радианах. sin ВЫРАЖЕНИЕ sin (эквивалентно sin \$ _)	

Таблица П1.3 (окончание)

Функция	Назначение и синтаксис	Ссылка
sqrt	Вычисляет квадратный корень числа. sqrt ВЫРАЖЕНИЕ sqrt (эквивалентно sqrt \$_)	
srand	Затравка перед первым вызовом rand() (в Perl 5.005 и выше вызывается автоматически при обращении к rand). srand ВЫРАЖЕНИЕ srand	

Таблица П1.4. Функции обработки массивов скаляров

Функция	Назначение и синтаксис	Ссылка
pop	Удаляет из массива скаляров последний элемент и возвращает его значение. Если массив пустой, то возвращает неопределенное значение undef. pop МАССИВ pop (в основной программе эквивалентно pop @ARGV, в подпрограмме эквивалентно pop @_)	
push	Рассматривает массив, заданный параметром МАССИВ, как стек и добавляет в конец массива элементы списка, определяемого параметром СПИСОК. Возвращает новое количество элементов полученного массива. push МАССИВ, СПИСОК	9.5.3
shift	Удаляет из массива скаляров первый элемент и возвращает его значение. После удаления элемента оставшиеся сдвигаются влево: второй становится первым, третий вторым и т. д. Если массив пустой, то возвращает неопределенное значение undef. shift МАССИВ shift (в основной программе эквивалентно shift @ARGV, в подпрограмме эквивалентно shift @_)	9.5.1
splice	Удаляет из массива заданное КОЛИЧЕСТВО элементов, начиная с элемента, определенного параметром НОМЕР. В случае задания параметра СПИСОК заменяет указанные элементы элемента списка. В списковом контексте возвращает удаленные элементы; в скалярном контексте — последний удаленный элемент. Если параметр КОЛИЧЕСТВО не задан, то удаляются все элементы после элемента с номером, определенным параметром НОМЕР. splice МАССИВ, НОМЕР[, КОЛИЧЕСТВО[, СПИСОК]]	

Таблица П1.4 (окончание)

Функция	Назначение и синтаксис	Ссылка
unshift	<p>Добавляет элементы списка, определенного параметром СПИСОК, в начало массива, заданного параметром МАССИВ. Выполняет действия, противоположные действиям функции shift.</p> <p>unshift МАССИВ, СПИСОК</p>	12.2.2

Таблица П1.5. Функции обработки списков

Функция	Назначение и синтаксис	Ссылка
grep	<p>Вычисляет блок операторов БЛОК или ВЫРАЖЕНИЕ для всех элементов списка, заданного параметром СПИСОК, локально устанавливая значение специальной переменной \$_ равным значению элемента списка. В списковом контексте возвращает список элементов, для которых ВЫРАЖЕНИЕ вычисляется равным Истина. В скалярном контексте — количество раз, когда ВЫРАЖЕНИЕ вычислялось равным Истина.</p> <p>grep БЛОК СПИСОК</p> <p>grep ВЫРАЖЕНИЕ, СПИСОК</p>	
join	<p>Соединяет отдельные строковые элементы списка параметра СПИСОК в одну строку, вставляя между ними разделитель, равный значению параметра ВЫРАЖЕНИЕ и возвращает полученную строку.</p> <p>join ВЫРАЖЕНИЕ, СПИСОК</p>	10.3
map	<p>Вычисляет блок операторов БЛОК или ВЫРАЖЕНИЕ для всех элементов списка, заданного параметром СПИСОК, локально устанавливая значение специальной переменной \$_ равным значению элемента списка. Возвращает список элементов, составленный из результатов указанных вычислений для каждого элемента заданного списка. Операторы из БЛОК и ВЫРАЖЕНИЕ вычисляются в списковом контексте.</p> <p>map БЛОК СПИСОК</p> <p>map ВЫРАЖЕНИЕ, СПИСОК</p>	
reverse	<p>В списковом контексте возвращает список значений элементов заданного параметром СПИСОК списка в обратном порядке. В скалярном контексте соединяет все элементы списка в одну строку и возвращает строку, в которой символы расположены в обратном порядке.</p> <p>reverse СПИСОК</p>	

Таблица П1.5 (окончание)

Функция	Назначение и синтаксис	Ссылка
sort	<p>Сортирует список значений, определенный параметром СПИСОК. Если параметры БЛОК или ПОДПРОГРАММА не заданы, то используется стандартная процедура сравнения строковых данных; если заданы, то операторы блока или подпрограмма используются в качестве процедуры сравнения при сортировке элементов списка. Возвращает отсортированный список значений исходного списка.</p> <p>sort [ПОДПРОГРАММА] СПИСОК</p> <p>sort [БЛОК] СПИСОК</p>	9.5.3
unpack	<p>Выполняет обратные действия относительно действий функции pack(): берет строку (параметр ВЫРАЖЕНИЕ), представляющую двоичную структуру упакованного массива или списка значений, и распаковывает ее в соответствии с заданным параметром ШАБЛОН шаблоном. В списковом контексте возвращает массив полученных значений, в скалярном — значение первого полученного элемента массива. Шаблон должен быть такой же, как и в функции pack().</p> <p>unpack ШАБЛОН, ВЫРАЖЕНИЕ</p>	

Таблица П1.6. Функции обработки хеш-массивов

Функция	Назначение и синтаксис	Ссылка
delete	<p>Удаляет из хеш-массива ключ и ассоциированное с ним значение. Ключ можно задавать как вычисляемое значение выражения. Возвращает значение, ассоциированное с удаленным ключом, или неопределенное значение, если заданного ключа в хеш-массиве не существует.</p> <p>delete ВЫРАЖЕНИЕ</p>	
each	<p>Возвращает следующую пару ключ/значение из хеш-массива, определенного параметром ХЕШ. Можно использовать в цикле для организации чтения хеша. В списковом контексте возвращает пару ключ/значение в виде двухэлементного списка, в скалярном контексте — только ключ. Если перебраны все элементы хеша, то в списковом контексте возвращает пустой список, а в скалярном контексте — неопределенное значение undef; следующий вызов функции each() начинает новый итерационный процесс.</p> <p>each ХЕШ</p>	

Таблица П1.6 (окончание)

Функция	Назначение и синтаксис	Ссылка
<code>exists</code>	<p>Проверяет, существует ли в хеш-массиве заданный ключ. Если существует, то возвращает булево значение Истина, иначе — Ложь. Параметр может быть выражением, возвращающим ссылку на элемент хеша с заданным ключом, например, <code>\$hash{\$key}</code>.</p> <p><code>exists</code> ВЫРАЖЕНИЕ</p>	
<code>keys</code>	<p>В списковом контексте возвращает список, элементами которого являются все ключи хеш-массива, заданного параметром <code>ХЕШ</code>; в скалярном контексте возвращает количество ключей хеш-массива.</p> <p><code>keys</code> <code>ХЕШ</code></p>	9.5.3
<code>values</code>	<p>В списковом контексте возвращает список, элементами которого являются все значения хеш-массива, заданного параметром <code>ХЕШ</code>; в скалярном контексте возвращает количество значений хеш-массива.</p> <p><code>values</code> <code>ХЕШ</code></p>	

Таблица П1.7. Функции ввода/вывода

Функция	Назначение и синтаксис	Ссылка
<code>binmode</code>	<p>Подготавливает файл, ассоциированный с дескриптором <code>ДЕСКРИПТОР</code>, для чтения и записи в двоичном формате, а не в текстовом, в операционных системах, которые различают двоичные и текстовые файлы. В Unix эта функция не имеет никакого эффекта; необходима в MS-DOS и других архаичных системах, иначе двоичный файл может быть испорчен.</p> <p><code>binmode</code> <code>ДЕСКРИПТОР</code></p>	
<code>dbmclose</code>	<p>Разрывает связывание хеш-массива и файлом базы DBM. Заменена функцией <code>untie()</code>.</p> <p><code>dbmclose</code> <code>ХЕШ</code></p>	
<code>dbmopen</code>	<p>Создает связывание хеш-массива с файлом базы DBM, имя которого определяется значением параметра <code>ФАЙЛ_DB</code>. Заменена функцией <code>tie()</code>. Файл базы данных открывается в режиме, указанном параметром <code>РЕЖИМ</code>.</p> <p><code>dbmopen</code> <code>ХЕШ</code>, <code>ФАЙЛ_DB</code>, <code>РЕЖИМ</code></p>	
<code>die</code>	<p>Вне тела функции <code>eval()</code> выводит значения элементов списка, заданного параметром <code>СПИСОК</code>, в стандартный файл ошибок <code>STDERR</code> и завершает выполнение сценария Perl с текущим значением специальной переменной <code>\$!</code>.</p>	7.2

Таблица П1.7 (продолжение)

Функция	Назначение и синтаксис	Ссылка
	<p>Если используется в теле функции <code>eval()</code>, то сообщение об ошибке помещается в переменную <code>\$@</code> и функция <code>eval()</code> завершается с неопределенным значением. Такое поведение позволяет использовать функцию <code>die()</code> для генерирования исключительных состояний.</p> <p><code>die</code> СПИСОК</p>	
<code>eof</code>	<p>Тестирует файл, ассоциированный с дескриптором, заданным параметром <code>ДЕСКРИПТОР</code>, на конец файла. Возвращает 1, если следующая операция чтения из файла возвратит конец файла. Употребленная без параметра, использует в качестве параметра файл, для которого выполнялась последняя операция чтения. Если используется с пустыми скобками <code>eof()</code>, то тестирует символ конца самого последнего файла из списка файлов, переданных в сценарий через командную строку.</p> <p><code>eof</code> <code>ДЕСКРИПТОР</code></p> <p><code>eof ()</code></p> <p><code>eof</code></p>	
<code>fileno</code>	<p>Возвращает числовой системный дескриптор для файла, ассоциированного с дескриптором, заданным параметром <code>ДЕСКРИПТОР</code>.</p> <p><code>fileno</code> <code>ДЕСКРИПТОР</code></p>	
<code>flock</code>	<p>Блокирует файл, ассоциированный с дескриптором, заданным параметром <code>ДЕСКРИПТОР</code>, для выполнения другими пользователями операций, определенных параметром <code>ОПЕРАЦИЯ</code>. Возвращает булево значение Истина в случае успешной блокировки файла и Ложь в противном случае. Блокирует полностью весь файл, а не отдельные записи.</p> <p><code>flock</code> <code>ДЕСКРИПТОР</code>, <code>ОПЕРАЦИЯ</code></p>	
<code>format</code>	Объявляет формат, используемый для вывода функцией <code>write()</code> .	8.1
<code>getc</code>	<p>Возвращает следующий символ из файла, открытого для чтения и ассоциированного с дескриптором, заданным параметром <code>ДЕСКРИПТОР</code>. Если достигнут конец файла или произошла ошибка чтения, возвращает неопределенное значение <code>undef</code>. Если используется без имени дескриптора, то читает из файла стандартного ввода <code>STDIN</code>.</p> <p><code>getc</code> <code>ДЕСКРИПТОР</code></p> <p><code>getc</code></p>	7.2

Таблица П1.7 (продолжение)

Функция	Назначение и синтаксис	Ссылка
print	<p>Выводит список строковых данных в файл, ассоциированный с дескриптором, заданным параметром <code>ДЕСКРИПТОР</code>. Если этот параметр опущен, то вывод осуществляется в стандартный файл вывода <code>STDIN</code> или текущий файл, установленный функцией <code>select()</code>. Если отсутствует параметр <code>СПИСОК</code>, то выводится содержимое специальной переменной <code>\$_</code>.</p> <pre>print ДЕСКРИПТОР СПИСОК print СПИСОК print</pre>	7.2
printf	<p>Выводит форматированный список строковых данных в файл, ассоциированный с дескриптором, заданным параметром <code>ДЕСКРИПТОР</code>. Если этот параметр опущен, то вывод осуществляется в стандартный файл вывода <code>STDIN</code> или текущий файл, установленный функцией <code>select()</code>. Параметр <code>ФОРМАТ</code> представляет строку, содержащую символы форматирования, полностью совпадающие с символами форматирования системной функции Unix <code>printf(3)</code>.</p> <pre>printf ДЕСКРИПТОР ФОРМАТ, СПИСОК printf ФОРМАТ, СПИСОК</pre>	
read	<p>Чтение заданного количества байт в скалярную переменную, определяемую параметром <code>СКАЛЯР</code>, из файла, связанного с заданным параметром <code>ДЕСКРИПТОР</code> дескриптором. Возвращает истинное количество прочитанных байтов, 0 в случае конца файла и неопределенное значение <code>undef</code> при возникновении ошибки. Параметр <code>СМЕЩЕНИЕ</code> задает позицию в строковой переменной <code>СКАЛЯР</code>, с которой вставляются прочитанные байты (по умолчанию с начала строки данных скалярной переменной).</p> <pre>read ДЕСКРИПТОР, СКАЛЯР, КОЛИЧЕСТВО[, СМЕЩЕНИЕ]</pre>	7.2
readdir	<p>В скалярном контексте читает следующее имя элемента каталога, ассоциированного с дескриптором каталога (параметр <code>ДЕСКРИПТОР</code>) функцией <code>opendir()</code>. В списковом контексте возвращает имена всех оставшихся элементов (файлов и каталогов). Если больше не осталось прочитанных элементов, возвращает неопределенное значение в скалярном контексте и пустой список в списковом контексте.</p> <pre>readdir ДЕСКРИПТОР</pre>	7.5

Таблица П1.7 (продолжение)

Функция	Назначение и синтаксис	Ссылка
rewinddir	Устанавливает текущую позицию каталога, ассоциированного с дескриптором ДЕСКРИПТОР, на начало. rewinddir ДЕСКРИПТОР	7.5
seek	Устанавливает текущую позицию файла, ассоциированного с дескриптором ДЕСКРИПТОР, смещенной на заданное параметром ПОЗИЦИЯ число байт относительно положения, определяемого параметром ПОЛОЖЕНИЕ (0 – начало файла; 1 – текущая позиция; 2 – конец файла). Возвращает 1 при успешном выполнении операции позиционирования, 0 в противном случае. seek ДЕСКРИПТОР, ПОЗИЦИЯ, ПОЛОЖЕНИЕ	7.2
seekdir	Устанавливает текущую позицию каталога, ассоциированного со своим дескриптором (параметр ДЕСКРИПТОР), для функции readdir(). Параметр ПОЛОЖЕНИЕ должен быть значением, возвращаемым функцией telldir(). seekdir ДЕСКРИПТОР, ПОЛОЖЕНИЕ	
select	Возвращает дескриптор текущего установленного файла вывода. Если задан параметр ДЕСКРИПТОР, делает файл, ассоциированный с этим дескриптором текущим установленным файлом вывода. Это приводит к тому, что функции print() и write() без дескрипторов будут осуществлять вывод именно в этот файл. select [ДЕСКРИПТОР]	7.2
syscall	Осуществляет вызов системной команды, заданной первым элементом передаваемого функции списка (параметр СПИСОК); остальные элементы списка передаются этой команде в качестве параметров. В случае не успешного выполнения системной команды, возвращает -1 и в переменной \$! содержится информация об ошибке. syscall СПИСОК	
sysread	Аналогична функции read(), за исключением того, что осуществляет не буферизованный ввод данных, не используя буферы стандартной системы ввода/вывода stdio. sysread ДЕСКРИПТОР, СКАЛЯР, КОЛИЧЕСТВО[, СМЕЩЕНИЕ]	7.2
sysseek		7.2

Таблица П1.7 (продолжение)

Функция	Назначение и синтаксис	Ссылка
<code>syswrite</code>	<p>Запись заданного количества байт из скалярной переменной, определяемой параметром <code>СКАЛЯР</code>, в файл, ассоциированный с заданным параметром <code>ДЕСКРИПТОР</code> дескриптором. Возвращает истинное количество прочитанных байтов или неопределенное значение <code>undef</code> при возникновении ошибки. Параметр <code>СМЕЩЕНИЕ</code> задает позицию в строковой переменной <code>СКАЛЯР</code>, с которой записываются в файл данные (по умолчанию с начала строки данных скалярной переменной); отрицательное значение этого параметра определяет запись заданного количества байт, начиная с конца строки. Осуществляет не буферизованный вывод данных, не используя буферы стандартной системы ввода/вывода <code>stdio</code>.</p> <p><code>syswrite</code> <code>ДЕСКРИПТОР</code>, <code>СКАЛЯР</code>, <code>КОЛИЧЕСТВО</code>[, <code>СМЕЩЕНИЕ</code>]</p>	7.2
<code>tell</code>	<p>Возвращает текущую позицию в файле, ассоциированном с заданным параметром <code>ДЕСКРИПТОР</code> дескриптором. Если параметр опущен, то подразумевается последний файл, для которого была выполнена операция чтения данных.</p> <p><code>tell</code> [<code>ДЕСКРИПТОР</code>]</p>	7.2
<code>telldir</code>	<p>Возвращает текущую позицию в каталоге, ассоциированном с заданным параметром <code>ДЕСКРИПТОР</code> дескриптором.</p> <p><code>tell</code> <code>ДЕСКРИПТОР</code></p>	
<code>truncate</code>	<p>Изменяет длину файла, ассоциированного с заданным дескриптором, устанавливая ее равной значению параметра <code>ДЛИНА</code>. Файл может быть увеличен или уменьшен.</p> <p><code>truncate</code> <code>ДЕСКРИПТОР</code>, <code>ДЛИНА</code></p> <p><code>truncate</code> <code>ВЫРАЖЕНИЕ</code>, <code>ДЛИНА</code></p>	7.3
<code>warn</code>	<p>Выводит в стандартный файл ошибок <code>STDERR</code> значения элементов списка, определяемого параметром <code>СПИСОК</code>. Если он пуст, то выводит содержимое специальной переменной <code>\$@</code> (обычно после выполнения функции <code>eval</code>), добавляя строку <code>"\t...caught"</code>, а если и она пуста, то выводит сообщение <code>"Warning: Something's wrong"</code></p> <p><code>warn</code> <code>СПИСОК</code></p>	

Таблица П1.7 (окончание)

Функция	Назначение и синтаксис	Ссылка
write	<p>Осуществляет форматный вывод в файл, ассоциированный с заданным параметром ДЕСКРИПТОР дескриптором, в соответствии с определенным форматом. Если параметр не задан, то выводит в стандартный файл вывода STDOUT или в файл, выбранный функцией select().</p> <p>write [ДЕСКРИПТОР]</p>	8

Таблица П1.8. Функции для работы с файлами и каталогами

Функция	Назначение и синтаксис	Ссылка
-X	<p>Обозначает большую группу именованных унарных операций проверки параметров и состояний файлов, заданных либо дескриптором, либо именем. Имена операций состоят из дефиса, за которым следует одна буква. Все они возвращают значение 1, если файл обладает проверяемым свойством, 0 в противном случае и неопределенное значение, если файл не существует. Если параметр опущен, то используется значение специальной переменной \$ _.</p> <p>-X ДЕСКРИПТОР</p> <p>-X ИМЯ_ФАЙЛА</p> <p>-X</p>	7.4
chdir	<p>Изменение текущего рабочего каталога на каталог, определяемый значением параметра ВЫРАЖЕНИЕ. Если параметр опущен, домашний каталог становится текущим. Возвращает булево значение Истина в случае успешного выполнения операции замены текущего каталога и Ложь в противном случае.</p> <p>chdir [ВЫРАЖЕНИЕ]</p>	7.5
chmod	<p>Изменение режима доступа к файлам, имена которых представлены в списке параметров. Первый элемент списка является восьмеричным числом, определяющий новый режим доступа. Возвращает количество файлов, для которых операция изменения режима доступа завершилась успешно.</p> <p>chmod СПИСОК</p>	7.3
chown	<p>Изменение владельца и группы владельца файлов, имена которых представлены в списке параметров. Первые два элемента списка являются числами, представляющими uid и gid. Возвращает количество файлов, для которых операция изменения владельца и группы завершилась успешно.</p> <p>chown СПИСОК</p>	7.3

Таблица П1.8 (продолжение)

Функция	Назначение и синтаксис	Ссылка
chroot	<p>Определяет новый корневой каталог для всех относительных (начинающихся с косой черты "/") имен файлов процесса пользователя и порожденных им процессов. Не меняет текущий рабочий каталог. В отсутствии параметра используется значение специальной переменной \$_. Может вызываться только суперпользователем.</p> <p>chroot ИМЯ_КАТАЛОГА</p>	
close	<p>Закрывает файл или программный канал, ассоциированный с дескриптором, заданным параметром ДЕСКРИПТОР. Возвращает булево значение Истина, если успешно очищен буфер и закрыт системный файловый дескриптор. Без параметра закрывает файл или программный канал, ассоциированный с текущим дескриптором, выбранным функцией select ().</p> <p>close [ДЕСКРИПТОР]</p>	7.2
closedir	<p>Закрывает каталог, ассоциированный с дескриптором каталога, заданным параметром ДЕСКРИПТОР. Возвращает булево значение Истина, если каталог успешно закрыт.</p> <p>closedir ДЕСКРИПТОР</p>	7.5
fcntl	<p>Реализует системную команду Unix fcntl(2). Перед использованием следует получить доступ к определениям системных констант оператором use Fcntl; Возвращаемое значение: если системная функция возвращает -1, то функция Perl — неопределенное значение; если системная функция возвращает 0, то функция Perl строку "0 but true"; если системная функция возвращает какое-либо другое значение, и функция Perl возвращает это же значение.</p> <p>fcntl ДЕСКРИПТОР, ФУНКЦИЯ, СКАЛЯР</p>	
glob	<p>Возвращает найденные в текущем каталоге файлы, имена которых удовлетворяют заданному шаблону (с использованием метасимволов Unix "*", "?"). Значением выражения должна быть строка, содержащая шаблон имен файлов.</p> <p>glob ВЫРАЖЕНИЕ</p>	
ioctl	<p>Реализует системную команду Unix ioctl(2). Перед использованием следует получить доступ к определениям системных констант оператором require "ioctl.ph";</p>	

Таблица П1.8 (продолжение)

Функция	Назначение и синтаксис	Ссылка
	<p>Возвращаемое значение: если системная функция возвращает -1, то функция Perl неопределенное значение; если системная функция возвращает 0, то функция Perl строку "0 but true"; если системная функция возвращает какое-либо другое значение, то и функция Perl возвращает это же значение.</p> <p>fcntl ДЕСКРИПТОР, ФУНКЦИЯ, СКАЛЯР</p>	
link	<p>Создает новую "жесткую" ссылку на файл, заданный параметром СТАРЫЙ. Возвращает булево значение Истина в случае успешного создания ссылки и Ложь в противном случае.</p> <p>link СТАРЫЙ, НОВЫЙ</p>	7.3
lstat	<p>Возвращает список значений полей структуры индексного дескриптора символической ссылки на файл. Если параметр опущен, то используется значение специальной переменной \$_. lstat [ДЕСКРИПТОР] lstat [ВЫРАЖЕНИЕ]</p>	7.4
mkdir	<p>Создание нового каталога с именем, заданным в параметре КАТАЛОГ, и режимом доступа, определяемым параметром РЕЖИМ. При успешном создании каталога возвращает булево значение Истина, в противном случае Ложь и в переменную \$! заносится сообщение об ошибке. mkdir КАТАЛОГ, РЕЖИМ</p>	7.5
open	<p>Открывает файл, имя которого является значением параметра ВЫРАЖЕНИЕ, и ассоциирует (связывает) его с дескриптором, определяемым параметром ДЕСКРИПТОР. Если параметр ВЫРАЖЕНИЕ не задан, то используется значение скалярной переменной с тем же именем, что и дескриптор. open ДЕСКРИПТОР[, ВЫРАЖЕНИЕ]</p>	7.2
opendir	<p>Открытие каталога, имя которого равно значению параметра ВЫРАЖЕНИЕ, и связывание его с дескриптором, определяемым параметром ДЕСКРИПТОР. Имена дескрипторов каталогов хранятся в собственном пространстве имен таблицы имен Perl. opendir ДЕСКРИПТОР, ВЫРАЖЕНИЕ</p>	7.5
readlink	<p>Возвращает значение символической ссылки, определяемой параметром ВЫРАЖЕНИЕ, если символические ссылки реализуются операционной системой; в противном случае — фатальная ошибка.</p>	

Таблица П1.8 (продолжение)

Функция	Назначение и синтаксис	Ссылка
	<p>Если при получении значения символической ссылки были получены системные ошибки, возвращает неопределенное значение и в специальную переменную \$! заносится сообщение об ошибке. Если параметр опущен, используется значение переменной \$ _.</p> <p><code>readlink [ВЫРАЖЕНИЕ]</code></p>	
<code>rename</code>	<p>Переименовывает файл. Возвращает 1 в случае успешного переименования и 0 в противном случае.</p> <p><code>rename СТАРОЕ_ИМЯ, НОВОЕ_ИМЯ</code></p>	7.3
<code>rmdir</code>	<p>Удаление каталога, заданного своим именем. Каталог удаляется, если он пустой. Возвращает 1 в случае успешного удаления и 0 в противном, сохраняя в переменной \$! сообщение об ошибке. Если параметр опущен, используется значение переменной \$ _.</p> <p><code>rmdir ИМЯ_КАТАЛОГА</code></p>	7.5
<code>stat</code>	<p>Возвращает список значений тринадцати полей структуры индексного дескриптора файла, заданного либо своим дескриптором (параметр ДЕСКРИПТОР), либо значением выражения (параметр ВЫРАЖЕНИЕ). Если параметр опущен, то используется значение специальной переменной \$ _ . В случае ошибки возвращает пустой список.</p> <p><code>stat [ДЕСКРИПТОР]</code></p> <p><code>stat [ВЫРАЖЕНИЕ]</code></p>	7.4
<code>symlink</code>	<p>Создание символической ссылки (параметр НОВОЕ_ИМЯ) на файл (параметр СТАРОЕ_ИМЯ). Возвращает 1 в случае успешного создания символической ссылки и 0 в противном случае.</p> <p><code>symlink СТАРОЕ_ИМЯ, НОВОЕ_ИМЯ</code></p>	7.3
<code>umask</code>	<p>Устанавливает маску режима доступа процесса, заданную значением параметра ВЫРАЖЕНИЕ (восьмеричное число), и возвращает предыдущее значение маски режима доступа.</p> <p><code>umask ВЫРАЖЕНИЕ</code></p>	
<code>unlink</code>	<p>Удаление файлов, определенных параметром СПИСОК. Возвращает количество успешно удаленных файлов.</p> <p><code>unlink СПИСОК</code></p>	7.3

Таблица П1.8 (окончание)

Функция	Назначение и синтаксис	Ссылка
utime	<p>Изменяет время доступа и модификации файлов, определенных в параметре СПИСОК. Первые два элемента списка должны быть числовыми и представлять, соответственно, время последнего доступа и время последней модификации. Возвращает количество файлов, для которых успешно была выполнена указанная операция.</p> <p>utime СПИСОК</p>	7.3

Таблица П1.9. Функции, относящиеся к управлению выполнением программы

Функция	Назначение и синтаксис	Ссылка
caller	<p>Возвращает контекст выполняемой подпрограммы, если вызвана без параметра. В скалярном контексте возвращается имя пакета, в котором находится вызывающая данную подпрограмма (т. е. если наша подпрограмма вызывается из другой подпрограммы, функции eval() или require()), или неопределенное значение в противном случае. В списковом контексте возвращает список с разнообразными сведениями. Можно передать в качестве параметра число, определяющее уровень вложенности получаемой информации, т. е. к какой подпрограмме она относится: 0 — наша подпрограмма; 1 — подпрограмма, вызывающая нашу, и т. д. Полный синтаксис выглядит так (см. документацию Perl):</p> <pre>(\$package, \$filename, \$line, \$subroutine, \$hasargs, \$wantarray, \$evaltext, \$is_require) = caller(\$i);</pre>	
die	<p>Вне тела функции eval() выводит значения элементов списка, заданного параметром СПИСОК, в стандартный файл ошибок STDERR и завершает выполнение сценария Perl с текущим значением специальной переменной \$!. Если используется в теле функции eval(), то сообщение об ошибке помещается в переменную \$@ и функция eval() завершается с неопределенным значением. Такое поведение позволяет использовать функцию die() для генерирования исключительных состояний.</p> <p>die СПИСОК</p>	10.3
do	<p>В действительности не является функцией. Выполняет последовательность операторов Perl, заданных в блоке БЛОК, и возвращает значение последнего выполненного оператора.</p>	11.1

Таблица П1.9 (продолжение)

Функция	Назначение и синтаксис	Ссылка
	<p>Если параметром является ВЫРАЖЕНИЕ, то рассматривает его значение как имя файла, который загружает и выполняет как последовательность операторов Perl. Если не удается прочитать файл, возвращает неопределенное значение и помещает в переменную \$! сообщение об ошибке; если не может откомпилировать файл, то также возвращает неопределенное значение и помещает сообщение об ошибке в переменную \$@. Если содержимое файла успешно откомпилировано, то возвращает значение последнего выполненного оператора.</p> <p>do БЛОК</p> <p>do ВЫРАЖЕНИЕ</p>	
dump	<p>Создание дампа ядра, которое системной командой undump может быть преобразовано в двоичный выполняемый файл. Если задан параметр МЕТКА, то при повторном вычислении двоичного файла, сначала выполнится оператор goto МЕТКА;. Функция устаревшая — существует компилятор из Perl в C, который выполняет ее функцию.</p> <p>dump МЕТКА</p>	
eval	<p>Если параметром является ВЫРАЖЕНИЕ, то осуществляется синтаксический разбор его строкового значения как программы Perl, которая, в случае успешной компиляции, немедленно выполняется. Компиляция мини-программы осуществляется каждый раз, когда вычисляется функция, и вне окружающего функцию контекста. Если параметр не задан, используется значение специальной переменной \$_.</p> <p>Если параметром является блок операторов БЛОК, то его содержимое компилируется один раз, тогда же, когда компилируются и окружающие функции операторы, и выполняется в контексте основной программы Perl.</p> <p>При обеих формах вызова функция возвращает значение последнего выполненного оператора мини-программы.</p> <p>eval ВЫРАЖЕНИЕ</p> <p>eval БЛОК</p>	10.3
exit	<p>Немедленно прекращает выполнение программы с кодом завершения, равным значению параметра ВЫРАЖЕНИЕ. Если параметр не задан, программа завершается с кодом 0.</p> <p>exit [ВЫРАЖЕНИЕ]</p>	

Таблица П1.9 (окончание)

Функция	Назначение и синтаксис	Ссылка
return	Выход из подпрограммы, функции eval() или do ФАЙЛ с кодом завершения равным значению параметра ВЫРАЖЕНИЕ. Если в подпрограмме, функции eval() или do ФАЙЛ не используется эта функция, то они завершаются с кодом, равным значению последнего выполненного оператора. return ВЫРАЖЕНИЕ	9.2.3; 11.2
sub	В действительности это не функция, а объявление подпрограммы: анонимной (без параметра ИМЯ, но с блоком операторов БЛОК), прототипа (без блока операторов БЛОК) и нормальной. sub БЛОК sub ИМЯ sub ИМЯ БЛОК	11.1
wantarray	Возвращает значение Истина, если контексту выполняемой подпрограммы требуется списковое значение, и Ложь, если требуется скалярное значение. Возвращает неопределенное значение, если подпрограмме требуется безразличный (void) контекст. wantarray	

Замечание

Для управления последовательностью выполнения программы Perl также используются операции continue, goto, last, next и redo, которые не являются истинными функциями, и поэтому мы не включили их в табл. П1.9.

Таблица П1.10. Функции, относящиеся к области видимости переменных

Функция	Назначение и синтаксис	Ссылка
caller	См. описание в табл. П1.9.	
import	Вставляет пространство имен одного модуля в другой. Это не встроенная функция, а всего лишь метод, наследуемый от модуля (параметр МОДУЛЬ), которому необходимо экспортировать свои имена (параметр СПИСОК) в другой модуль. import МОДУЛЬ СПИСОК	12.3
local	Создает временные копии для глобальных переменных, перечисленных в списке ее параметров (динамическая область видимости), видимые в блоке операторов {...}, файле или функции eval().	11.3.2

Таблица П1.10 (окончание)

Функция	Назначение и синтаксис	Ссылка
	<p>Если в списке более одной переменной, то он должен быть заключен в круглые скобки. Созданные копии не являются истинными локальными переменными.</p> <p><code>local ПЕРЕМЕННАЯ</code></p> <p><code>local (СПИСОК)</code></p>	
<code>my</code>	<p>Объявляет переменные, перечисленные в списке параметров, как локальные (лексически) для блока операторов <code>{...}</code>, файла или функции <code>eval()</code>. Если в списке более одной переменной, то он должен быть заключен в круглые скобки. Созданные переменные являются истинными локальными переменными.</p> <p><code>my ПЕРЕМЕННАЯ</code></p> <p><code>my (СПИСОК)</code></p>	11.3.1
<code>package</code>	<p>Определяет отдельное глобальное пространство имен (пакет): все неопределенные динамические идентификаторы (включая те, которые объявлены через <code>local()</code>, но не <code>my()</code>) будут храниться в нем. Для доступа к ним вне пакета следует указывать префикс, представляющий имя пакета с последующими двумя символами двоеточий <code>::</code>. Область видимости переменных пакета распространяется до конца блока операторов, в котором расположен пакет или до нового объявления пакета. Если опущено имя пакета, то предписывает, чтобы все идентификаторы были определены, включая имена функций.</p> <p><code>package [ИМЯ_ПАКЕТА]</code></p>	12.1
<code>use</code>	<p>Загружает модуль во время компиляции; если модуль не доступен, то компиляция всей программы прекращается.</p> <p><code>use МОДУЛЬ СПИСОК</code></p> <p><code>use МОДУЛЬ</code></p> <p><code>use МОДУЛЬ ВЕРСИЯ СПИСОК</code></p> <p><code>use ВЕРСИЯ</code></p>	12.3.1

Таблица П1.11. Функции, относящиеся к модулям Perl

Функция	Назначение и синтаксис	Ссылка
<code>do</code>	См. описание в табл. П1.9.	11.1
<code>import</code>	См. описание в табл. П1.10.	12.3

Таблица П1.11 (окончание)

Функция	Назначение и синтаксис	Ссылка
no	Отказ от импортирования символических имен (параметр СПИСОК) модуля (параметр МОДУЛЬ) во время компиляции программы. Является антонимом функции use (см. ее описание в табл. П1.10). no МОДУЛЬ СПИСОК	12.3.3
package	См. описание в табл. П1.10.	12.1
require	Загружает модуль или класс, имя которого является значением параметра ВЫРАЖЕНИЕ, во время выполнения программы. Если параметр опущен, то используется значение специальной переменной \$_. Если значением параметра ВЫРАЖЕНИЯ является число, то оно определяет минимальную версию Perl, необходимую для выполнения программы. require [ВЫРАЖЕНИЕ]	12.2.1
use	См. описание в табл. П1.10.	12.3.1

Таблица П1.12. Функции, относящиеся к классам и объектно-ориентированным технологиям

Функция	Назначение и синтаксис	Ссылка
bless	После выполнения функции <code>bless()</code> ссылка на субъект Perl (переменную, массив или хеш), определенная параметром ССЫЛКА, становится ссылкой на объект, определяемый пакетом с именем, равным значению параметра ИМЯ_КЛАССА. Если этот параметр не задан, то подразумевается текущий пакет. Возвращаемое значение — ссылка на объект. <code>bless ССЫЛКА, ИМЯ_КЛАССА</code> <code>bless ССЫЛКА</code>	13.1
dbmclose	См. описание в табл. П1.7.	
dbmopen	См. описание в табл. П1.7.	
package	См. описание в табл. П1.10.	12.1
ref	Если параметр ВЫРАЖЕНИЕ является ссылкой, то возвращает тип субъекта ссылки; в противном случае Ложь. Если параметр опущен, то используется переменная \$_. Возвращаемым значением может быть: REF, SCALAR, ARRAY, HASH, CODE, GLOB или имя пакета, если ссылка была переопределена функцией <code>bless()</code> . <code>ref [ВЫРАЖЕНИЕ]</code>	9.1; 13.1

Таблица П1.12 (окончание)

Функция	Назначение и синтаксис	Ссылка
<code>tie</code>	<p>Связывает переменную, определенную параметром ПЕРЕМЕННАЯ, с классом ИМЯ_КЛАССА, который будет обеспечивать реализацию методов доступа для этой переменной, список которых зависит от типа связываемой переменной (скаляр, массив, хеш, дескриптор файла). Операции доступа к связанной переменной (например, получение и присваивание значения для скалярной переменной или присваивание значения элементу массива и т. п.) автоматически вызывают соответствующие методы определенного класса. Элементы параметра СПИСОК передаются соответствующим конструкторам этого класса в зависимости от типа связываемой переменной: <code>TIESCALAR()</code>, <code>TIEARRAY()</code>, <code>TIEHASH()</code> или <code>TIEHANDLE()</code>. Возвращаемым значением функции является объект класса, к которому привязывается переменная.</p> <p><code>tie ПЕРЕМЕННАЯ, ИМЯ_КЛАССА, СПИСОК</code></p>	
<code>tied</code>	<p>Возвращает объект класса, с которым связана переменная, определенная параметром ПЕРЕМЕННАЯ, с помощью функции <code>tie()</code>.</p> <p><code>\$object = tied ПЕРЕМЕННАЯ</code></p>	
<code>untie</code>	<p>Разрывает созданную функцией <code>tie()</code> связь между переменной, определенной параметром ПЕРЕМЕННАЯ, и классом, реализующим методы доступа к ней.</p> <p><code>untie ПЕРЕМЕННАЯ</code></p>	
<code>use</code>	См. описание в табл. П1.10.	12.3.1

Таблица П1.13. Разные функции

Функция	Назначение и синтаксис	Ссылка
<code>defined</code>	<p>Проверяет, не равно ли значение выражения, заданного параметром ВЫРАЖЕНИЕ, неопределенной величине <code>undef</code>; если не равно, то возвращает булево значение Истина, в противном случае Ложь. Если параметр функции опущен, то используется значение специальной переменной <code>\$_</code>.</p> <p><code>defined [ВЫРАЖЕНИЕ]</code></p>	10.3
<code>formline</code>	<p>Внутренняя функция, используемая операцией <code>format</code>, но ее можно вызывать и явным образом. Выводит значения элементов списка, представленного параметром СПИСОК, в специальную переменную <code>\$^A</code> — аккумулятор форматного вывода.</p>	

Таблица П1.13 (окончание)

Функция	Назначение и синтаксис	Ссылка
	<p>Вывод осуществляется в соответствии с заданной параметром ШАБЛОН строкой шаблонов. Управляющие символы "~" и "~~" игнорируются. Всегда возвращает значение Истина.</p> <p>formline ШАБЛОН, СПИСОК</p>	
reset	<p>Используется для очистки переменных и переустановки команды поиска ?образец? (обычно в блоке continue цикла). Значением параметра ВЫРАЖЕНИЕ должна быть строка, представляющая символ, с которого начинается имя переменной (допустим дефис "-" для задания интервала символов). Если параметр опущен, то переустанавливает команду поиска ?образец?, чтобы она могла осуществить очередной поиск в соответствии с заданным образцом. Действует только на переменные текущего пакета. Всегда возвращает значение 1.</p> <p>reset [ВЫРАЖЕНИЕ]</p>	
scalar	<p>Вычисляет выражение, определяемое параметром ВЫРАЖЕНИЕ, в скалярном контексте и возвращает вычисленное значение.</p> <p>scalar ВЫРАЖЕНИЕ</p>	
undef	<p>Унарная операция, присваивающая lvalue, представленным параметром ВЫРАЖЕНИЕ, неопределенное значение. Применяется к скалярным переменным (\$), массивам скаляров (@), хешам (%), подпрограммам (&) и глобальным типам (*). Всегда возвращает неопределенное значение. Вызванная без параметра, просто вычисляет неопределенное значение, которое можно использовать, например, в качестве возвращаемого значения функции.</p> <p>undef [ВЫРАЖЕНИЕ]</p>	Пример 11.7
vec	<p>Рассматривает строковое значение параметра ВЫРАЖЕНИЕ как последовательность целых чисел без знака и возвращает значение битового поля, номер которого определяется значением параметра СМЕЩЕНИЕ (первое поле в строке всегда имеет номер 0). Параметр БИТЫ_ПОЛЯ задает ширину битового поля в битах. Может использоваться в качестве левого значения операции присваивания (в этом случае параметры должны быть заключены в скобки).</p> <p>vec ВЫРАЖЕНИЕ, СМЕЩЕНИЕ, БИТЫ_ПОЛЯ</p>	

Таблица П1.14. Функции для работы с процессами и группами процессов

Функция	Назначение и синтаксис	Ссылка
alarm	<p>Определяет количество секунд (параметр СЕКУНДЫ), через которое должен быть выдан сигнал SIGALARM. Если параметр отсутствует, используется значение специальной переменной \$ _.</p> <p>alarm [СЕКУНДЫ]</p>	
exec	<p>Выполняет заданную параметром СПИСОК команду, прекращая дальнейшее выполнение программы Perl. Никогда не возвращает кода возврата выполнения команды, только в случае, если команда не существует, возвращает булево значение Ложь. Если СПИСОК состоит из более чем одного элемента, вызывает системную команду <code>execvp(3)</code> и передает ей в качестве параметров значения списка, которая вызывает заданную первым элементом списка команду, интерпретируя оставшиеся элементы как ее параметры. Если список представлен одной скалярной переменной или массивом из одного элемента, то его значение проверяется на наличие метасимволов командного интерпретатора shell. Если таковые обнаружены, то вся строка передается анализатору shell (в Unix это <code>/bin/sh -c</code>); в противном случае она разбивается на слова и передается в качестве параметра системной команде <code>execvp()</code>. В системной переменной \$0 сохраняется имя выполняемой команды. В форме с параметром ПРОГРАММА выполняет команду, заданную этим параметром, а в системную переменную \$0 заносится содержимое первого элемента списка. Таким образом можно скрыть от программы Perl имя истинной выполняемой команды.</p> <p>exec СПИСОК</p> <p>exec ПРОГРАММА СПИСОК</p>	
fork	<p>Осуществляет вызов системной функции <code>fork(2)</code>, создающей новый процесс. Возвращает идентификатор порожденного процесса в родительский процесс, значение 0 в порожденный процесс и неопределенное значение undef, если не удалось создать новый процесс.</p> <p>fork</p>	
getpgrp	<p>Возвращает текущую группу процесса с заданным идентификатором PID. Если значение параметра PID равно 0, или он опущен, то возвращается текущая группа текущего процесса.</p> <p>getpgrp PID</p>	
getppid	<p>Возвращает идентификатор процесса (ID) родительского процесса.</p> <p>getppid</p>	

Таблица П1.14 (продолжение)

Функция	Назначение и синтаксис	Ссылка
getpriority	Возвращает текущий приоритет процесса, группы процесса или пользователя (см. описание системной функции <code>getpriority(2)</code>). <code>getpriority WHICH, WHO</code>	
kill	Посылает сигнал процессам, определенным в элементах списка параметра СПИСОК. Первым элементом списка должен быть посылаемый сигнал; если он отрицательный, то уничтожаются группы процессов, а не сами процессы. Возвращает количество процессов, которым передан сигнал. <code>kill СПИСОК</code>	
pipe	Открывает пару соединенных дескрипторов файлов, образуя программный канал (pipe). Записанные в один файл данные, можно прочитать из другого. При передаче данных используются буферы стандартной системы ввода/вывода <code>stdio</code> . <code>pipe ДЕСКРИПТОР_ЧТЕНИЯ, ДЕСКРИПТОР_ЗАПИСИ</code>	
setpgrp	Устанавливает для процесса с заданным идентификатором (параметр PID) текущую группу равной значению параметра PGRP. Значение 0 идентификатора процесса соответствует текущему процессу. Если оба параметра опущены, то по умолчанию их значения принимаются равными 0. <code>setpgrp PID, PGRP</code>	
setpriority	Устанавливает текущий приоритет процесса, группы процесса или пользователя (см. описание системной функции <code>setpriority(2)</code>). <code>setpriority WHICH, WHO, PRIORITY</code>	
sleep	Приостанавливает выполнение программы Perl на заданное значением параметра ВЫРАЖЕНИЕ количество секунд или навсегда, если параметр отсутствует. Выходит из состояния ожидания раньше указанного времени, если процесс получает сигнал <code>GIGALARM</code> . Возвращает действительное количество секунд нахождения программы в состоянии ожидания. <code>sleep ВЫРАЖЕНИЕ</code>	
system	Аналогична функции <code>exec()</code> , но для выполнения команды порождает новый процесс, окончания которого ожидает родительский процесс, прежде чем продолжить свое выполнение. Все, что сказано относительно параметров функции <code>exec()</code> , распространяется и на параметры функции <code>system()</code> .	

Таблица П1.14 (окончание)

Функция	Назначение и синтаксис	Ссылка
	Возвращает такой же код завершения команды, что и функция <code>wait()</code> ; для получения истинного кода завершения полученное значение следует разделить на 256. <code>system</code> СПИСОК <code>system ПРОГРАММА СПИСОК</code>	
<code>times</code>	Возвращает четырехэлементный список, содержащий пользовательское и системное время для процесса и порожденного им процесса. <code>(\$user,\$system,\$cuser,\$csystem) = times;</code>	
<code>wait</code>	Ожидает завершения порожденного процесса и возвращает идентификатор завершенного порожденного процесса или <code>-1</code> в случае, если порожденных процессов не существует. В специальной переменной <code>\$?</code> сохраняется статус завершения. <code>wait</code>	
<code>waitpid</code>	Ожидает завершения процесса с заданным в параметре <code>PID</code> идентификатором процесса (<code>-1</code> означает любой процесс) и возвращает идентификатор завершенного порожденного процесса или <code>-1</code> в случае, если порожденных процессов не существует. Параметр <code>FLAGS</code> представляет набор флагов, уточняющих действия функции. Например, флаг <code>WHOANG</code> означает не блокирующее ожидание завершения любого процесса. Во всех системах реализован флаг <code>0</code> , означающий блокирующий вызов. В специальной переменной <code>\$?</code> сохраняется статус завершения. <code>waitpid PID, FLAGS</code>	

Таблица П1.15. Низкоуровневые функции работы с сокетами

Функция	Назначение и синтаксис	Ссылка
<code>accept</code>	Принимает входящее подключение через сокет; работает аналогично системной команде <code>accept(2)</code> . Возвращает упакованный адрес сокета в случае успешного подключения; иначе булево значение Ложь. <code>accept НОВЫЙ_СОКЕТ, РОДОВОЙ_СОКЕТ</code>	
<code>bind</code>	Назначает сокету, определенному параметром <code>СОКЕТ</code> , сетевой адрес, заданный параметром <code>ИМЯ</code> . Имя сокета представляет собой упакованный адрес сокета соответствующего типа.	

Таблица П1.15 (продолжение)

Функция	Назначение и синтаксис	Ссылка
	Возвращает булево значение Истина в случае успешного назначения и Ложь в противном случае. <code>bind СОКЕТ, ИМЯ</code>	
<code>connect</code>	Осуществляет подключение сокета, определенного параметром СОКЕТ, к удаленному сокету, имя которого задано параметром ИМЯ. Имя удаленного сокета представляет собой упакованный адрес сокета соответствующего типа. Возвращает булево значение Истина в случае успешного назначения и Ложь в противном случае. <code>connect СОКЕТ, ИМЯ</code>	
<code>getpeername</code>	Возвращает упакованный адрес удаленного сокета, к которому подключен СОКЕТ функцией <code>connect()</code> . <code>getpeername СОКЕТ</code>	
<code>getsockname</code>	Возвращает упакованный адрес сокета программы, который подключен функцией <code>connect()</code> к удаленному сокету. <code>getsockname СОКЕТ</code>	
<code>getsockopt</code>	Возвращает значение затребованной опции (параметр ИМЯ_ОПЦИИ) сокета (параметр СОКЕТ) заданного уровня (параметр УРОВЕНЬ) или неопределенное значение в случае возникновения ошибки. <code>getsockopt СОКЕТ, УРОВЕНЬ, ИМЯ_ОПЦИИ</code>	
<code>listen</code>	Включает режим приема для указанного сокета, регистрируя его как сервер. Возвращает булево значение Истина в случае успешного включения и Ложь в противном случае. <code>listen СОКЕТ, РАЗМЕР_ОЧЕРЕДИ</code>	
<code>recv</code>	Получение заданного значением параметра ДЛИНА количества байтов через указанный сокет (параметр СОКЕТ) и сохранение их в скалярной переменной СКАЛЯР. Возвращает адреса удаленного сокета, из которого прочитаны данные, или неопределенное значение в случае возникновения ошибки при получении данных. В действительности вызывает С-функцию <code>recvfrom()</code> ; параметр ФЛАГИ полностью соответствует аналогичному параметру указанной функции С. <code>recv СОКЕТ, СКАЛЯР, ДЛИНА, ФЛАГИ</code>	

Таблица П1.15 (окончание)

Функция	Назначение и синтаксис	Ссылка
send	<p>Посылает строку сообщения (параметр СООБЩЕНИЕ) через сокет, заданный параметром СОКЕТ. Параметр ФЛАГИ имеет тот же смысл, что и при вызове системной функции с одноименным названием. Если сокет не подключен к удаленному сокету, то параметр АДРЕС определяет адрес сокета, к которому следует подключиться. Возвращает количество переданных символов или неопределенное значение в случае возникновения ошибки.</p> <p>send СОКЕТ, СООБЩЕНИЕ, ФЛАГИ[, АДРЕС]</p>	
setsockopt	<p>Устанавливает заданную параметром ИМЯ_ОПЦИИ опцию сокета. Возвращает неопределенное значение undef в случае возникновения ошибки.</p> <p>setsockopt СОКЕТ, УРОВЕНЬ, ИМЯ_ОПЦИИ, ЗНАЧЕНИЕ</p>	
shutdown	<p>Закрывает указанный в параметре СОКЕТ сокет на выполнение определенных операций, задаваемых параметром ДЕЙСТВИЕ: 0 — прекращается чтение, 1 — прекращается запись, 2 — прекращается использование сокета.</p> <p>shutdown СОКЕТ, ДЕЙСТВИЕ</p>	
socket	<p>Открывает сокет указанного типа и ассоциирует его с дескриптором сокета, заданного параметром СОКЕТ. Параметры ОБЛАСТЬ, ТИП и ПРОТОКОЛ аналогичны таким же параметрам при вызове системной функции с таким же именем. Перед использованием этой функции следует оператором use Socket; импортировать необходимые определения.</p> <p>socket СОКЕТ, ОБЛАСТЬ, ТИП, ПРОТОКОЛ</p>	
socketpair	<p>Создает безымянную пару двунаправленных сокетов указанного типа в заданной области. Параметры ОБЛАСТЬ, ТИП и ПРОТОКОЛ аналогичны таким же параметрам при вызове системной функции с таким же именем. Возвращает булево значение Истина в случае успешного создания пары сокетов.</p> <p>socketpair СОКЕТ1, СОКЕТ1, ОБЛАСТЬ, ТИП, ПРОТОКОЛ</p>	

Замечание

Большинство программистов Perl для создания сокетов и управления ими предпочитает использовать не низкоуровневые встроенные функции из табл.

П1.15, а работать с функциями высокоуровневого интерфейса модулей IO::Socket::INET и IO::Socket::UNIX.

Таблица П1.16. Функции для работы со временем и датой

Функция	Назначение и синтаксис	Ссылка
gmtime	<p>Преобразует значение даты и времени, полученное функцией time() в 9-элементный массив, соответствующий временной зоне Гринвича. Все элементы массива являются числами. Месяцы (\$mon) нумеруются целыми числами от 0 до 11; дни недели (\$wday) целыми от 0 до 6, причем 0 соответствует воскресенью; год (\$year) отсчитывается от года 1900. В скалярном контексте возвращает такую же структуру даты, что и системная функция ctime(3): "Thu Oct 13 04:54:34 2000"</p> <p>(\$sec, \$min, \$hour, \$mday, \$mon, \$year, \$wday, \$yday, \$isdst) = gmtime(time);</p>	
localtime	<p>Полностью соответствует функции gmtime() за исключением того, что значения элементов возвращаемого массива соответствуют зоне местного времени, а не гринвичской.</p> <p>(\$sec, \$min, \$hour, \$mday, \$mon, \$year, \$wday, \$yday, \$isdst) = localtime(time);</p>	
time	<p>Возвращает число секунд, прошедших от начала эпохи — 1 января 1970 года.</p> <p>time</p>	
times	<p>Возвращает четырехэлементный массив, содержащий пользовательское и системное время для процесса и порожденного им процесса.</p> <p>(\$user, \$system, \$cuser, \$csystem) = times;</p>	

Получение информации из системных файлов

В языке Perl реализован большой набор функций, позволяющий получить информацию из разных системных файлов. По существу, все эти функции являются "двойниками" соответствующих системных функций с такими же именами. Функции с именами, начинающимися с get, предназначены для

получения и представления в программе Perl информации из одной записи определенного системного файла.

Группа функций `getgr*()` получает информацию из записей файла групп `/etc/group`, а функции группы `getpw*()` — из файла паролей `/etc/passwd`. (Здесь, и в дальнейшем, символ "*" означает любую последовательность букв.) В списковом контексте функции этих групп возвращают следующие массивы, которые полностью соответствуют структуре записей файлов, обрабатываемых этими функциями (см. документацию по операционной системе UNIX):

```
($name, $passwd, $uid, $gid,
    $quota, $comment, $gcos, $dir, $shell, $expire) = getpw*
($name, $passwd, $gid, $members) = getgr*
```

Если какие-то поля записи не поддерживаются операционной системой, то значение соответствующих элементов в возвращаемом списке равно пустой строке. Сказанное может относиться к элементам `$quota`, `$comment`, `$gcos` и `$expire` записи файла паролей. Элемент `$members` записи файла групп содержит разделенные пробелами регистрационные имена пользователей группы.

В скалярном контексте эти функции возвращают первый элемент списка — регистрационное имя пользователя или имя группы, если они вызываются без параметров. Если в вызове функции указан параметр — имя или числовой идентификатор, то именно значение этого поля записи и возвращается.

Таблица П1.17. Информация о пользователях и группах

Функция	Назначение и синтаксис	Ссылка
<code>endpwent</code>	Заккрытие файла паролей после завершения обработки его содержимого. <code>endpwent</code>	
<code>getpwent</code>	Возвращает следующую запись файла паролей при очередном вызове. <code>getpwent</code>	
<code>getpwnam</code>	Получение записи из файла паролей от пользователя по его регистрационному имени. <code>getpwnam</code> ИМЯ	
<code>getpwuid</code>	Получение записи из файла паролей от пользователя по его числовому идентификатору, заданному параметром <code>UID</code> . <code>getpwuid</code> <code>UID</code>	

Таблица П1.17 (окончание)

Функция	Назначение и синтаксис	Ссылка
setpwent	Позиционирование файла паролей на первую запись. Последующий вызов функции <code>getpwent()</code> начинает его обработку с первой записи. <code>setpwent</code>	
endgrent	Закрытие файла групп после завершения обработки его содержимого. <code>endgrent</code>	
getgrent	Возвращает следующую запись файла групп при очередном вызове. <code>getgrent</code>	
getgrgid	Получение записи из файла групп о группе по ее числовому идентификатору, заданному параметром <code>GID</code> . <code>getgrgid GID</code>	
getgrnam	Получение записи из файла групп о группе по ее имени, заданному параметром <code>ИМЯ</code> . <code>getgrnam ИМЯ</code>	
setgrent	Позиционирование файла групп на первую запись. Последующий вызов функции <code>getgrent()</code> начинает его обработку с первой записи. <code>setgrent</code>	
getlogin	Возвращает регистрационное имя пользователя на рабочей станции, где выполняется программа Perl. <code>getlogin</code>	

Еще четыре группы функций — `gethost*()`, `getnet*()`, `getproto*()`, `getserv*()` — позволяют получить информацию о хостах (файл `/etc/hosts`), сетях (файл `/etc/networks`), протоколах (файл `/etc/protocols`) и сервисах (файл `/etc/services`). В списковом контексте функции этих групп возвращают следующие массивы, которые полностью соответствуют структуре записей файлов, обрабатываемых этими функциями (см. документацию по операционной системе UNIX):

```

($name,$aliases,$addrtype,$length,@addrs) = gethost*
($name,$aliases,$addrtype,$net)           = getnet*
($name,$aliases,$proto)                   = getproto*
($name,$aliases,$port,$proto)             = getserv*

```

В скалярном контексте эти функции возвращают первый элемент списка, если они вызываются без параметров. Если в вызове функции указан пара-

метр — имя или числовой идентификатор, то именно значение этого поля записи и возвращается.

Таблица П1.18. Сетевая информация

Функция	Назначение и синтаксис	Ссылка
endhostent	Заккрытие файла хостов после завершения обработки его содержимого. endhostent	
gethostbyaddr	Получение записи из файла хостов о хосте с заданным IP-адресом и типом. gethostbyaddr ADDR, ADDRTYPE	
gethostbyname	Получение записи из файла хостов о хосте с заданным именем. gethostbyname ИМЯ	
gethostent	Возвращает следующую запись файла хостов при очередном вызове. gethostent	
sethostent	Позиционирование файла хостов на первую запись. Последующий вызов функции gethostent() начинает его обработку с первой записи. sethostent STAYOPEN	
endnetent	Заккрытие файла сетей после завершения обработки его содержимого. endnetent	
getnetbyaddr	Получение записи из файла сетей о сети с заданным адресом и типом. getnetbyaddr ADDR, ADDRTYPE	
getnetbyname	Получение записи из файла сетей о сети с заданным сетевым именем. getnetbyname ИМЯ	
getnetent	Возвращает следующую запись файла сетей при очередном вызове. getnetent	
setnetent	Позиционирование файла сетей на первую запись. Последующий вызов функции getnetent() начинает его обработку с первой записи. setnetent STAYOPEN	
endprotoent	Заккрытие файла протоколов после завершения обработки его содержимого. endprotoent	

Таблица П1.18 (окончание)

Функция	Назначение и синтаксис	Ссылка
getprotobyname	Получение записи из файла протоколов о протоколе с заданным именем. getnetbyname ИМЯ	
getprotobynumber	Получение записи из файла протоколов о протоколе с заданным номером. getnetbyname НОМЕР	
getprotoent	Возвращает следующую запись файла протоколов при очередном вызове. getprotoent	
setprotoent	Позиционирование файла протоколов на первую запись. Последующий вызов функции getprotoent() начинает его обработку с первой записи. setprotoent STAYOPEN	
endservent	Закрытие файла служб после завершения обработки его содержимого. endservent	
getservbyname	Получение записи из файла служб о службе с заданным именем и протоколом. getnetbyname ИМЯ, ПРОТОКОЛ	
getservbyport	Получение записи из файла служб о службе с заданным номером порта и протоколом. getnetbyname ПОРТ, ПРОТОКОЛ	
getservent	Возвращает следующую запись файла служб при очередном вызове. getservent	
setservent	Позиционирование файла служб на первую запись. Последующий вызов функции getservent() начинает его обработку с первой записи. setservent	

Межпроцессное взаимодействие

В набор стандартных функций Perl входят функции взаимодействия между процессами, реализованные в Unix System V. Для получения правильных значений констант, используемых этими функциями, следует до обращения

к какой-либо функции этой группы подключить модуль `IPC::SysV`, в котором содержатся объявления всех необходимых констант.

Таблица П1.19. Функции взаимодействия между процессами

Функция	Назначение и синтаксис	Ссылка
<code>msgctl</code>	<p>Вызывает системную функцию <code>msgctl(2)</code>. Если команда, заданная параметром <code>CMD</code>, является командой <code>IPC_STAT</code>, то параметр <code>ARG</code> должен быть переменной, которая будет содержать возвращаемую структуру <code>msgid_ds</code>. Возвращаемое значение функции аналогично возвращаемому значению функции <code>ioctl()</code>: если системная функция возвращает <code>-1</code>, то функция Perl — неопределенное значение; если системная функция возвращает <code>0</code>, то функция Perl — строку <code>"0 but true"</code>; если системная функция возвращает какое-либо другое значение, то и функция Perl возвращает это же значение. См. также документацию по модулям <code>IPC::SysV</code> и <code>IPC::SysV::Msg</code>.</p> <p><code>msgctl ID, CMD, ARG</code></p>	
<code>msgget</code>	<p>Создает очередь сообщений с заданным параметром КЛЮЧ ключом, вызывая системную функцию <code>msgget(2)</code>. Возвращает идентификатор очереди сообщений, если она успешно создана, или неопределенное значение в противном случае. См. также документацию по модулям <code>IPC::SysV</code> и <code>IPC::SysV::Msg</code>.</p> <p><code>msgget КЛЮЧ, ФЛАГИ</code></p>	
<code>msgrcv</code>	<p>Вызывает системную функцию <code>msgrcv</code> для получения сообщения из очереди сообщений с идентификатором, заданным параметром <code>ID</code>, и сохранения его в переменной, определяемой параметром ПЕРЕМЕННАЯ. Максимальная длина сообщения не может превышать значения, заданного параметром ДЛИНА. Если сообщение получено, то в переменной перед самим сообщением сохраняется его тип. Возвращает булево значение Истина в случае успешного получения сообщения или Ложь в противном случае. См. также документацию по модулям <code>IPC::SysV</code> и <code>IPC::SysV::Msg</code>.</p> <p><code>msgrcv ID, ПЕРЕМЕННАЯ, ДЛИНА, ТИП, ФЛАГИ</code></p>	
<code>msgsnd</code>	<p>Вызывает системную функцию <code>msgsnd</code> для отправки сообщения, заданного параметром СООБЩЕНИЕ, в очередь сообщений с идентификатором, заданным параметром <code>ID</code>. Сообщение должно начинаться с типа сообщения, представленного длинным целым числом, которое можно создать функцией <code>pack("l", \$type)</code>.</p>	

Таблица П1.19 (продолжение)

Функция	Назначение и синтаксис	Ссылка
	<p>Возвращает булево значение Истина в случае успешной отправки сообщения или Ложь в противном случае. См. также документацию по модулям IPC::SysV и IPC::SysV::Msg.</p> <p>msgsnd ID, СООБЩЕНИЕ, ФЛАГИ</p>	
semctl	<p>Вызывает системную функцию <code>semctl()</code>. Если команда, заданная параметром CMD, является командой IPC_STAT или GETALL, то параметр ARG должен быть переменной, которая будет содержать возвращаемую структуру <code>semid_ds</code> или массив значений семафора. Возвращаемое значение функции аналогично возвращаемому значению функции <code>ioctl()</code>: если системная функция возвращает -1, то функция Perl неопределенное значение; если системная функция возвращает 0, то функция Perl строку "0 but true"; если системная функция возвращает какое-либо другое значение, то и функция Perl возвращает это же значение. См. также документацию по модулям IPC::SysV и IPC::Semaphore.</p> <p>semctl ID, SEMNUM, CMD, ARG</p>	
semget	<p>Создает набор семафоров, вызывая системную функцию <code>semget</code>. Возвращает идентификатор семафора, если набор успешно создан, или неопределенное значение в противном случае. См. также документацию по модулям IPC::SysV и IPC::SysV::Semaphore.</p> <p>semget KEY, NSEMS, FLAGS</p>	
semop	<p>Вызывает системную функцию <code>semop</code> для выполнения операций с семафором, например, сигнализация или ожидание. Параметр OPSTRING должен быть упакованным массивом структур <code>semop</code>, каждая из которых создается функцией <code>pack("sss", \$semnum, \$semop, \$semfkg)</code>. Количество операций с семафором определяется числом элементов массива OPSTRING. Возвращает булево значение Истина в случае успешного выполнения операций с семафором или Ложь в противном случае. См. также документацию по модулям IPC::SysV и IPC::SysV::Semaphore.</p> <p>semop KEY, OPSTRING</p>	
shmctl	<p>Вызывает системную функцию <code>shmctl</code>. Если команда, заданная параметром CMD, является командой IPC_STAT, то параметр ARG должен быть переменной, которая будет содержать возвращаемую структуру <code>shmid_ds</code>.</p>	

Таблица П1.19 (окончание)

Функция	Назначение и синтаксис	Ссылка
	<p>Возвращаемое значение функции аналогично возвращаемому значению функции <code>ioctl()</code>: если системная функция возвращает <code>-1</code>, то функция Perl неопределенное значение; если системная функция возвращает <code>0</code>, то функция Perl строку <code>"0 but true"</code>; если системная функция возвращает какое-либо другое значение, то и функция Perl возвращает это же значение. См. также документацию по модулю <code>IPC::SysV</code>.</p> <p><code>shmctl ID, CMD, ARG</code></p>	
<code>shmget</code>	<p>Создает область памяти для совместного использования (разделяемая область памяти), вызывая системную функцию <code>shmget</code>. Возвращает идентификатор разделяемой области памяти, если она успешно создана, или неопределенное значение в противном случае. См. также документацию по модулю <code>IPC::SysV</code>.</p> <p><code>shmget KEY, SIZE, FLAGS</code></p>	
<code>shmread</code>	<p>Читает в переменную <code>VAR</code> заданное параметром <code>SIZE</code> количество байт из разделяемой области памяти с идентификатором, определяемым параметром <code>ID</code>, начиная с позиции, указанной параметром <code>POS</code>. Возвращает булево значение Истина в случае успешного чтения данных или Ложь в противном случае. См. также документацию по модулям <code>IPC::SysV</code>.</p> <p><code>shmread ID, VAR, POS, SIZE</code></p>	
<code>shmwrite</code>	<p>Записывает заданное параметром <code>SIZE</code> количество байт из строки данных <code>STRING</code> в разделяемую область памяти с идентификатором, определяемым параметром <code>ID</code>, начиная с позиции, указанной параметром <code>POS</code>. Если строка данных содержит меньшее количество байт, то добавляются нулевые значения <code>"\0"</code>. Возвращает булево значение Истина в случае успешной записи данных или Ложь в противном случае. См. также документацию по модулям <code>IPC::SysV</code>.</p> <p><code>shmwrite ID, STRING, POS, SIZE</code></p>	



Стандартные модули

Стандартная поставка Perl включает в себя большое число модулей, в которых представлены разные средства и функции для решения многих задач, возникающих при программировании на Perl. Простым подключением соответствующего модуля к своей программе можно получить необходимые средства для решения конкретной задачи. Более того, все перечисленные ниже с кратким описанием модули присутствуют в системах программирования на языке Perl на всех известных платформах. Поэтому, если использовать их в своих разработках, то проблем с переносимостью программ Perl никогда не возникнет. Мы рекомендуем читателю потратить некоторое время на внимательный просмотр этого приложения. Эта работа с лихвой окупится в дальнейшем — не надо будет прикладывать усилий для решения проблемы, которая уже решена в каком-либо модуле. Все стандартные модули в табл. П2.1 представлены в алфавитном порядке.

Замечание

Более подробную информацию о работе с модулем Perl можно получить, обработав его одной из программ `pod2html`, `pod2text`, `pod2latex` или `pod2man`, которые извлекают документацию в формате POD из самого модуля и представляют ее в соответствующих форматах. Перечисленные программы входят в стандартную поставку Perl.

Таблица П2.1. Стандартные модули Perl

Модуль	Краткое описание
AnyDBM_File	Этот модуль всего лишь загружает один из модулей для связывания хешей Perl с файлами DBM в следующей последовательности: <code>NDBM_File</code> , <code>DB_File</code> , <code>GDBM_File</code> , <code>SDBM_File</code> и <code>ODBM_File</code> . Если установлен модуль <code>NDBM_File</code> , то загружается он; если его нет, то ищется следующий из приведенного списка и т. д. Модуль <code>SDBM_File</code> всегда присутствует, так как он входит в стандартную поставку Perl.

Таблица П2.1 (продолжение)

Модуль	Краткое описание
AutoLoader	Предоставляет стандартный механизм для отложенной загрузки функций, хранящихся в отдельных файлах на диске. Используется при создании собственных модулей расширения Perl.
AutoSplit	Содержит функции, разбивающие программу или модуль Perl на файлы, которые впоследствии может обрабатывать модуль AutoLoader. Основное применение — построение библиотечных модулей Perl с автозагрузкой.
Benchmark	Содержит набор функций для проведения тестов на быстродействие выполнения отдельных фрагментов кода программы.
Carp	Содержит функции <code>carp()</code> и <code>croak()</code> — аналоги функций <code>warn()</code> и <code>die()</code> . Они отображают сообщение с указанием подпрограммы и номера ее строки, откуда вызываются эти функции, тогда как <code>warn()</code> и <code>die()</code> отображают номер строки в коде всей программы.
Config	Содержит набор функций для доступа к конфигурационной информации Perl, которую сценарий <code>Configure</code> занес в компьютер при установке Perl.
Cwd	Содержит функции для определения полного имени текущего рабочего каталога и его изменения. Эти функции работают надежнее и быстрее, чем соответствующие встроенные функции Perl.
DB_File	Позволяет использовать в программе Perl возможности, предоставляемые библиотекой Berkeley DB, предлагая единообразный интерфейс доступа к базам данных разных форматов.
Devel::SelfStubber	Генерация шаблонов (заготовок) для самозагружающихся модулей.
diagnostics	Позволяет отображать более полные и развернутые диагностические сообщения компилятора и интерпретатора perl по сравнению с используемой по умолчанию краткой диагностикой.
DirHandle	Предоставляет объектный интерфейс к функциям работы с каталогами: <code>opendir()</code> , <code>closedir()</code> , <code>readdir()</code> и <code>rewinddir()</code> .
DynaLoader	Определяет стандартный интерфейс Perl к механизмам динамической загрузки, доступным на большинстве компьютерных платформ. Используется при создании собственных модулей расширения Perl.

Таблица П2.1 (продолжение)

Модуль	Краткое описание
Env	Perl поддерживает работу с переменными среды через ассоциативный массив <code>%ENV</code> . Модуль позволяет использовать переменные среды как обычные скалярные переменные Perl, используя имена этих переменных в качестве идентификатора переменных Perl. Например, <code>\$PATH</code> , <code>\$HOME</code> и т. д.
Exporter	Реализует метод <code>import()</code> , который обычно наследуется другими модулями вместо определения ими собственных методов импортирования своих определений и функций.
ExtUtils::Install	Предоставляет две функции <code>install()</code> и <code>uninstall()</code> , необходимые модулю <code>ExtUtils::MakeMaker</code> для выполнения зависимой от платформы установки и удаления расширений Perl.
ExtUtils::Liblist	Содержит функцию <code>ext()</code> , которая преобразует список библиотек в строки, пригодные для включения в файл <i>Makefile</i> расширения Perl на текущей платформе.
ExtUtils::MakeMaker	Автоматизирует создание файла <i>Makefile.pl</i> , необходимого для построения расширения Perl.
ExtUtils::Manifest	Содержит функции для создания файла <i>MANIFEST</i> , который содержит список имен файлов (одно имя в строке с необязательным комментарием). Содержимое файла легко прочитать командой <code>awk '{print \$1}' MANIFEST</code> . Этот модуль используется модулем <code>ExtUtils::MakeMaker</code> для создания списка файлов, содержащих функции модуля.
ExtUtils::Miniperl	Содержит единственную функцию <code>writemain()</code> , получающую список каталогов, содержащих архивные библиотеки, которые необходимы модулям Perl и которые следует включить в новый двоичный файл Perl. Обычно используется из файла <i>Makefile</i> , сгенерированного модулем <code>ExtUtils::MakeMaker</code> . Программист непосредственно не работает с этим модулем.
ExtUtils::Mkbootstrap	Единственная функция этого модуля <code>mkbootstrap()</code> обычно вызывается из файла <i>Makefile</i> соответствующего расширения и создает файл <code>*.bs</code> , необходимый для динамической загрузки в некоторых архитектурах операционных систем.
ExtUtils::Mksymlists	Единственная функция этого модуля <code>Mksymlists()</code> создает файлы, которые используются компоновщиком

Таблица П2.1 (продолжение)

Модуль	Краткое описание
(прод.)	некоторых операционных систем при создании совместно используемых (shared) библиотек для динамических расширений Perl. Обычно вызывается из файла <i>Makefile</i> , сгенерированного модулем <code>ExtUtils::MakeMaker</code> при построении расширения.
<code>ExtUtils::MM_OS2</code>	Перекрывает методы модуля <code>ExtUtils::MM_Unix</code> , используемые модулем <code>ExtUtils::MakeMaker</code> , для работы в операционной системе OS2.
<code>ExtUtils::MM_Unix</code>	Методы этого модуля используются модулем <code>ExtUtils::MakeMaker</code> . Программист никогда не работает с ними, если только он не улучшает модуль <code>ExtUtils::MakeMaker</code> в связи с его переносимостью.
<code>ExtUtils::MM_VMS</code>	Перекрывает методы модуля <code>ExtUtils::MM_Unix</code> , используемые модулем <code>ExtUtils::MakeMaker</code> , для работы в операционной системе VMS.
<code>Fcntl</code>	Этот модуль всего лишь трансляция заголовочного файла <code>fcntl.h</code> языка C, содержащего определения различных констант.
<code>File::Basename</code>	Функции этого модуля осуществляют синтаксический разбор полного имени файла, выделяя ключевые единицы. Можно использовать синтаксис имен файлов разных операционных систем. Например, в UNIX ключевыми единицами будут путь, имя файла и расширение.
<code>File::CheckTree</code>	Единственная функция этого модуля <code>validate()</code> позволяет выполнять унарные именованные операции проверки над группой файлов, причем для каждого файла группы можно задать свои операции. Все операции проверки выполняются за одно обращение к функции.
<code>File::Copy</code>	Содержит единственную функцию <code>copy()</code> , которая копирует группу файлов за одно обращение.
<code>File::Find</code>	Содержит две функции поиска файлов по критерию, определенному в пользовательской подпрограмме. Функция <code>find()</code> отыскивает все файлы в каталогах заданного списка, последовательно переходя на поиск в каталогах нижнего уровня. Функция <code>finddepth()</code> также отыскивает файлы в каталогах заданного списка, но начинает поиск с последнего вложенного каталога, последовательно переходя на поиск в каталогах верхнего уровня.

Таблица П2.1 (продолжение)

Модуль	Краткое описание
<code>File::Path</code>	Содержит две функции <code>mkspath()</code> и <code>rmtree()</code> , которые, соответственно, создают и удаляют каталоги. В отличие от встроенной функции создания каталога <code>mkdir()</code> функция этого модуля создает каталоги верхнего уровня, если они не существуют. Функция удаления каталога, в отличие от встроенной функции <code>rmdir()</code> , удаляет любой, не обязательно пустой каталог, причем возвращает количество удаленных файлов.
<code>FileCache</code>	Предоставляет функцию <code>casheout()</code> , которая открывает файл, не обращая внимания на установленное в соответствующей переменной среды количество одновременно открытых файлов. Ее использование для открытия файлов позволяет открыть произвольное число файлов.
<code>FileHandle</code>	Предоставляет объектный интерфейс для работы с дескрипторами файлов.
<code>GDBM_File</code>	Позволяет программе Perl использовать возможности, предоставляемые библиотекой <code>gdbm</code> , распространяемой на условиях лицензии GNU. См. описание модуля <code>DB_File</code> .
<code>Getopt::Long</code>	Реализует функцию <code>GetOption()</code> , позволяющую получать и обрабатывать опции командной строки, с которыми была запущена программа Perl. Параметры этой функции определяют синтаксис правильных опций: опция с обязательным/необязательным строковым/целым/числовым параметром, опция без параметров.
<code>Getopt::Std</code>	Содержит функции <code>getopt()</code> и <code>getopts()</code> , реализующие простой механизм обработки односимвольных опций (например, <code>-v</code>). Опции можно объединять в группы.
<code>I18N::Collate</code>	Предоставляет объекты, которым можно присвоить строки данных, и в дальнейшем сравнивать и упорядочивать их в соответствии с таблицей символов национального алфавита. Для правильной работы объектов и методов этого модуля необходимо иметь установленный модуль <code>POSIX</code> языка Perl. Системе также должны быть доступны <code>POSIX</code> -функции <code>setlocale(3)</code> и <code>strxfrm(3)</code> .
<code>integer</code>	Использование в программе этого модуля предписывает компилятору применять арифметику целых чисел, начиная от строки вызова модуля (<code>use integer;</code>) до конца блока, в котором он вызван.

Таблица П2.1 (продолжение)

Модуль	Краткое описание
IPC::Open2	Содержит функцию <code>open2()</code> , которая порождает новый процесс и выполняет в нем команду, определяемую значением параметра функции. Ассоциируя со стандартными файлами ввода и вывода два дескриптора, позволяет программе Perl получать и передавать информацию для порожденного процесса.
IPC::Open3	Содержит функцию <code>open3()</code> , которая работает аналогично функции <code>open2()</code> модуля <code>IPC::Open2</code> за исключением того, что открывается еще третий файл для обмена информацией — стандартный файл вывода ошибок.
lib	Упрощает работу со специальной переменной <code>@INC</code> во время компиляции. Позволяет добавлять дополнительные каталоги поиска для операторов <code>use</code> и <code>require</code> .
Math::BigFloat	Позволяет создавать объекты, представляющие вещественные числа с произвольным количеством цифр в мантиссе, и выполнять над ними арифметические операции.
Math::BigInt	Позволяет создавать объекты, представляющие целые числа с произвольным количеством цифр в мантиссе, и выполнять над ними арифметические операции.
Math::Complex	Позволяет создавать объекты, представляющие комплексные числа, и выполнять над ними арифметические операции.
NDBM_File	Позволяет связывать функцией <code>tie()</code> хеши с файлами NDBM. См. описание модуля <code>DB_File</code> .
Net::Ping	Содержит функцию <code>pingecho()</code> , которая использует TCP-службу эхо (<code>echo</code>) (не <code>ICMD</code>) для определения, доступен ли удаленный хост.
ODBM_File	Позволяет связывать функцией <code>tie()</code> хеши с файлами ODBM. См. описание модуля <code>DB_File</code> .
overload	Позволяет осуществлять перегрузку стандартных операций Perl, т. е. при вызове в программе стандартной функции выполнять вместо предусмотренных стандартных операций методы каких-либо классов или собственные подпрограммы. Модуль постоянно изменяется, поэтому рекомендуем внимательно ознакомиться с документацией установленного на компьютере интерпретатора perl.
POSIX	Обеспечивает доступ ко всем (или почти всем) идентификаторам стандарта POSIX версии 1003.1. Для многих из этих идентификаторов реализованы Perl-подобные интерфейсы.

Таблица П2.1 (продолжение)

Модуль	Краткое описание
Pod::Text	Содержит функцию <code>pod2text()</code> , которая в программе Perl позволяет преобразовать файл документации в формате POD в обычный текстовый ASCII файл. (В стандартную поставку Perl входит отдельная программа <code>pod2text</code> , которую можно запустить из командной строки.)
Safe	Создает специальную область, в которой можно выполнить небезопасный код Perl. Эта область имеет собственное пространство имен и ассоциированную маску операций. Код, выполняемый в ней, не может ссылаться на переменные вне своего пространства имен. Компилируемый вне этой области код может поместить некоторые свои переменные в пространство имен этой области. Тогда они станут доступны коду, компилируемому в ней. Задаваемая маска операций позволяет исключить из выполняемого в безопасной области кода Perl потенциально опасные операции. По умолчанию любые операции доступа к системным ресурсам не выполняются в безопасной области.
SDBM_File	Позволяет связывать функцией <code>tie()</code> хеши с файлами SDBM. См. описание модуля <code>DB_File</code> .
Search::Dict	Содержит функцию <code>look()</code> , которая устанавливает указатель позиции в файле на строку, содержимое которой больше или равно некоторому строковому значению, переданному в функцию в качестве параметра.
SelectSaver	Позволяет создать объект <code>SelectSaver</code> , с помощью которого можно выбрать новый дескриптор файла, при этом старый дескриптор сохраняется и восстанавливается при уничтожении объекта <code>SelectSaver</code> .
SelfLoader	Этот модуль предоставляет механизм отсроченной загрузки функций, объявления которых расположены после лексемы <code>__DATA__</code> пакета, определенного в файле программы Perl. Функция будет автоматически загружена при первом к ней обращении. Этот модуль отличается от модуля <code>AutoLoader</code> , который работает с функциями в отдельных самостоятельных файлах.
Shell	Позволяет вызывать утилиты UNIX, доступные из командной строки, как будто они являются подпрограммами Perl. Параметры, включая ключи, передаются в утилиты как строковые данные.
sigtrap	Позволяет инициализировать обработчик сигнала для сигналов, переданных в качестве параметров модулю при его подключении оператором <code>use</code> .

Таблица П2.1 (продолжение)

Модуль	Краткое описание
Socket	Загружает определения констант и функций заголовочного файла <code>socket.h</code> языка C, а также специальных функций обработки некоторых структур данных, необходимых для работы с сетью.
strict	Позволяет ограничить небезопасные конструкции: переменные, ссылки и простые слова, не являющиеся заранее объявленными подпрограммами.
subs	Осуществляет объявление подпрограмм, чьи имена передаются в списке при подключении модуля в программу Perl. Это объявление позволяет вызывать в сценарии эти подпрограммы, задавая их параметры без скобок, даже до их определения. Можно перекрыть встроенные функции.
Symbol	Предоставляет функцию <code>gensym()</code> , которая создает анонимную глобальную переменную и возвращает ссылку на нее. Такую переменную можно использовать в качестве дескриптора файла или каталога. Функция <code>qualify()</code> уточняет имена переменных, добавляя к ним префикс, состоящий из имени пакета, где они определены, и двойного двоеточия <code>::</code> .
Sys::Hostname	Предоставляет функцию <code>hostname()</code> , которая всеми возможными способами пытается получить имя хоста, последовательно вызывая разные системные программы.
Sys::Syslog	Реализует Perl-интерфейс к программе <code>syslog(3)</code> операционной системы UNIX. Для использования этого модуля следует программой <code>h2ph</code> создать файл <code>syslog.ph</code> .
Term::Cap	Позволяет создать объект, который извлекает из базы данных терминалов и сохраняет информацию о свойствах терминала заданного типа. См. man-страницу <code>termcap(5)</code> для получения информации об использовании базы данных терминалов.
Term::Complete	Содержит функцию <code>Complete()</code> , которая посылает сообщение через текущий дескриптор файла и читает ответ пользователя, выполняя специальные действия, если в ответе пользователя встречаются коды следующих клавиш: <code><Tab></code> , <code><Ctrl>+<D></code> , <code><Ctrl>+<U></code> или <code></code> .
Test::Harness	Этот модуль используется модулем <code>ExtUtils::MakeMaker</code> . Если строится расширение Perl и в каталоге расширения имеются тестовые сценарии, имена которых удовлетворяют шаблону <code>t/*.t</code> , то функция <code>runtest()</code> выполнит все тестовые сценарии, переданные ей в качестве параметров.

Таблица П2.1 (продолжение)

Модуль	Краткое описание
Text::Abbrev	Позволяет создать таблицу неповторяющихся сокращений для элементов списка из их значений. Функция <code>abbrev()</code> просматривает содержимое каждого элемента переданного ей списка и конструирует для него простым отсечением все возможные сокращения с учетом того, чтобы ни одно из полученных сокращений не совпадало со значениями других элементов в списке. Эти сокращения используются в качестве ключей в хеше, переданном в функцию <code>abbrev()</code> ; для всех подобных ключей значением становится значение самого элемента списка.
Text::ParseWords	Содержит функцию <code>quotewords()</code> , которая выделяет из содержимого строк, заданных элементами массива, слова, ограниченные определяемым пользователем разделителем (может быть задан регулярным выражением).
Text::Soundex	Реализует алгоритм <code>soundex</code> , описанный в томе 3 книги Д. Кнут "Искусство программирования". Этот алгоритм разбивает слова на более мелкие части, используя простую модель, которая аппроксимирует звучание английского слова. Каждому слову в соответствии с его разбиением присваивается четырехсимвольный код, первым символом которого является буква, а остальные три представляют цифры.
Text::Tabs	Предоставляет две функции — <code>expand()</code> и <code>unexpand()</code> , которые, соответственно, преобразуют все символы табуляции в строке в символы, определяемые пользователем, и наоборот — символы, определяемые пользователем, в символы табуляции.
Text::Wrap	Предоставляет функцию <code>wrap()</code> , которая форматирует длинную строку данных в абзац заданной ширины, определяя отступы для первой и последующих строк абзаца. Перенос осуществляется по словам.
Tie::Hash	Эти модули позволяют создать шаблоны (заготовки) для методов классов, реализующих действия для связываемых функцией <code>tie()</code> хешей.
Tie::StdHash	
Tie::Scalar	Эти модули позволяют создать шаблоны (заготовки) для методов классов, реализующих действия для связываемых функцией <code>tie()</code> скаляров.
Tie::StdScalar	
Tie::SubstrHash	Предоставляет табличный хеш-интерфейс к массиву с ключами и записями постоянной длины.

Таблица П2.1 (окончание)

Модуль	Краткое описание
Time::Local	Содержит две эффективные функции вычисления местного времени (timelocal()) и времени по Гринвичу (timegm()).
vars	Этот модуль позволяет объявить переменные, имена которых переданы ему в качестве параметров, как глобальные.

Модули CPAN

Если предлагаемые стандартными модулями Perl средства не подходят для решения поставленной задачи, то прежде чем начинать собственную разработку, рекомендуется обратиться к архиву модулей Perl в Internet по адресу:

<http://www.perl.com/CPAN/modules/>

Здесь расположены свободно распространяемые модули Perl. Рекомендуем перейти по соответствующей ссылке на страницу, где все модули классифицированы по категориям их применения. Ниже мы представляем все перечисленные на ней категории с соответствующим русским переводом:

Part 2 — The Perl 5 Module List

(Часть 2 — Список модулей Perl 5)

1. Module Listing Format
(Формат информационной записи о модуле)
2. Perl Core Modules, Perl Language Extensions and Documentation Tools
(Модули ядра Perl, расширения языка Perl и средства документирования)
3. Development Support
(Поддержка разработки)
4. Operating System Interfaces, Hardware Drivers
(Интерфейсы к операционным системам, драйверы устройств)
5. Networking, Device Control (modems) and InterProcess Communication
(Работа в сети, управление модемами и межпроцессное взаимодействие)
6. Data Types and Data Type Utilities
(Типы данных и утилиты работы с типами данных)
7. Database Interfaces
(Интерфейсы баз данных)
8. User Interfaces
(Интерфейсы пользователя)

9. Interfaces to or Emulations of Other Programming Languages
(Интерфейсы к другим языкам программирования или их эмуляция)
10. File Names, File Systems and File Locking (see also File Handles)
(Имена файлов, файловые системы и блокировка файлов
(также см. Обработка файлов))
11. String Processing, Language Text Processing, Parsing and Searching
(Обработка строковых данных, обработка текстов, синтаксический разбор и поиск)
12. Option, Argument, Parameter and Configuration File Processing
(Обработка опций, параметров и конфигурационных файлов)
13. Internationalization and Locale
(Локализация программ, многоязыковая поддержка)
14. Authentication, Security and Encryption
(Идентификация пользователя, безопасность и кодирование информации)
15. World Wide Web, HTML, HTTP, CGI, MIME
16. Server and Daemon Utilities
(Серверные утилиты и утилиты создания программ-демонов)
17. Archiving, Compression and Conversion
(Архивизация, сжатие и преобразование)
18. Images, Pixmap and Bitmap Manipulation, Drawing and Graphing
(Изображения, обработка битовых и пиксельных растров, графика)
19. Mail and Usenet News
(Почта и группы новостей)
20. Control Flow Utilities (callbacks and exceptions etc)
(Утилиты управления выполнением программ (функции отклика на события, обработка исключений и т. д.))
21. File Handle, Directory Handle and Input/Output Stream Utilities
(Обработка файлов, обработка каталогов и утилиты потокового ввода/вывода)
22. Microsoft Windows Modules
(Модули для операционной системы Microsoft Windows)
23. Miscellaneous Modules
(Разные модули)
24. Interface Modules to Commercial Software
(Модули, предоставляющие интерфейс к коммерческому программному обеспечению)

Приложение 3

Специальные переменные



В предыдущих главах мы встречались с некоторыми переменными, имеющими для интерпретатора perl специальное значение. Среди них есть скалярные переменные, массивы и хеш-массивы, глобальные и локальные переменные для текущего блока. Специальные предопределенные переменные используются, например, в интерпретаторе команд UNIX shell или в программе awk. Но нигде они не встречаются в таком количестве, как в языке Perl. Мы познакомились лишь с небольшой их частью. Многие специальные переменные используются в тех разделах языка, которые не обсуждались в данной книге. Поэтому здесь мы решили привести сводку специальных переменных Perl.

- ❑ **\$_**
Используется по умолчанию во многих функциях и операциях, в том числе:
 - в качестве области поиска в операциях сопоставления с образцом `m//`, замены `s///` и транслитерации `y///`, когда область поиска не задана явно операцией связывания `=~`;
 - в качестве области ввода, когда условное выражение оператора `while` состоит из единственной операции ввода `<>`;
 - в качестве параметра функций `chop`, `split`, `print`;
 - в качестве переменной цикла в операторе `foreach`, если переменная цикла не задана явно.
- ❑ **\$nn**
Переменные `$1`, `$2`, ... содержат подобразцы из соответствующих наборов круглых скобок в последней успешной операции сопоставления с образцом.
- ❑ **\$&**
Часть строки, найденная при последней успешной операции сопоставления с образцом.
- ❑ **\$`**
Часть строки, стоящая перед совпавшей частью при последней успешной операции сопоставления с образцом.
- ❑ **\$'**
Часть строки, стоящая после совпавшей части при последней успешной операции сопоставления с образцом.

□ \$+

Содержит подобраец из последнего набора круглых скобок в последней успешной операции сопоставления с образцом. Например, в результате операции

```
"1234.5678" =~ m/(\d+)\.(\d+)/;
```

переменная \$+ получит значение 5678.

□ \$*

Установить значение 0, чтобы в операциях сопоставления с образцом строка рассматривалась как мультистрока, состоящая из нескольких строк, разделенных символом новой.

Установить значение 1, чтобы в операциях сопоставления с образцом строка рассматривалась как одна строка. В этом случае метасимвол "." соответствует любой одиночный символ, в том числе и новой строки.

Устарела, для указанных целей рекомендуется в операциях сопоставления с образцом использовать ключи `m` и `s`, соответственно.

Значение по умолчанию равно 0.

□ \$.

Номер последней строки, считанной из того файла, для которого выполнялась последняя операция чтения.

□ \$/

Разделитель записей, считываемых из входного файла. По умолчанию равен символу новой строки. Может состоять из нескольких символов. Если значение установлено неопределенным при помощи функции `undef`, то при чтении из входного файла границей записи является признак конца файла.

□ \$|

Обычно данные, выводимые в файл функциями `print` или `write`, предварительно помещаются в буфер. Когда буфер заполняется, его содержимое записывается в файл. Буферизация повышает эффективность операций вывода. По умолчанию Perl использует буферизацию для каждого выходного файла, что соответствует нулевому значению переменной `$|`. Чтобы ее отменить, следует выбрать файл при помощи функции `select` и установить значение переменной `$|` не равным 0.

□ \$,

Разделитель полей выходных записей для функции `print`. Параметры функции `print` при выводе разделяются символом, который является значением переменной `$_`. По умолчанию этим значением является нулевой символ, т. е. выводимые элементы печатаются друг за другом. Если переменной `$,` присвоить, например, значение символа новой строки, то каждый параметр функции `print` будет напечатан в отдельной строке.

- `$\`
Разделитель выходных записей для функции `print`. Добавляется в конец списка параметров функции `print`. По умолчанию равен нулевому символу. Если установить его значение равным символу новой строки, каждый вызов функции `print` будет завершаться переводом на новую строку.
- `$"`
Когда переменная-массив, заключенная в двойные кавычки, передается в качестве параметра функции `print`, ее элементы при выводе разделяются последовательностью символов, содержащейся в переменной `$"`. По умолчанию ее значением является символ пробела.
- `$;`
Символ, используемый в качестве разделителя индексов при эмуляции многомерных массивов. Значение по умолчанию `\034`. Интерпретатор, встретив запись `$array{"bim", "bom"}`, преобразует ее к виду `$array{"bim" . $; . "bom"}`.
- `$#`
Формат, используемый по умолчанию для вывода чисел. Значение по умолчанию `%.20g`, что означает представление выводимых чисел в формате с 20 знаками после десятичной точки. Не рекомендуется использовать в версиях Perl 5 и старше.
- `$%`
Каждый файл, открытый программой Perl для вывода, имеет свою копию переменной `$%`, в которой хранится номер текущей страницы. В каждый момент значение переменной `$%` равно номеру текущей страницы текущего файла вывода.
- `$=`
Длина текущей страницы текущего файла вывода. По умолчанию значение равно 60.
- `$-`
Число строк, оставшихся на текущей странице текущего файла вывода.
- `$~`
Имя текущего формата для текущего файла вывода. Значением по умолчанию является имя дескриптора этого файла.
- `$^`
Имя текущего формата заголовка страницы для текущего файла вывода. Значением по умолчанию является имя дескриптора файла, к которому добавлен суффикс `_TOP`.
- `$:`
Текущее множество символов переноса слова.
Если поле вывода в формате начинается с символа `^`, то интерпретатор помещает в это поле выводимое слово только тогда, когда в нем достаточно

места для этого слова. Чтобы определить, может ли слово поместиться в поле формата, интерпретатор подсчитывает число символов между следующим выводимым символом и следующим символом переноса слова. Символ переноса слова обозначает или конец слова, или место, где слово может быть разбито на две части. По умолчанию значением переменной `$:` является строка, состоящая из символов пробела, новой строки и дефиса: " \n-".

□ `$^L`

Признак перехода на новую страницу, используемый в шаблонах формата. Значение по умолчанию `\f`.

□ `$^A`

Аккумулятор, используется функцией `write()` для временного хранения выводимых отформатированных строк.

□ `$?`

Информация, возвращаемая последней операцией ```, операцией закрытия программного канала или вызовом функции `system`. Представляет 16-битное целое число, состоящее из двух частей по 8 бит. Старшие 8 разрядов содержат код завершения процесса, младшие — дополнительную системную информацию о завершении процесса.

□ `$!`

Некоторые функции Perl обращаются к функциям операционной системы. Если системная функция генерирует ошибку, код ошибки сохраняется в переменной `$!`. В числовом контексте переменная `$!` дает код системной ошибки, в строковом контексте — соответствующее текстовое сообщение.

□ `$^E`

Информация о системной ошибке, специфическая для текущей операционной системы. Отличается от переменной `$!` для платформ Win32, OS/2 и VMS, для остальных платформ эти переменные совпадают.

□ `$@`

Сообщение об ошибке, сгенерированное интерпретатором в результате последнего вызова функции `eval()`. Нулевое значение означает, что функция `eval()` завершилась успешно.

□ `$$`

Идентификатор процесса выполняющегося интерпретатора perl.

□ `$<`

Действительный идентификатор пользователя данного процесса.

□ `$>`

Эффективный идентификатор пользователя данного процесса.

□ `$()`

Действительный идентификатор группы данного процесса.

- ☐ `$)`
Эффективный идентификатор группы данного процесса.
- ☐ `$0`
Имя файла, содержащего выполняющуюся Perl-программу.
- ☐ `$[`
Индекс первого элемента массива и первого символа в подстроке. Значение по умолчанию 0. Не рекомендуется изменять.
- ☐ `$]`
Номер версии интерпретатора perl.
- ☐ `$^D`
Текущее значение флагов отладки, которые передаются при вызове интерпретатора с ключом `-D`.
- ☐ `$^F`
Максимальный номер дескриптора системного файла. Обычно системными файлами считаются стандартные файлы `STDIN`, `STDOUT`, `STDERR`, которые в ОС UNIX имеют дескрипторы 0, 1, 2 соответственно. Поэтому значением по умолчанию является 2.
- ☐ `$^H`
Значение, содержащее информацию о том, какие проверки синтаксиса заданы директивой `use strict`.
- ☐ `$^I`
При вызове интерпретатора perl с флагом `-iextension` входные файлы, полученные при помощи операции `<>`, можно редактировать непосредственно на месте. При этом для резервной копии файла используется имя с расширением `extension`. Это расширение и сохраняется в переменной `$^I`. Если ее значение сделать неопределенным при помощи функции `undef`, редактирование на месте будет запрещено.
- ☐ `$^M`
Эта переменная используется только в специально скомпилированной версии интерпретатора perl для создания резервного буфера памяти.
- ☐ `$^O`
Имя операционной системы, в которой была осуществлена компиляция данного интерпретатора perl.
- ☐ `$^P`
Содержит внутренние флаги отладчика Perl. Если значение равно 0, режим отладки отключен.
- ☐ `$^R`
Результат последнего удачного выполнения конструкции `{?{code}}`, представляющей расширенный синтаксис регулярного выражения, которая в книге не рассматривалась.

- ☐ **\$^T**
Время, когда была запущена программа. Измеряется в секундах относительно начала 1970 года.
- ☐ **\$^W**
Значение ключа командной строки `-w`. Значение 0 подавляет вывод предупреждающих сообщений о возможных синтаксических и других ошибках, значение 1 — разрешает.
- ☐ **\$^X**
Имя, по которому была вызвана выполняющаяся программа.
- ☐ **\$ARGV**
Имя текущего файла при чтении из `<>`.
- ☐ **@ARGV**
Содержит аргументы командной строки выполняющейся программы.
- ☐ **@INC**
Содержит имена каталогов, в которых следует искать сценарии Perl, подлежащие выполнению в конструкциях `do filename`, `require` или `use`. Первоначально содержит: имена каталогов, переданные при запуске интерпретатору perl в качестве параметра ключа `-I`; имена библиотечных каталогов по умолчанию (зависят от операционной системы); символическое обозначение текущего каталога `"."`.
- ☐ **@_**
Внутри подпрограммы содержит список переданных ей параметров.
- ☐ **%INC**
Содержит по одному элементу для каждого файла, подключенного при помощи функций `do` или `require`. Ключом является имя файла в том виде, как оно указано в качестве аргумента функций `do` или `require`, а значением — его полное маршрутное имя.
- ☐ **%ENV**
Содержит текущие значения переменных среды.
- ☐ **%SIG**
Служит для настройки обработки сигналов. Ключом является мнемоническое имя сигнала, значением — либо строка `'IGNORE'` (игнорировать сигнал), либо строка `'DEFAULT'` (восстановить реакцию на сигнал по умолчанию), либо строка, задающая квалифицированное имя подпрограммы-обработчика сигнала. Механизм сигналов, используемый в ОС UNIX, здесь не рассматривался.
- ☐ **@EXPORT**
Содержит имена, которые пакет экспортирует по умолчанию.

❑ @EXPORT_OK

Содержит имена, которые пакет экспортирует по запросу вызывающей программы.

❑ ARGV

Дескриптор, ассоциированный с текущим файлом ввода, из которого осуществляется считывание при помощи операции `<>`. Последовательно ассоциируется с файлами, переданными в программу в качестве аргументов командной строки, имена которых сохраняются в массиве `@ARGV`.

❑ DATA

Специальный дескриптор файла, ассоциированный с частью файла, которая расположена после лексемы `__END__`. Чтение из дескриптора `DATA` означает считывание строки, расположенной в файле сразу за лексемой `__END__`, что позволяет поместить программу и данные в один файл.:

```
#!/usr/local/bin/perl
$line = <DATA>;
print ("$line");
__END__
```

Эта строка расположена в одном файле с Perl-программой.

❑ STDERR

Дескриптор стандартного файла диагностики, который обычно связан с экраном.

❑ STDIN

Дескриптор стандартного файла ввода, обычно ассоциированный с клавиатурой.

❑ STDOUT

Дескриптор стандартного файла вывода, обычно ассоциированный с экраном.

Имена специальных переменных Perl состоят из префикса, определяющего тип переменной: `$`, `@`, `%`, за которым, как правило, следует символ, не являющийся алфавитно-цифровым. Такой выбор имен уменьшает вероятность случайного переопределения специальной переменной пользователем, но имеет очевидные неудобства, связанные с ее запоминанием. Поэтому у многих специальных переменных имеются более удобные мнемонические имена, являющиеся псевдонимами имен, рассмотренных выше. Для того чтобы иметь возможность использовать псевдонимы, необходимо в начало программы поместить директиву

```
use English;
```

Мнемонические имена специальных переменных приведены в табл. ПЗ.1.

Таблица ПЗ.1. Мнемонические имена специальных переменных Perl

Переменная	Псевдоним
\$_	\$ARG
\$0	\$PROGRAM_NAME
\$<	\$REAL_USER_ID или \$UID
\$>	\$EFFECTIVE_USER_ID или \$EUID
\$(\$REAL_GROUP_ID или \$GID
\$)	\$EFFECTIVE_GROUP_ID или \$EGID
\$]	\$PERL_VERSION
\$/	\$INPUT_RECORD_SEPARATOR или \$RS
\$\	\$OUTPUT_RECORD_SEPARATOR или \$ORS
\$,	\$OUTPUT_FIELD_SEPARATOR или \$OFS
\$"	\$LIST_SEPARATOR
\$#	\$OFMT
\$@	\$EVAL_ERROR
\$?	\$CHILD_ERROR
\$!	\$OS_ERROR или \$ERRNO
\$.	\$INPUT_LINE_NUMBER или \$NR
\$*	\$MULTILINE_MATCHING
\$;	\$SUBSCRIPT_SEPARATOR или \$SUBSEP
\$:	\$FORMAT_LINE_BREAK_CHARACTERS
\$\$	\$PROCESS_ID или \$PID
\$\$A	\$ACCUMULATOR
\$\$D	\$DEBUGGING
\$\$E	\$EXTENDED_OS_ERROR
\$\$F	\$SYSTEM_FD_MAX
\$\$I	\$INPLACE_EDIT
\$\$L	\$FORMAT_FORMFEED
\$\$O	\$OSNAME
\$\$P	\$PERLDB
\$\$T	\$BASETIME
\$\$W	\$WARNING
\$\$X	\$EXECUTABLE_NAME
\$&	\$MATCH
\$'	\$PREMATCH
\$'	\$POSTMATCH
\$+	\$LAST_PAREN_MATCH
\$~	\$FORMAT_NAME
\$=	\$FORMAT_LINES_PER_PAGE
\$-	\$FORMAT_LINES_LEFT
\$^	\$FORMAT_TOP_NAME
\$	\$OUTPUT_AUTOFLUSH
\$\$%	\$FORMAT_PAGE_NUMBER

Предметный указатель

Б

Блок, 111
именованный, 132

В

Выражение, 57; 94
терм, 95
побочные эффекты, 25

Д

Дескриптор, 148
-, 169
STDERR, 150
STDIN, 150
STDOUT, 150
предопределенный, 150
файла, 148

К

Ключевые слова, 31
Команда управления циклом, 125
last, 126
next, 128
redo, 129
Конструкторы
массива скаляров, 42
Контекст, 53
скалярный, 53
списковый, 53

Л

Литерал
строковый, 33
числовой, 31

М

Модификатор, 105
foreach, 109
if, 106
unless, 106
until, 107
while, 107

О

Оператор, 104
goto, 133
ветвления, 113
простой, 104
составной, 110
цикла, 116

Оператор цикла

until, 116
while, 116

Операция, 57

!~, 78
", 77
%, 58
&, 71
*, 58
**, 58
, и =>, 87
. (конкатенация), 62
/, 58
^, 72
|, 71
~, 72
+, 58
++, 61
<<, 72
<>, 80; 140; 156
<> без аргумента, 141
=, 74
=~ , 78
->, 78
>>, 72
format, 176
lvalue, 74
qx, 137
арифметическая, 58
выбора, 87
диапазон, 81
"документ здесь", 91
заклЮчения в кавычки, 89
заклЮчения в обратные кавычки ``, 137
"запятая", 86
контекст, 100
логическая, 68
отношения, 65
побитовая, 70
приоритет, 96
разыменования ссылки, 78
составного присваивания, 75
сочетаемость, 99
списковая, 88
списковая операция, 98
ссылка, 77
укороченная схема вычисления, 69
унарный + и -, 60
x (повторение строки), 63
-X (проверка файлов), 170

П

Параметры командной строки, 140

Переменная, 38; 52

\$!, 154

\$., 156

\$/ , 156

динамическая, 112

лексическая, 112

подстановка, 39

пространство имен, 53

скалярная, 39

хеш, 47

Перенаправление стандартного

ввода/вывода, 150

Предопределенные файлы

STDERR, 140

STDIN, 140

STDOUT, 140

Простое слово, 41

Р

Работа с каталогами, 172

дескриптор каталога, 172

С

Структура индексного дескриптора, 168

Т

Тип данных, 29

массив, 42

хеш-массивы, 47

Ф

Файл

владелец, 163

двоичный, 171

жесткие ссылки, 165

заккрытие, 155

запись, 149; 158

константы режимов доступа, 152

открытие, 149

переименование, 166

получение информации, 168

права доступа, 153; 164

режимы доступа, 151

символические ссылки, 165

текстовый, 171

текущая позиция, 158

удаление, 166

усечение, 166

Файловая система UNIX, 162

gid, 162

uid, 162

Формат, 175

верхний колонтитул, 181

переменная \$%, 182

переменная \$^L, 184

переменная \$~, 182

переменная \$=, 181

символы форматирования, 177

строка переменных, 176

строка шаблонов, 176

шаблоны, 177

Функция

chmod(), 164

chown(), 163

close(), 155

die(), 154

getc(), 159

glob, 144

length(), 160

link(), 165

local, 112

lstat(), 169

map, 111

my, 111

open(), 149; 151

print(), 145; 158

read(), 160

rename(), 166

seek(), 159

select(), 158

stat(), 168

symlink(), 165

sysopen(), 152

sysread(), 161

sysseek(), 161

syswrite(), 161

tell(), 158

truncate(), 166

unlink(), 166

utime(), 164

write(), 176

Э

Элементы языка

блок, 26

выражение, 25

знаки операций, 24

идентификатор, 22

ключевые слова, 23

лексема, 22

литерал, 23

операторы, 22

пробельные символы, 24

простой оператор, 25

разделитель, 24