Волгоградский государственный университет
Кафедра информатики и экспериментальной математики

Обработка текста средствами Perl. Краткий конспект лекций

В.А.Клячин

Волоград 2007

Глава 1

Организация структур компонент текста средствами Perl

1.1 Обзор возможностей языка Perl

1.1.1 Введение в Perl

Где взять Perl. В каком программировании пртменяется Perl обработка текста + CGI + системное программиорвание в среде UNIX (реже в Windows). Автоматически интерпретатор Perl поставляется с дистрибутивами UNIX подобных систем. Официальный сайт www.perl.com. Версию под Windows можно взять у меня.

Переменные в Perl

В Perl бывают переменные трех типов. Скалярна переменная число или строка, массив список скаляров с числовой индексацией и хеш список скаляров с строковым типом индексации. Эти три типа характеризуются символами перед именем пременной:

- $ – для скаляров
- @ – для массивов
- % – для хешей.

Для задания строк можно использовать как двойные так и одинарные кавычки. Но двойные кавычки производят интерполяцию переменных и обратной косой черты, а одинарные кавычки подавляют эту интерполяцию. Примеры:

```
$answer =43     --          целое число

$pi=3.14159     --          число с п<лвающей точкой

$a=6.02e23      --          число в виде экспоненты

$pet="Camel"    --          строка

$sign="I love my $pet" --   строка с интерполяцией

$cost='It costs $100' --    строка без интерполяции

$exit=system("vi $file") -- статус завершения программы

$cwd=`pwd`               -- строка выдачи команды
```

Инициализация массива
@home=("стол", "стул","печка");
Обратно, присвоить значения переменным, значениями из массива @home можно так

```
($a,$b,$c)=@home;
```

Обращение к элементу массива просходит в скалярном контексте:

```
$home[0]="стол";

$home[1]="стул";
 ...
```

Функции push(), pop() позволяют добавлять новый элемет в конец массива и удалять последний элемент массива.

```
$new=2;
push(%home,$new);

print $home[3];

$old=pop(%home);
```

```
print $old;

print $home[3];
```

Хеш представляет собой неупорядоченный список скаляров, к которым обращение производится не по индексу, а по строке. В программе хеши задаются так

```
n (ключе1, значе1, ключе2, значе2, ... , ключеn, значеn)
```

Приведем пример

```
%longday=(

"Пн" => "Понедельник",

"Вт" => "Вторник",

"'Ср" => "Среда",

"Чт" => "Четверг",

"Пт" => "Пятница",

"Сб" => "Суббота",

"Вс" => "Воскресение"

);
```

Обращение происходит с помощью фигурных скобок, вот так

```
$longday{"Ср"}="Среда";
```

Построение многомерных массивов и хешей. Предположим, что имеется хеш @friends, где индексом является имя человека, а значением имя его друга. Но что делать если друзей больше одного. К сожалению, естественное решение в виде

```
$friends{"Vanya"}=("Sasha","Sveta","Petya");
```

будет не верным, поскольку слева стоит скаляр, а справа массив. Тем не менее в Perl существует явное указание присвоение сниска скаляру (который в свою очередь представляет собой ссылку на массив!!) с помощью квадратных скобок:

```
$friends{"Vanya"}=["Sasha","Sveta","Petya"];
```

Теперь обращаемся к прлученным данным так

```
$friends{"Vanya"}[0]="Sasha";

$friends{"Vanya"}[1]="Sveta";
```

Теперь предположим, для каждого друга мы хотим включить в хеш его сестер и братьев (друга имеется в виду). Заметьте естественность решения задачи:

```
$brothersesisterseofefriends{"Vanya"}={

"Sasha" => ["Oleg","Natasha"],

"Sveta" => ["Igor","Dima"],

"Petya" =>["Viktor","Olya","Jenya"]

};
```

К такому хешу доступ осуществляется тоже естествнным образом:

```
$brothersesisterseofefriends{"Vanya"}{"Sveta"}[1]="Igor";
```

Для вывода в окно значений переменных используется функция print:

```
$hel="Hello, friends!\n";

print $helo;
```

выводится строка "Hello, friends!"с переходом на следующую строку. Если имеется хеш

```
@longdayemassive=%longday
```

превращает его в список

```
("Пн","Понедельник","Вт","Вторник",...,"Вс","Воскресенье");
```

Перечислим некоторые полезные функции работы с хешами. Функция keys(Функция values(

Функция keys(

Функция values(

```
$list=keys(%longday);
```

Выполнение скрипта. Простейшая программа:

```
#!/bin/usr/perl

#етR кRмме-тарис

$hel="Hello, friends!\n";

print $helo;
```

Запускаем скрипт из командной строки

```
% perl prog.pl

Hello, friends!

%
```

"Дскрипторы файлов. Система ввода вывода

Дескриптор файла – специальная переменная, предназначенная для выполнения файловых операций ввода/вывода.

Идентификаторы STDIN, STDOUT используются в качестве ввода вывода в консоль. Для создания дескриптора файла используется функция open().

```
open(FILE,"filename") чтение из существующего файла

open(FILE,"<filename") чтение из существующего файла

open(FILE,">filename") создать файл и производить в него запись

open(FILE,">>filename") дописывание к существующему файлу

open(FILE,"| выходная программа конвейера") организовать выходной
фильтр

open(FILE," входная программа конвейера |") организовать входной
фильтр
```

Заметим, что следующая операция

```
print STDOUT "Hello";
```

эквивалентна

```
print  "Hello";
```

Для чтения с клавиатуры используется оператор <>:

```
print "Введите строку\n";

$in=<>;

print "Вы ввели $in\n";
```

Для чтения из файла в треугольных скобках необходимо указать дескриптор файла, открытого для чтения.

```
print "Hello!!\n";

open(FIL,">c:\\some.txt");
```

```
print FIL "Hello all!!!\n";

open(DAT,">data.dat");

print "Input value i\n";

$i=<>;

 print "Input value j\n";

 $j=<>;
print DAT $i,$j;

close(DAT);

open(DAT,"<data.dat")or die "Can't to open file";

$i=<DAT>; $j=<DAT>;

print $i+$j , "\n";
```

Чтение из файла прооисходит по-строчно до символа перевода строки. Причем этот символ добавляется в прочитанную строку. Для его удаления используется функции chop(), chomp(): первая удаляет последний симвл в строке, тогда как вторая удаляет "/ n".

```
$line=<FILE>;

chomp($line);
```

Пример

```
$line="Hello!!!\n";

print $line;

chomp($line);

print $line;

print "You see";
```

Скалярный и списочный контексты

Всякая операция Perl выполняется в конкретном контексте. Существует два основных контекста скалярный и списочный. Например, если происходит опреация присвоения скалярной переменной , скалярному элементк массива или хеша, то эта опрерация выполняется в скалярном контексте:

```
$x=somefunk();

$x[1]=somefunk();

$x{"ray"}=somefunk();
```

А вот присвоения в списочном контексте

```
@x=somefunk();

@x[1]=somefunk();

@x{"ray"}=somefunk();

%x= somefunk();
```

Вот пример выполнения функции в скалярном контексте,

```
@list=(1,2,3,4,5,"10");

print $list[2];
```

а вот ее выполнение в списочном контексте

```
@list=(1,2,3,4,5,"10");

print @list;
```

Проверьте в чем разница вывода на консоль в этих двух примерах. Упомянем здесь еще функцию join() осуществляющую соединение массива в строку с заданным разделителем:

```
@list=(1,2,3,4,5,"10");

 $temp=join($",@list);
print $temp."\n";

print "@list";
```

Наконец функция split(); Первый ее аргумен разделтель (в общем случае регулярное выражение), второй аргумент строка для разделения. Пример

```
$str="To be or not to be";

 @temp=split($",$str);

$"="\n";

print "@temp";
```

Пакеты и пространства имен

Для классификации имен переменных и для удобства слежением за ними в Perl введены понятия пространства имен, которе задается в т.н. пакетах. Предположим у нас имеется пакет с именем Camel. В самом начале пакета мы указывает на пространство имен таким образом:

pakcage Camel;

Это означает, что при использовании внутри данного модуля переменные вида

```
 $fido
```

на самом деле будет иметь вид

```
$Camel::fido
```

Для использования переменных и функций из пакета Camel мы должны в программе записать строку

```
use Camel;
```

Тогда возникает естественный вопрос что будет означать переменная

```
$fido
```

используемая в программе? Таким образом мы подошли к очень важному вопросу поиску имени. Прежде всего заметим, что в Perl существует два вида пространств имен таблицы имен и области лексической видимости. Таблицы имен представляют собой хеши глобальных переменных из пакетов, области лексической видимости это анонимные временные выделения памяти (динамически выделяемая память) в которых переменная становится видимой в пределах определенного блока в программе. Для определения переменных с лексичойской областью предназначен опрератор my. В общем случае при использовании переменной типа $ var подразумевается принадлежность ее определенному пакету (пространству имен). Следует привести алгоритм поиска имени интерпретатором Perl

- Если в ближайшем блоке, содержащем переменную есть ее объявление с помощью my , то эта переменная считается с лексической областью видимости действительна внутри этого блока и всех блоках вложенных в него. Если такого объявления нет интерпретатор переход на 1 блок выше и осуществляет поиск там. Таким образом определяются переменные с лексической областью видимости, максимальной из которых является исходный файл.

- На данном этапе интерпретатор считает, что переменная не является переменной с лексической областью видимости и пытается найти подключенный пакет, в котором может находится объявление переменной. Но если включена директива use strict; то интерпретатор выдает ошибку, если не находит пакета с соответствующим определением. Будьте осторожны при использовании PerlBuilder в нем use strict не работает!!! Найдя нужный пакет или если отключено strict интерпретаотор автоматически добавляет к имени имя пакета $ Camel::var – для найденного пакета, или

  ```
  $::var==$main::var -- во втором случае (без strict).

  $Camel::var -- для найденного пакета,

  или

  $::var==$main::var -- во втором случае (без strict).
  ```

Все вышеперечисленное относится и к функциям, о чем будет особо отмечено ниже.

Операции и операторы Perl

Бинарные арифметические операции с числовыми данными:

```
$a+$b -- c<Rже-ие,

$a*$b -- ум-Rже-ие,

$a%$b -- де<е-ие с RстаткRм

$a/$b -- де<е-ие

$a**$b -- вRзведе-ие в степе-ь
```

Для строковых данных определена операция сложения строк (операция конкатенации) оператор точка (.) :

```
$a=123;

$b=456;

print $a+$b -- вывRдит 579

print $a.$b  -- вывRдит 123456
```

Есть так же опреация умножение строк:

```
$a=123;

$b=3;

print $a*$b -- вывRдит 369

print $a x $b -- вывRдит 123123123
```

Как и в языке Си определена операция присвоения с вычислением операции.

```
$a (Rперация)= выраже-ие
```

еапример

```
$line.="\n";
```

означает, что в конец строки добавлен символ "/n". А так же допустимы операции инкремента и декремента:

```
++$i, $i++ -- прибавить 1
```

```
--$a, $i-- вычесть 1
```

Что есть истина? Истинным является выражение значение которого не равно 0, или не равно "0", или не равно пустой строке. Любое определенное значение является истинным. Любая ссылка есть истина даже, если указывает на переменную имеющую значение false. Так, что

```
10 - 10 -- ложь
```

```
0.00 -- ложь
```

```
"0"  -- ложь
```

```
"0.00" -- истина
```

Логические операции

```
$a && $b или $a and $b -- И
```

```
$a || $b или $a or $b -- ИЛИ
```

```
$a xor $b исключающее ИЛИ
```

```
! $a  not $a  -- НЕ
```

Операторы сравнения:

Операция	число	строка
Равно	==	eq
Не равно	!=	ne
меньше	<	lt
больше	>	gt
меньше или равно	<=	le
больше или равно	>=	ge

Операторы проверки файлов

Пример	Название	Результат
-e $a	Существует	Истина, если файл с именем $a существует
-r $a	Доступен для чтения	Истина, если файл с именем $a доступен для чтения.
-w $a	Доступен для записи	Истина, если файл с именем $a доступен для записи.
-f $a	Файл	Истина, если файл с именем $a является обычным файлом.

-d $a	Каталог	Истина, если файл с именем $a является каталогом.
-t $a	Текстовый файл	Истина, если файл с именем $a является текстовым файлом.

Операторы

Условные операторы

```
if(выраже-ие) {


} elsif(выраже-ие) {



} . . .

else {


}
```

Пример.

```
$city=<CITIES>;

if($city eq "Volgograd") {

print "Volgograd on the right bank of Volga \n";

 }
 elsif($city eq "Saratov")
{
```

```
print "Saratov on the right bank of Volga \n";

} elsif($city eq "Samara") {

print "Samara on the left bank of Volga\n";

} else { print "Sorry, ....\n" }
```

Следующий оператор

```
unless(условие) {

} else {

}
```

Операторы цикла

```
while(условие) {

# тело цикла
 }
```

Результат некоторых операций можно рассматривать в условном (булевом) контексте т.е. в условных выражениях. Операция присваивания возвращает скалярное значение в скалярном контексте и именно оно участвует в определении истинности. Например

```
while($line=<FILE>)

{

}
```

Здесь в цикле читается строки из файла до тех пор пока в $ line не будет прочитана пустая строка, соответствующая концу файла. При вводе пустой строки в файл к нему автоматически добавляется символ /n по-настоящему пустой строкой будет строка, возвращенная в $ line после прочтения всех строк данного файла. Вот еще примр использования в булевом контексте

```
@arg=(1,2,3,4,5,6,7);

$n=@arg;

print $n;
```

данная программа выведет на экран значение 7. Поэтому в цикле

```
while(@ARGV) { print shift @ARGV }
```

происходит вывод элементов массива. Здесь функция shift выбирает из массива оче- редной элемент. Следующий оператор

```
for(выр1,выр2,выр3) {

}
```

Здесь выр1 начальное выражениеб выр2 условное выражение истинность которого обеспечивает выполнение тела цикла на каждой итерации и выр3 выражение приращения.

```
for($i=0;$i<100; $i++) { $s+=$i; }
```

Оператор foreach. Синтаксис:

```
foreach $var (@massiv) { # тело цикла }
```

Здесь последовательно переменной $ var присваиваются значения из массива @massiv. Следует обратить внимание, что переменная $ var ссылается на сам элемент массива, а не на его копию, поэтому модификация переменной цикла в теле цикла приводит к изменению элемента массива.

```
@arg=(1,2,3,4,5,6,7); foreach $var (@arg) { $var+=3; } print
"@arg";
```

Досрочный выход из цикла можно осуществить с помощью операторов next и last. Первый является аналогом оператора continue, а второй оператора break в языке Си. Пример

```
@users=("somebody","root","apache","mysql","postrgesql","msql","firebird","user");
foreach $user (@users) { print $user."\n"; if($user eq "root" ) {
next; } if($user eq "mysql") { print "Find user $user.\n"; last; }

}
```

А вот вывод этой программы

```
somebody

root

apache

mysql

Find user mysql
```

Разноообразие кавычек

В Perl вместо кавычек " можноиспользовать произвольные разделители из буквенно- цифрового набора, за исключением пробела и перевода строки. Вот соответствующая таб- лица

обычные	общие	значение	Интерполяция
''	q//	строка	нет
""	qq//	строка	да
``	qx//	выполненеие команды	да
()	qw//	заключение слов в кавычки разделенных пробелами	нет
//	m//	поиск по шаблону	да

```
s///              s///      замена по шаблону      да

y///               tr///     трансляция символов   нет

""                 qr//      регулярное выражение   да
```

Вот, к примеру предыдущий пример

```
@users=qw!somebody root apache mysql postrgesql msql firebird
user!;
 foreach $user (@users) {
  print $user."\n";
  if($user eq
"root" ) {
 next;
  }

  if($user eq "mysql") {

   print "Find user
$user.\n"; last;

 }

}
```

А вот еще интересней

```
@users=qw!somebody root apache mysql postrgesql msql firebird
user!;

print qq?@users?;
```

Этот пример выводит всех пользователей.

Встроенный документ

Встраивание документа осуществляется следующей конструкцией

```
print <<END;

some text of

document

and other and other ... END
```

Между << и идентификатором не должно быть пробелов если только он не заключен в кавычки.

Подпрограммы

Синтаксис объявления функции, процедуры или подпрограммы.

```
sub NAME

sub NAME PROTO

sub NAME       ATTRS

sub NAME PROTO ATTRS
```

Здесь NAME - имя подпрограммы, PROTO, ATTRS – прототип и атрибуты. Для определения подпрограммы необходимо добавить блок BLOCK

```
sub NAME             BLOCK

sub NAME PROTO       BLOCK

sub NAME       ATTRS BLOCK

sub NAME PROTO ATTRS BLOCK
```

Так же возмож-но бъявление и определение неименноанной подпрограммы

```
sub                   BLOCK

sub      PROTO        BLOCK

sub             ATTRS BLOCK

sub      PROTO ATTRS BLOCK
```

Но в этом случае необходимо иметь способ вызова такой процедуры. Например

```
$subref = sub BLOCK
```

В этом случае возвращаемое значение получается во время компиляции. Перечислим способы вызова подпрограмм

```
NAME(LIST)

NAME LIST -- если имеет объявление или импортирована из модуля

&NAME  -- передает в функцию текущее значение глобальной
переменной @_

sub prog{ print @e; }

prog 34;
```

Подпрограмма получает параметры в виде одного списка скаляров, помещаемых в массив @_. Доступ к отдельным элементам этого массива, т.е. отдельным параметрам, осуществляется так же как и для обычного массива.

```
$_[0] -- 1-й аргумент

$_[2] -- 2-й аргумент и т.д.
```

Функция может возвращать значение либо с явным указанием в операторе return, либо как последнее вычисленнRe значение:

```
sub prog{ @r=($e[0],$e[1]);
```

```
return @r;
 }

@s=prog 34,56;

print "@s";
```

В следующем примере вызов функции осуществляется в скалярном контексте, поэтому программа напечатает размер возвращаемого массива, т.е. число аргументов подпрограммы.

```
sub prog{
 @r=($_[0],$_[1],$_[2]);
 return @r;
 }

$s=prog 34,56; print "$s";
```

Отметим, что передача параметров осуществляется по ссылке, т.е. меняя значения массива @_ мы автоматически меняем значение фактического аргумента. Пример

```
sub prog{ $e[0]=23;
 @r=($_[0],$_[1]);
 return @r;
 }
  $i=34;
@s=prog $i,56;

print "$i";  -- печатает 23, а не 34!!!
```

Поэтому часто используют локальные переменные в фуукции с лексической областью видимости

```
sub prog{

my ($a,$b)=@_;

$a=23;
```

```
@r=($a,$b);

return @r;

}

$i=34;

@s=prog $i,56;

print "$i";
```

Определение функции можно вынести в конец файла, но при этом необходимо предварительно ее объявить:

```
sub prog;

$a=88;

$i=34;

@s=prog $i,56;

print "$i"." $a";

sub prog{

my ($a,$b)=@_;

$a=23;

@r=($a,$b);

return @r;

}
```

Можно прямо в функции проверить в каком контексте она вызывается в списочном или в скалярном. Для этого служит ключевое слово wantarray. Вот примеры

```
sub prog{

my ($a,$b)=@_;

@r=($a,$b);

return wantarray ? @r:11;

}

@x=prog(44,55);

print @x; -- напечатает 44,55

или

sub prog{

my ($a,$b)=@e;

@r=($a,$b);

return wantarray ? @r:11;

}

$x=prog(44,55);

print $x; -апечатает 11!!!
```

Заметим, что без wantarray в скалярном контексте возвращается размер массива (см. выше).

e-Reда пR<ез-Rс мRжет быть фу-кция shift();

```
shift(ARRAY)

shift
```

Эта функция возвращает первый элемент массива и сдвигает его уменьшая размер на единицу. Если в подпрограмме отсутствует массив, то считается массив @_. Например в последнем примере

```
sub prog{

my $a=shift; my $b=shift;


@r=($a,$b);

return wantarray ? @r:11;

}
```

В функцию можно передать хеш.

```
sub prog{

my %a=@_;
my $i=$a{"1"}; print "$i"; }

prog("1" =>"one","2"=>"two","3"=>"three");
```

Отметим каким образом используется прототип функций. Это понятие несколько отличается от аналога в Си. Прототип задает возможный вариант для количества и типов параметров функций.

```
sub mylink ($$)   mylink $old, $new

sub myreverse(@)  myreverse $a,$b,$c

sub myjoin($@)   myjoin ":",$a,$b,$c

sub mypop (\@)  mypop @array

sub mysplice (\@$$@)  mysplice @array, @array,0,@pushme

sub mykeys (\%) mykeys %{hashref}
```

```
sub myindex ($$;$) myindex $a,$b

                  myindex $a -- ошибка
                  myindex $a ,$b,$c,$d -- ошибка
а в этом примере ошибки нет

sub prog($$$$$;$@)  prog(1,2,3,4,5,6,7);
```

Отметим некоторые особенности применения прототипа. Каждый символ прототипа с обратной косой чертой представляет фактический аргумент, который должен начинаться с того же символа. Каждый символ % без косой черты подавляет все остальные аргументы представляя аргументы в виде одного списка. Символ & должен быть представлен в вызове ссылкой на функцию.

Здесь ошибка.

```
sub prog(\@) { print "@e"; }

@x=(1,2,3,4,5);

 prog(1,2,3,4,5); -- ошибке
 prog(@x); -- нет ошибки
```

Несколько примеров

```
sub prog($) {
 #$p=$e[0];
 print $e[0]->[0];
  }

$m[0]="Hello"; $m[1]="world!";

prog \@m;

Выводит "Hello";

sub prog(\@) {
```

```
$p=\@e;

print $p->[0]->[0]; # or

print ", ".@e[0]->[1];

}

$m[0]="Hello"; $m[1]="world!"; prog @m;
```

А вот так можно передавать функции функцию в качестве аргумента

```
sub prog(\&$) {

my ($f,$x)=@_;

&$f($x);

}

sub myf {

print "Hello"."$_[0]\n";
 }

prog (&myf,", world!");
```

Эта программа выводит Hello, world!.
И здесь то же

```
sub prog(&$) {
 my ($f,$x)=@e;

&$f($x);
 }
  sub myf {
  print "Hello"."$e[0]\n";
```

```
  }

prog (\&myf,", world!");
```

Функция eval()

Синтаксис этой функции такой

```
eval BLOCK

eval EXPR

eval
```

Функция sort()

Синтаксис этой функции такой

```
sort USERSUB LIST

sort BLOCK LIST

sort LIST
```

Функция сортирует массив , заданный в парамтре LIST и возвращает отсортированный массив.

USERSUB если присутствует, представляет собой имя подпрограммы, которая возвращает целое число, меньше, равное или больше 0, в зависимости от того как должны быть упорядочены элементы массива. Параметры этой функции не передаются через массив @_, а через глобальные переменные $ a, $ b того пакета, в который была скомпилирована sort(). Для сортировки обычных числовых массивов

```
sub numerically {$a <=> $b}

@sorted=sort numerically @m;
```

А так можно отсортировать по убыванию

```
sub numerically {$b <=> $a}

@sorted=sort numerically @m;
```

Наконец, данную функцию мжно применить для сортировки хеша по значениям. Вот решение задачи

```
sub byval {$h{$a}<=>$h{$b}}

for $key (sort byval keys %h){

print "$key => $h{$key}";

}
```

Ссылки

Понятие ссылки в Perl несколько отличается от понятия указателя в Си. Пусть задана переменная

```
$x="Linux";
```

Тогда ссылку на эту переменную можно определить так

```
$p=\$x;
```

Можно представлять все дело так. Интерпретатором создается таблица имен переменных, и с каждым именем связан адрес расположения в памяти значения адреса. Ссылка – обычная переменная, имеющая значеним адрес , где содержится значение переменной. Такие ссылки называются жесткими. Бывают еще ссылки символическими, когда ссылка указывает не на адрес с значением переменной, а на имя переменной в таблице имен. Если в качестве переменной фигурирует массив, то с его именем связывается адрес расположения указателя на первый элемент массива. Поэтому значения ссылок

```
@mass=("xx","yy","zz");

$p=\@mass;

$pp=\$_[0];
```

имеют разные значения.

Оператор косой черты создает ссылку на объект:

```
$scalref=\$x;

$cjnstref=\123;

$arrayref=\@ARGV;

$heshref=\%ENV;

$coderef=\&prog;
```

Ссылки можно создавать на неименнованные переменные. Например как выше

```
$ref=\123; --ссылка на значение
```

А вот ссылка на анонимный массив:

```
$p=[1,2,[3,4,5]];
```

Заметим, что это ссылка на массив, который состоит из двух чисел и ссылки на другой массив. Ссылки на безымянный хеш создаются точно так же :

```
$hashref={'1'=> "one", "2"=> "two","3"=>"three"};
```

Или ссылка на анонимную подпрограмму

```
$coderef= sub { print $_[0];};
```

Теперь

```
&$coderef(12,23); выводит 12.
```

Для обращения к значениям переменных, на которые указывает ссылка используется операция разыменования. Чтобы ее применить, достаточно перед ссылкой указать символ определяющий объект, на которую ссылается ссылка:

```
$a=12.3;

$pa=\$a;

print $$pa; --печатает 12.3

@mass=("Ivanov", "Petrov", "Sidorov");

$pmass=\@mass;

print "@$pmass";
```

Для ссылки на подпрограмму см. выше. Можно создавать ссылки на ссылки. Тогда, для разыменования необходимо применять соответствующий символ нужное число раз:

```
$x="Mathematics"; $ppp=\\\$x; print $ppp; print $$ppp;

print $$$ppp; print $$$$ppp;
```

Для массивов, хешей и подпрограм предназначен третий способ обращения к переменной по ссылке – сиспользованием оператора стрелка ($->$). Приведем примеры. Пусть задан массив

```
@m=("ftp","http","smtp","pop3");

$x=\@m;
```

Тогда следующие обращения к массиву эквивалентны

```
print $$x[1];

print $x->[1];
```

Аналогично для хешей

```
%h=("ftp"=>21,"http"=>80,"smtp"=>25,"pop3"=>110);

$x=\%m;
```

```
print $$x["http"];

print $x->["http"];
```

И для подпрограмм:

```
 sub prog ($$){
  return $_[0]<$_[1]?$_[0]:$_[1];
}

$refcode=\&prog; print &$refcode(1,3);
```

Или

```
 sub prog ($$){
  return $_[0]<$_[1]?$_[0]:$_[1];
}

$refcode=\&prog; print

$refcode->(1,3);
```

Рассмотрим работу с символическимим ссылками. Символическая ссылка появляется , если попытаться раыменовать переменную не явлющуюся жесткой ссылкой:

```
$name="var";
$$name="value";
print $var."\n";
$$name[0]="new value\n";
print "@var";
$$name{"one"}=123;
print $var{"one"};
```

Все это выводит на экран

```
value

new value

123
```

Организация структур данных и доступа к ним

Прежде всего рассмотрим создаие и использование двумерных массивов. Инициализация двумерного массива возможна, например в такой форме

```
@m=(["q","w","e","r"],[1,2,3,4],["a","s","d","f"]);

print $m[0][0];
```

Или

```
$m=[["q","w","e","r"],[1,2,3,4],["a","s","d","f"]];

print $m->[0][0];
```

Обращения к элементам массива в первом и втором примерах эквивалентны следующим

```
$m[0]->[0];
```

и

```
$m->[0]->[0];
```

соответственно.

Теперь будем строить массив динамически. Применим такую констврукцию. Предположим мы читаем строки из файла и строим массив из слов разделяемых пробелами.

```
open (FILE,"<text.txt");

while(<FILE>)

{ @tmp=split;

push @mm, [@tmp];
 }
```

Здесь подразумевается, по умолчанию запись в переменную $_ из файла и применение функции shift то же к той же строке. Заполнив массив, прочитаем первую строку файла

```
$p=$mm[0];
 print "@$p";
```

Другими словами мы формируем двумерный массив, представляющий собой одномерный массив ссылок на массивы слов.

Если мы формируем ссылку на массив массивов, то поступаем так

```
open (FILE,"<text.txt");

while(<FILE>)

{ @tmp=split;

push @$refmm, [@tmp];

$p=$$refmm[0];

print "@$p";
 }
```

Срезы массивов

Срезом называется часть массива, представляющая собой самостоятельный массив. Например, если задан массив

```
@m=(1,2,4,8,16,32,64,128,256,512,1024);
```

то можно построить срез

```
@part=@m[2..7];

print "@part"; -- печатает 4,8,16,32,64,128
```

Или так

```
@part=@m[2,5,6];

print "@part"; --печатает 4,32,64
```

Теперь рассмотрим срезы двумерных массивов.

```
@m=([1,2,4],[8,16,32],[64,128,256],[512,1024]);

@part=(); for($y=0;$y<2;$y++)

{ push @part, $m[2][$y]; }

print "\n@part";
```

Рассмотрим как получить двумерный срез – например для $ x в диапазоне 1..3 и $ y в диапазоне 0..2,

```
@new=();

for($startx=$x=1;$x<=3;$x++)

{

for($starty=$y=0;$y<=2;$y++) {
$new[$x-$startx][$y-$starty]=$m[$x][$y]; } } $p=$new[1];

print "\n@$p";
```

Хеши массивов

Инициализация хеша массивов проводится приблзительно так же, как и массив массивов

```
%h=(services=>["http","ftp","smtp","telnet","pop3"],
ports=>[80,21,25,23,110]);
```

Для добавления в хеш нового массива достаточно написать

```
$h{hosts}=["127.0.0.1","10.10.4.116","10.10.4.114","10.10.4.46"];

print "\n$h{hosts}->[2]"; -- печатает 10.10.4.114
```

Для распечатки всего хеша применяем следующий цикл.

```
for $all (keys %h)
{ $p=$h{$all};

print "\n$all :@$p";

}
```

Массив хешей

Формируем массив хешей

```
@mh=( {"1"=>"Programming","2"=>"PHP","3"=>"PHP","4"=>"KompGraph"},
{"1"=>"MMKG","2"=>"SRV","3"=>"Perl"}, {"1"=>"SPPO","2"=>"Perl"} );

print $mh[1]{"2"};
```

Добавляем новый хеш

```
push @mh, {"1"=>"SPPO","2"=>"SPPO"};

print "\n$mh[3]{'1'}";
```

Рассмотрим заполнение массива хешей из файла.

```
open(FILE,"<text.txt");

while(<FILE>) {

$rec={};

for $field (split)

{ ($key,$value)=split "=", $field;

$rec->{$key}=$value;

}

push @mh, $rec;
```

```
}

print $mh[1]{second};
```

Здесь предполагается, что в файле text.txt данные записаны в виде

```
first=Programming second=PHP third=PHP

first=MMKG second=SRV third=Perl fourth=OSO

first=SPPO second=SPPO
```

Хеш хешей

Сначала строим хеш анаонимных хешей

```
%tetraedr=(
face1=>{"0"=>"P0", "1"=>"P1","2"=>"P3"},

face2=>{"0"=>"P1", "1"=>"P2","2"=>"P3"},

face3=>{"0"=>"P0", "1"=>"P3","2"=>"P2"},

face4=>{"0"=>"P0", "1"=>"P2","2"=>"P1"}, );

#теперь выводим значения хеша face3:

$p=$tetraedr{face3};

for $keys (keys %$p){
print "\n". $$p{"$keys"};

}
```

Сгенерируем теперь хеш из файла. Пусть в файле text.txt имеются записи вида

```
P0:x=10.2 y=2.3 z=4.4
```

Тогда формируем хеш, например так

```
open(FILE,"<text.txt");

while($line=<FILE>)

{

@point=split(":",$line); @coor=split(" ",$point[1]);
@x=split("=",$coor[0]); @y=split("=",$coor[1]);
@z=split("=",$coor[2]); $hh{$point[0]}{$x[0]}=$x[1];
$hh{$point[0]}{$y[0]}=$y[1]; $hh{$point[0]}{$z[0]}=$z[1]; }

$p=$hh{"P1"};
for $key (keys %$p)
{

print "\n".$key." = ".$$p{$key};

}
```

Или тело внутреннего цикла можно сделать короче

```
@point=split(":",$line);

@coor=split(" ",$point[1]);

for $str (@coor) {
 @v=split("=",$str);
$hh{$point[0]}{$v[0]}=$v[1];

}
```

Теперь добавим запись в хеш:

```
$hh{"P4"}{'x'}=3.2;
```

Наконец, выведем весь хеш хешей:

```
for $k (keys %hh)
{ $p=$hh{$k}; print "\n\n$k :";
for $key (keys %$p)
{ print "\n".$key." = ".$$p{$key}; }

}
```

Сложные структуры данных

Пример из геометрии – скрипт определяет видимость граней выпуклого замкнутого многогранника. Здесь задается ссылка PolySurf на хеш состоящий из ссылок на двумерные анонимные массивы. Первый такой массив хранит координаты точек, а второй информацию о точках из первого массива, которые образуют грани поверхности тела.

```
sub scal(\@\@) {

my $a=$_[0]; my $b=$_[1];

my $c=($a->[0])*($b->[0])+($a->[1])*($b->[1])+($a->[2])*($b->[2]);

 return $c
 }

 sub vekt(\@\@)
 {
 my ($a,$b)=@_;

 my @c;
 @c=($a->[1]*$b->[2]-$b->[1]*$a->[2],
 $a->[2]*$b->[0]-$b->[2]*$a->[0],
 $a->[0]*$b->[1]-$b->[0]*$a->[1]);
 return @c;
 }
 $pi=3.14159/180;
 $phi=45*$pi;
 $psi=0*$pi;
 @e1=(cos($phi),sin($phi),0);
```

```
 @e2=(-sin($phi)*sin($psi),cos($phi)*sin($psi),cos($psi));
 @e3=vekt(@e1,@e2);



$PolySurf={
POINTS=>[[0,0,0],[1,0,0],[1,1,0],[0,1,0],[0,0,1],[1,0,1],[1,1,1],[0,1,1]],
FACES=>[[0,1,2,3],[0,4,5,1],[4,7,6,5],[2,6,7,3],[1,5,6,2],[0,3,7,4]]
};

#Calculate the normals to faces and answer the question:

#can we see the faces?

$G=$PolySurf->{FACES};

@faces=@$G;

for $face (@faces)

{ $P=$PolySurf->{POINTS};

@POINTS=@$P;

@face_points=@$face; @a=($POINTS[$face_points[1]]->[0] -
$POINTS[$face_points[0]]->[0], $POINTS[$face_points[1]]->[1] -
$POINTS[$face_points[0]]->[1], $POINTS[$face_points[1]]->[2] -
$POINTS[$face_points[0]]->[2]);

@b=($POINTS[$face_points[3]]->[0] - $POINTS[$face_points[0]]->[0],
$POINTS[$face_points[3]]->[1] - $POINTS[$face_points[0]]->[1],
$POINTS[$face_points[3]]->[2] - $POINTS[$face_points[0]]->[2]);

@norm=vekt(@a,@b);

if(($s=scal(@norm,@e3))<=0)
    {
    print "The face "."@face_points"." is not visible\n";
```

```
    }
}
```

1.1.2 Модули и пакеты

Любая часть программы на Perl компилируется в определенном пакете. По умолчанию это пакет main. Пакет это просто пространство имен. Если мы устанавливаем текущим пакет, скажем с названием Geometry, то мы должны сказать так

```
package Geometry
```

В этом случае, использование переменной, например

```
@vector=(1.0,2.3,-3.1);
```

означает, что на самом деле эта переменная получает имя @Geometry::vector. Если мы определяем функцию

```
sub scal(\@\@) {

my $a=$_[0]; my $b=$_[1];

$c=($a->[0])*(b->[0])+($a->[1])*($b->[1]);

 return $c
 }

 @a=(-2,0);
 @b=(-1,2);
 print scal(@a,@b);
```

то на самом деле эта функция получает название & Geometry::scal. Щбратите внимание на следующий пример. В этом примере функция определена в одном пакете, в том же пакете определены пара массивов, потом происходит переключение пакета и в новом контексте мы определяем переменные с теми названиями, но при передаче в функцию важно какие именно массивы передаются – из какого пакета. Так, что первый вызов напечатает 2, а второй число 3.

```
package Geometry; sub scal(\@\@) {

my $a=$_[0]; my $b=$_[1];

$c=($a->[0])*(b->[0])+($a->[1])*($b->[1]);

 return $c
 }

 @a=(-2,0);
 @b=(-1,2);
package Music;

@a=(-3,0);
 @b=(-1,2);

 print Geometry::scal(@Geometry::a,@Geometry::b);
 print Geometry::scal(@a, @b);
```

Таким образом пакет определяет какие переменные и подпрограммы доступным тому или иному участку кода. Пакет которому принадлежит текущий код называется текущим пакетом. Первоначально текущий пакет по умолчанию называется main. В любой момент пакет можно сменить используя оператор package (см. пример выше).

Любые перемнные, необъявленные с ключевым словом my, ассоциируются с каким либо пакетом. Область видимости пакета простирается от самого объявления и до конца охватывающего блока или до объявления другого пакета.

С каждым пакетом связана таблица имен, в которую помещаются переменные объявленные внутри пакета без использования ключевого слова my. Эта таблица представляет собой обычный хеш, который можно просмотреть так

```
  foreach $symname (sort keys %Geometry::)
 {
  *sym=$Geometry::{$symname};

 print "\$$symname is defined \n" if defined $sym;
  print "\@$symname is defined \n"  if defined @sym;
   print "\%$symname is defined \n" if defined %sym;
   print "\&$symname is defined\n" if defined &sym;
```

```
}
```

Здесь *sym обозначает специальный тип переменных typeglob, и представляют собой записи о хранении существующих имен переменной в качестве скаляра, массива , хеша или подпрограммы. Такая перемнная может разыменновываться любым из способов: $ sym, @sym, % sym, & sym.

1.1.3 Модули

Имеется два вида модулей – традиционный и объектно-ориентированный. Мы рассмотрим сначала традиционный вид модулей. Такие модули определяют какие подпрограммы и переменные могут импортироваться в другие программы. Модули включаются в программу с помощью команды

```
use MODULE LIST;
```

или просто

```
use MODULE;
```

Эта команда осуществляет предварительную загрузку модуля и импорт запрошенных имен, которые становятся доступными на момент компиляции. Если не задать список нужных имен LIST, используются имена, которые содержатся во внутреннем массиве модуля @EXPORT. Для такого действия необходимо в модуле использовать модуль Exporter.

```
use SomModule;

prog();   -- при условии, что в файле модуля SomeModule.pm имеется
@EXPORT=qw(prog);
```

Все модули в Perl обычно имеют раширение .pm. Удвоенное дветочие транслируется в разделитель каталогов

```
Geometry::Vector::Plane соответствует /Geometry/Vector/Plane.pm
```

Perl ищет модули в каждм из каталогов, перечисленных в @INC. Смотрим в нашем случае:

```
print "@INC"; c:\perl\site\lib, c:\perl\lib
```

Так, что модуль Geometry::Vector::Plane соответсвует файлу

$$c:\backslash perl\backslash site\backslash lib\backslash Geometry\backslash Vector\backslash Plane.pm.$$

1.1.4 Создание модулей

Допустим мы разрабатываем модуль для работы с комплексными числами. Назовем файл модуля complex.h и разместим его в каталог C:/perl/lib/Complex. Поэтому полное название модуля Complex::complex. Рассмотрим его содержимое

```
package Complex::complex;

use Exporter;

our @ISA= "Exporter";

our  @EXPORT=qw(square);

sub square(@) { my @z; $z[0]=$_[0]*$_[0]-$_[1]*$_[1];
$z[1]=2*$_[0]*$_[1];

return @z;

}
```

Здесь мы задаем пространство имен с помощью директивы package , загружаем модуль Exporter, определяем массив экспортируемых по умолчанию имен. Далее определена подпрограмма, которая может быть вызвана в другой программе. Смысл ключевого слова our состоит в том, что создаваемая переменная доступна и в программе:

```
print @Complex::complex::EXPORT; -- печатает square
```

Помимо массива EXPORT могут быть определены массивы EXPORT_OK для экспорта имен по требованию в инструкции use MODULE LIST, и EXPORT_TAGS – для экспорта групп имен:

```
use Complex::complex;

@z=(1,1); print Complex::complex::square(@z);

print @Complex::complex::EXPORT_OK;

$Complex::complex::scal="free";
```

```
package Complex::complex;

print $scal;

print $orig;
```

1.1.5 Оъектно-ориентированные модули

Небольшое введение в теринологию объектов и классов. Класс – это обединение сложно составленных переменных с общими свойствами, определяемые как переменные-члены класса. Представитель класса называется объектом. Поведение объекта задается набором методов – функций членов класса. Основные свойства классов – инкапсуляция, т.е. объединение данных и кода в единое целое. Поиморфизм – использование одного и того же мтода в зависимости от ситуации. Типичным проявлением полиморфизма является перегрузка функций и операторов. Наследуемость – способность переносить свойства одного класса другому с уточнением свойств и поведения второго.

Определение класса находится в том или ином модуле. В Perl нет специальных средств для ООП и реализовано оно существующими средствами:

- *Объект* – просто ссылка. Поскольку ссылки позволяю создавать весьма сложные структуры данных, то они могут быть использованы в ООП.
- *Класс* – является просто пакетом. Пакет служит классом благодаря возможности хранения в нем подпрограмм и переменных. Обычно один или несколько классов реализуют в модуле.
- *Метод* – является просто подпрограммо, реализованной в соответствующем пакете.

В Perl существуют две формы вызова методов. В любом случае в подпрограмму реализующую метод передается дополнительный параметр, назыаемый инвокатором. Если для вызова используется класс – передается имя класса, если объект – передается ссылка. Рассмотрим первую форму вызова метода

```
INVOCANT->METHOD(LIST)

INVOCANT->METHOD
```

Например, если имеется класс MyClass и его методы new и show, то что бы создать новый объект и вызвать для него метод show необходимо поступить так

```
$obj = MyClass->new();

$obj->show();
```

Ассоциативность слева оператора стрелки позволяет это осуществить и так

```
$obj = MyClass->new()->show();
```

Далее, имя метода можно передать в переменную:

```
$method="new";

$obj=MyClass->$method();
```

Вторая форма вызова методов называется списочной и имеет такой синтаксис

```
METHOD INVACANT (LIST)

METHOD INVACANT LIST

METHOD INVACANT
```

Тогда предыдущий пример можно переписать и так

```
$obj = new MyClass;

show $obj;
```

Одна из проблем, которая может возникнуть при использовании второго способа вызова методов заключается в возможности существования в текущем пакете подпрограммы с тем же именем, что и имя класса или метода. Так вызов

```
$obj = new MyClass;
```

может привести к тому, что будет вызвана подпрограмма new, если таковая имеется в пакете. Для разрешения этой проблемы применяется следующий вариант

```
$obj = new MyClass::;
```

1.1.6 Создание объектов

Все объекты являются ссылками. Для того, что бы ссылка стала объектом необходимо сообщить Perl об этом, т.е. сообщить интерпретатору какому пакету объект принадлежит. Это действие осуществляется средством использования функции bless. Эта функция может принимать один или два аргумента. Первый является ссылкой, а второй именем пакета, в котором находится объект ссылки. По умолчанию считается текущим пакетом. Пример

```
$obj={}; -- ссылка на хеш

bless($obj); -- сделать хеш объектом в текущем пакете

bless($obj, "Paket"); -- сделать хеш объектом в пакете Paket
```

Обычно bless применяется в конструкторах. Например

```
package Paket::MyPaket;

use Exporter;

our @ISA ="Exporter"; -- массив @ISA используется в механизме
наследования

 our @EXPORT =qw(creat);


sub creat {

my $self =[];

bless $self , "Paket::MyPaket";

return $self;

}
```

Имея такое определение, можно создать объект так

```
use Paket::MyPaket;

$obj=Paket::MyPaket->creat;

$obj[0]=1;

$obj[1]="One";

print "@obj\n";
```

Обычно в модуле указываются способ экспорта и список экспортируемых имен:

```
our  @EXPORT=qw(square);-- экспорт по умолчанию

our @EXPORT_OK=qw(scal); -- экспорт по требованию т.е. в форме

use Paket "scal";

our %EXPORT=(
"first" => ["$square"],

"second" => ["$scal"]

); --экспорт по группам в форме

use Paket ":first";
```

1.1.7 Наследуемые конструкторы

Предположим у нас имеется класс Parent, который наследуется классом Child, причем в классе Child нет собственного метода creat. Использование может быть следующим

```
Parent -> creat;    =     Parent::creat("Parent");

Child -> creat;     =     Parent::craet("Child");
```

А реализация метода выглядит следующим образом

```
sub creat{

my $class=shift; -- считываем имя класса

my $self={};

bless($self,$class);

return $self;

 }
```

Аналогично можно реализовать конструктор с инициализатором.

```
sub creat{

my $invocant =shift;


 my $class=ref($invocant)  || $invocant;

my $self={@_}; оставшиеся аргументы конструктора

bless($self,$class);

return $self;

 }

В этом случае вызов примет вид


$obj=User->creat(name=>"Sasha", passwd="parol");
```

Еще пример. В модуле:

```
sub creat {

my $invocant =shift;

my $class= ref($invocant) || $invocant;

my $self =[ @_ ];

bless $self , "Paket::MyPaket";

return $self;

}
```

В скрипте

```
use Paket::MyPaket;

$obj=Paket::MyPaket->creat(1,2,3,4);
 print "@$obj\n";
```

И с хешами

```
my $invocant =shift;

my $class= $invocant;

 my $self ={  @_ };

$obj=Paket::MyPaket->creat("1"=>2,"3"=>4);

%h=%$obj;

print "$h{'1'}\n";
```

Возможно задание парметров по умолчанию

```
 my $self ={  "1"=> "ONE",
"3"=>"TWO",  @_  };

И в скрипте

$obj1=Paket::MyPaket->creat;
%h1=%$obj1;
print $h1{"1"}."\n"; -- печатает ONE

$obj=Paket::MyPaket->creat("1"=>2,"3"=>4);
%h=%$obj;
print $h{"1"}."\n"; -- печатает 2;
```

1.1.8 Наследование классов

Язык Perl поддерживает наследование классов. Каждый элемент массива @ISA данного пакета содержит имя другого пакета, в котором производится поиск отсутствующих методов. Например в следующем примере класс Polytop становится подклассов класса Surf

```
package Polytop;

our @ISA = "Surf";
```

Допустим, что в ссылке $ obj находится объект класса Polytop и для него необходимо вызвать метод Rotate()

```
$obj->Rotate(12);
```

Поскольку этот объект принадлежит классу Polytop, то интерпретатор сначала ищет этот метод в пакете Polytop, т.е. имя Polytop::Rotate(). Если таакого метода не обнаруживается, то интерпретатор обращается к массиву @Polytop::ISA, находит там первое имя и обращается к соответствующему пакету для поиска метода Surf::Rotate(). Если и эта подпрограмма не найдена, то интерпретатор переход в пакеты укаханные в массиве @Surf::ISA. По такой схеме реализуется как одиночное так и множественное наследование. При этом поиск осуществляется в порядке слева направо и поиск ведется сначала вглубь. Классы Perl наследуют методы, а не данные. Если необходимо осуществить наследование данных, то это можно сделать через наследование методов.

Управление данными экземпляров класса можно осуществлять с помощью встроенных методов.

Если, к примеру, необходимо реализовать класс User, в области данных котрого необходимо хранить имя пользователя в хеше с ключем login, то соответствующий хеш оформляется в виде анонимного хеша, ссылка на который и будет объектом. Далее создаем пару методов:

```
package User;

use  Exporter;

our @ISA= "Exporter";

our  @EXPORT=qw(new get_name set_name);

sub new {

my $h={};

bless $h;

return $h;

}

sub get_name {

my $self =shift; // один аргумент -- инвокант

return $self->{login};

 }

sub set_name {

my $self =shift; // один аргумент -- инвокант
```

```
$self->{login}=shift; // другой аргумент -- имя пользователя

 }
```

Тогда можно использовать все это дело в таком виде

```
use User;

$user=User->new();

$user->set_name("root");

$d=$user->get_name;

print $d;
```

Порождение классов с помощью пакета Class::Struct

Стандартный модуль Class::Struct позволяет облегчить разработку классов. Он создает все необходимое для начала работы с классом в целом: генерируется конструктор с именем new(), методы доступа к полям данных, указанных в этой структуре. Например

```
package Person;

use Class::Struct;

 struct Person =>{

 name => '$',

 login =>'$',

 passw=>'$'
 };
1;
```

```
use Person;

my $usr=Person->new();

$usr->name("Petrov");

$usr->login("petrov");

$usr->passw("parol");

print $usr->passw;
```

Перегрузка операторов

Перегрузка операторов определяет способ обработки результата операции, если в качестве опрндов присутствуют объекты некоторых классов, не являющиеся базовыми типами. Для перегрузки операторов применяется прагма

```
use overload;
```

Например,

```
package Paket;

use overload '+'=>\&add,
             '>'=>"more",
             'abs'=>sub{return @_};
```

Здесь в первом случае указана ссылка на процедуру, которая будет вызвана если попыться сложить два объекта класса Paket. Такие процедуры мы будем называть обработчиками. Для бинарных операторов обработчик вызывается если первый операнд или второй операнд являются объектами класса, если для первого не задан обработчик. Так, что можно написать

```
$ob+12;
```

или

```
12+$ob;
```

Для объектов разных классов вызывается обработчик левого операнда.

При выполнении перегрузки соответствующий обработчик получает три аргумента. Первые два – его операнды. Для унарных операторов – первый операнд, второй undef – неопределен.

Третий параметр указывает на то, могут ли быть переставлены аргументы местами. Если они могут меняться местами, от третий параметр имеет значение true. Приведем пример класса с перегрузкой операторов.

```
package ClipByte;

use overload '+'=> \&clip_add,
             '-'=>\&clip_sub;

sub myprint { my $class = shift;

my $v=$$class; print $v => "\n"; }


 sub new {

my $class = shift;

my $value = shift;

return bless \$value => $class;

 }

 sub clip_add
 {

my ($x,$y)=@_;

my ($value)=ref($x)?$$x:$x;

$value += ref($y)?$$y:$y;

$value= 255 if $value > 255;
```

```
$value =0 if $value <0;

return bless \$value => ref($x);

 }

sub clip_add
 {

my ($x,$y)=@_;

my ($value)=ref($x)?$$x:$x;

$value -= ref($y)?$$y:$y;

$value= 255 if $value > 255;

$value =0 if $value <0;

return bless \$value => ref($x);

 }

package main;

$byte1=ClipByte->new(200);

$byte2=ClipByte->new(100);

$byte3=$byte1+$byte2;

$byte4=$byte1-$byte2;

$byte5=150-$byte2;

$byte3->myprint;
```

```
 $byte4->myprint;
```

Приведем список пергружаемых операторов в Perl.

```
Арифметические +, -, *, /, %, **

Логические !

Поразрядные & | << >>

Присваивания += -= /= %= **= ++ --

Сравнения < > == <= >= != lt le gt ge eq  ne cmp

Математические atan2 cos sin exp log sqrt
```

Наконец приведем пример перегрузки преобразования в строку.

```
package Mans;

use overload q("")=> \&as_string;

sub new {

my $class = shift;

return bless { @_ } => $class;

 }

 sub as_string
 {

my $self = shift; my ($key,$value,$result);
while(($key,$value)= each %$self)
{ $result.="$key => $value\n"; } return $result;
```

```
 }

package main;

$obj=Mans->new("user1"=>"login1",
"user2"=>"login2","user3"=>"login3");

print "$obj";
```

В данном примере вместо ожидаемого вывода вроде HASH(0x23654) мы увидим

```
user1 => login1

user2 => login2

user3 => login3
```

1.1.9 Обзор модулей и функций Perl для системного и сетевого программирования

Модуль IO::File

Указанный модуль предоставляет объект для работы с файлами. Сначала приведем пример.

```
use IO::File; my $file="file.txt"; my $counter =0;

my $fh=IO::File->new($file); while (my $line =$fh->getline) {
$counter++;

}

print $counter."\n";
```

Данный скрипт подсчитывет число строк в файле file.txt. Ниже перечислены другие методы класса IO::File .

```
IO::File ->new($filename,$mode);

$filename -- имя открываемого файла,

$mode -- режим открытия файла Параметр не обязателен и режим может
быть указан как и для функции open();
```

Метод возвращает дескриптор файла.

```
$fh=IO::File->new(">file.txt");
```

Имеется возможность переоткрыть файл через его дескриптор и метод open()

```
$fh->open($filename,$mode);

Аргументы те же, что и для метода new().
```

Обычно метод используется для переопределения стандартных дескрипторов:

```
STDOUT->open(">log.txt");
```

так, что теперь вызов функции print() привдет к записи в фай log.txt.

```
$fh->print(@arg) -- запись в файл массива @arg;

$fh->printf($format,@args) -- форматированный вывод в файл.

$line = $fh->getline() -- получить строку из файла

@lines = $fh->getlines() -- получить массив  строк из файла


$result=$fh->read ($buffer,$length,$offset) -- читает из файла
$length байт и начиная со смещения $offset данные  записываются в
$buffer.

$result=$fh->sysread ($buffer,$length,$offset) -- то же, чот и
выше, только эта функция не применяет стандартную буферизацию
ввода-вывода.
```

```
$result=$fh->syswrite($buffer,$length,$offset) -- записывает в
файл указанное число байт из буфера, начиная с указанного
смещения.
```

Расмотрим небольшой пример

```
use IO::File; my $file="file1.txt";

my $fh=IO::File->new($file,"w");

$data="Hello!\n\r"."Helo!!!";

$fh->syswrite($data,100);

$fh->open($file,"r");

@line= $fh->getlines;

print "@line";
```

Межпроцессное взаимодействие

Сначала рассмотрим предоставляемую в Perl систему сигналов для указания процессам о наступлении некоторых событий.

Небольшой пример.

```
my $inter=0;

$SIG{INT}=\&handler;

while($inter < 3) {

print "I am sleeping\n"; sleep(1);

}

sub handler { $inter++;
```

```
$SIG{INT}=DEFAULT; -- восстанавливает обработчик сигнала

}
```

Если вместо DEFAULT написать IGNORE – сигнал полностью игнорируется.

Тот или иной сигнал посылается процессом посредством функции

```
$count=kill($signal, @processes);

Посылается сигнал $signal процессам перечисленным в массиве
@processes.
```

следующий способ взаимодействия процессов – запуск внешней команды и получения вывода этой команды в скрипте посредством обратных кавычек:

```
$res=‘dir‘;

print $res; -- для Windows

$res=‘ls -l‘;

print $res; -- для UNIX
```

А вот так лучше не делать

```
$res=‘perl io.pl‘;

print $res;
```

Следующая функция дублирует процесс.

```
$pid=fork(); if($pid==0) {

print "I’m child\n"; }
 else {
 print "My child with $pid\n"

 }
```

Эта функция является аналогом функции дублирования процесса в UNIX. Вместе с функциями и операторами создания каналов образуют весьма эффективны механизм общения родительского и дочернего процессов.

В первую очередь какнал может быть открыт посредством функции open(). Рассмотрим пример.

```
open(CHNL,"dir |"); while($str=<CHNL>) { push @str,$str; }

print @str;
```

Здесь канал открывается для чтения результата работы команды dir.

А теперь рассмотрим сспециальную функцию для создания именованных каналов.

```
$result = pipe(READHAND,WRITEHAND);
```

Функция создает канал, связывающий два открытых файла – первый для чтения, второй для записи. При успешной операции возвращается истина.

Рассмотрим пример.

```
$arg=shift || 10;

pipe(READ,WRITE);

if(fork==0) {

close READ;

factorial($arg);

exit 0; }

if(fork==0)

{ close READ;

my $res=fibonachi($arg); exit 0;

}
```

```
close WRITE;


while($lin=<READ>) {print $lin; }

#print while <READ>;

sub factorial { my $t=shift;

for(my $r=1, my $i=1; $i<=$t;$i++)

 {

 print WRITE "factorial($i)=", $r*=$i,"\n";

 }

}

 sub fibonachi {

 my $t=shift;

 my ($a,$b)=(1,0);

 for(my $i=1;$i<=$t; $i++)

{ my $c=$a+$b;

print WRITE "fibonachi($i)= $c\n";

($a,$b)=($b,$c); } }
```

Модули для работы с серверами FTP и HTTP

Язык Perl предоставляет широкие возможности сетевого программирования как на уровне сокетов, так и на уровне приложений для работы с екоторыми стандартными серверами. Мы рассмотрим модули для работы с ftp сервером и web сервером.

```
use Net::FTP;

$host="localhost";

$dir="./";

my $ftp=Net::FTP->new($host);

$ftp->login("klchnv","tanyavarya");

@files=$ftp->dir($dir);

for $file(@files) { print $file."\n"; }

$ftp->quit;


print "Select file name for upload\n";

$filename=<>; chomp($filename);

$ftp->get($filename); open(FL,$filename);

while($line=<FL>) { print $line."\n"; } $ftp->quit;
```

Вкратце опишем используемые методы класса Net::FTP:

```
$ftp=Net::FTP->new($host,[%param]); создает новое ftp соединение с
удаленным сервером на $host. Хеш параметров задает параметры
соединения.
```

```
Firewall - имя proxy-сервера FTP, которое должно быть указано,
если компьютер подключен к сети, защищенной боандьауэром.

Port - номер порта, по умолчанию 21

Timeout -- Значениетайм-аута для различных операций в секундах
(120сул -- по умолчанию)

Passive -- включает пассивный режим -- необходим для некоторых
брандмауэров.

Пример

$ftp=Net::FTP->new($host,Firewall => "10.10.4.1", Timeout => 100);
```

Для авторизации на сервере используется функция login:

```
$success=$ftp->login($login,$passw)

$login, $passw -- параметры авторизации.

$ftp->pwd -- возвращает текущий удаленный каталог.

$ftp->mkdir -- создает удаленный каталог

$ftp->cwd -- меняет текущий удаленный рабочий каталог

$ftp->rmdir -- удалить указанный каталог

$ftp->dir -- получить массив из списка файлов в текущем каталоге

$ftp->get -- загрузить файл из текущего удаленного каталога в
текущий локальный каталог

$ftp->put -- загрузить на сервер заданный файл

$ftp->size -- определить размер файла
```

Для работы с протоколом HTTP и другими протоколами (FTP, SMTP, HTTPS) в 1995 появилась впервые библиотекм LWP, написанная Мартином Костером и Гисле Аасом.

В следующем примере демонстрируется использование этой библиотеки для получения удаленного документа по заданному URL в командной строке. Создаваемый объект User-Agent способен осуществлять запросы на удаленный web сервер. Сам запрос представляет собой объект класса HTTP::Request, а ответ сервера мы получаем в виде объекта класса HTTP::Response. У последнего есть метод content возвращающий строку ответа сервера без заголовка.

```
use LWP;

my $url=shift || "http://127.0.0.1";

my $agent=LWP::UserAgent->new;

my $request=HTTP::Request->new(GET => $url);

my $rsp=$agent->request($request);

print $rsp->content;
```

Можно упростить этот код, используя модуль LWP::Simple

```
use LWP::Simple;

my $url=shift || "http://localhost";

getprint($url);
```

Отметим, что модуль LWP имеет не только средства для посылки запросов и для разбора HTML и XML документов и преобразования их в текстовый читабельный формат. Отметим следующие модули : HTML::Parser, HTML::Formatter, HTML::FormatText и др.

Глава 2

Использование регулярных выражений

2.1 Регулярные выражения

2.1.1 Введение в регулярные выражения

Регулярные выражения представляют собой специальный язык для задания шаблонов символьных строк с целью осуществления операций поиска
замены строк и их подстрок. Механизм регулярных выражений является одним из эффективных методов обработки текстовой информации. В качестве простейшего примера подобных шаблонов можно привести шаблоны для задания имен файлов и масок поиска. Например строка *.? задает имя файла с расширением из одного символа.

Регулярные выражения состоят из двух типов символов. Специальные символы вроде *,? в файловых шаблонах носят название метасимволы. Все остальные символы называются литералами.

Существует несколько утилит, в которые встроена работа с регулярными выражениями. В первую очередь стоит отметить программу grep
egrep. Данная программа имеется как для UNIX подобных ОС так и для Windos. Приведем синтаксис этой команды.

```
egrep -hinvw шаблон имя_файла
```

Утилита позволяет находить соответствия шаблону в строках из списка файлов. Если файл не определен, то поиск строк производится в стандартном потоке ввода stdin. Все строки соответствующие шаблону выводятся в stdout. Если определена опция -h, то строки предваряются соответствующим именем файла. Опция -n заставляет нумеровать строки, а -i – заставляет игнорировать регистр в шаблоне. Опция -v выводит строки, которые не соответствуют заданному шаблону, а -w – ограничивает соответствие только словами.

Приведем небольшой пример.

```
#egrep '^*.-$' file.txt
```

Данное регулярное выражение находит в заданном файле все строки с переносом в конце строки или слова разделенные дефисом и одно из которых перенесено на слдующую строку. В этом примере введены метасимволы $ и ^ для обозначения соответственно конца и начала строки. А так же метасимвол точка (.) для обозначения произвольного символа и * для обозначения повторения предшествующего символа произвольное число раз. Правильно читать такое регулярное выражение надо так: "^*.-$"совпадает, если мы находимся в начале строки за которым следует произвольная пследовательность символов, оканчивающаяся дефисом, за которым следует сразу же конец строки.

Символьным классом называется группа символов, определяющих один из альтернативных символов в шаблоне. Символьные классы задаются в квадратных скобках. Например символьный класс [A-Z] представляет один из символов из указанного диапазона, т.е. одну из заглавных букв латинского алфавита. Или, например символьный класс [a-zA-Z_!?] представляет либо букву латинского алфавита, или символ подчеркивания, или восклицательный или вопросительный знак. Следует обратить внимание на то, что дефис в символьном классе играет особую роль – он является разделителем при указании диапазона символов. Однако, если он стоит на первом месте, то он имеет смысл обычного дефиса.

Инвертирование символьного класса – это определеие исмвольного класса не входящих в список символов. Для этого сразу после открывающей квадратной скобки следует поставить знак ^. Например, если мы хотим найти в тексте строки, в которых присутствуют символы не входящие в алфавитно-цифровой набор мы сформируем такое регулярное выражение $[\hat{} - A - Za - z0 - 9]$.

Рассмотрим поисл строк, в которых после "ж"или "ш"идет гласная не и – "[ж|ш][^ и]". Но это выражение подойдет и для слова "шутка". Поэтому лучше использовать регулярное выражение "[ж|ш]ы". Здесь мы воспользовались метасимволом ", который буквально означает союз "или". Таким образом регулярное выражение "[ж|ш]ы"мы должны читат так – строка содержит символ "ж"или "ш", за которым сразу же следует символ "ы". Учитывая, что буквы могут быть заглавными, окончательно получаем регулярное выражение "([Жж])|([Шш])[ыЫ]"или "[ЖжШш]ы". Круглые скобки в приведенном примере являются метасимволами, которые в данном случае огранчивают действие метасимвола |. Отметим, что метасимвол выбора или альтернативы отличается от символьного класса тем, что соответсвуют выбору одного из двух регулярных выражений, когда как символьный класс соответствует единичному символу.

Перейдем к рассмотрению метасимволов, отделяющих слова. Границы слов определяются метапоследовательностями \ < \ >. Однако в некоторых системах используются мета последовательности $\backslash b\backslash b$. Например регулярное выражение $\backslash b[0-9]$ ищет слова, начинающиеся с цифры. Что бы разъяснить ситуацию отметим, что граница слова – это позиция в строке соответствующая началу или концу последовательности алфавитно-цифровых символов. Например в выражении $123 началом слова будет символ "1". Проверьте.

Специальные метасимволы, которые определяют количество символов называют квантификаторами. Отметим некоторые из них.

- '+' – означает повторение предшествующего символа не менее 1 раза.
- '*' – означает повторение предшествующего символа любое количество раз
- '?' – означает присутствие предшествующего символа не более 1 раза
- {n,m} – означает повторение предшествующего символа от n до m раз
- {n} – означает повторение предшествующего символа ровно n раз.

Пример. Если мы хотим определить нахождение IP адреса в строке, то можно использовать такое регулярное выражение [0-9]{1,3}[0-9]{1,3}[0-9]{1,3}[0-9]{1,3} . При построении данного примера мы воспользовались метапоследовательностью \. означающее экранирование метасимвола в шаблоне. Т.е. если мы хотим использовать в шаблоне метасимвол как литерал, то этого можно достигнуть с помощью экранирования метасимвола.

Рассмотрим пример поиска тегов в html файле, которые определяют горизонтальную строку определенной ширины. Например, <hr size = 12>. Соответствующее регулярное выражение выглядит так

```
"<hr( +size *= *[0-9]+)? *>".
```

Заметим, что "*"вынесен за круглые скобки, что бы регулярное выражение было способно находить теги вида $< hr \quad >$. Если оставить в скобках, то произвольное количество символов может присутствовать перед закрывающей треугольной скобкой тега только в случае присутствия атрибута size. Конструкция "+"наоборот внесена в скобки для того, чтобы соответствовать тегу вида <hr> с атрибутами по умолчанию.

Во многих системах использующих регулярные выражения круглые скобки могут запоминать текст, расположенный внутри них. При этом можно использовать метапоследовательности $\backslash 1, \backslash 2, \backslash 3...$ можно ссылаться на этот текст. Например, для нахождения двух повторяющихся в тексте слов можно воспользоваться регулярным выражением

```
\b([A-Za-z]+) +\1\b
```

При построении регулярного выражения приходится следить за тем, что бы регулярное выражение

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

Примеры.

Имена переменных.

Имена переменных во многих языках программирования состоят из алфавитно-цифровых символов и знаков подчеркивания, но не могут начинаться с цифры. Это правило определяет регулярное выражение

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Если длина идентификатора ограничена 32 символами, тогда звездочку можно заменить на $\{0, 31\}$.

Адрес URL HTTP.

Адреса URL могут иметь весьма сложную структуру, но мы ограничимся рассмотрением адресов, задаваемых в виде

```
http://host/path.html
```

Для адреса хоста ограничимся рассмотрением выражений вида www.duel.ru. Которые находятся с помощью регулярного выражения

```
[a-z0-9_.]+
```

Для указания пути воспользуемся перечислением символов, которые могут участвовать в его формировании:

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

Но можно и попроще

```
[^ ]*\.html
```

Время в формате 12:20 pm

Первое, что приходит в голову регулярное выражение

```
[0-9]?[0-9].[0-9][0-9] (am|pm)
```

Однако такое время 99:99 pm тоже будет подходить. Учитывая, что час , если состоит из двух цифр, то первая может быть только 1, получим следующее регулярное выражение

```
(1[012]|[1-9]).
```

Поступая и для минут аналогично, окончательно получим

```
(1[012]|[1-9]):[0-5][0-9]? (am|pm)
```

Наконец, приведем список метапоследовательностей

```
\0 -- совпадение с символом "нулевой байт"

\A -- совпадение в начале строки

\b -- граница слова

\B -- не граница слова

\cX -- совпадение с управляющим символом Ctr-X (\cZ,...)

\d -- совпадает с любой цифрой

\D -- совпадает с любым символом -- не цифрой.

\l -- нижний регистр для следующего символа

 \n -- совпадение с символом новой строки

\t -- совпадение с символом табуяции

\r -- возврат каретки

\s -- совпадение с любым пробельным символом

\S -- совпадение с не пробельным символом

\w -- символ слова
```

```
\W -- не символ слова

\u -- верхний регистр для следующего символа

\z -- конец строки

\Z -- конец строки или перевод строки.
```

Пример. Соответствие строки "We proof Lusin theorem"регулярному выражению "$\backslash ul$".

Упражнение. Построить регулярное выражение для поиска времени в 24-часовом формате: "21:12". Построить регулярное выражение для поиска даты в формате "12/03/2007".

Для проверки соответствия средствами языка Perl используется оператор

```
$str=~m/reg_exp/
```

После закрывающей косой черты может стоять так называемый модификатор шаблона.

Приведем список модификаторов.

```
/i -- игнорировать регистр при поиске

/x -- игнорировать пробельные символы

/g -- глобальный поиск всех соответствий

/o -- подавление повторой компиляции

/s -- рзрешать . устанавливать соответствие переводу строки

/m -- разрешить ^ $  устанавливать соответствие рядом со
встроенным \n
```

Что касается модификатора s. Вот иллюстрирующий пример.

```
$str="The aim is future.\n And who are  you?";

if($str=~/. A/s) { print " OK!\n"; } else { print "NO\n"; }
```

И аналогично для m

```
$str="The our is future.\n And who are  you?"; if($str=~/^ /) {
print " OK!\n"; } else { print "NO\n"; }
```

Для формирования регулярного вырааажения можно использовать переменные Perl. Как например в следующем примере.

```
$sign='[+-]?';

$digits='\d+';

$decimal ='\.?';

$more_digits='\d*';

$number="$sign$digits$decimal$more_digits";

$num=10.23;

if($num=~m/^$number$/o) { print "OK!\n";

} else { print "No match\n"; }
```

В списочном контексте оператор m// возвращает список подстрок, соответствующим найденным с помощью регулярного выражения. Приведем примеры.

```
$_="x=1&y=2&z=3"; if(@m=$_ =~ /([a-z]+)=([0-9]+)/g)

 { print "@m\n";
 }

Распечатывает массив : x 1 y 2 z 3.
```

Можно использовать хеш

```
$_="x=1&y=2&z=3";
if(%m=$_ =~ /([a-z]+)=([0-9]+)/g)
{
for $key (keys %m){
print "$key = $m{$key}\n"; } }
```

Модификатор g позволяет использовать функцию pos() для определения смещения сразуже за найденным соответствием.

```
$str="Hello world!";

while($str=~/l/g)
 {
 $p=pos($str);
 print "We find l in $p\n";
 }
```

И еще пример выводящий все найденные соответствия

```
$str="x=1 yX=2 x=3";

if(@m=$str=~/x=/ig)

{ $n=@m; print $n."\n"; print "@m"; }
```

2.1.2 Один пример

Рассмотрим задачу. Требуется написать скрипт, переводящий введенный размер, котрый указан в сантиметрах или дюймах, в другую соответствующую единицу измерения. Для такой реализации воспользуемся следующей возможностью использования языка Perl. Мы знаем, что выражения в круглых скобках запоминаются в переменные \1, \2, и т. д. Однако эти переменные доступны исключительно только внутр регулярного выражения. В Perl введены дополнительные переменные $ 1, $ 2,.. , которые соответствуют вышеуказанным и доступны вне регулярного выражения.

Вот пример решения указанной выше задачи.

```
sub obr { $_[0]=$_[0]*2.5; }

print "Enter the length (i.e. 12.3 cm or 20.51 in)\n";

$str=<STDIN>;

chop($str);
```

```
if($str=~/^([-+]?[0-9]+(\.[0-9]*)?) +(cm|in)$/) {

$ch=$1; $ed=$3; print $ch."  ".$ed."\n";

if($ed eq "in")

{ $str=obr($str);

$str.=" cm"; }

else { if($ed eq "cm")

{ $str=$str*2/5; $str.=" in"; } } print $str."\n"; }

else { print $str." -- is not number\n";

}
```

Отметим, что в используемом регулярном выражении есть пара круглых скобок, расположенных внутри другой пары, причем эта пара скобок предназначена для группировки указания квантификатору. Однако эта часть строки будет скопирована в переменную $ 2. Поэтому мы и использовали для определения единицы измерения $ 3. С целью избежать такого побочного эффекта используем группировку без сохранения (?:). То есть регулярное выражение приобретает следующий вид.

```
/^([-+]?[0-9]+(\.[0-9]*)?) +(cm|in)$/
```

Кроме переменных $ 1, $ 2, .. в Perl введены переменные

```
$‘ -- для текста слева от найденного соответствия,

$& -- для самого соответствия,

$’ -- для текста справа от найденного соответствия.
```

Небольшой пример.

```
$str="The values x=1 yX=2 x=3";

$str=~/x/; print $‘."\n"; print $&."\n"; print $’."\n";
```

2.1.3 Модификация строки

`s/шаблон/подстановка/модифик`

Здесь модификаторы имеют такой же смысл, что и для оператора поиска. Оператор подстановки ищет подстроки, соответствующие шаблону и заменяют их строкой подстановки.

Как в скалярном так и в списочном контексте оператор замены возвращает число найденных соответствий. В строке замены можно использовать переменные

`$1, $2, ...`

`$‘ , $& , $’`

Вот пример.

```
%h=('1'=>"one","2"=>"two","3"=>"three");
$i=0;

$str="The numbers  1 2 3  are replaced by strings";

print $str."\n";

$str=~s/([0-9]{1})/$h{$1}/g;

print $str;
```

Предположим необходимо сократить десятичные дроби вида 12.3087622100 до 12.308, но 1.230211155643 необходимо вывести в виде 1.23. Вот решение

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

Иногда необходимо, что бы новая, модифицированная строка не портила старой. Вместо очевидного решения

```
$tm=$str; $tmp=~s/.../.../;
```

можно объединить

```
($tmp=$str)=~s/.../.../;
```

Оператор замены можно применять и к массиву строк. Например так как показано в примере, в котром все числовые данные окружаются HTML тегом выделения текста полужирным шрифтом.

```
$str1="The first number is 12\n";

$str2="The second number is 23\n";

$str3="The third number is -12\n";

@m=($str1,$str2,$str3);

foreach(@m) { s/([-+]?[0-9]+)/<b>$1<\/b>/; } print "@m";
```

Заметим, что здесь по умолчанию используется переменная $_, которая является переменной цикла и строкой , к которой применяется оператор замены. С целью сохранения первоначального массива, можно использовать одновременное присваивание и оператор замены как и в скалярном случае.

```
for(@new=@m) { s/([-+]?[0-9]+)/<b>$1<\/b>/;
 }
 $"="\n";
 print "@m";
print "\n"; print "@new";
```

Модификатор /e используется в случаях, когда необходимо считать строку замену не строкой а кодом Perl. Например, регулярное выражение

```
s/([0-9]+)/sprintf("%x",$1)/ge
```

преобразует десятичные целые в 16-чные.

В последующих примерах используется расширенный список метапоследовательностей регулярных выражений в Perl.

```
(?: ) -- группировка без захвата,

(?= ) -- истина, если выполняется утверждение после данной точки

(?! ) -- истина, если  не выполняется утверждение после данной
```

точки

(?<=) -- истина, если выполняется утверждение перед данной точкой

(?<!) -- истина, если не выполняется утверждение перед данной точкой

(?{ }) -- выполнить встроенный код Perl.

Метапоследовательности (?=) и (?<=) имеют специальное название. Первая называется позитивной опережающей проверкой, а вторая позитивной ретроспективной проверкой. Сначала совсем простые примеры.

(?=\d) совпадает с той позицией текста, после которой идет цифра,

(?<=\d) совпадает с той позицией текста, перед которой стоит цифра.

Отметим одно важное свойство обеих проверок – они не поглащают текст. По существу они находят не совпадения, а местоположение совпадения. Вот пример.

```
$str="abc1def";

if($str=~/(?=\d)/) {

print "Find - ".$1; -- печатает Find -

}


$str="abc1def";

if($str=~/(?=\d)(\d)/) {

print "Find - ".$1; -- печатает Find - 1

}
```

Рассмотрим решение задачи расстановки десятичной точки в многозначном числе. То есть числа вида 45123 необходимо заменить на 45.123 . Вот решение

```
$num=1234567;

$num=~s/(?<=\d)(?=(\d\d\d)+$)/\./g;

print $num;
```

Отметим еще одно решение этой задачи посредством регулярного выражения с использованием негативной опережающей проверки.

```
$num=~s/(\d)(\d\d\d)(?!\d)/$1.$2/g;
```

К сожалению этот вариант правильно работать не будет. Псокольку имеется только одно соответсвтвие шаблону – "четыре цифры и далее не цифра". Положение исправляет применение цикла

```
while($num=~s/(\d)(\d\d\d)(?!\d)/$1.$2/g) { } print $num;
```

Предположим в файле temp.txt записаны строки вида

```
From: nnn@mail.ru

To: mmm@yandex.ru

Date: Mon, 17 Jul 2007 09:00:00

Subject: Hello!
```

Требуется составить хеш, ключами которого будут значения поле From, No,..., а значения – текст справа от двоеточия.

```
open(OUT,"temp.txt"); while(<OUT>) { /(^.*?):(.*)/; $h{$1}=$2; }

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

2.1.4 Еще пару слов об квантификаторах

Напомним, квантификаторы отвечают за число повторений определенного символа. Мы уже знаем, что , например, $\{2,6\}$ – представляет собой квантификатор, означающий повторение предшествующего символа от 2 до 6 раз. К примеру, регулярному выражению $/lo\{1,2\}p/$ удовлетворяют строки

```
"The loop ...";

"The slope ..."
```

Одноко у квантификатора есть еще два свойства – свойство минимальности и свойство максимальности. Минимальный кватификатор ищет минимальное число символов для совпадения в пределах ему отпущенных, а максимальный пытается найти максимальное число соответвтующих символов. Приведем список квантификаторов с указанными свойствами

```
*   -- 0 или брльше совпадений (максимльный)

+ -- 1 или более символов (максимальный)

? -- 1 или 0 совпадений (максимальный)

{n} -- ровно n  совпадений

{m,} -- не меньше m совпадений (максимальный)

{m,n} -- не меньше m и не больше n  совпадений (максимальный)

*? -- 0 или более совпадений (минимальный)

+? -- 1 или более совпадений (минимальный)

?? -- 0 или 1 совпадений (минимальный)

{m,}? -- не менее m  совпадений (минимальный)

{m,n}? -- н менее m  и не более n совпадений (минимальный).
```

Примеры.

Разберите самостоятельно следующие примеры , иллюстрирующие работу максимальных минимальных квантификаторов.

```
$str="123abc1234567890";

if(@m=$str=~/([0-9]{1,2})/) {

print @m; -- печатает 12

 }
$str="123abc1234567890";

if(@m=$str=~/([0-9]{1,2}?)/) {

print @m; -- печатает 1

}


$str="123abc1234567890";

if(@m=$str=~/([0-9]{1,2}[a-z]+?)/) {

print @m; -- печатает 23a


if(@m=$str=~/([0-9]{1,2}[a-z]+)/) {

print @m; -- печатает 23abc

}
```

Следующие примеры предназначены для самостоятельного разбора в качестве подготовки к следующей лекции. Тема будущей лекции – "Механизм обработки регулярных выражений". Вообще говоря для правильного и эффективного составления регулярного выражения необходимо понимание механизма с помощью которого осуществляется поиск совпадений. Существует три разновидности таких механизов. Но по существу их два. Один управляется регулярным выражением, другой – символьной строкой.

Пример 1.

Попробуйте объяснить в чем причина разницы в количестве строк вывода в первом примере и во втором.

```
$tr="12ab34d56adch";

@m=$tr=~/([0-9]{1,2}[a-z].{3})/g;

$"="\n";

print "@m";
```

Пример 2.

```
$tr="12ab34d56adch";

@m=$tr=~/([0-9]{1,2}[a-z].{1})/g;

$"="\n";

print "@m";
```

Пример 3.

Что будет напечатано в следующих примерах?

```
$str="The dragging belly indicates your cat is to fat";

@m=$str=~/(fat|cat|belly|your)/;

print @m;
```

Пример 4.

```
$s="About 24 characters long";

@m=$s=~/(^.*[0-9][0-9])/;
```

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

Пример 5.

```
$s="Copyright 2007.";

@m=$s=~/(^.*([0-9]+))/;

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

Пример 6. Почему в следующем примере ничего не печатается

```
$s="fox";

@m=$s=~/(x*)/;

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

2.1.5 Механизм обработки регулярныых выражений

Механизм обработки регулярных выражений зависит от программы, предлагающей услуги использования регулярных выражений. Существует, в основном два различных механизма НКА и ДКА – недетермированный и детермированный конечные автоматы.

```
ДКА -- awk, egrep, MySQL

НКА -- Java, grep, .NET, Perl, PHP, Python, Ruby, sed, vi, Tcl.
```

Упрощенно говоря, первый механизм управляется текстовой строкой, второй – регулярным выражением.

Машина НКА начинает работу прямо перед первым символом и пытается найти соответствие всему шаблону с этой точки. Весь шаблон соответствует тогда и только тогда, когда машина достигнет шаблона раньше, чем она соскочит с конца строки. Если совпадение найдено, она сразу же пекращает работу. Механизм просматривает регулярное выражение

по одному компоненту, и проверяет совпадает и он с текущим текстом. В случае совпадения проверяется следующий компонент. Пример. В строке "1231214..."происходит поиск по регулярному выражению "1(21|23|25)". Машина НКА устанавливает текущую позицию – перед первым символом строки. Далее перебирает компоненты регулярного выражения. Первый компонент – символ 1. Проверка завершается удачей. Тогда машина рассматривает следующий компонент – список альтернатив. Перебирает их слева на право. Каждая альтернатива так же рассматривается по компонентно. На второй альтернативе механизм находит совпадение и прекращает работу.

Машина ДКА сканирует строку и следит за всеми потенциальными совпадениями. Каждый следующий сканируемый символ обновляет список потенциальных совпадений. Пример. В строке "1234..."происходит поиск по регулярному выражению "1(23|24|25)". Выбирая символ "1"механизм ДКА находит одно потенциальное совпадение – первый сомвол в шаблоне. Перейдя ко второму символу строки, ДКА находит три потенциальных совпадения – $12_{\blacktriangle}3, 12_{\blacktriangle}4, 12_{\blacktriangle}5$. Рассмотривая третий символ в строке машина ДКА отказывается от второго и третьего совпадений.

Приведем общие правила для механизма обработки регулярных выражений.

- **Правило 1.** Предпочтение отдается тому совпадению, которое начинается раньше. Правило ничего не говорит о длине совпадения. Это значит , что в примере

```
$s="fox";

@m=$s=~/(x*)/;

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

ничего не будет распечатано, поскольку первое совпадение – пустая строка. По той же причине в примере

```
$str="The dragging belly indicates your cat is to fat";

@m=$str=~/(fat|cat|belly|your)/;

print @m;
```

будет напечатано слово belly, как первое встречное, удовлетворяющее условию регулярного выражения. Кроме этого, если машина регулярного выражения не находит

совпадения с текущего символа выбирается следующий символ и происходит поиск совпадений с новой позиции. Если совпадение найдено, то машина сразу же прекращает работу. Когда машина обнаруживает набор альтернатив (разделенных символом |) она опробует их слева направо. Но останавливает выбор в таком порядке: если с текущей позиции нет совпадения **со всем регулярным выражением**, то только после этого переходим к следующей позиции. Это еще раз объясняет, что в предыдущем примере будет напечатано именно слово belly, а не слово fat.

- **Правило 2.** Квантификаторы *, ?, +, $\{m, n\}$ работают максимально. Механизм регулярных выражений соглашается на значения меньше максимума только в одном случае – если слишком большое число повторений не позволяет найти для какой-либо последующей части регулярного выражения. Например, регулярное выражение $/\backslash b\backslash w+s\backslash b/$предназначено для поиска слов, оканчивающихся на s. Часть вырежения $\backslash w+$ может совпасть со всем, например , словом programms, но оставшаяся часть регулярного выражения не найдет совпадения. Поэтому выражению $\backslash w+$ соответствует часть слова – programm.

Рассмотрим более подробно механизм НКА, точнее его одну из важнейших концепций – возврат.

Возврат в механизме НКА

Основной принцип работы механизма НКА заключается в следующем: он последовательно просматриват все подвыражения или компоненты, и когда приходится выбирать между двумя равноправными вариантами – выбирает один и запоминает другой, что бы вернуться к нему в случае необходимости. Такое поведение механизма НКА связано с использованием квантификаторов и конструкций выбора.

Если выбранный вариант и все выражение успешно совпадают, то процесс поиска завершается. Если какая-либо из оставшихся частей регулярного выражения приводит к неудаче, механизм регулярных выражений возвращается в точку ветвления и продолжает поиск с другим вариантом. Такое поведение напоминает алгоритм поиска выхода из лабиринта.

Рассмотрим пример поиска совпадения в строке "1231211256"по регулярному выражению "1(21|23|25)". Дойдя до выбора альтернатив машина выбирает вариант (21), а два другиз запоминает. Проверяя выбранный вариант,машина приходит к неудаче и не переходит к новой позиции в строке, а возвращается к ранее оставленным вариантам. Выделим два правила, справедливые для процесса возврата.

- В тех ситациях, когда механизм выбирает между попытками найти совпадение или отказаться от его поиска (например при использовании квантификаторов ?, *, и т.п.)

механизм всегда сначала пытается найти совпадение для максимальных квантификаторов или пропустить совпадение для минимальных квантификаторов.

- При локальной неудаче происходит возврат к последнему из сохраненных вариантов.

Рассмотрим простые примеры. Строка – "123", регулярное выражение – "12?3". После первого совадения символа 1, текущее состояни выглядит так, строка – $1_{\blacktriangle}23$ выражение $1_{\blacktriangle}2?3$. Теперь механизм решает проблему – пытаться найти 2 или нет. Первое из вышеприведенных замечания утврждат, что механизм будет пытааься найти символ 2, но запомнит такое состояние, что бы в случае неудачи к нему вернуться: в строке $1_{\blacktriangle}23$, в выражении $12?_{\blacktriangle}3$. Это значит, что механизм позднее продолжит поиск с компонента, следующего в регулярном выражении после 2? и сопоставит его с текстом, находящимся до 2. Другими словами литерал 2 пропускается. Сохранив это состояние, машина НКА переход в состояние: строка $12_{\blacktriangle}3$, выражение $12?_{\blacktriangle}3$, после успешного найденного символа 2. После чего находится символ 3 и машина останавливается.

Рассмотрим поиск того же шаблона, но в строке 134. Все повторяется так же как и в предыдущем примере, но не найдя символа 2, машина возвращается в последнее сохраненное состояние, которе в нашем случае имеет вид: строка $1_{\blacktriangle}34$, шаблон $12?_{\blacktriangle}3$. После успешно найденного символа 3, машина останавливается.

Теперь рассмотрим строку 124 для поиска с тем же шаблоном. Прежде всего заметим, что машина для поиска символа 2, как и в первом случае сохранит состояние: строка $1_{\blacktriangle}24$, шаблон $12?_{\blacktriangle}3$. Символ 2 находится в строке, но дальнейшая проверка пиводит к неудаче, поскольку в строке отсутствует символ 3 и поэтому машина возвращается к приведенному выше состоянию и продолжают поиск. Но эта попытка так же приводит к неудаче, поскольку в строке символ 2 не совпадает с символом 3 в регулярном выражении. Но на этом машина не останавливается, а продолжает применять регулярное выражение с новой позиции в строке – в позиции между первым и вторым символом.

Теперь поменяем регуляярное выражение. Строка 123, шаблон 12??3, т.е. рассмотрим регулярное выражение с минимальным квантификатором. После первого обнаружения символа 1, состояние машины таково: строка $1_{\blacktriangle}23$, шаблон $1_{\blacktriangle}2??3$. Поскольку в шаблоне встречается минимальный квантификатор, то из двух вариантов – попытаться найти символ 2 или пропустить его, выбирается второй вариант. В список состояний сохраняется состояние: строка $1_{\blacktriangle}23$, шаблон $1_{\blacktriangle}23$. После этого, машина переводится в состояние: строка $1_{\blacktriangle}23$, шаблон $1_2\blacktriangle??3$, пропуская символ 2. Но символ 2 не совпадает с символом 3, поэтому машина возвращается в последнее сохраненное состояние и удачно находит совпадение шаблона сос строкой.

С небольшими изменениями машина аналогично работает и квантификаторами *,+. Рассмотрим пример. Предположим в строке "The sum is 1234 or more"ищем соответствие

по шаблону [0-9]+. Как только машина дойдет до цифр она будет перебирать их, сохраняя промежуточные состояния

The sum is 1▲234...
The sum is 12▲34...
The sum is 123▲4...
The sum is 1234▲...

Последние три состояния отражают тот факт, что найденные совпадения являются не обязательными, но т.к. квантификатор максимальный, то машина пытается найти как можно больше соответствий. дойдя до последнего состояния , машина обнаруживает пробел, не подходящий для [0-9] и , естественно возвращается к последнему состоянию продолжая работу, но заметим конец регулярного выражения останавливается, найдя общее совпадение. Заметим, в списке состояний нет состояния The sum is ▲1234... поскольку для квантификатора + необходимо как минимум 1 совпадение, но и для шаблона $[0-9]*$ это состояние не появится. Более того, для этого шаблона машина НКА не дойдет вообще до цифр. Почему?

Теперь рассмотрим работу машины НКА в случае квантификатора *. Пусть в строке "The sum is 1234 or more"ищется подстрока с шаблоном $.*[0-9][0-9]$. Во-первых механизм поиска для компонента .* регулярного выражения дойдет до конца строки и сохранит все промежеточные состояния как необязательные. После этого управление получает следующий компонент – класс [0-9]. Попытка найти соответствие приводит к неудаче. Поэтому происходит возврат к последнему сохраненному состоянию. То есть : строка The sum is 1234 or mor▲e. Однако сравнение e с [0-9] приводит к неудаче и откат происходит дальше – до первого удачного совпадения с символом 4. Но сравнение пробела со вторым классом приводит к неудаче и машина делает еще один возврат. При этом машина забывает о первом удачном совпадении. В новом состоянии механизм обнаруживает необходимое совпадение и завершает свою работу.

Для более глубокого понимания механизма возврата в машине НКА необходимо рассмотреть один весьма примечательный пример. Этот пример мы уже рассматривали ранее. Сейчас вернемся к нему. Суть задачи состоит в том, что бы в дробной части чисел 12.980076543 сохранить три или отлько две цифры в зависимости от того третья цифра равна или не равна нулю. Так , что приведенное число нужно заменить на 12.98, а число 321.116700333046 на 321.116. Эта проблема решается регулярным выражением $\$number =\sim s/(\backslash.\backslash d\backslash d[1-9]?)\backslash d*/\$1$. Конечно, данное решение без сомнения правильно. Но в случае, когда дробь имеет вид, например 2.123, то замена происходит следующая: дробная часть .123 заменяется на .123. Другими словами , на самом деле, ничего не надо менять. Поставим перед собой задачу добиться такого эффекта – производить замену, только если это требуется. Ясно, что замена необходима, если часть регулярного

выражения $\backslash d*$ совпадает хотя бы с одной цифрой. Кажется, что достаточно использовать такое регулярное выражение $\$number =\sim s/(\backslash.\backslash d\backslash d[1-9]?)\backslash d+/\$1$. Для чисел вида 1.34590075654377 все остается без изменения, для чисел вида 4.54 подвыражение $\backslash d+$ не имеет соответствия и замена не происходит. Но числа вида 1.123 регулярное выражение заменит на 1.12. Действительно, как только машина обнаружит за цифрой 3 несоответствие для $\backslash d+$ механизм осуществит возврат, поскольку часть $[1-9]$ была не обязательной! В новом состоянии имеется соответствие $\backslash d+$ и произойдет указанная выше замена. Мы приходим в тупиковую ситуацию. Некоторые диалекты регулярных выражений позволяют решить проблему с помощью так называемых атомарных группировок.

И так, обратимся к тому же примеру. В этом примере мы хотим, по существу , добиться несовпадения, которое нас больше устроит, чем совпадение. Совпадение происходит из-за возврата в сохраненное состояние благодаря квантификатору "?". Или так – если имеется совпадение для [1-9], то вариант без совпадения с [1-9] не нужно сохранять. С этой целью используются скобки (?> ...) – атомарная группировка. Поиск совпадения для нее проходит как обычно, но когда процесс поиска выходит за пределы этой конструкции, все сохраненные состояния удаляются. Теперь решение задачи выполняется регуляным выражением $\$number =\sim s/(\backslash.\backslash d\backslash d(?>[1-9]?))\backslash d+/\$1$. Действительно, если совпадения для [1-9] нет, то внутри скобок машина возвращается к сохраненному состоянию и затем выходит из конструкции для поиска соответствий для $\backslash d+$. Если же совпадение для [1-9] было найдено, то машина сначала сохраняет состояние для не совпадения с [1-9] , находит соответствие с ним и выходит из атомарной группировки, теряя сохраненное состояние, к которому уже не возвращается.

Захватывающие квантификаторы

К захватывающим квантификаторам относятся квантификаторы $?+$, $*+$, $++$, $\{n,m\}+$. Эти квантификаторы действуют так же как и максимальные квантификаторы, но захватив часть строки, с ней не расстаются. Попросту говоря, эти квантификаторы не создают сохраненных состояний. Вот пример

```
$str="Programms";

if($str=~/\b\w++s\b/)
{

 print "OK!\n";

 }
```

```
 else
 {
print "NO!\n";
 }
```

Конструкция типа $\backslash w++$ почти ни чем не отличается от $(?>\backslash w+)$.

Возврат при позиционной проверке

Аналогично атомарной группировке действует позиционная проверка. Напомним. Имеется 4 вда позиционной проверки: позитивная и негативная опережающая проверки и позитивна и негативная ретроспективные проверки. Такая проверка осуществляется отдельно от основного выражения. В процессе проверки подвыражения машина НКА сохраняет возможные состояния и возвращается к ним. Однако при выходе за рамки подвыражения все сохраненные состояния внутри соответствующего подвыражения теряются. Рассмотрим пример.

```
$str="1234.12534789";

if($str=~/(\.\d+)5(.\d\d\d)/) {

print "OK!\n".$1.$2;// печатает OK! .123478

 }

else { print "NO!\n"; }
```

Это понятно, нов случае

```
$str="1234.12534789";

if($str=~/(?=\.\d+)5(.\d\d\d)/)

{
 print "OK!\n".$1;
 }
  else
   {
   print "NO!\n";
    }
```

будет распечатано NO! Это следствие того, что подвыражение $(? = \backslash.\backslash d+)$ успешно находит позицию перед точкой. Переходя к следующему компоненту регулярного выражения – цифре 5, машина отбрасывает все сохраненные состояния. Поэтому, в отличие от предыдущего примера соответствие в целом не находится.

2.2 Обсуждение примеров решений задач

2.2.1 Пример модификации текстового файла в формат HTML

Приведем пример форматирования простого текста в формат HTML. Будем считать, что наш текс состоит из одной строки. Средстами Perl это делается так

```
undef $/;

$text=<FILE>;
```

На первом шаге заменяем специальные символы

```
$text=~s/&/&amp;/g;

$text=~s/>/&gt;/g;

$text=~s/</&lt;/g;
```

Следующий шаг – разбиение на параграфы, которые в HTML выделяются тегом <p>. К сожаление простое решение

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

не подходит из-за того, что мы рассматриваем весь текст файла в виде одной строки. Поэтому мы должны использовать модификатор m.

```
$text=~s/^$/<p>/gm
```

Или, учитывая возможные пробельные символы в пустой строке, окончательно будем иметь

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

Затем переведем электронные адреса в ссылки. Другими слова, если в тексте встречается выражение вида name@volsu.ru, то его следует заменить на гиперссылку типа mailto

```
<a \href="mailto:name@volsu.ru">name@volsu.ru</a>
```

В целом выражение для замены будет иметь вид

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

Для поиска имени простейшим вариантом будет

```
\w[-.\w]*
```

Для имени хоста

```
[-a-z0-9]+(\.[-a-z0-9]+)*\.[a-z]{2,4}
```

Таким образом полное решение замены адресов выглядит так

```
$text=~s/\b(\w[-.\w]*\@[-a-z0-9]+(\.[-a-z0-9]+)*\.[a-z]{2,4})\b/<a
href="mailto:$1">$1<\/a>/gi
```

Так как это не помещается в одной строке, Perl предоставляет следующую возможность

```
$text=~s{

\b

(\w[-.\w]*

\@

[-a-z0-9]+(\.[-a-z0-9]+)*\.[a-z]{2,4})

\b

}

 {

 <a href="mailto:$1">$1<\/a>
}gix
```

x – позволяет вставлять пропуски и комментарии. То есть пробельные символы превращаются в метасимволы. Что бы обозначить в этом случае пробельные символы как литералы, используйте $\backslash s$.

2.2.2 Поиск IP адресов

Требуется в тексте отыскать IP адреса, т.е. последовательности цифр, разделенных точками вида 127.2.3.1. Но иногда такой адрес может быть записан в виде 127.002.0003.001.

Первое решение

```
/^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$/
```

Но в таком или даже упрощенном виде

```
/^\d+\.\d+\.\d+\.\d+$/
```

может соответствовать строкам вида 2345.1111.556774332345.4 Тогда мы требуем присутствия не более трех цифр. Этого можно достигнуть двумя способами (в зависимости от того поддерживаются диалектом регулярных выражений квантификаторы $\{n, m\}$ или нет).

```
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
```

```
/^\d\d?\d?\.\d\d?\d?\.\d\d?\d?\.\d\d?\d?$/
```

Теперь займемся вопросом построения регулярного выражения с учетом того, что числа в IP адресе должны лежать в диапазоне 0-255.

Во-первых, заметим, что если число состоит из 1 или 2 цифр, то проверять принадлежность диапазону не нужно. Проверка не нужна и в случае когда трехзначное число начинается с нуля или единицы. Тогда можно сформировать такую конструкцию выбора

```
\d|\d\d|[01]\d\d
```

Далее, если первая цифра 2, то число может быть не больше 255. Поэтому, если вторая цифра меньше 5, то число правильное. Если оно равно 5, то третья должна быть меньше 6. Это можно записать так

```
2[0-4]\d|25[0-5].
```

В результате получаем выражение

```
\d|\d\d|[01]\d\d|2[0-4]\d|25[0-5]
```

или так

```
[0-1]?\d\d?|2[0-4]\d|25[0-5].
```

Окончательно регулярное выражение теперь примет вид

```
[0-1]?\d\d?|2[0-4]\d|25[0-5]\.[0-1]?\d\d?|2[0-4]\d|25[0-5]\.
```

```
[0-1]?\d\d?|2[0-4]\d|25[0-5]\.[0-1]?\d\d?|2[0-4]\d|25[0-5]
```

Однако в реальной ситуации можно воспользоваться одним из первых вариантов, сохранив числа в переменные $\$1, \$2, \$3, \$4$ проверив принадлежность необходимому варианту с помощью операторов Perl.

Оглавление