Perl: Llama book distilled

Александр Владимирович Коновалов
avkon@imm.uran.ru

11 февраля 2003 г.

Содержание

1 Perl **1**
1.1 Требования к ПО . . . 1
1.2 Мотивы проектных решений . . . 1
1.3 Литература . . . 1
1.4 Основные структуры данных . . . 2
1.4.1 Скаляры . . . 2
1.4.2 Массивы . . . 2
1.4.3 Хеши . . . 2
1.5 Контексты . . . 4
1.6 Регулярные выражения . . . 5
1.7 Скобочки . . . 6
1.8 Управляющие конструкции . . . 6
1.9 Пример . . . 6
1.10 Локализация . . . 7
1.11 Области применения языка . . . 8
1.12 Задания . . . 8

1 Perl

1.1 Требования к ПО

Любой осмысленный Юникс, XWindow.

1.2 Мотивы проектных решений

“...Иногда вспоминают, что программы пишутся людьми для людей” L.Wall Неоднозначности и диалекты в естественных языках никого не пугают.

1.3 Литература

1. Larry Wall, Tom Christiansen, Randal Schwartz, **Programming Perl**, O’Reilly, 2nd Edition, September 1996. Замечательная книжка от Автора Перла. 2-е изд. доступно в электронном виде. По-русски вышел перевод 3-го издания (Символ-Плюс, 2002).

2. Т. Кристиансен, Н. Торкингтон **Perl: библиотека программиста** - СПб: Изд. "Питер", 2000. Название оригинала более информативно: Perl Cookbook. Оригинал есть в электронном виде.

3. D.Conway **Object Oriented Perl** Manning Publications Company; (August 1999), 490 p. Есть в электронном виде. Полезна обсуждением традиционных языкостроительных проблем в применении к Перлу.

4. S. Srinivasan **Advanced Perl Programming** O'Reilly, 1997. Книжка полезна рассказом о внутренностях Перла. Есть в электронном виде.

5. Р. Шварц, Т. Кристиансен **Изучаем Perl** Базовое подмножество языка. По-моему, слишком базовое, но по случайному совпадению "все" на этом подмножестве и пишут. Предисловие Л.Волла рекомендуется прочитать каждому! Перевод есть в электронном виде.

6. Отдельно необходимо выделить *крайне качественную* документацию, входящую в поставку.

В настоящее время на русском языке вышло довольно много всяких текстов про Перл. Здесь есть 2 тонких момента:

- обычно не имеет смысла покупать скверно переведённый буквальный пересказ документации,

- издательства Sams Publishing и Que замечательны тем, что толковых книжек принципиально не выпускают (дюжину их я просмотрел).

1.4 Основные структуры данных

Магические символы.

- скаляры (символ `$`),

- массивы (они же списки; возможно, это слегка экзотично, но в данном тексте эти понятия будут считаться синонимами) (символ `@`, скобочки `[]`).

- хеши (символ `%`, скобочки `{}`).

1.4.1 Скаляры

Очень хорошие **скаляры**. Всё входит, лишь бы памяти хватило. И строки, и целые, и плавающие — сразу.

```
$a = 12; $a .= "1";
$a++; # <--- 122
```

1.4.2 Массивы

Очень хорошие **массивы**. Динамически растут, пока хватает памяти. Нумеруются с нуля.

```
@a = (1, "aa", 3); # $a[1] хранит "aa"
@b = (44, 55, @a, 66); # @b хранит 44, 55, 1, "aa", 3, 66
```

Возможны "дырки", то есть `$foo[3] = 3; $foo[100_000_000] = 'a';` Что в остальных элементах? Ничего (`undef`). Чудес, однако, не бывает — запаситесь 4 Б на элемент массива, даже если там `undef`.

Безымянные массивы могут возникать как слева, так и справа, например, самый понятный способ поменять местами 2 переменные это `($a, $b) = ($b, $a);`, а переставить первый элемент в конец можно так: `@l = (@l[1..$#l], $l[0]);`. Массивы слева, однако, жадные — после `($a, @l, $b) = (1, 2, 3);` `$a` будет 1, `@l` — `(2, 3)`, а `$b` — неопределённое значение, т.е. пусто. Добавить элемент в список можно просто `@l = (@l, $a);`, но более эффективный способ `push @l, $a`. Аналогично с удалением: `$a = pop @l` вернёт последный элемент и удалит его из списка. Можно манипулировать и с началом (`unshift/shift`) и даже с серединой (`splice`), ужимая массив или вставляя элементы в середину.

Наивные попытки поместить массив в массив приведут к каким-то совсем другим результатам:

```
@l = (1, 2, 3);
@l1 = ('a', @l, 'b'); # станет ('a', 1, 2, 3, 'b'), что естественно
$l2[0] = @l;          # как не странно, 3, а вовсе не вложенный массив
```

Вложенные структуры данных в языке, естественно, есть, реализуются они через **ссылки**, но в данном тексте ссылки более упоминаться не будут. В отсутствие ссылок вложенность можно эмулировать, создавая вре́менные строки с содержимым вложенных массивов. Благодаря быстрым строковым операциям особого урона производительности обычно не наносится.

1.4.3 Хеши

Хеши (правильнее, но слишком длинно, говорить "хеш-таблицы", это же в некоторых других местах называется "ассоциативные массивы") встроены прямо в язык. Коротко напомним, что это такое. Имеем множество потенциальных ключей, с ключом связано значение, требуется обеспечить дешёвые (за постоянное время) операции вставки/удаления/выборки по ключу, при этом операция "дай следующего за текущим ключом" интереса не представляет. Естественно, что если потенциальных ключей немного, их можно бесхитростно сопоставить индексам массива, и все проблемы разрешатся. Если же потенциальных ключей много, массив станет недопустимо большим. Решением проблемы может быть применение к ключам хеш-функции[1], которая сопоставит ключу это самое "место в массиве". Естественно, хорошая хеш-функция должна обеспечить, чтобы два ключа "почти никогда" не отображались на одно значение хеш-функции. Понятно, что придумать такую функцию весьма непросто, если мощность множества ключей сильно больше можности множества значений хеш-функции. Вот, например, что используется в текущем (5.6.1) Перле[2]:

```
#define PERL_HASH(hash,str,len) \
     STMT_START { \
    register const char *s_PeRlHaSh = str; \
    register I32 i_PeRlHaSh = len; \
    register U32 hash_PeRlHaSh = 0; \
    while (i_PeRlHaSh--) \
        hash_PeRlHaSh = hash_PeRlHaSh * 33 + *s_PeRlHaSh++; \
    (hash) = hash_PeRlHaSh + (hash_PeRlHaSh>>5); \
    } STMT_END
```

Итак, процесс помещения в хеш происходит так[3]: берём ключ, считаем от него хеш-функцию, по ней определяем ячейку в массиве, помещаем туда ключ и связанное с ним значение. Проверка, есть ли ключ в таблице, устроена аналогично. А что будет, если разные ключи всё-таки отобразятся на одну ячейку? На самом деле, в ячейке хранится не элемент, а ссылка на список всех элементов, попавших в эту ячейку. Итак, в плохом случае мы получаем вместо хеша линейный список, а в хорошем — операции выполняются за независящее от размера хеш-таблицы время.

Итак в хеше хранятся неупорядоченные пары "ключ-значение". Каждому ключу соответствует единственное значение, возможно, неопределённое. И ключ, и значение — скаляры.

```
%h = ('a', 1, 'bb', 2, 'ccc', 3); # теперь $h{'bb'} будет 2
@words = ('a', 'bb', 'ccc'); # то же в общем виде
foreach $k (@words) {
    $h{$k} = length $k;
}
```

[1] Интеллигентные люди иногда говорят "функция расстановки".

[2] "Трудно придумать функцию много лучшую, чем эта, а вот худшую — легко" Керниган и Пайк, *The Practice of Programming*

[3] На самом деле картина сложнее, но вреда от упрощения быть не должно.

Рассмотрим типичные приёмы работы с хешем как с целым.

Получаем все пары “ключ-значение”.

```
while (($k, $v) = each %h) {
    print "$k=>$v\n";
}
```

Получить список всех ключей можно вызовом `keys`. Так, печать в столбик можно записать и так:

```
for $k (keys %h) {
    print "$k=>$h{$k}\n";
}
```

Необходимо понимать, что `keys` строит список всех ключей, что будет весма заметно для больших хешей. Это не значит, что вариант с `each` всегда лучше — `keys` может быть частью конвейера! Все значения можно получить, вызвав `values`. Понятно, что ключи будут извлекаться совсем не в том порядке, в каком их туда помещали, на то он и хеш, но гарантируется, что для `keys`, `values` и `each` порядок будет одинаковым. Удаляются занчения вызовом `delete $h{'aaa'};`. Если написать `$h{'aaa'} = undef`, то ключ из хеша никуда не пропадёт, просто ему будет соответветствовать неопределённое значение `undef`.

Обращение по ключу выполняется за постоянное время, что позволяет изящно решать многие задачи. Например,

```
open IN, "<besy.txt" or die;
undef $/;
@words = split /[\s"!?.,]+/, <IN>;
@uniq{@words} = ();
@list = keys %uniq;
```

поместит в @list слова из файла `besy.txt` без повторений (в неопределённом, правда, порядке). Здесь возникают несколько тонких моментов. Мы пишем именно `@uniq`, а не `%uniq` потому, что работаем не с хешем как целым, а со списком его элементов. Ключи хеша `%uniq` — неопределённые значения, в пустом списке `()` их хватит на всех. Ту же нехитрую идею можно выразить явным циклом

```
for $l (@words) {
    if (! $uniq{$l}) {
        push @list, $l;
        $uniq{$l}=1;
    }
}
```

Заметим, что решение этой задачи в “школьном” стиле со сканированием массивов и вложенными циклами не только менее асимптотически эффективно ($O(n^2)$ вместо $O(n)$), но и хуже читается, чем вышеприведённые[4]. Приятный пример компромисса между эффективностью и легкочитаемостью!

1.5 Контексты

Как и в любом естественном языке, в Перле значение языковой конструкции зависит от контекста, в котором встречается. Рассмотрим пример.

[4]Автору приходилось наблюдать студентов 4-го курса матмеха, написавших упомянутые циклы и удивлявшихся “А почему программа работает для маленьких данных, а на **Бесах** зависает?” Незабываемые впечатления. Кстати, каждый может проверить свою программистскую интуицию и оценить, как будут различаться времена работ для 207 тыс. слов, из которых 35 тыс. уникальных.

```
$a = ('a', 'b', 'c'); # случай 1: $a будет 'c'
@l = ('a', 'b', 'c');
$b = @l;              # случай 2: $b будет 3
```

Результат разный, и понятно почему — конструкция `a = (a, b, c);` в C уже имеет смысл[5], и нужно стараться не травмировать C-программиста. Интереснее способ, которым этого добились — контексты. Контексты бывают:

- Скалярные.
- Списочные.
- А вот хешевого не бывает!

Общая идея проста — функция, ждущая скаляра, создаёт для вызванного скалярный контекст. Присваивание оказывается ассиметричным: контекст задаётся слева.

Итак, работа примера объясняется тем, что в 1-ом случае запятая почувствовала, что от неё ждут скаляра, и стала работать соответственно, а во 2-м произошла интерпретация списка в скалярном контексте, что в данном случае означает "вернуть последний индекс". Узнать текущий контекст можно (функцией `wantarray`), как и установить требуемый, но лучше без особо веских причин этим не увлекаться.

1.6 Регулярные выражения

Мощный механизм, использующий для работы со строками формализм *конечных автоматов* (finite state machine). Две основные возможности:

- Найти нечто в строке. `$s =~ /рег.выр./`
- Заменить нечто в строке. `$s =~ s/рег.выр./зам./`

Сложности здесь двоякого рода: во-первых, регэкспы — это новый (непроцедурный) язык программирования, к которому надо привыкнуть, а во вторых, дело осложняется изобилием в этом языке крайне значимых закорючек-зюквочек. Дело, однако, стоит потраченных усилий: те же цели достигаются на "наивном C" в десятки раз более длинными программами, а читать и модифицировать их — одно мучение.

Перейдём к основам. Обычный символ (буква, цифра, пробел) сопоставляется сам с собой. Точка `.` — с любым символом. Перечисление в квадратных скобках значит "любой из". Итак, `/a.[bc]/` сопоставится с `azb`, `a b` и `a1bc`, но не `ab`. `\s` означает "любой символ, похожий на пробел", т.е. табуляция `\t`, перевод строки `\n`, возват каретки `\r` и пр., включая собственно пробел. `+` означает "допустим повтор 1 или более раз", `*` — "0 или более", а `?` — 0 или 1 раз. `|` обеспечивает отбор одного из вариантов. Итак, `/a(bc|1)+d/` сопоставится с `abcd` и `a1bcbcd`, а с `a1bcbd` — нет.

Какой же толк в сопоставлении? Толк здесь примерно 3-х сортов: можно узнать, что сопоставилось, можно узнать, что именно сопоставилось, и можно сопоставившиеся заменить.

Узнать, сопоставилось ли, можно, вызвав сопоставление в скалярном контексте: если ДА, то вернут истину.

Что именно сопоставилось, можно узнать двумя способами: убедить сопоставление вернуть список из сопоставившегося (после `@l = $s =~ /a(.+)b(.)+c/;` в @l будет ('zz', 3), либо прочитать нечто похожее из специальных read-only переменных (`$&` — сопоставившаяся часть строки, `` $` `` — строка до сопоставившейся части, `$'` — строка после, `$1` — содержимое 1-х круглых скобок, `$2` — содержимое 2-х и т.п., например, после `$s = 'ab11cde'; $s =~ /b(.+)c/;` в `$&` попадёт `b11c`, в `` $` `` — `a`, в `$'` — `de`, а в `$1` — `11`).

[5]Может возникнуть вопрос: "А причём тут вообще C?" При том, что одинаково выглядящее должно работать одинаково, а первоначальный круг писателей на Перле читал C без словаря.

Пример замены сопоставившегося: `$s = 'abcdef'; $s =~ s/(.).(.)/$2$1/;`. Теперь в `$s` будет `cadef`. Часто бывает нужно заменить все вхождения, а не только первое. Это обеспечивает ключ g: после `$s = 'abcdef'; $s =~ s/(.).(.)/$2$1/g;` в `$s` будет `cafd`. Иногда нужно не *подставить*, а *исполнить* то, что образовалось справа. Это делается указанием ключа `e`. Так, после `$s = '1234567'; $s =~ s/(.).(.)/$1+$2/eg;` в `$s` будет `4107`, если же `e` убрать, то, понятно, совсем даже `1+34+67`.

Создавая регулярное выражение, приходится решать сразу 2 задачи:

- Все нужные строки должны сопоставляться.
- Все ненужные сопоставляться не должны.

Иногда написание удаётся резко упростить, если ориентироваться на входной поток специального вида, а не решать задачу в общем случае.

Разберём содержательный пример. Имеем строку `$s = 'Колбаса $8, сало $4.13, тонер $3';`, содержащую цены в долларах. Попытаемся написать регулярное выражение, преобразующее цены в рубли. Оказывается, это довольно просто: `$s =~ s/\$(\d+(\.\d+)?)/$1*31/ge;`

1.7 Скобочки

Уже к настоящему моменту видно, что Перл изобилует скобочками. Ну, про `{}`, `()` и `[]` мы говорить не будем, остановимся сначала на `''` и `""`. Разница между ними проявляется, когда внутри встречаются переменные. А именно, после `$a = "abc"; $b="123$a 45"` `$b` станет равным `123abc 45`. Этот процесс называется математико-механическим словом *интерполяция*. Проводится ровно один раунд интерполяции, то есть если после первого круга подстановок результирующая строка вновь содержит пробелы, это уже никого не волнует. Находясь в двойных кавычках, избегнуть интерпретации `$` как начала имени переменной можно, если "заэскейпить" его: после `$a=1; $b=2; $c="a\$a$b";` в `$b` будет `a$a2`.

Одинарные кавычки отличаются от двойных тем, что в них интерполяция не выполняется. Для сравнения: `$a=1; $b=2; $c='a\$a$b';` поместит в `$b` строчку `a\$a$b`.

1.8 Управляющие конструкции

Любимые россиянами `if`, `for` и `while` никуда, естественно, не делись. Правда, рисование скобок `{ }` — обязательно. Блоки можно именовать, что позволяет вываливаться из вложенных блоков. `break` и `continue` из C подверглись переименованию — в Перле это `last` и `next`.

В формате `for $e (список)` оператор `for` превращается в совсем другой оператор: `@l=(10..33); for $e (@l) {print $e;}` создаст список из целых от 10 до 33 включительно, а затем напечатает все элементы списка. Вместо `for` можно писать `foreach`, что значит ровно то же.

Замечательной разновидностью "управляющих конструкций" в Перле являются пришедшие из функционального программирования `grep` и `map`. Принцип их работы прост и понятен: `map { $_+1 } (1, 2, 3)` вернёт `(2, 3, 4)`, т.е. первый аргумент применяется к каждому элементу списка и результат возвращается, при этом текущий элемент помещается в *переменную по умолчанию* `$_`. grep устроен аналогично одноимённой Unix-овской утилите: `grep { $_ % 2 } (0..10)` вернёт нечётные числа в диапазоне от 0 до 10, т.е. если выражение в фигурных скобках для текущего элемента — истина, то элемент копируется в результат. Отметим, что иногда нам нужен не результат, а побочный эффект от вызовов функций, например, `map {print "$_\t"} @l` распечатает элементы списка @l, разделив их табуляциями.

Возникает вопрос: зачем нужны `map` и `grep`, если всем известные операторы цикла уже есть? Огромное преимущество этих конструкций —

в возможности объединять их в цепочки-конвейеры, например, `map { "$k{$_}:$_" } sort { $k{$b} <=> $k{$a} } grep {! $k{$_}++ } @l` вернёт список уникальных элементов `@l` с указанным перед элементом количеством повторений. Конечно, нет никаких проблем написать это, используя циклы, но читаемость map-sort-grep варианта выше: стадии здесь чётко отделены друг от друга.

1.9 Пример

Представлен CGI-скрипт, разбирающий журнал регистрации веб-сервера и выдающий 10 самых популярных URL-ов на данном сервере. Каждый клиент считается лишь однажды, вне зависимости от того, сколько раз он заходил на данный URL. Заведомо вредительские либо служебные обращения игнорируются.

```
#!/usr/bin/perl -w
my $i;

@ignore_url = qw{ /al.css /scripts/root.exe?/c+dir
  /[cd]/winnt/system32/cmd.exe?/c+dir
  /(_vti_bin|_mem_bin)/..%255c../..%255c../..%255c../winnt/system32/cmd.exe?/c+dir
  /scripts/..(%C1%1C|%c0%2f|%C0%AF|%C1%9C)../winnt/system32/cmd.exe?/c+dir
  /MSADC/root.exe?/c+dir };
# @ignore_url = map { s/[.+?]/\\$&/g; $_ } @ignore_url;
@ignore_url = map { s/[.+?]/\\$&/g; qr/$_/ } @ignore_url;

open LOG, "/var/log/httpd/access_log" or die "Can't open ($!)";
LINE: while ($line = <LOG>) {
    if (($host, $url) = $line =~ /^(\S+) .+\"GET (\S+) HTTP/) {
        if ($visited{$url}
            and -1!=index $visited{$url}, " $host ") {
            next;
        }
        for $patt (@ignore_url) {
            next LINE
                if $url =~ /$patt/;
        }
        $visited{$url} .= " $host ";
        $hosts{$url}++;
    }
}
close LOG;

print "Content-type: text/html\n\n<html><body>\n";
for $url (sort { -$hosts{$a} <=> -$hosts{$b} } keys %visited) {
    if (++$i > 10) {
        last;
    }
    print "<a href=\"$url\">$hosts{$url}</a>\n";
}
print "</body></html>\n";
```

Пример лога.

```
213.243.76.33 - - [07/Jul/2002:21:04:25 +0600] "GET /alpha/f90www/ HTTP/1.0" 200 5805
213.243.76.33 - - [07/Jul/2002:21:04:30 +0600] "GET /alpha/f90www/image16.gif HTTP/1.0" 200 1990
213.243.76.33 - - [07/Jul/2002:21:04:30 +0600] "GET /alpha/f90www/Image14.gif HTTP/1.0" 200 7136
213.243.76.33 - - [07/Jul/2002:21:04:30 +0600] "GET /alpha/f90www/image15.gif HTTP/1.0" 200 991
213.243.76.33 - - [07/Jul/2002:21:20:33 +0600] "GET /alpha/fortran-imm/fortran_sites.html HTTP/1.0" 200 16076
213.243.76.33 - - [07/Jul/2002:21:29:17 +0600] "GET /alpha/fortran-imm/fortran_sites.html HTTP/1.0" 304 -
213.146.148.40 - - [07/Jul/2002:21:30:06 +0600] "GET /alpha/f90www/rel_op_rus.html HTTP/1.1" 200 3468
213.180.194.163 - - [07/Jul/2002:21:32:39 +0600] "GET /robots.txt HTTP/1.1" 200 0
213.180.194.163 - - [07/Jul/2002:21:32:39 +0600] "GET /1000/pvfs-4iods-perf.jpg HTTP/1.1" 304 -
66.196.65.27 - - [07/Jul/2002:22:28:47 +0600] "GET /robots.txt HTTP/1.0" 200 0
```

```
213.146.148.40 - - [08/Jul/2002:02:42:57 +0600] "GET /alpha/f90www/log_expr_rus.html HTTP/1.1" 200 7026
68.33.218.20 - - [08/Jul/2002:05:28:28 +0600] "GET /pubs/progr01-averbukh.ps.gz HTTP/1.1" 200 26530
81.19.66.8 - - [08/Jul/2002:07:02:33 +0600] "GET /robots.txt HTTP/1.0" 200 0
81.19.66.8 - - [08/Jul/2002:07:10:33 +0600] "GET /robots.txt HTTP/1.0" 200 0
81.19.66.8 - - [08/Jul/2002:07:27:09 +0600] "GET /robots.txt HTTP/1.0" 200 0
194.248.137.189 - - [08/Jul/2002:07:29:18 +0600] "GET /scripts/..%255c%255c../winnt/system32/cmd.exe?/c+dir" 404 -
66.28.250.174 - - [08/Jul/2002:08:39:25 +0600] "GET /robots.txt HTTP/1.0" 200 0
66.28.250.174 - - [08/Jul/2002:08:39:25 +0600] "GET /100/graphics.txt HTTP/1.0" 200 7766
66.28.250.171 - - [08/Jul/2002:09:34:46 +0600] "GET /robots.txt HTTP/1.0" 200 0
66.28.250.171 - - [08/Jul/2002:09:34:46 +0600] "GET /pubs/abrau00-skis.html HTTP/1.0" 200 9097
212.122.17.131 - - [08/Jul/2002:11:16:05 +0600] "GET /alpha/f90www/prim7.html HTTP/1.1" 200 3621
```

Время работы составляет 7.5 сек. с `qr` (и 28.5 с без) для лога из 92 тыс. строк (9 МБ), чтение файла занимает 0.4 сек.

1.10 Локализация

Вставте

```
use locale;
use POSIX qw (locale_h);
setlocale(LC_CTYPE, "Russian_Russia.1251");
```

с правильной кодировкой ("Russian_Russia.1251"или "Russian_Russia.866", гордые жители Юникса могут выбрать из выдачи `locale -a` по вкусу), и будет вам счастье (в виде пречисления русских букв к буквам, перевода больших букв в маленькие и работающей лексикографической сортировки).

1.11 Области применения языка

- Универсальный ЯП высокого уровня. Прототипы. + — хороший переносимый язык, CPAN. − — инструментальный средства.
- “Строительные леса” для разработчика и “швейцарская армейская бензопила” сисадмина, т.е. собственно “язык сценариев”. + — компактные мощные программы, − — если не поменять стиль, по с увеличением объема получится УЖАС.
- Web-программизм. Уходящая натура родом из 97-го. CGI. + — исторические. − — квалификация существующих Web-программистов требует других средств.

1.12 Задания

1. Павлом Чистопольским написан мини-http сервер phttp (140 строк на Перле). Задание состоит в его развитии (поддержка раздачи картинок, многопользовательской работы и пр.)
2. Для всех хешей в Перле используется одна и та же хеш-функиция. Найдите ключи, вызывающие коллизию, и посмотрите, что произойдёт с производительностью программ, использующих хеши.
3. Всё-таки 7.5 сек. — это довольно много. Найдите узкие места и ускорьте разбиратель лога.
4. Напишите раскрашиватель исходных текстов программы для LaTeX-а (на входе — например C, на выходе — красивый LateX).