Регулярные выражения в Perl

Оригинал: http://www.perl.ru/

Определения

Регулярные выражения в perl одна из самых мощных его возможностей. regex позволяют в perl сопоставлять текст с указанным шаблоном, разбивать текст в массив по шаблону, производить замену текста по шаблону и многое многое другое. Так-же иногда регекспами называются операторы поиска и замены.

Оператор q(text) заменяет строку text на строку, заключенную в одинарные кавычки(например если в q(text) поставить символ q(text\n), то напечатает text\n , т.е. \n это два

символа, подобно print 'amam $file' напечатает amam $file). В данном случае почти все специальные символы не будут интерпретироваться внутри q(), исключая '\'

```
$some=q(Don't may be);
```

Оператор qq~text~; (вместо значка ~ можно ставить например знак |) позволяет работать со строками и многострочными текстами. пользуясь этим оператором можно выводить целые куски html-кода и писать в этом коде имена скалярных переменных.

Оператор qw("text") разбивает строку на массив слов.

```
@mass=qw("я вышел погулять и увидел как через реку строят новый мост");
#хотя с настроенной локалью будет работать и
@mass=qw(я вышел погулять и увидел как через реку строят новый мост);
for(@mass){print $_,"\n"}
```

Оператор qr/pattern/ ключи - imosx работает подобно регулярному выражению s/.../.../

```
$rex=qr/my.STRING/is;
s#$rex#foo#;
#тоже самое, что и
s/my.STRING/foo/is;
```

Результат может использоваться подобно вызову подпрограммы(см perldoc perlop Regex quote like operator)

```
$re=qr/$pattern/;
$string=~/foo${re}bar/;
$string=~$re;
$string=~/$re/;
```

Ключи imosx стандартные(см. ниже)

Оператор qx/STRING/ работает как системная команда, подобно $output = `cmd 2>$1`;. Программа, иллюстрирующая использование данного оператора:

```
#!/usr/bin/perl
qx[dbfdump --fs="\x18" --rs="\x19"  pdffile.dbf >pdffile.txt];
```

файл pdffile.dbf содержит memo-поля(memo-поле содержит ссылку, подобно функции seek, на текст в файле с расширением *.fpt), которые при помощи DBI.pm мне когда-то давно выудить не удалось. Принимает разрешения FoxBASE4 и дампит файлы со встроенными memo-полями в текстовый вид. Т.е. таким образом получилось вытащить информацию из файла memo-типа *.fpt.

Допустим используя команду $perl_info = qx(ps $$); мы выводим информацию о текущем процессе запущенного скрипта(каждая запущенная программа в UNIX имеет свой собственный уникальный идентификатор, который содержится во встроенной переменной $$ - достаточно

уникальное число, можно использовать почти как счетчик случайных чисел). Если сказать `$shell_info = qx'ps $$';` то выведет информацию о самом `ps`. Т.е. скобки осуществляют своеобразное экранирование от двойной кавычки.

В перл есть три основных оператора, работающих со строками:

`m/.../` - проверка совпадений (matching),
`s/.../.../` - подстановка текста (substitution),
`tr/.../` - замена текста (translation).

Опертаор `m/.../` анализирует входной текст и ищет в нем подстроку совпадающую с указанным шаблоном (он задан регулярным выражением). Оператор `s/.../.../` выполняет подстановку одних текстовых фрагментов вместо других, при помощи регулярных выражений. Оператор `tr/.../.../` заменяет выходной текст, но при этом он не использует регулярные выражения, осуществляя замену посимвольно.

Оператор `m/шаблон/` - поиск подстроки по определенному шаблону. Например `print "$1 г.\n" while m!((\d){4})!g` найдет и выведет все даты в переменной `$_`. В шаблоне не важно, что будет его ограничителем. Например при поиске гиперссылок, которые зачастую содержат символы `/`, разумнее пользоваться не `/`, а например `#` или `!` как символами ограничителями. В таком случае шаблон будет более прост для понимания другим программистам, да и немного короче. В perl оператор `m/.../` используется очень часто, и поэтому используется сокращение, без начальной буквы `m`. Если начальная буква есть, то в качетсве символов ограничителей можно исползовать любой другой символ.

Для оператора `m/pattern/` есть 6 параметров: `gimsxo`

`m/foo/g` говорит компилятору найти все `foo` в тексте, в то время как `m/foo/` найдет только первое вхождение подстроки `foo` в строке `$_`. В строке `$_` содержится обычный текст, как и в переменной `$text`$, `$_` такая-же переменная, только она существует всегда и вводится, когда не определена специально другая, по умолчанию.

Например можно сказать `for (@mass){print $_,"\n"}` или `for $elem (@mass){print $elem,"\n"}`. Эти две строчки делают одно и то-же, но в первом случае запись короче, да и зачастую бывает удобно использовать переменную `$_`, например, когда нужно выделить при помощи регулярного выражения определенные данные, пользуясь перебором массива(функция map):

```
@res=map{/(\d\d\d\d)/} split /\s/, $texts;
```

что эквививалентно коду

```
push @res, $1 while m!((\d){4})!g; #(в данном случае $_=$texts)
```

или что эквивалентно конструкции

```
foreach(split /\s/, $texts){
push @res, $1 if(/(\d\d\d\d)/g)
}
```

Следующий параметр m/foo/i, говорит о том, что не нужно учитывать регистр при поиске по подстроке.

Параметр m/foo/s говорит от том, что строка, по которой производится поиск, состоит из одной строчки.

Например нужно выцепить все url картинок из странички www.astronomynow.com, чтобы сделать локальное зеркало этой странички и пользователи могли с интересом читать последние новости астрономии:

```
#!/usr/bin/perl -wT
use LWP::Simple;
$page=get "http://www.astronomynow.com";
&getlink($page);
sub getlink{
  local $_=$_[0];
  push(@res, "http://$2")
    while m{SRC\s*=\s*(["'])http://(.*?)\1\s*(.*?)WIDTH="100" HEIGHT="100"(.*?)>}igs
}
```

В подпрограмме заводится при помощи функции local переменная, видимая только в области действия подпрограммы. Этой переменной присваивается значение переменной $page, в которой содержится текст выкачанной Simple.pm странички.

Можно сделать немного по другому, сохранить скачанную страничку в файл на диск и затем следующее:

```
$/="\001";
open F, "<page.html"; $page=<F>; close F;
&getlink($page);
...
```

Встроенная переменная $/ содержит символ разделителя входных записей. Это может быть перевод каретки или, при upload far'ом на сервер файлов в не ASCI виде, она приобретают на конце строчки хитрый символ ^M.

Если $/ переопределить, то можно свободно пользоваться дескрипторами открытия файлов для просмотра многострочного текста(m/pattern/s). Например когда открывается файл при помощи функции open F, "<file.txt"; @mass=<F>, то присваивая дескриптор F массиву в массиве появятся строчки, разделенные символом, содержащимся в $/.

Переопределив $/ можно запросто написать:

```
open F, "<file.txt"; $mass=<F>
```

и в переменной $mass будет содержаться многострочный текст с точки зрения человека, но программа будет видеть этот текст как одну строку и по тексту можно будет запросто пройтись поиском m/pattern/igs и выделить все необходимые подстроки.

Параметр m/foo/o говорит от том, что шаблон нужно компилировать только один раз. Если оператор используется в сочетании с операциями привязки =~ и отрицание !~, то строкой, в которой ведется поиск, является переменная, стоящая слева от операции привязки. В противном случае поиск ведется в строке $_.

Оператор s!pattern!substring! - поиск в строке по шаблону pattern и замена найденного текста на substring. Как и для оператора m/.../, косую черту можно не ставить, пригоден любой символ, который не находится в противореции с заданным выражением. Не рекомендуется использовать в качестве ограничителей ? и '.

s!/usr/local/etc/!/some/where/else! - заменяет путь.
s(/usr/local/etc/)(/some/where/else)g - заменяет все встречающимеся пути до файла.

параметры: egimsxo e - указывает, что substring нужно вычислить.

например нужно переделать все escape последовательности, для этого вызывается соответствующая подпрограмма:

```
$text =~ s/(&.*?;)/&esc2char($1)/egs;
```

т.е. из регулярного выражения происходит вызов подпрограммы.

g - заменить все одинаковые компоненты, а не один, как в отсутствии ключа g.
i - не учитывать регистр.
m - строка, в которой происходит поиск, состоит из множества строк.
s - строка, в которой происходит поиск, состоит из одной строки.
x - сложный шаблон, т.е. можно писать не в строчку, а для упрощения понимания разбивать шаблон на несколько строк, примеры об этом ниже.
o - компилировать шаблон один раз.

Допустим нужно сделать поисковик, который ходит по директориям на сервере, но некоторые директории типа /cgi-bin/ и т.п. индексировать нельзя. Объявляем переменную, которая будет содержать регулярное выражение, в данном случае перечисление или img или image или temp или tmp или cgi-bin:

```
$no_dir = '(img|image|temp|tmp|cgi-bin)';
```

Ключи регулярного выражения m#no_dir#io говорят о том, что компилировать содержимое $no_dir нужно только один раз(ключ o) и также еще не учитывать регистр(ключ i).

Оператор tr/выражение1/выражение2/, ключи cds

Смысл: замена выражения1 на выражение2. Если указан ключ c, то это инверсия выражения1, т.е. в выражение один не входят содержащиеся в нем символы. если указа ключ d, то значит стереть замененные символы. Если указан ключ s, то значит заменить многочисленные повторяющиеся символы на одиночный символ.

Оператор y/выражение1/выражение2/(ключи cds), равносилен оператору tr.

Например в поисковой системе нужно приводить запрос в нижний регистр, дабы не зависеть от настроек локали:

```
$CAP_LETTERS = '\xC0-\xDF\xA8';
$LOW_LETTERS = '\xE0-\xFF\xB8';

$code = '$html_text =~ ';
$code .= "tr/A-Z$CAP_LETTERS/a-z$LOW_LETTERS/";
$down_case = eval "sub{$code}";
```

одиночные символы

В регулярном выражении любой символ соответствует самому себе, если только он не является метасимволом со специальным значением (такими метасимволами являются \, |, (,), [, {, *, +, ^, $, ? и .). В следующем примере проверяется, не ввел ли пользователь команду "quit" (и если это так, то прекращаем работу программы):

```
while(<>){
  if(m/quit/){exit;}
}
```

Правильнее проверить, что введенное пользователем слово "quit" не имеет со-седних слов, изменяющих смысл предложения. (Например, программа выполнит заведомо неверное действие, если вместо "quit" пользователь введет команду "Don't quit!".) Это можно сделать с помощью метасимволов ^ и $. Заодно, что-бы сравнение было нечувствительно к разнице между прописными и заглавными буквами, используем модификатор i:

```
while (<>)
{if (m/^quit$/i) {exit;} }
```

Кроме обычных символов perl определяет специальные символы. Они вводятся с помощью обратной косой черты (escape-последовательности) и также могут встречаться в регулярном выражении:

- \077 - восьмеричный символ,
- \a - символ BEL (звонок),
- \c[- управляющие символы (комбинация Ctrl + символ, в данном случае это управляющий символ ESC),

- \d - соответствует цифре,
- \D - соответствует любому символу, кроме цифры,
- \e - символ escape (ESC),
- \E - конец действия команд \L, \U и \Q,
- \f - символ прогона страницы (FF),
- \1 - следующая литера становится строчной (lowercase),
- \L - все последующие литеры становятся строчными вплоть до командй \E,
- \n - символ новой строки (LF, NL),
- \Q - вплоть до команды \E все последующие метасимволы становятся обычными символами,
- \r - символ перевода каретки (CR),
- \s - соответствует любому из "пробельных символов" (пробел, вертикальная , или горизонтальная табуляция, символ новой строки и т. д.),
- \S - любой символ, кроме "пробельного",
- \t - символ горизонтальной табуляции (HT, TAB),
- \u - следующая литера становится заглавной (uppercase),
- \U - все последующие литеры становятся заглавными вплоть до команды \E,
- \v - символ вертикальной табуляции (VT),
- \w - алфавитно-цифровой символ (любая буква, цифра или символ подчеркивания),
- \W - любой символ, кроме букв, цифр и символа подчеркивания,
- \x1B - шестнадцатиричный символ.

Bat также можете "защитить" любой метасимвол, то есть заставить perl рассматривать его как обыкновенный символ, а не как команду, поставив перед метасимволом обратную косую черту \. Обратите внимание на символы типа \w, \d и \s, которые соответствуют не одному, а любому символу из некоторой группы. Также заметьте, что один такой символ, указанный в шаблоне, соответствует ровно одному символу проверяемой строки. Поэтому для задания шаблона, соответствующего, например, слову из букв, цифр и символов подчеркивания, надо использовать конструкцию \w+, как это сделано в следующем примере:

```
$text = "Here is some text."
$text =~ s/\w+/There/;
print $text;
There is some text.
```

классы символов

Символы могут быть сгруппированы в классы. Указанный в шаблоне класс символов сопоставляется с любым из символов, входящим в этот класс. Класс - это совокупность символов, заключенный в квадратные скобки [и]. Можно указывать как отдельные символы, так и их диапазон (диапазон задается двумя крайними символами, соединенными тире). Наример, следующий код производит поиск гласных:

```
$text ="Here is the text.";
if ($text =" /[aeiou]/) {print "Vowels: we got 'em.\n";}
Vowels: we got 'em.
```

Другой пример: с помощью шаблона [A-Za-z]+ (метасимвол + означает утверждение: "один или более таких символов") ищется и заменяется первое слово:

```
$text = "What is the subject.";
$text =" s/[A-Za-z]+/Perl/;
print $text;
Perl is the subject;
```

Если требуется задать минус как символ, входящий в класс символов, перед ним надо поставить обратную косую черту \-. Если сразу после открывающей квадратной скобки стоит символ ^, то смысл меяется на противоположный. А именно, этот класс сопоставляется любому символу, кроме перечисленных в квадратных скобках. В следующем примере производится замена фрагмента текста, составленного не из букв и не из пробелов:

```
$text = "perl is the subject on page 493 of the book.";
$text =~ s/[^A-Za-z\s]+/500/;
print $text;
perl is the subject on page 500 of the book.
```

альтернативные шаблоны

Вы можете задать несколько альтернативных шаблонов, используя символ | как разделитель. Альтернативные шаблоны позволяют превратить процедуру поиска из однонаправленного процесса в разветвленный: если не подходит один шаблон perl подставляет другой и повторяет сравнение, и так до тех пор, пока не иссякнут все возможные альтернативные комбинации. Например, следующий фрагмент проверяет, не ввел ли пользователь "exit", "quit" или "stop":

```
while (<>){
  if(m/exit|quit|stop/){exit;}
}
```

Чтобы было ясно, где начинается и где заканчивается набор альтернативных шаблонов, их заключают в круглые скобки - иначе символы, расположенные справа и слева от группы шаблонов, могут смещаться с альтернативыми шаблонами.

В следующем примере метасимволы ^ и $ обозначают начало и конец строки и отделяются от набора альтернативных шаблонов с помощью скобок:

```
while (<>){
  if(m/^(exit|quit|stop)$/){exit;}
}
```

Альтернативные варианты перебираются слева направо. Как только найдена первая альтернатива, для которой выполняется совпадение с шаблоном, перебор прекращается. Участки шаблона, заключенные в круглые скобки, выполняют специальную роль при выполнении операций поиска и замены. Если символ \ находится в квадратных скобках, он интерпретируется как обычный символ. Поэтому если вы используете конструкцию шаблона вида [Tim|Tom|Tam], то она будет эквивалентна классу символов [Tioam|]. Точно так же

большинство других метасимволов и команд, специфичных для регулярных выражений - в частности, квантификаторы и мнимые символы, описанные в двух последующих разделах, - внутри квадратных скобок превращаются в обычные символы или escape-последовательности текстовых строк.

квантификаторы

Квантификаторы в регулярных выражениях

Квантификаторы указывают на то, что тот или иной шаблон в строке может повторяться определенное количество раз. Например, можно использовать квантификатор `+` для поиска мест неоднократного, повторения подряд латинской буквы `e` и их замены на одиночную букву `e`:

```
$text = "Hello from Peeeeeeeeeeeeeeerl.";
$text =~ s/e+/e/g;
print $text;
Hello from perl.
```

мнимые символы

Мнимые символы в регулярных выражениях

В perl имеются символы (метасимволы), которые соответствуют не какой-либо литере или литерам, а означают выполнение определенного условия (поэтому в английском языке их называют assertions, или утверждениями). Их можно рассматривать как мнимые символы нулевого размера, расположенные на границе между реальными символами в точке, соответствующей определенному условию:

- `^` - начало строки текста,
- `$` - конец строки или позиция перед символом начала новой строки, распо-ложенного в конце,
- `\b` - граница слова,
- `\B` - отсутствие границы слова,
- `\A` - "истинное" начало строки,
- `\Z` - "истинный" конец строки или позиция перед символом начала новой строки, расположенного в "истинном" конце строки,
- `\z` - истинный конец строки,
- `\G` - граница, на которой остановился предыдущий глобальный поиск, выполняемый командой `m/.../g`,
- `(?= шаблон)` - после этой точки есть фрагмент текста, который соответствует указанному регулярному выражению,
- `(?! шаблон)` - после этой точки нет текста, который бы соответствовал указанному регулярному выражению,
- `(?<= шаблон)` - перед этой точкой есть фрагмент текста, соответствующий указанному регулярному выражению,

- (?<! шаблон) - перед этой точкой нет фрагмента текста, соответствующего указанному регулярному выражению.

Например, вот как выполнить поиск и замену слова, используя метасимволы границы слов:

```
$text = "Here is some text.";
$text = s~/\b([A-Za-z)+)\b/There/;
print $text;
There is some text.
```

perl считает границей слова точку, расположенную между \w и \W, независимо от того, в каком порядке следуют эти символы . В следующем примере выводится сообщение о том, что пользователь ввел слово "yes", при условии, что оно единственное, что ввел пользователь. Для этого шаблон включает мнимые символы начала и конца строки:

```
while (<>) {
 if (m/^yes$/) {
  print "Thank you for being agreeable.\n";
  }
}
```

Приведенный выше пример требует комментария. Прежде всего, бросается в глаза наличие двух групп метасимволов для начала и конца строки. В большинстве случаев они означают одно и то же, так как обычно символы новой строки (то есть \n), встречающиеся внутри текстового выражения, не рассматриваются как вложенные строки. Однако если для команды m/.../ или s/.../.../ указан модификатор m, то текстовое выражение будет рассматриваться как многострочный текст, в котором границами строк выступают символы новой строки \n. В случае многострочного текста метасимвол ^ сопоставляется с позицией после любого символа новой строки, а не только с началом текстового выражения. Точно также метасимвол $ - это позиция перед любым символом новой строки, расположенным внутри текстового выражения, а не обяательно конец текстового выражения или же позиция перед концевым символом \n. Однако метасимвол \A - начало текстового выражения, а метасимвол \Z - конец текстового выра-жения или позиция перед концевым символом \n, даже если в текстовом выражении имеются вложенные символы \n и при выполнении операции поиска или йены указан модификатор m. Метасимвол точка (.) соответствует любому символу, кроме символа новой строки \n. Независимо от того, задан ли модификатор m, она не будет сопоставляться ни с внутренними, ни с концевыми символами \n. Единственный способ заставить точку рассматривать \n как обычный символ - использовать модификатор s.

Отсюда понятна разница между метасимволами \Z и \z. Если в качестве текстового выражения используется результат чтения входного потока данных, то с большой вероятностью данное выражение заканчивается символом \n, за исключением того варианта, когда программа предусмотрительно "отщипнула" его с помощью функции chop или chomp. Метасимвол \Z игнорирует концевой символ \n если он случайно остался на месте, рассматривая обе ситуации как "конец строки". В отличие от него метасимвол \z оказывается более

пунктуальным и рассматривает концевой символ \n как неотъемлемую часть проверяемого текстового выражения, если только пользователь не позаботился об удалении этого символа.

Отдельно следует остановиться на метасимволе \G. Он может указыватьсяв регулярном выражении только в том случае, если выполняется глобальный поиск (то есть если команда m/.../ имеет модификатор g). Метасимвол \G, указанный в шаблоне, соответствует точке, на котброй остановилась предыдущая операция поиска.

ссылки на найденный текст

Иногда нужно сослаться на подстроку текста, для которой получено совпадение с некоторой частью шаблона. Например, при обработке файла, HTML может потребоваться выделять фрагменты текста, ограниченные открывающими и закрывающими метками HTML (например, <A> и </A>). В начале уже приводился пример, в котором выделялся текст, ограниченнуй метками HTML <A> и <B>. Следующий пример позволяет выделять текст, расположенный между любыми правильно закрытыми метками:

```
$text = "<A>Here is an anct1or.</A>";
if($text=~m%<([A-Za-z]+)>[\w\s\.]+</\1>%i){
}
```

Вместо косой черты в качестве ограничителя шаблона использован другой символ. Это позволяет использовать символ косой черты внутри шаблона без предшествующей ему обратной косой черты. Каждому фрагменту шаблона, заключенному в круглые скобки, соответствует определенная внутренняя переменная. Переменные пронумерованы, так что на них можно ссылаться внутри шаблона, поставив перед номером обратную косую черту (\1, \2, \3,...). На значения переменных можно ссылаться внутри шаблона, как на обычный текст, поэтому </\1> соответствует </\A>, если открываю-щей меткой служит<A>, и , если открывающей меткой служит . Эти же самые внутренние переменные можно использовать и вне шаблона, ссылаясь на них как на скаляры с именами $1, $2, $3,..., $n:

```
$text = "I have 4 apples.";
if ($text =- /(\(\d+)/) {
print "Here Is the number of apples: $1.\n";
Here is the number of apples: 4.
```

Каждой паре скобок внутри шаблона после завершения операции поиска будет соответствовать скалярная переменная с соответствующим номером. Это можно 'использовать при выделении нужных для последующей работы фрагментов ана-лизируемой строки. В следующем примере мы изменяем порядок трех слов в тек-стовой строке с помощью команды s/.../.../:

```
$text = "I see you.";
$text=-s/^(\w+) *(\w+) *(\w+)/$3 $2 $1/;
print $text;
you see I.
```

Переменные, соответствующие фрагментам шаблона, нуумеруются слева направо с учетом вложенности скобок. Например, после следующей операции поиска будут проинициализированы

шесть переменных, соответствующих шести парам скобок:

```
$text = "ABCDEFGH";
$text =- m/(\w(\w)(\w))((\w)(\w))/;
print "$1/$2/$3/$4/$5/$6/";
ABC/B/C/DE/D/E
```

Кроме переменных, ссылающихся на найденный текст, можно использовать специальные переменные perl.

Функции, использующие регулярные выражения

Фактически, есть три функции, которые в качестве разделителя могут использовать регулярные выражение: `split`, `grep`, `map` и еще можно воспользоваться специальными операторами `...` и `..` и используемыми совместно с ними условиями `if`, `unless` и просто логическими операторами.

split

Если необходимо разделить данные из `STDIN` по нужному разделителю, то можно воспользоваться локализацией `$/`:

```
sub example_local{
  local $/ = undef;
  @mass= split /pattern/, <>;
  return 1;
}
print scalar(@mass);
```

Можно разделять данные из файла и так:

```
undef $/;
@res=split /pattern/, <F>;
```

что эквивалентно:

```
while (<F>) {push @asdf, split}
```

После split можно ставить вместо запятой и стрелочку:

```
@mass = split /(\d){4}/ => $file;
```

В функции сплит можно воспользоваться макисмальным квантификатором `*`, который в том числе и о символов, позволит разделить строку на символы, которых там нет(в силу того, что `*` это `0` и более символов), т.е. посимвольно:

```
@ruru = split /\001*/ => "lalalalalala";
#массив @ruru будет содержать элементы по одной букве.
```

Если строка состоит из нескольких строк, то можно поставить разделителем и символ начала новой строки:

```
$str = "asdf\nghjk\nqwer\n";
@lines = split /^/ => $str;
```

Вобщем, в split можно вставлять любой поиск по шаблону.

grep

Функция grep так-же позволяет запонять массив значениями. Например нужно получить список расширений файлов в заданной директории:

```
while(<$dir/*.*>){push @files, $_}             #читаем директорию
@test = grep { s|.*/(.*?)\.(.*)|$2| } @files;  #оставляем в директории только расширения файлов
```

можно использовать признак четности для занесения в массив:

```
@test1=qw(1 2 3 4 5 6 7 8 9);
@evens = grep($_%2 == 1) @test1;
```

Или более сложное регулярное выражение для вытаскивания всех e-mail адресов из текстовой странички:

```
@mass=grep{s/(.*) ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})(.*)/$2/ig} split /\n/, $test;
```

Здесь используется укороченная запись:

```
@mass=grep {/pattern/} split /\n/, $test;
```

которая эквивалента записи из двух сторчек:

```
@uuu=split /\n/, $test;
@mass=grep {/pattern/} @uuu;
```

map

Функция map похожа по своей работе на обычное условие `if`, допустим нужно разделить записи на блоки, разделенные четырьмя пробелами:

```
@probel = map m!\s{4}!, split /\n/, $test;
```

other

Вывод строк из заданного интервала для данной строки:

```
if(/pattern1/i .. /pattern2/i){...}
#истинность первого оператора включает конструкцию, а второго е  выключает.
if($nomer1 .. $nomer2){...}
```

`...` не возвратит истину, в отличии от `..`, если условия выполняются в одной строке.

```
if(/pattern1/i ... /pattern2/i){...}
if($nomer1 ... $nomer2){...}
```

для многострочного файла

```
print -ne 'print if 3 .. 15' file.txt
```

выведет строки файла с 3 по 15 строчку, та-же самая опреация но немного по другому:

```
open F, "<file";
while(<F>){
  print if(3 .. 15)
}
```

или с какой нибудь начальной и конечно разметкой, например есть вспомогательный файл шаблонов(просто различные виды html, в зависимости от действия пользователя) для разных определенны случаев, которые нужны исходя из контекста программы:

```
open F, "<file";
while(<F>){
  print if(/<!--begin welcome-->/i ... /<!--end welcome-->/i)
}
```

Такая конструкция позволяет выводить куски многострочного html кода(для однострочного нужно ставить оператор `..`). Условия в таких операторах можно ставить и разнотипными

```
$file=qr/2345/;
while(<F>){
  print if(/^$/ .. 10); #увидим, что находится от пустой до 10-й строки
  print if(/^\001/ .. /$file/); #выведет все, что после нуля и до того что задано qr
}
```

Программа чтения почтовых адресов из mbox или sent-mail:

```
while(<F>){
  next unless /^From:?\s/i .. /^$/;
  while (/([^<>(,;)\s]+\@[^<>(,;)\s]+)/)g{
    print "$1\n" unless $test{$1}++;
  }
}
```

запускается `./regex.pl /root/mail/sent-mail` и выводит каждый емейл по одному разу.

Использование встроенных переменных

- `$'` - подстрока, следующая за совпадением.
- `$&` - совпадение с шаблоном поиска
- `$`` - подстрока, расположенная перед совпадением

- $^R - результат вычисления утверждения в теле шаблона
- $n - n-ный фрагмент совпадения
- \n - n-ный фрагмент совпадения вызываемый в самом шаблоне
- $+ - фрагмент совпадения
- $* - разрешает выполнять поиск в многострочных файлах
- @- - спецмассив, который содержит начальную позицию найденного слова
- @+ - массив, содержащий позицю последнего найденного слова

$& - совпадение с шаблоном поиска, при последней операции поиска или замены. В отличии от переменной $_, эту переменную переопределять как вздумается нельзя.

$' подстрока за совпадением с шаблоном поиска, е также можно только читать.

$` - подстрока, расположенная перед совпадением, разрешается только е чтение.

$^R - результат вычисления утверждения в теле шаблона для последнего вычисления шаблона, если в нем идет счет или вызывается внешняя программа:

```
$qwer="lala";
$qwer=~ /x(?{$var=5})/;
print $^R;
5
```

$+ - фрагмент совпадения в шаблоне, который в нем был последним в круглых скобках. Разрешается только чтение $+.

$* - разрешает выполнять поиск в многострочных файлах, булева переменная, если она взведена в 1, то символы шаблона поиска ^ и $ сопоставляются позициям перед и после внутренних символов новой строки, если 0, то от начала текста и до конца текста:

```
$kim="lala\nfa\eti\nzvuki...";
$kim=~~ /^eti/; #совпадение не нашлось
$*=1;
$kim=~~ /^eti/; #совпадение нашлось
```

$n - n-ный фрагмент совпадения:

```
print "$1 $2 $3\n" if(/^(\d)(\w)(\W)$/);
```

\n - n-ный фрагмент совпадения вызываемый в самом шаблоне, например поиск гиперссылок:

```
/a href=(['"])(.*?)\1>/
```

Например нужно занести в массив только цифры из строчки "12@#34@@#@@###34@@##67##@@#@#@34":

```
$_='12@#34@@#@@###34@@##67##@@#@#@34';
s/@/#/g;
```

```
s/(#)\1+/$1/g;
print join /\n/, split /#/, $_;
```

Регулярное выражение s/(#)\1+/$1/g; использует повторение переменной $1 (квантификатор +) и если оно есть, то заменяет все подряд идущие # между цифрами на одну #, содержащуюся в$1(переменная $1 существует, если часть шаблона или шаблон указать в круглых скобках).

Допустим нужно определить, все ли цифры числа различны. Попробуем найти хотя-бы одно повторяющееся число:

```
if(/(\d).*(?=\1)/g){
print "по крайней мере одна цифра $1 различна\n";
}
```

Выражение берет 1-ю цифру и ищет е совпадения со всеми остальными, если есть, то говорит, что найдено и заканчивает работу. Регулярное выражение берет первое число при помощи (\d) и начинает его сравнивать со всеми остальными числами при помощи .*(?=\1). Если первое число в строке уникально, регулярное выражение начнет сопостовлять второе число со всеми восемью оставшимися числами. Если и второе число в строке уникально, то берется третье число и сравнивается со всеми остальными. И т.д., если совпадение было найдено, то регулярное выражение возвращает true и заканчивает свою работу, даже если в строке еще есть повторяющиеся числа. Чтобы можно было просмотреть все повторяющиеся числа, можно воспользоваться модификацией предыдущего кода:

```
$_ = '2314152467';
my @a = m/(\d)(?=\d*\1)/g ;
if (@a){
  print join(',',@a)," - Repeat\n";
}
else{
  print "Ok\n" ;
}
```

Этот усовершенствованный код работает до тех пор, пока не будут найдены все совпадения, если таковые вообще есть.

В perl 5.6 вводятся переменные @- и @+, комбинация которых может заменять переменные $`, $&, и $'. После совпадения шаблона переменная $-[0] содержит начало соответсвия текста шаблону, а переменная $+[0] содержит конец соответсвия текста шаблону. В начале поиска обе являются нулями. Это значит, что можно вычислить значения $`, $&, и $':

```
$do      = substr($stroka, 0, $-[0]);
$sovpalo = substr($stroka, $-[0], $+[0] - $-[0]);
$posle   = substr($stroka, $+[0]);
```

Например:

```
$test="11-231234";
$test=~/\d{2}-\d{6}/;
print "$-[0], $+[0]";
0, 9
```

Соответственное переменные $#- и $#- указывают размерность массивов @- и @+.

Переменная $^N.

Как работают регулярные выражения

Регулярные выражения, использующие квантификаторы, могут порождать процесс, который называется перебор с возвратом (backtracking). Чтобы произошло совпадение текста с шаблоном, надо построить соответствие между текстом и всем регулярным выражением, а не его частью. Начало шаблона может содержать квантификатор, который поначалу срабатывает, но впоследствии приводит к тому, что для части шаблона не хватает текста или возникает несоответствие между текстом и шаблоном. В таких случаях perl возвращается назад и начинает построение соответствия между текстом и шаблоном с самого начала, ограничивая "жадность" квантификатора (именно поэтому процесс и называется "перебор с возвратом"). Перечислим квантификаторы perl:

- * - ноль или несколько совпадений,
- + - одно или несколько совпадений,
- ? - ноль совпадений или одно совпадение,
- {n} - ровно n совпадений,
- {n,} - по крайней мере n совпадений,
- {n,m} - от n до m совпадений.

Например квантификатор + соответствует фразе "один или несколько" и является жадным. Рассмотрим пошагово принцип перебора с возвратом на примере квантификатора +:

```
'aaabc' =~/a+abc/;
```

a+ сразу в силу жадности совпадает с тремя a:

```
(aaa)bc
```

но после aaa не следует строка "abc", а следует "bc". Поэтому результат - failed поэтому анализатор должен откатиться назад и вернуть с помощью a+ два a: (aa)abc т.е. на втором шаге шаблон найдет совпадение.

Рассмотрим пример работы еще одного жадного квантификатора *(ноль или несколько совпадений):

```
   amxdemxg    /.*m/
```

Сначала будет найдена вся строка abcdebfg в силу жадности .*, потом квантификатору нужно будет найти сравнение с буквой m, произойдет ошибка. Квантификатор .* отдаст одну букву и его содержимое будет уже amxdemx. На конце снова нет буквы m. Будет отдана еще одна буква

и снова не будет найдено совпадение со всем шаблоном и наконец квантификатор .* будет содержать подстроку amxde, за которой уже стоит символ m. И поиск на этом и закончится не смотря на то, что в строке amxdemxg содержится не одна буква m. Потому и говорят, что квантификаторы обладают жадностью, т.е. находят максимально возможное совпадение.

Допустим нужно найти совпадение:

```
$uu="How are you? Thanks! I'm fine, you are ok??";
$uu=~s/.*you//;
print $uu;
```

Квантификатор .* оставит текст " are ok??", а вовсе не "? Thanks! I'm fine, you are ok??". Если же поставить ограничитель ?, который вместе со знаком квантификатора означает максимально возможное совпадение

```
$uu="How are you? Thanks! I'm fine, you are ok??";
$uu=~s/.*you//;
print $uu;
```

то переменная $uu будет содержать текст "? Thanks! I'm fine, you are ok??".

Предположим нужно найти совпадения типа network workshop, т.е. перекрытия.

```
$u='network';
$m='workshop';
print "перекрытие $2 найдено: $1$2$3\n" if("$u $m" =~/^(\w+)(\w+) \2(\w+)$/);
```

$1 сразу берет все слово в $u, но дальше идет еще один максимальный квантификатор (\w+), которому тоже чего-то надо и он забирает из переменной \1 букву k(причем только одну):

```
#!/usr/bin/perl
$uu="asdfg asdf";
$uu=/(\w+)(\w+)\s(\w+)(\w+)/;
print "$1 $2##$3 $4";
asdf g##asd f
```

далее пошаговая работа regex выглядит примерно так:

```
1: 'networ''k'=> '\sk' совпадает ли с '\sworkshop'     falure
2: 'netwo''rk'=> '\srk' совпадает ли с '\sworkshop'    falure
3: 'netw''ork'=> '\sork' совпадает ли с '\sworkshop'   falure
4: 'net''work'=> '\swork' совпадает ли с '\sworkshop'  ok
```

и в результате программа выдаст:

```
перекрытие work найдено: networkshop
```

Данный регексп не сработает, если

```
$u='networkwork';
$m='workshop';
```

шаблон найдет перекрытия workwork, а не work. Чтобы этого избежать, нужно сделать минимальным \1: /^(\w+?)(\w+) \2(\w+)$/

Квантификатор действует только на предшествующий ему элемент шаблона. Например, конструкция \d{2}[a-z]+ будет соответствовать последовательности из одной или нескольких строчных латинских букв, начинающейся с двух цифр, а не последовательности, составленной из чередующихся цифр и букв. Для выделения группы элементов, на которую действует квантификатор, используются круглые скобки: (\d{2}(a-z])+

Логические операции в регулярных выражениях

В регулярных выражениях perl есть синтаксические выражение, позволяющие в шаблонах использовать простые логические конструкции:

- (?= шаблон) - после этой точки есть фрагмент текста, который соответствует указанному регулярному выражению
- (?! шаблон) - после этой точки нет текста, который бы соответствовал указанному регулярному выражению,
- (?<= шаблон) - перед этой точкой есть фрагмент текста, соответствующий указанному регулярному выражению,
- (?<! шаблон) - перед этой точкой нет фрагмента текста, соответствующего указанному регулярному выражению.
- (?#текст) - комментарий. Текст комментария игнорируется.
- (?:шаблон) или (?модификаторы:шаблон) - группирует элементы шаблона. В отличие от обычных круглых скобок, не создает нумерованной переменной. Например, модификатор i не будет делать различия между строчными и заглавными буквами, однако область действия этого модификатора будет ограничена только указанным шаблоном.
- (?=шаблон) - "заглядывание вперед". Требует, чтобы после текущей точки находился текст, соответствующий данному шаблону. Такая, конструкция обрабатывается как условие или мнимый символ, поскольку не включается В результат поиска. Например, поиск с помощью команды /w+(?=\s+)/ найдет слово, за которым следуют один или несколько "пробельных символов", однако сами они в результат не войдут.
- (?!шаблон) - случай, противоположный предыдущему. После текущей точки не должно быть текста, соотносимого с заданным шаблоном. Так, если шаблон w+(?=\s) - это слово, за которым следует "пробельный символ", то шаблон w+(?!\s) - это слово, за которым мет "пробельного символа".
- (?<=шаблон) - заглядывание назад. Требует, чтобы перед текущей точкой находился соответствующий текст. Так, шаблон (?<=\s)w+ интерпретируется как слово, перед которым имеется пробельный символ (в отличие от заглядывания вперед, заглядывание назад может работать только с фиксированным числом проверяемых символов).
- (?<!шаблон) - отрицание предыдущего условия. Перед текущей точкой не должно быть текста, соотносимого с заданным шаблоном. Соответственно, от команды /(?<!\s)w+/

требуется найти слово, перед которым нет пробельного символа.

- (?{код}) - условие (мнимый символ), которое всегда выполняется. Сводится к выполнению команд perl в фигурных скобках. Вы можете использовать эту конструкцию, только если в начале сценария указана команда use re 'eval'. При последовательном соотнесении текста и шаблона, когда perl доходит до такой конструкции, выполняется указанный код. Если полного соответствия для оставшихся элементов найти не удалось, то при возврате левее данной точки шаблона вычисления, проделанные с локальными переменными, откатываются назад. (Условие является экспериментальным. В документации, прилагаемой в perl, можно найти довольно детальное рассмотрение (с примерами) работы этого условия и возможных трудностей в случае его применения.)
- (?>шаблон) - "независимый" или "автономный" шаблон. Используется для оптимизации процесса поиска, поскольку запрещает "поиск с возвратом". Такая конструкция соответствует подстроке, на которую налагается заданный шаблон, если его закрепить в текущей точке без учета последующих элементов шаблона. Например, шаблон (?>a*)ab в отличие от a*ab не может соответствовать никакой строке. Если поставить в любом месте шаблон a*, он съест все буквы a, не оставив ни одной шаблону ab. (Для шаблона a*ab "аппетит" квантификатор * будет ограничен за счет работы поиска с возвратами: после того как на первом этапе не удастся найти соответствие между шаблоном и текстом, perl сделает шаг назад и уменьшит количество букв a, захватываемых конструкцией a*.)
- (?(условие)шаблон-да|шаблон-нет) или (?(условие)шаблон-да) - условный оператор, который подставляет тот или иной шаблон в зависимости от выполнения заданного условия. Более подробно описан в документации perl.
- (?модификаторы) - задает модификаторы, которые локальным образом меняют работу процедуры поиска. В отличие от глобальных модификаторов, имеют силу только для текущего блока, то есть для ближайшей группы круглых скобок, охватывающих конструкцию, Например, шаблон ((?i)text) соответствует слову "text" без учета регистра.

Поиск повторяющихся слов в регулярном выражении осуществляется при помощи т.н. обратных ссылок. Выше уже был приведен пример их использования для выбирания всех адресов рисунков с www.astronomynow.com:

```
m{SRC\s*=\s*(["'])http://(.*?)\1\s+(.*?)WIDTH="100" HEIGHT="100"(.*?)>}igs
```

(["']) - найти либо " либо ' либо ничего, т.к. src=http:// может быть без кавычек. Как только был надено что-либо из этих трех позиций, через минимальное количество символов(регулярное выражение (.*?)) символов оно заносится в специальную переменную \1, которая вне m/.../ может быть вызвана как $1(в s/.../.../ она вызывается в его левую половину как $1). Дальше после *.gif|*.jpg|*.bmp и т.д. должен обязательно идти хотя-бы один пробел \s+, т.к. броузеры воспримут подстроку src=file.gifborder=0 как файл картинки с расширением gifborder=0. Поэтому данное регулярное выражение вполне исправно работает, хотя оно было сделано для сайта, где в img src ставится полный адрес, т.е. начинающийся с http:// Для других сайтов придется выстраивать полные пути в ссылках используя base href, если есть или его url. Если

нужно найти какое-то по счету совпадение шаблона в строке, то это реализуется примерно так:

```
while($str=~/WHAT/g){$n++}
$n++ while $str=~/WHAT/g;
$n++ while $str=~/(?=WHAT)/g;#для перекрывающихся совпадений
for($n=0; $n=~/WHAT/g; $n++){}
```

Каждое кратное совпадение

```
(++$n % 6) == 0;
```

Нужное Вам совпадение:

```
$n=($str=~/WHAT/gi)[6]; #допустим шестое
```

Или каждое четное совпадение

```
@mass=grep{$n++ %2==0} /WHAT/gi;
```

для нечетного нужно написать внутри `grep`: `$n++ %2==1` Логические операции внутри регулярных выражений. Если нужно найти последнее совпадение, то можно воспользоваться отрицанием опережающей проверки `(?!WHAT)`:

```
m#PATTERN(?!.*PATTERN)$#
```

т.е. нийти какой-то `PATTERN`, при этом не должно найтись что-то еще(`.*`) и `PATTERN`, т.е. результат - последнее совпадение;

Минимальные квантификаторы `*?`, `+?`, `??`,`{}?`

допустим нужно найти двойку, перед которой не стоит 3 или пробел:

```
print "$1\n" while m%2(?![3\s])gm%;
```

используется условие по отрицанию, `A(?!B)`: найти `A`, перед которым не находится `B`. Чтобы найти двойку, за которой стоит 3 или пробле(`\s`), то можно воспользоваться:

```
print "$1\n" while m%2(?=[3\s])gm%;
```

или

```
print "$1\n" while m%2(?![^3\s])gm%;
```

где используется `^`, `[^3\s]`, который значит следущее: в класс символов, которые нужно найти, не входят `3` и пробел, или другими словами найти все кроме `3` и `\s`.

Допустим существует HTML-документ, в котором произвольное число вложенных таблиц [<table>.*</table>]. Требуется "вырезать" по очереди самые вложенные таблицы (не содержащие внутри [<table>.*</table>]), и, соответственно, выводить. И так - рекурсивно до конца вырезать изнутри всю таблицу. Ниже представлена программа, реализующая эту задачу при помощи логического оператора `(?!...)`:

```
#!/usr/bin/perl -wT

$file=qq|s<table>aaa bbb
      <table>cc<table>ccc
      <table> 2<table>bb</table> <table>cc</table>    </table></table>cc
  </table>
     ddd</table>d
|;

print $file;
&req($file);
sub req {
  if($file=~m%(<table>((?!.*<table>).*?)</table>)%igs){
    $file=~s%(<table>((?!.*<table>).*?)</table>)%%igs;
    print "Virezali --$1--";
    &req($file);
  }
return $file;
}
```

Продолжаем рассматривать логические операторы в регулярных выражениях на опретаорах типа OR, AND или NOT.

Регексп истиннен, если `/AM|BMA/` или `/AM/ || /BMA/` и если есть перекрытие типа `/BMAM/`. Так-же и `/AM/ && /BMA/`:

```
/^(?=.*AM)(?=.*BMA)/s
```

Выражение истинно если `/AM/ и /BMA/` совпадают при перекрытии которое не разрешено:

```
/AM.*BMA|BMA.*AM/s
```

Выражение истинно, если шаблон `/ABC/` не совпадает:

```
!~/ABC/
```

или

```
/^(?:(?!ABC).)*$/s
```

Выражение истинно, если `ABC` не совпадает, а `VBN` совпадает:

```
/(?=^(?:(?!ABC).)*$)VBN/s
```

Несовпадение можно проверить несколькими способами:

```
unless($str =~ /MMM/){...}
if(!($str =~ /MMM/)){...}
if($str !~ /MMM/){...}
```

Для обязательного совпадения в двух шаблонах:

```
unless ($str !~ /MMM/ && $str !~ /BBB/){...}
#или
if ($str =~ /MMM/ && $str =~ /BBB/){...}
```

Хотя бы в одном

```
unless ($str !~ /MMM/ || $str !~ /BBB/){...}
#или
if ($str =~ /MMM/ || $str =~ /BBB/){...}
```

Регулярные выражения - основа работы с операторами m/.../ и s/.../.../, так как они передаются последним в качестве аргументов. Разберемся, как устроено регулярное выражение \b([A-Za-z)+)\b, осуществляющее поиск отдельных слов в строке:

```
$text = "Perl is the  subject.";
$text =~/\b([A-Za-z]+)\b/;
print $1;
```

Выражение \b([A-Za-z]+)\b включает в себя группирующие метасимволы (и), метасимвол границы слова \b, класс всех латинских букв [A-Za-z] (он объединяет заглавные и строчные буквы) и квантификатор +, который указывает на то, что требуется найти один или несколько символов рассматриваемого класса. Поскольку регулярные выражения, как это было в предыдущем примере, могут быть очень сложными, разберем их по частям. В общем случае регулярное выражение состоит из следующих компонентов:

Совпадение с любым символом

В perl имеется еще один мощный символ - а именно, точка (.). В шаблоне он соответствует любому знаку, кроме символа новой строки. Например, следующая команда заменяет в строке все символы на звездочки (использован модификатор g, обеспечивающий глобальную замену):

```
$text = "Now is the time.";
$text =~ s/./*/g;
print $text;
********************
```

А что делать, если требуется проверить совпадение именно с точкой? Символы вроде точки (конкретно, \|()[{^$*+?.), играющие в регулярном выражении осббую роль) называются, как уже было сказано выше, метасимволами, и если вы хотите, чтобы они внутри шаблона интерпретировались как обычные символы, метасимволу должна предшествовать обратная косая черта. Точно так же обратная косая черта предшествует символу, используемому в качестве ограничителя для команды m/.../, s/.../.../ или tr/.../.../, если он встречается внутри шаблона и не должен рассматриваться как ограничитель. Рассмотрим пример:

```
$line = ".Hello!";
if ($1ine =- m/\./) {
```

```
  print "Shouldn't start a sentence with a perlod!\n";
}
Shouldn't start a sentence with a perlod!
```

Если нужно найти самый короткий текстовый фрагмент /QQ(.*?)FF/ в "QQ ff QQ ff FF", однако оно найдет "ff QQ ff". Шаблон всегда находит левую строку минимальной длины, которая соответствует всему шаблону, т.е. это вся строка в этом примере. Для правильного шаблона нужно воспользоваться логическими операторами в регулярных выражениях: /QQ((?:(?!QQ).)*)FF/, т.е. сначала QQ, потом не QQ, потом FF.

Конструкции (?<=шаблон) и (?<!шаблон) работают только с шаблонами, соответствующими фиксированному числу символов. Иными словами, в шаблонах, указываемых для (?<=...) и (?<!...), не должно быть квантификаторов.

Эти условия полезны, если нужно проверить, что перед определенным фрагментом текста или после него находится нужная строка, однако ее не требуется включать в результат поиска. Это бывает необходимо, если в коде используются спе-циальные переменные $& (фрагмент, для которого найдено соответствие между текстом и регулярным выражением), $` (текст, предшествующий найденному фрагменту) и $' (текст, следующий за найденным фрагментом). Более гибким представляется применение нумерованных переменных $1, $2, $3, ... в которые заносятся отдельные части найденного фрагмента.

В следующем примере ищется слово, за которым следует пробел, но сам пробел не включается в результат поиска:

```
$text = "Mary Tom Frank ";
while ($text =~ /\w+(?=\s)/g) {print $& . "\n";}
Mary
Tom
Frank
```

Того же результата можно добиться, если заключить в круглые скобки интересу-ющую нас часть шаблона и затем использовать ее как переменную $1:

```
$text = "Mary Tom Frank ";
while ($text =~ /(\w+)\s/g) {
  print $1 . "\n";
}
Mary
Tom
Frank
```

Следует четко понимать, что вы имеете в виду, когда используете то или иное условие. Рассмотрим следующий пример:

```
$text="Mary+Tom";
if($text=~m|(?!Mary\+)Tom|){
  print "Tom is without Mary!\n";
```

```
}
else{
  print "Tom is busy...\n";
}
```

Вопреки нашим ожиданиям, perl напечатает: `Tom is without Mary!` Это произойдет по следующей причине. Пробуя различные начальные точки входной строки, от которой начинается сопоставление шаблона и текста, per1 рано или поздно доберется до позиции, расположенной прямо перед именем "`Tom`". Условие `(?!Mary\+)` требует, чтобы после текущей точки не находился текст *`Mary+`", и это условие для рассматриваемой точки будет выполнено. Далее, perl последовательно проверяет, что после текущей точки следуют буквы "T", "o" и "m", и это требование также в силе (после проверки условия `(?!Mary\+)` текущая точка остается на месте). Тем самым найдено соответствие между подстрокой "Tom" и шаблоном, поэтому команда поиска возвращает значение истина.

Регулярное выражение `(?!Mary\+)....Tom`, резервирующее четыре символа под текст "`Mary+`", для приведенного выше случая выведет то, что требовалось, но выдаст ошибочный ответ, если перед именем "Tom" нет четырех символов:

```
$text="O, Tom! ";
if($text =~ m|(?!Mary\+)....Tom|){
  print "Tom is without Mary!\n";
}
else{
  print "Tom is busy...\n";
}

Tom is busy...
```

Наконец, если более точно сформулировать, чего требуется, получится нужный результат:

```
$text="Mary+Tom";
if($text=~m|(?<!Mary\+)Tom|){
  print "Tom is without Mary!\n";
}
else{
  print "Tom is busy...\n";
}

Tom is busy...
```

Вспомнить и написать про строчку вида

```
push @mass, $li unless($li=~m/(([2 .. 12]).*?1995)|(([6 .. 12]).*?2001)|/)
```

; perldoc perlop [0-9.]

Модификаторы команд m/.../ и s/.../.../

В perl имеется несколько модификаторов, используемых с командами m/.../ и s/.../.../:

- i - игнорирует различие между заглавными и строчными буквами.
- s - метасимволу "точка" разрешено соответствовать символам \n.
- m - разрешает метасимволам ^ и $ привязываться к промежуточным символам \n, имеющимся в тексте. Не влияет на работу метасимволов \A, \Z и \z.
- x - игнорирует "пробельные символы" в шаблоне (имеются в виду "истинные" пробелы, а не метасимволы \s и пробелы, созданные через escape-последовательности). Разрешает использовать внутри шаблона комментарии.
- g - выполняет глобальный поиск и глобальную замену.
- c - после того как в скалярном контексте при поиске с модификатором g не удалось найти очередное совпадение, не позволяет сбрасывать текущую позицию поиска. Работает только для команды m/.../ и только вместе с модификатором g.
- o - запрещает повторную компиляцию шаблона при каждом обращении к данному оператору поиска или замены, пользователь, однако, должен гарантировать, что шаблон не меняется между вызовами данного фрагмента кода.
- e - показывает, что правый аргумент команды s/.../.../ - это фрагменты выполняемого кода. В качестве текста для подстановки будет использовано возвращаемое значение - возможно, после процесса интерполяции.
- ee - показывает, что правый аргумент команды s/.../.../ - это строковое выражение, которое надо вычислить и выполнить как фрагмент кода (через функцию eval). В качестве текста для подстановки используется возвращаемое значение - возможно, после процесса интерполяции

Особенности работы команд m/.../ и s/.../.../

До сих пор мы рассматривали регулярные выражения, используемые в качестве шаблонов для команд m/.../ и s/.../.../, и не особо интересовались, как работают эти команды. Настало время восполнить пробелы.

Команда m/.../ ищет текст по заданному шаблону. Ее работа и возвращаемое значение сильно зависят от того, в скалярном или списковом контексте она используется и имеется ли модификатор g (глобальный поиск).

Команда s/.../.../ ищет прототип, соответствующий шаблону, и, если поиск оказывается успешным, заменяет его на новый текст. Без модификатора замена производится только для первого найденного совпадения, с модификатором g выполняются замены для всех, совпадений во входном тексте. Команда возвращает в качестве результата число успешных замен или пустую строку (условие ложь false), если ни одной замены сделано не было. В качестве анализируемого текста используется $_ (режим по умолчанию) или выражение, присоединенное к шаблону с помощью оператора =~ или !~. В случае поиска (команда m/.../) конструкция, расположенная слева от операторов =~ или !~, может и не быть переменной. В случае замены

(команда s/.../.../) в левой части должна стоять скалярная переменная, или элемент массива, или элемент хэша, или же команда присвоения одному из указанных объектов.

Вместо косой черты в качестве ограничителя для аргументов команд m/.../ и s/.../.../ можно использовать любой символ, за исключением "пробельного символа", буквы или цифры. Например, в этом качестве можно использовать символ комментария, который будет работать как ограничитель:

```
$text="ABC-abc";
$text =~ s#B#xxx#ig;
print $text;
AxxxC-axxxc
```

В качестве ограничителей не стоит использовать вопросительный знак и апостроф (одинарную кавычку) - шаблоны, с такими ограничителями обрабатываются специалиным образом. Если команда m/.../ использует символ косой черты в качестве разделителя, то букву m можно опустить:

```
while (defined($text = <>))
{ if ($text =~/^exit$/i) {exit;} }
```

Если в качестве ограничителя для команды m/.../ используется вопросительный знак, то букву m также можно опустить. Однако шаблоны, ограниченные символом ?, в случае поиска работают особым образом (независимо от наличия или отсутствия начальной m). А именно, они ведут себя как триггеры, которые срабатывают один раз и потом выдают состояние ложь (false), пока их не взведут снова, вызвав функцию reset (она очищает статус блокировки сразу всех конструкций ?...?, локальных для данного пакета). Например, следующий фрагмент сценария проверяет, есть ли в файле пустые строки:

```
while (<>)
if (?^$?) {print ."There is an empty line nere.\n";} continue {
reset if eof;   #очистить для следующего файла
}
```

Диагностическое сообщение будет напечатано только один раз, даже если в файле присутствует несколько пустых строк. Команда поиска с вопросительным знаком относится к подозрительным командам, а потому может не войти в новые версии perl. 1 В качестве ограничителей можно также использовать различные (парные) койструкции скобок:

```
while (<>){
  if(m/^quit$/i){exit;}
  if(m(^stop$)i){exit;}
  if(m[^end$]i) {exit;}
  if(m{^bye$}i) {exit;}
  if (!1)<^exit$>i) {exit;}
}
```

В случае команды s/.../.../ и использования скобок как ограничителей для первого аргумента, ограничители второго аргумента могут выбираться независимо:

```
$text =~ "Perl is wonderful";
$text =~ s/is/is very/;
$text =~ s[wonderful]{beautiful};
$text =~ s(\.)/!/;
print $text;
Perl is very beautiful!
```

Предварительная обработка регулярных выражений

Аргументами команд m/.../ и s/.../.../ являются регулярные выражения, которые перед началом работы интерполируются подобно строкам, заключенным в двойные кавычки В отличие от текстовых строк, для шаблона не выполняется интерполяция имен типа $), $| и одиночного $ - perl считает, что такие конструкции соответствуют метасимволу конца строки, а не специальной переменной. Если же в результате интерполяции шаблон поиска оказался пустой строкой, perl использует последний шаблон, который применялся им для поиска или замены.

Если вы не хотите, чтобы perl выполнял интерполяцию регулярного выражения, в качестве ограничителя надо использовать апостроф (одиночную кавычку), тогда шаблон будет вести себя, как текстовая строка, заключенная в апострофы. Однако, например, в случае команды замены s/.../.../ с модификатором e или ee (их работа описывается чуть дальше) для второго аргумента будет выполняться интерполяция даже в том случае, если он заключен в апострофы.

Если вы уверены, что при любом обращениик команде поиска или замены шаблон остается неизменным (например, несмотря на интерполяцию, скалярные переменные внутри шаблона не будут менять своего значения), то можно задать модификатор o. Тогда perl компилирует шаблон в свое внутреннее представление только при первой встрече с данной командой поиска или замены. При остальных обращениях к команде будет использовать откомпилированное значение. Однако, если внезапно изменить значение переменных, задействованных в шаблоне, perl этого даже не заметит.

Команда замены s/.../.../ использует регулярное выражение, указанное в качестве второго аргумента, для замены текста. Поскольку оно обрабатывается (интерполируется) после того, как выполнена очередная операция поиска, в нем можно, в частности, использовать временные переменные, созданные на этапе поиска. В следующем примере мы последовательно заменим местами пары слов, заданных во входном тексте, оставив между ними по одному пробелу:

```
$text = "One   Two   Three   Four  Five   Six";
$text =- s/(\w+)\s*(\w+)/$2$1/g;
Two One Four Three Six Five
```

Однако perl допускает и более сложные способы определения заменяющего текста. Так, если для команды s/.../.../ указать модификатор e, то в качестве второго аргумента надо указать код, который необходимо выполнить (например, вызвать функцию). Полученное выражение будет использованокак текст для подстановки. При этом после вычисления

текстового значения, но пер д его подстановкой будет выполнен процесс интерполяции, аналогичный процессу интерполяции текстовых строк, заключенных в двойные кавычки. Еще более сложная схема реализуется, если задан модификатор ee. В этом слу-чае второй аргумент команды s/.../.../ - это строковое выражение, которое сперва надо вычислить (то есть интерполировать), затем выполнить в качестве кода (вызвав встроенную функцию eval) и только после второй интерполяции полученный результат подставляется вместо найденного текста.

Работа команды m/.../ в режиме однократного поиска В скалярном контексте и без модификатора g команда m/.../ возвращает логическое значение - целое число 1 (истина (true)), если поиск оказался успешным, и пустую строку "" (ложь (false)), если нужный фрагмент текста найти не удалось. Если внутри шаблона имеются группы элементов, заключенные в круглые скобки, то после операции поиска создаются нумерованные переменные $1, $2, ..., в которых содержится текст, соответствующий круглым скобкам. В частности, если весь шаблон заключить в круглые скобки, то в случае успешного поиска переменная $1 будет содержать текст, соотнесенный с шаблоном. После успешного поиска можно также использовать специальные переменные $&, $', $' и $+

```
$text = "---one---two---three---";
$scalar = ($text =' m/(\w+)/);
print "Result: $scalar ($1).";
Result: 1 (one).
```

Если вы используете команду m/.../ в списковом контексте, то возвращаемое значение сильно зависит от того, есть ли группы из круглых скобок в вашем шаблоне. Если они есть (то есть если создаются нумерованные переменные), то после успешного поиска в качестве результата будет получен список, составленный из нумерованных переменных ($1, $2,...):

```
$text = "---one,   two,   three---";
 array = ($text ='m/(\w+),\s+(\w+),\s+(\w+)/);
print join "=",  array;
one=two=three.
```

В отличие от ранних версий, perl 5 присваивает значения нумерованным переменным, даже если команда поиска работает в списковом контексте:

```
$text = "---one,   two,   three--- ";
($Fa, $Fb, $Fc) = ($text=-m/(\w+),\s+(\w+),\s+(\w+)/);
print "/$Fa/$Fb/$Fc/\n";
print "$1=$2=$3.\n";
/one/two/three/
one=two::three.
```

Если же в шаблоне нет групп, выделенных круглыми скобками, то в случае успешного поиска возвращается список, состоящий из одного элемента - числа 1. При неудачном поиске независимо от того, были ли в шаблоне круглые скобки, возвращается пустой список:

```
$text = "---one,   two,   three--- ";
@array = ($text=~ m/z\w+/);
print "Result: /", @array, "/\n";
print "Size: ", $#array+1, ".\n";
Result://
Size: 0.
```

Обратите внимание на разницу между пустым и неопределенным списками.

Работа команды m/.../ в режиме глобального поиска

Команда m/.../ работает иначе, если указан модификатор g, задающий глобальный поиск всех вхождений шаблона по всему тексту. Если оператор используется в списковом контексте и в шаблоне есть группы круглых скобок, то в случае удачного поиска возвращается список, состоящий из всех найденных групп, расположенных друг за другом:

```
$text = "---one---two-~-three---";
@array = ($text =~m/(-(\w+))/);
print "Single: [", join(", ",  array),"].\n";
@array = ($text =~m/(-(\w+))/g);
print "Global: [", join(", ",  array),"].\n";
Single: [-one, one].
Global: [-one, one, -two, two, -three, three].
```

Если же в шаблоне нет групп круглых скобок, то оператор поиска возвращает список всех найденных прототипов шаблона, то есть ведет себя так, как если бы весь шаблон был заключен в круглые скобки:

```
$text = "---one---two---three--";
@array = ($text =~m/\w+/);
print "Result: (", join(", ", @array), ").\n";
Result: (one, two, three).
```

В случае неудачного поиска, как и в предыдущих вариантах, возвращается пустой список. В скалярном контексте и с модификатором g команда m/.../ ведет себя сивершенно особым образом. Специальная переменная $_ или переменная, стоящая слева от оператора =~ или !~, при поиске с модификатором g получает дополнительные свойства - в нее записывается последнее состояние. При каждом последующем обращении к данному фрагменту кода поиск будет продолжаться с того места, на котором он остановился в последний раз. Например, следующая команда подсчитывает количество букв x в заданной строке текста:

```
$text = "Here is texxxxxt.";
$counter = 0;
while ($text =~ m/x/g){
print "Found another x.\n";
$conter++;
print "Total amount = $counter.\n";
```

```
Found another x.
Found another x.
Found another x.
Found another x.
Found another x.
Total amount = 5.
```

Состояние (точнее, позиция) поиска сохраняется даже в случае перехода к следующему оператору поиска, имеющему модификатор `g`. Неудачный поиск сбрасывает значение в исходное состояние, если только для команды `m/.../` не указан модификатор `c` (то есть команда должна иметь вид `m/.../gc`). Изменение текстового буфера, для которого выполняется поиск, также сбрасывает позицию поиска в исходное состояние. В следующем примере из текстовой строки последовательно извлекаются и выводятся пары имя/значение до тех пор, пока строка не закончится:

```
$text = "X=5; z117e=3.14l6; temp=lQ24;";
$docycle = 1; $counter = 0;
while ($docycle) {
  undef $name; undef $value;
  if ($text =~ m/(\w+)\s*=\s*/g) {$name = $1;}
  if ($text =~ m/([\d\.\*\-]*)\s*;/g) {$value = $1;}
  if (defined($name) and defined($value)) {
    print "Name=$name, Value=$value.\n";
    $counter++,
  }else{
    $docycle = 0;
  }
}
print "I have found $conter values.\n";
Name=X, Value=5.
Name=z117e, Value=3.1416.
Name=temp, Value=1024.
I have found 3 values.
```

Позиция, на которой остановился поиск, может быть прочитана и даже переустановлена с помощью встроенной функции perl `pos`. В шаблоне на текущую позицию поиска можно ссылаться с помощью метасимвола `\G`. В следующем примере из строки последовательно извлекаются буквы `p`, `o` и `q` и выводится текущая позиция поиска:

```
$index = 0;
$_ = "ppooqppqq";
while ($index++ < 2) {
print "1: '";
print $1 while /(o)/gc; print "', pos=", pos, "\n";
print "2: '";
print $1 if /\G(q)/gc; print "', pos=';' pos, "\n";
print "3: '";
```

```
print while /(p)/gc; print "', pos=",pos, "\n";
}

1: 'oo', pos=4;
2: 'q', pos=7;
3: 'pp', pos=4;
1: '', pos=7;
2: 'q', pos=8;
3: '', pos=8;
```

В документации perl приводится основанный на этом механизме интересный пример последовательного лексического разбора текста. В нем каждая последующая команда поиска очередной лексической единицы начинает выполнятьсяс того места, где завершила свою работу предыдущая. Советую внимательно разобраться с этим примером (страница руководства perlop, раздел "Regexp Quote-Uke Operators", описание команды m/PATTERN/), если вы хотите расширить доступный вам инструментарий perl!

Замена строк с помощью команды tr/.../.../

Кроме команд m/.../ и s/.../.../ строки можно обрабатывать с помощью команды tr/.../.../ (она же - команда y/.../.../):

```
tr/список1/список2/модификаторы;
y/список1/список2/модификаторы;
```

В отличие от m/.../ и s/.../.../, эта команда не использует шаблоны и регулярные выражения, а выполняет посимвольную замену, подставляя в текст вместо литер из первого списка соответствующие им литеры из второго списка. Например, в следующем случае производится замена литер "i" на "o": $text = "My name is Tim."; $text =~ tr/i/o/; print $text; My name is Tom.

В качестве списков используются идущие друг за другом символы, не разделяемые запятыми (то есть это скорее строки, чем списки). В отличие от шаблонов команд m/.../ и s/.../.../, аргументы команды tr/.../.../ не интерполируются (то есть подстановки значений вместо имен переменных не происходит), хотя escape-последовательности, указанные внутри аргументов, обрабатываются правильно. Подобно m/.../ и s/.../.../, команда tr/.../.../ по умолчанию работает с переменной $_:

```
while (<>){
tr/iI/jJ/;
print;
```

В качестве списков можно указывать диапазоны символов - как, например в следующем фрагменте кода, заменяющем строчные буквы на заглавные: $text = "Here is the text."; $text =~ tr/a-z/A-Z/; print $text; HERE IS THE TEXT.

Как и в случае m/.../ и s/.../.../, команда tr/.../.../ не требует использовать именно знаки косой черты в качестве ограничителей. Можно использовать практически любой символ, отличный от "пробельных", букв и цифр, а также парные скобочные конструкции.

Команда tr/.../.../ возвращает число успешных замен. В частности, если не было сделано никаких замен, она возвращает число ноль. Это позволяет, например, подсчитать с помощью команды tr/.../.../ количество вхождений буквы x в строку $text, не меняя содержимого этой переменной: $text = "Here is the text."; $xcount = ($text =~tr/x/x/); print $xcount; 1 Если у команды tr/.../.../ нет модификаторов (см. далее раздел "Модификаторы команды tr/.../.../"), то ее аргументы при обычных условиях должны быть одинаковой длины. Если второй аргумент длиннее первого, то он усекается до длины первого аргумента - так, команда tr/abc/0-9/ эквивалентна команде tr/abc/012/. Если первый аргумент длиннее второго и второй не пуст, то для второго аргумента необходимое число раз повторяется его последний символ - так, команда tr/0-9/abc/ эквивалентна команде tr/0123456789/abccccccccc/. Если же второй, аргумент пуст, то команда tr/.../.../ подставляет вместо него первый аргумент.

Как легко заметить, если второй аргумент пуст, то (при отсутствии модификаторов) команда tr/.../.../ не производит никаких действий, а возвращаемое ею значение равно числу совпадений между первым аргументом и обрабатываемым текстом. Например, следующая команда подсчитывает количество цифр в строке:

```
$text = "Pi=3.1415926536, e=2.7182";
$digit_counter=($text =~ tr/0-9//);
print $digit_counter;
16
```

Команда tr/.../.../ работает без рекурсии, просто последовательно заменяет символы входного текста. Например, для замены заглавных букв на строчные, и на-оборот, достаточно выполнить команду:

```
$text = "MS Windows 95/98/NT";
$text =" tr/A-Za-z/a-zA-Z/;
print $text;
ms WINDOWS 95/98/nt
```

Если в списке, указанном в качестве первого аргумента, есть повторяющиеся символы, то для замены используется первое вхождение символа:

```
$text = "Billy Gates";
$text =~ tr/ttt/mvd/;
print $text;
Billy Games
```

Модификаторы команды tr/.../.../

Команда tr/.../.../ допускает использование следующих модификаторов:

- d - удаляет непарные символы, не выравнивая аргументы по длине.

- c - в качестве первого аргумента использует полный список из 256 символов за вычетом указанных в списке символов.
- s - удаляет образовавшиеся в результате замены повторяющиеся символы.

Если указан модификатор d, а первый аргумент команды длиннее второго, то все символы из первого списка, не имеющие соответствия со вторым списком, удаляются из обрабатываемого текста. Пример: удаляем строчные латинские буквы и заменяем пробелы на слэши:

```
$text = "Here is the text.";
$text =~ tr[ a-z][/]d;
print $text;
H///.
```

Наличие модификатора d - единственный случай, когда первый и второй аргументы не выравниваются друг относительно друга, В остальных вариантах второй аргумент либо усекается, либо последний символ в нем повторяется до тех пор, пока аргументы не сравняются, либо, если второй аргумент пуст, вместо Второго аргумента берется копия первого.

Если указан модификатор c, то в качестве первого аргумента рассматриваются все символы, кроме указанных. Например, заменим на звездочки все символы, кроме строчных латинских букв:

```
$text = "Here is the text,";
$text =' tr/a-z/*/c;
print $text;
*ere*is*the*text*
```

Если указан модификатор s, то в случае если замещаемые символы образуют цепочки из одинаковых символов, они сокращаются до одного. Например, заменим слова, состоящие из латинских букв, на однократные символы косой черты:

```
$text = "Here is the text.";
$text ="tr(A-Za-z)(/)s;
print $text;
/ / / /.
```

Без модификатора s результат был бы другим:

```
$text = "Here is the text.";
$text =' tr(A-Za-z)(/);
print $text;
//// // /// ////.
```

Примеры:
1. Заменить множественные пробелы и нетекстовые символы на одиночные пробелы:

```
$text = "Here   is   the   text."
$text =~ tr[\000-\040\177\377][\040]s;
```

```
print $text;
Here is the text.
```

2. Сократить удвоенные, утроенные и т.д. буквы;

```
$text = "Here is the texxxxxxt.";
$text =~ tr/a-zA-Z/s;
print $text;
Here is the text.
```

3. Пересчитать количество небуквенных символов:

```
$xcount=($text =~ tr/A-Za-z//c);
```

4. Обнулить восьмой бит символов, удалить нетекстовые символы:

```
$text =- tr{\200-\377}{\000-\l77};
$text =~ tr[\000-\037\177][]d;
```

5. Заменить нетекстовые и 8-битные символы на одиночный пробел:

```
$text =~ tr/\021-\176/ /cs;
```

Поиск отдельных слов

Чтобы выделить слово, можно использовать метасимвол \S соответствующий символам, отличным от "пробельных":

```
$text = "Now is the time.";
$text =- /(\S+)/;
print $1;
Now
```

Однако метасимвол \S соответствует также и символам, обычно не используемым для идентификаторов. Чтобы отобрать слова, составленные из латинских букв, цифр и символов подчеркивания, нужно использовать метасимвол \w:

```
$text = "Now is the time.";
$text =~ /(\w+)/;
print $1;
Now
```

Если требуется включить в поиск только латинские буквы, надо использовать класс символов:

```
$text = "Now is the time.";
$text =~ /([A-Za-z]+)/;
print $1;
Now
```

Более безопасный метод состоит в том, чтобы включить в шаблон мнимые символы границы слова:

```
$text = "How is the time.";
$text=~/\b([A-Za-z]+)\b/;
print $1;
Now
```

Привязка к началу строки

Началу строки соответствует метасимвол (мнимый символ) ^. Чтобы шаблон к началу строки, надо задать этот символ в начале регулярного выражения. Например, вот так можно проверить, что текст не начинается с точки:

```
$line = ".Hello!";
if($line=~m/^\./){
  print "Shouldn't start a sentence with a period!\n";
}
Shouldn't start a sentence with a period!
```

Чтобы точка, указанная в шаблоне, не интерпретировалась как метасимвол перед ней пришлось поставить обратную косую черту.

Привязка к концу строки

Чтобы привязать шаблон к концу строки, используется метасимвол (мнимый символ) $. В нашем примере мы используем привязку шаблона к началу и к концу строки, чтобы убедиться, что пользователь ввел только слово "exit":

```
while(<>){
  if(m/"exlt$/) {exit;}
}
```

Поиск чисел

Для проверки того, действительно ли пользователь ввел число, можно использо-вать метасимволы \d и \D. Метасимвол \D соответствует любому символу, кроме цифр. Например, следующий код проверяет, действительно ли введенный текст представляет собой целое значение без знака и паразитных пробелов:

```
$test = "Hello!";
if($text =~ /\D/){
print "It is not a number.\n";
}
It is not a number.
```

То же самое можно проделать, использовав метасимвол \d:

```
$text = "333";
if($text =~ /^\d+$/){
print "It is a number.\n";
}
It is a number.
```

Вы можете потребовать, чтобы число соответствовало привычному формату. То есть число может содержать десятичную точку, перед которой стоит по краййей мере одна цифра и, возможно, какие-то цифры после нее:

```
$text= "3,1415926";
if($text =~ /^(\d+\.\d*|\d+)$/){
print "It is a number.\n";
}
It is a number.
```

Кроме того, при проверке можно учитывать тот факт, что перед числом может стоять как плюс, так и минус (или пустое место):

```
$text = "-2.7182";
if ($text =~ /^([+-]*\d+)(\.\d*|)$/) {
print "It is a number.\n";
```

Поскольку плюс является метасимволом, его надо защищать обратной косой чертой. Однако внутри квадратных скобок, то есть класса символов, он не может быть квантификаторам. Знак "минус" внутри класса символов обычно играет роль оператора диапазона и поэтому должен защищаться обратной косой чертой. Однако в начале или в конце шаблона он никак не может обозначать диапазон, и поэтому обратная косая черта необязательна. Наконец, более строгая проверка, требует, чтобы знак, если он присутствует, был только один:

```
$text = "+0.1428571428571428571";
if ($text =~ /^(+|-|)\d+(\.\d*\)$/) {
  print "It is a number.\n";
}
It is a number.
```

Альтернативные шаблоны, если они присутствуют, проверяются слева направо. Перебор вариантов обрывается, как только найдено соответствие между текстом и шаблоном. Поэтому, например, порядок альтернатив в шаблоне (\.\d*|) мог бы стать критичным, если бы не привязка к концу строки. Наконец, вот как можно произвести проверку того, что текст является шестна-дцатеричным числом без знака и остальных атрибутов:

```
$text = "1AO";
unless (ftext =~ m/^[a-fA-F\d]+$/) {
print "It is not a hex number, \n";
}
```

Проверка идентификаторов

С помощью метасимвола \w можно проверить, состоит ли текст только из букв, цифр и символов подчеркивания (это те символы, которые perl называет словесными (word characters)):

```
$text="abc";
if($text=~/^\w+$/){
```

```
  print "Only word characters found. \n";
}
Only word characters found.
```

Однако, если вы хотите убедиться, что текст содержит латинские буквы и несодержит цифр или символов подчеркивания, придется использовать другой шаблон:

```
$text = "abc";
if($text=~ /^[A-Za-z]+$/)
{ print "Only letter characters found.\n";}
Qnly letter characters found.
```

Наконец, для проверки, что текст является идентификатором, то есть начинается с буквы и содержит буквы, цифры и символы подчеркивания, можно испольровать команду:

```
$text = "X125c";
if($text=~ /^[A-Za-z]\w+$/)
{ print "This is identifier.\n";}
This is identifier.
```

Как найти множественные совпадения

Для поиска нескольких вхождений шаблона можно использовать модификатор `g`. Следующий пример, который мы уже видели ранее, использует команду `m/.../` с модификатором `g` для поиска всех входжений буквы `x` в тексте:

```
$text="Here is texxxxxt";
while($text=~m/x/g){
  print "Found another x.\n";
}
Found another x.
Found another x.
Found another x.
Found another x.
Found another x.
```

Модификатор `g` делает поиск глобальным. В данном (скалярном) контексте perl помнит, где он остановился в строке при предыдущем поиске. Следующий поиск продолжается с отложенной точки. Без модификатора `g` команда `m/.../` будет упорно находить первое вхождение буквы `x`, и цикл будет продолжаться бесконечно.

В отличие от команды `m/.../` команда `s/.../.../` с модификатором `g` выполняет глобальную замену за один раз, работая так, будто внутри нее уже имеется встроенный цикл поиска, подобный приведенному выше. Следующий пример за один раз заменяет все вхождения `x` на `z`:

```
$text = "Here is texxxxxt.";
$text =~ s/x/z/g;
```

```
print $text;
Here is tezzzzzt.
```

Без модификатора g команда s/.../.../ заменит только первую букву x. Команда s/.../.../ возвращает в качестве значения число сделанных подстановок, что может оказаться полезным:

```
$text= "Here is texxxxxt.";
print (text =~ s/x/z/g)
5
```

Поиск нечувствительных к регистру совпадений

Вы можете использовать модификатор i, чтобы сделать поиск нечувствительным к разнице между заглавными и строчными буквами. В следующем примере про-грамма повторяет на экране введенный пользователем текст до тех пор, пока не будет введено Q, или q (сокращение для QUIT или quit), после чего программа прекращает работу:

```
while(<>){
  chomp;
  unless (/^q$/i){
    print
  }
  else {
    exit;
  }
}
```

Выделение подстроки

Чтобы получить найденную подстроку текста, можно использовать круглые скобки в теле шаблона. Если это более удобно, можно также использовать встроенную функцию substr. В следующем примере мы вырезаем из текстовой строки нужный нам тип изделия:

```
$record = "Product number:12345
           Product type: printer
                  Product price: $325";
if($record=~/Product type:\s*([a-z]+)/i){
  print "The product's type Is^$1.\n";
}
product's type is printer.
```

Вызов функций и вычисление выражений при подстановке текста

Используя для команды s/.../.../ модификатор e, вы тем самым показываете, что правый операнд (то есть подставляемый текст) - это то выражение perl, которое надо вычислить. Например, с помощью встроенной функции perl uc (uppercase) можно заменить все строчные буквы слов строки на заглавные:

```
$text = "Now is the time.";
$text=~ s/(\w+)/uc($1)/ge;
```

```
print $text;
NOW IS THE TIME.
```

Вместо функции uc($l) можно поместить произвольный код, включая вызовы программ.

Поиск n-го совпадения

С помощью модификатора g перебираются все вхождения заданного шаблона. Но то делать, если нужна вполне определенная точка совпадения с шаблоном, например, вторая или третья? Оператор цикла while в сочетании с круглыми скобками, выделяющими нужный образец, поможет вам:

```
$text = "Name:Anne Nanie:Burkart Name:Glaire Name: Dan";
while ($text =~ /Name: \s*(\w+)/g){
  ++$match;
  print "Match number $match is $1.\n";
}

Match number 1 is Anne
Match number 2 is Burkart
Match number 3 is Claire
Match number 4 is Dan
```

Этот пример можно переписать, используя цикл for:

```
$text = "Name:Anne Name:Burkart Name:Ciaire Name:Dan";
for ($match = 0;
     $text =~ /Name:\s*(\w+)/g;
         print "Match number ${\match} is $1.\n")
{}
Match nuwber 1 Is Anne
Match number 2 is Burkart
Match number 3 is Claire
Match number 4 is Dan
```

Если же вам требуется определить нужное совпадение не по номеру, а по содержанию (например, по первой букве имени пользователя), то вместо счетчика $match можно анализировать содержимое переменной $1, обновляемой при каждом найденном совпадении. Когда требуется не найти, а заменить второе или третье вхождение текста, можно применить ту же схему, использовав в качестве тела цикла выражение perl, вызываемое для вычисления заменяющей строки:

```
$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";
$match =0;
$text =~ s/(Name:\s*(\w+))/   # начинается код perl
         if (++$match == 2)   # увеличить счетчик
                   {"Name:John ($2)"}# вернуть новое значение
         else {$1}            # оставить старое значение
         /gex;
```

```
print $text;
Name:Anne Name:John (Burkart) Name:ClaireName:Dan
```

В процессе глобального поиска при каждом найденном совпадении вычисляется выражение, указанное в качестве второго операнда. При его вычислении увеличивается значение счетчика, и в зависимости от него в качестве замены подставляется либо старое значение текста, либо новое. Модификатор `x` позволяет добавить в поле шаблона комментарии, делая код более прозрачным. Обратите внимание, что нам пришлось заключить весь шаблон в круглые скобки, чтобы получить значение найденного текста и подставить его на прежнее место полностью.

Как ограничить "жадность" квантификаторов

По умолчанию квантификаторы ведут себя как "жадные" объекты. Начиная с текущей позиции поиска, они захватывают самую длинную строку, которой может соответствовать регулярное выражение, стоящее перед квантификатором. Алгоритм перебора с возвратами, используемый perl, способен ограничивать аппетит квантификаторов, возвращаясь назад и уменьшая длину захваченной строки, если не удалось найти соответствия между текстом и шаблоном. Однако этот механизм не всегда работает так, как хотелось бы. Рассмотрим следующий пример. Мы хотим заменить текст "That is" текстом "That's". Однако в силу "жадности" квантификатора регулярное выражение "`.*is`" сопоставляется фрагменту текста от начала строки и до последнего найденного "is":

```
$text = "That is some text, isn't it?";
$text =~ s/.*is/That's/;
print $texts;
That'sn't it?
```

Чтобы сделать квантификаторы не столь жадными, а именно заставить их захватывать минимальную строку, с которой сопоставимо регулярное выражение, после квантификатора нужно поставить вопросительный знак. Тем самым квантификаторы принимают следующий вид:

- `*?` - ноль или несколько совпадений,
- `+?` - одно или несколько совпадений,
- `??` - ноль совпадений или одно совпадение,
- `{n}?` - ровно `n` совпадений,
- `{n,}?` - по крайней мере `n` совпадений,
- `{n,m}?` - совпадений по крайней мере `n`, но не более, чем `m`.

Оратите внимание, что смыслквантификатора от этого не меняется; меняется только поведение алгоритма поиска. Если в процессе сопоставления шаблона и текста прототип определяется однозначно, то алгоритм поиска с возвратами увеличит "жадность" такого квантификатора точно так же, как он ограничивает аппетит собрата. Однако если выбор неоднозначен, то результат поиска будет другим:

```
$text = "That is some text, isn't it?";
$text =~ s/.*?is/That's/;
print $texts;
That's some text, isn't it?
```

Как удалить ведущие и завершающие пробелы

Чтобы отсечь от строки начальные "пробельные символы", можно использовать, следующую команду:

```
$text = "          Now is the time.";
$text =~ s/^\s+//;
print $texts;
Now is the time.
```

Чтобы отсечь "хвостовые" пробелы, годится команда:

```
$text = "Now is the time.          ";
$text =~ s/\s+$//;
print $texts;
Now is the time.
```

Чтобы отсечь и начальные, и хвостовые пробелы лучше вызвать последовательно эти две команды, чем использовать шаблон, делающий отсечение ненужных пробелов за один раз. Поскольку процедура сопоставления шаблона и текста достаточно сложна, на эту простую операцию может уйти гораздо больше времеви, чем хотелось бы.

Например в тексте нужно найти текст, находящийся между открывающим и закрывающим тегом:

```
$text="<a>blah-blah</a>";
if($text=~m!<([a|b])>(.*?)/\1!ig){
print "$2\n";
}
```

найдет все слова, стоящие между тегами <a> и .

В регулярных выражениях пристутствует своя семантика: быстрота, торопливость и возврат. Если квантификатор * совпадает во многих случаях, то в результате быдет выведен наибольший по длинне результат. Это жадность. Быстрота: поиск старается найти как можно быстрее. "Text"=~/m*/, по смыслу символов m нет, но в результате будет возвращено значение 0. Т.е. формально 0 и более символов.

```
$test="aaooee ooaao";
$test=~s/o*/e/;
print $test;
eaaooee ooaao
```

потому что 1 элемент сторки - 0 и более символов.

Если добавить квантификатор g, то результат будет таким:

```
eaeaeeeeee eeaeaee
```

т.к строка содержит 13 мест, где может встречатся o, в том числе и пустых.

Модификаторы:

- /i игнорировать регистр
- /x игнорировать пропуски в шаблоне и разрешить комментарии.
- /g модификатор разрешающий выполнение поиска/замены везде, где это возможно
- /gc не сбрасывается позиция при неудачном поиске.
- /s разрешается совпрадение . с \n, игнорируется $*.
- /m разрешить совпадение ^ и $ для начала и конца строки во внутренних переводах строк
- /o однократная компиляция
- /e правая часть s/// представляет собой выполняемый код
- /ee правая часть s/// выполняется, после чего возвращаемое значение интерпретируется снова.

при вызове use locale учитываются локальные настройки. Модификатор /g может заполнить массив значений @nums = m/(\d+)/g; но это сработает для ненакладывающихся совпадений. Чтобы поймать совпадения нужно воспользоваться оператором ?=... Если ширина = 0, то механизм поиска остался на прежнем месте. Найденые данные остаются внутри скобок. Если есть модификатор /g, то текущая позиция остается прежней, но происходит перемещение на один символ вперед. $numbers="123456789"; @one=$numbers=~/(\d\d\d)/g; @two=$numbers=~/(?=(\d\d\d))/g; print "@one \n"; print "@two \n";

Модификаторы m и s нужны для поиска последовательностей символов, содержащих перевод строки. При s точка совпадает с \n и игнорируется $*. m делает совпадающими ^ и $ до и после \n. e правая часть выполняется как программный код: perl -i -n -p -e 's/(.)/lc($1)/g' *.html приводит все литеры во всех файлах *.html текущей директории к нижнему регистру.

Встроенные переменные в regex.

$1, $2, $3, $4, ..., $n ... содержат ссылки на найденный текст, только в том случае если regex был в круглых скобках:

```
s%<f(.*?)><(.*?)"><(.*?)">%$1 $2 $3%g;
```

внутри regex можно использовать переменные типа \1, \2, \3, \4, ... \n, ...

```
s/a href=(["'])(.*?)\1>/$2/g
```

найдет все урл, заключенные в двойные, одинарные и вообще без кавычек, находящиеся в документе.

для /(a.*b)|(mumu)/ в переменной $+ содержится $1 или $2.

$& содержит полный текст совпадения при последнем поиске.

$' и $` содержатся строки до и после совпадения

Если нужно скопировать и сделать подстановку, то нужно действовать примерно так:

```
($at = $bt) =~ s!m(.*?)o!! #для строк
for(@mass1 = @mass2){s/umka/maugli/} #для массивов


$u = ($m=~s/a/b/g); #поменять $m и занести в $u число замен.
```

Если нужно выцепить только алфавитные символы, с учетом настроек locale, то регексп примерно такой: `/^[^\W\d_]+$/` в нем учитываются все не алфавитные символы, не цифры и не подчеркивания(для случая "ванька-встанька"), симвлол отрицания в группе `[]` - `^`, т.е. найти все, что не `[\W\d_]`, можно было написать и скажем так `!~m/^(\W|\d|_)*/`.

Для упрощения понимания сложных регулярных выражений можно воспользоваться их комментированием. Иногда правда можно только по виду регулярного выражения определить зачем оно предназначено:

```
$mmm{$1} = $2 while ($nnn =~ /^([^:]+):\s+(.*)$/m);
```

читаем регулярное выражение:

нужно найти в файле все что до двоеточия не двоеточие и все что после двоеточия(включая возможные повторения после первого **: .*?: .*?: .*?:**, потому что была найдена первая позиция: выделить все что не есть двоеточие до первого двоеточия)

Что это может быть, вполне вероятно, что оно нужно для составления статистики писем, выцепление заголовка письма и его названия из mbox в хеш. По крайней мере это регулярное выражение подходит для данной задачи.

Рабочие программы, использующие регулярные выражения

В принципе регулярные выражениея это вовсе не вещь в себе, хотя иногда и может встретится задача, фактически полностью реализуемая при помощи regex. Ниже приведены программы, иллюстрирующие использование реглуярных выражений:

Выделение чисел в математической записи

Пример использования логических условий для нахождения любых чисел в том числе и в общепринятой математической записи:

```
#!/usr/bin/perl
$_=qq~
1234
34 -4567
 3456
-0.35e-0,2
    56grf45
```

```
 -.034 E20
      -.034 e2,01   -,045 e-,23
   -,034 e201  3e-.20
 -,045 e-,23 e-0.88

 4 E-0.20
 22
 E-21
  -0.2 w         4 3
 345
 2 ^-,3
 ~;
 print "$1\n" while
  m%(([+-]?(?=\d|[\.,]\d)\d*([\.,]\d*)?((\se|e|\s?\^)
     ([-+]?\d*[,\.]?)\d+)?)|([+-]?e[+-]?\d*[,.]?\d+))%gxi;
```

программа исправно выводит все числа. Разберем регулярное выражение

```
  m%(([+-]?(?=\d|[\.,]\d)\d*([\.,]\d*)?((\se|e|\s?\^)
     ([-+]?\d*[,\.]?)\d+)?)|([+-]?e[+-]?\d*[,.]?\d+))%gxi;
```

в переменной $1 содержится то, что регулярное выражение находит в результате, т.е. m%(...)%gmi. m%((что-то)|([+-]?e[+-]?\d*[,.]?\d+))%gmi нужно для того, чтобы находить числа вида e-20 или E21(так в математике обозначают десятку в какой-то степени, например e-0,20 = $10^{-0,20}$ или E20 = 10^{21}). Рассмотрим левое регулярное выражение "что-то" для чисел вида не e20 или E21:

```
 ([+-]?(?=\d|[\.,]\d)\d*([\.,]\d*)?((\se|e|\s?\^)([-+]?\d*[,\.]?)\d+)?)
```

[+-]? - есть ли в перед числом знак + или -. ? - если вообще есть что-то, находящееся внутри впереди стоящего [...]. Выкинем проверку знака, регексп сократится до

```
 (?=\d|[\.,]\d)\d*([\.,]\d*)?((\se|e|\s?\^)([-+]?\d*[,\.]?)\d+)?
```

рассмотрим regex (?=\d|[\.,]\d)\d* логический оператор (?=B) требует, чтобы перед числов было B. В данном случае B представляет из себя regex \d|[\.,]\d Regex \d|[\.,]\d значит, что перед каждым числом должно быть что-то либо просто число, либо число, перед которым стоит либо запятая, либо точка, т.е. находим все числа вида ,2 .2 или просто числа 2(2 выбрано для примера, может быть и 3). Далее скобка закрывается и идет \d*, т.е. число вида ,2 точно пройдет(например ,2 e-,23 где перед запятой забыли поставить нолики, но мало ли бывает, забыли, надо и это предусмотреть. Вообще когда пишешь программу, надо предполагать, что е использовать будет ленивый склеротический чайник, правда не всегда возможно предугадать что учудит юзер, но к этому надо стремится), а вот число вида ,223 не пройдет. Да и regex (?=\d|[\.,]\d) говорит о том, что нужно найти только одну цифру после запятой. Для остальных цифр и нужен квантификатор \d*, который значит любое количество цифр, в том числе и ноль, т.е. оно работает и для числе вида .2 или ,2 Далее

идет регулярное выражение ([\.,]\d*)? которое говорит о том, есть ли вообще точка и запятая(здесь всю полную строчку в принципе можно усовершенствовать) и число \d*(в том числе и его отсутствие, ведь квантификатор * значит любой символ в том числе и ноль). Отбрасывая все что было выше от этого большого регулярного выражения остается строчка:

```
((\se|e|\s?\^)([-+]?\d*[,\.]?)\d+)?
```

Эта строчка отвечает за поиск в строке $_ математических обозначений степеней типа e201, E,20(число в степени 0,20 например $a^{-0,20}$) и т.д. но только для подстрок вида -,034 e201. Заметьте, что в конце стоит знак вопроса, т.е. если степенное обозначение вообще существует. (\se|e|\s?\^) есть ли числа вида -,034 e201 или -,034e201 и числа в "компьютерной" записи вида 2 ^-,3 = $2^{-0,3}$, т.е. этим регекспом мы разрешили пользователю ставить или не ставить пробел при указании степени и разрешили писать значек ^ с пробелом перед ним(если есть). Далее идет выражение ([-+]?\d*[,\.]?), которое говорит о том, что степень может быть с + или - (типа e,-23 где юзер забыл поставть нолик, а на самом деле хотел написать $a^{-0,23}$). Дальше идет цифра \d* (а может и не идет, т.к. квантификатор то *). Потом идет либо точка либо запятая(причем тут негласно введено ограничение на использование запятой/точки, после e, если степень дробная или вообще есть, точка или запятая должна быть, иными словами не имеет смысла написать -2,34e-,23, хотя юзер на самом деле хотел написать число $-2,34^{-0,23}$). Наконец мы добрались до конца: идет \d+, но тут уж, пользователь, будь добр напиши хотя бы одно число, т.к. квантификатор +, а не * после \d. Т.е. наложили своего рода ограничения здравого смысла, можно просто написать 2, а можно написать и 2e,- что суть бессмысленно. И еще, m%(что-то)%igm стоит квантификатор i, который разрешает e быть и заглавным и квантификатор x, который разрешает разносить регулярное выражение на несколько строк.

Прошу прошения что не ставил иногда знаки препинания, которые есть точка и запятая, тогда Вы бы подумали, что что-то лишнее написно и не подсечено как спецсимвол при помощи бэкслэша \.

Итак, регулярным выражением

```
m%(([+-]?(?=\d|[\.,]\d)\d*([\.,]\d*)?((\se|e|\s?\^)
   ([-+]?\d*[,\.]?)\d+)?)|([+-]?e[+-]?\d*[,.]?\d+))%gxi;
```

были предусмотрены числа степенного порядка, просто числа, числа со знаком, нецелые числа вида ,3(которое есть 0,3 или 0.3), ошибки пользователя при вводе чисел(типа -.034 e2,01 хотя надо бы писать либо -,034 e2,01 либо -.034 e2.01 хотя по смыслу перед точками и запятыми нужно ставить нули, но мы предусмотрели и это) и числа в "компьютерном" представлении.

Конечно, данное регулярное выражение не претендует на абсолютную работу, т.к. оно успешно не работает на подстроках вида -,045 e -,23 e-0.88 считая -,045 отдельным числом, а -,23 возводит в степень e-0.88, хотя по идее должно было бы быть два числа -,045 e -,23 и e-0.88, в таком случае еще одно ограничение пользователю: если хочется, чтобы степенные

числа понимались корректно(для этой программы), то нельзя ставить пробел перед степенью e.

Облегчение поиска работы

Допустим Вы оказались без работы, развалилась ваша фирма или еще какая-нибудь причина. Вам требуется найти новую. Для упрощения этой задачи естьь следующий скрипт, который выцепливает по нужной позиции(веб программирование, зарплата от 200$ и т.д.) с www.job.ru все заявки за последние 10-15 дней, точнее емейлы, куда нужно слать резюме, что значительно убыстряет поиск работы(имея базы адресов легче разослать одно и то-же резюме, используя нехитрый список рассылки):

```
#!/usr/bin/perl -wT
$url0="http://www.job.ru/cgi/list1.cgi?GR_NUM=";
$url1="%31&TOPICID=9&EDUC=2&TP=&Gr=&SEX=&AGEMIN=23&AGEMAX=&MONEY=200&CDT=";
$url2="&LDAY=99&ADDR=%ED%CF%D3%CB%D7%C1&KWORD=&KW_TP=AND";
use LWP::Simple;
foreach($i=1; $i<=57; $i++){#57 число листаемых страниц
$plus.="%31%2B";
$test=$url0.$plus.$url1.$url2,"\n";
@mass=grep{s/(.*) ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})(.*)/$2/ig} split /\n/, get "$test";
$test.=join "\n", @mass;
$test.=\n";
}
@un=grep{!$test{$_}++} split /\n/, $test;
print join "\n", @un;
print "\nВы можете отправлять по вашей специальности $#un резюме\n";
```

Что делает эта программа, она составляет GET запрос из параметров, которые скрыты в hidden полях навигации по результатам запроса на www.job.ru. Программа при помощи Simple.pm отправляет запрос на сервер и как бы листает странички с поиском. Критерий ваших профессиональных навыков составлен в GET-запросе и осталось только разослать почту(для этого можно написать список рассылки) по адресам, которые выдала программа. Разберем регулярное выражение для вытаскивания почтового адреса из текущей странички s/(.*) ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3})(.*)/$2/ig.

[\w+\-\.]\@ - найти все что содержит буквы, тире и точки до символа @, ведь почтовый адрес по спецификации может быть вида aa.ss-ss@chto-to.ru. Тоже самое после символа @ - [\w\-\.]+ далее может быть точка \. и любая буква от 2 до 3 символов \w{2,3}, т.е. окончание, самый верхний домен .com, .ru, .cz и т.д. Далее регулярное выражение состоит из трех классов скобок (.*) - переменная $1, ([\w+\-\.]+\@[\w\-\.]+\.\w{2,3}) переменная $2 и все остальное в (.*) - $3. Пробел перед $2 стоит потому, что так устроен html, отдаваемый пользователю поиском по базе предложений о работе www.job.ru. Нам нужно содержимое $2, в котором находится e-mail работодателя. Пишем его во вторую часть s/наш regex/$2/ig. Квантификатор i нужен для того, чтобы не различать регисты Vasya@pupkin.ru и vasya@pupkin.ru, квантиикатор g задействова на тот случай, если работодатель указывает 2 адреса, по которым нужно высылать резюме. На 23 августа 2001 года на 20 часов 10 минут

прогамма выдала 410 e-mail адресов(пролистав за 3-4 минуты 57 страниц), где вас ждут, как потенциального сотрудника.

Остается написать скрипт почтовой рассылки по e-mails, выданным данным скриптом. Но это в другой главе.

Примером выше был получен спсиок email адресов. Теперь необходимо проверить, действительно ли существуют домены, на которых заведены такие пользщователи(примитивная - но проверка).

```
#!/usr/bin/perl
use Socket;                       #загрузить inet_addr
s{                                #
  (                               #Сохранить имя хоста в $1
    (?:                           #Группирующие скобки
          (?! [-_] )                  #ни подчеркивание, ни дефис
          [\w-] +                     #кусок имени хоста
          \.                          #и точка домена
        )+                            #повторить несколько раз
        [A-Za-z]                      #следующий символ - буква
        [\w-]+                        #домен верхнего уровня
  )                               #конец записи $1
}{                                #Заменить следующим:
  "$1" .                          #исходн часть + пробел
    (($addr = gethostbyname($1)) #Если имеется адрес
        ? "[" . inet_ntoa($addr). "]"#отформатировать
        : "[???]"                     #иначе пометить как сомнительный
        )
}gex
```

Переписываем исходную программу с учетом вышеприведенного кода

```
#!/usr/bin/perl -wT
$url0="http://www.job.ru/cgi/list1.cgi?GR_NUM=";
$url1="%31&TOPICID=9&EDUC=2&TP=&Gr=&SEX=&AGEMIN=23&AGEMAX=&MONEY=200&CDT=";
$url2="&LDAY=99&ADDR=%ED%CF%D3%CB%D7%C1&KWORD=&KW_TP=AND";
use Socket;
use LWP::Simple;
foreach($i=1; $i<=57; $i++){
  $plus.="%31%2B";
  $test=$url0.$plus.$url1.$url2,"\n";
  @mass=grep{s/(.*) ([\w+\-\.]+\@[\w+\-\.]+\.\w{2,3})(.*)/$2/ig} split /\n/, get
"$test";
  $test1.=join "\n", @mass;
  $test1.="\n";
}
@res=split /\n/, $test1;
@un=grep{!$test{$_}++} @res;
```

```
  foreach $file(@un){
    $file=~s/(.*)\@(.*)/www\.$2/;
  =pod
    $file=~s{((?:(?![-_])[\w-]+\.)+[A-Za-z][\w-]+)}
      {"$1".(($addr=gethostbyname($1))?"[".inet_ntoa($addr)."]":"[???]")}gex;
    print $file,"\n" if($file !~/\?\?\?/);
  =cut
  $file=~s{
    (
      (?:
        (?![-_])
        [\w-]+
        \.
      )+
      [A-Za-z]
      [\w_]+
    )
  }{
    "$1".
    (($addr = gethostbyname($1))
     ? "[".inet_ntoa($addr)."]"
     : "[???]"
    )
  }gex;
  print $file,"\n" if($file !~/\?\?\?/);
  }
```

Между строчками можно комментировать целые куски кода.

```
  =pod
    $file=~s{((?:(?![-_])[\w-]+\.)+[A-Za-z][\w-]+)}
      {"$1".(($site=gethostbyname($1))?"[".inet_ntoa($site)."]":"[???]")}gex;
    print $file,"\n" if($file !~/\?\?\?/);
  =cut
```

Эта программа успешно удалила некторые из адресов, которые Socket.pm показались подозрительными. Все-таки какую-никакую, а проверку существования e-mail адресс окольными путями при помощи perl провести можно. Автору сего текста все-таки больше нравится вариант, заключенный в комментарии =pod(.*?)=cut. Он просто короче. Да и если научится читать сложные регулярные выражения, то можно написать полный регексп e-mail адресов, который занимается тем, что выделяет адреса в точности с соответствующим RFC(занимает это регулярное выражение несколько страгниц). Но впрочем ниже будет подглава, посвященная чтению монстрообразных, на первый взгляд, регекспов налету, со множеством примеров, выше же мы уже попытались угадать предназначение регулярного выражения только по его виду.

Ключи, которые использовались в вышеприведенном регулярном выражении

g - глобальная замена
e - выполнение
x - улучшенное форматирование.
Если написать это регулярное выражение в одну строчку, то оно врядли там поместится:

```
s{((?:(?![-_])[\w-]+\.)+[A-Za-z][\w-])}#здесь силовой перевод каретки
{"$1".(($addr=gethostbyname($1))?"[".inet_ntoa($addr)."]":"[???]")}gex
```

Разберем один интересный момент в данном регекспе:

```
s/regex/условие?да:иначе/
```

Тут проявляется пожалуй одна из действительно сильнейших особенностей regex, возможность в одном регулярном выражении избежать многострочных условий с циклом. В приведенном примере работает все примерно так: Если $addr=gethostbyname($1) - да, то ставить ip-адрес(inet_ntoa($addr)), если нет(не откликнулся сервер, сбой на линии и пр) то метить этот урл как подозрительный [???]. В принципе в программе ничего человеку делать не нужно, т.к. подозрительные отметаются условием print $file,"\n" if($file !~/\?\?\?/); Общее время работы программы 10-15 минут.

Очень простое решение для зеркала новостной ленты

Допустим нужно сделать зеркало какой-либо зарубежной новостной ленты вместе с загрузкой картинок с удаленного сервера, чтобы не ждать по несколько минут отображения содержимого полностью загруженной большой таблицы. Приведенный скрипт запускается при помощи crontab каждые 5 часов:

```
#!/usr/bin/perl -w
$/="\001";
print "content-type: text/html\n\n";
$dir="/var/www/docs/html/news/images";
$imgurl="http://www.qwerty.ru/news/images";
use LWP::Simple;
use LWP::UserAgent;
$page=get "http://www.astronomynow.com";
$page=~s/face="(.*?)"//igs;
&getimg($page);
$page=~s!/images/grafix/listdot.gif!../../listdot.gif!igs;
$page=~s!/images/grafix/spacer.gif!../../spacer.gif!igs;
$page=~s!images/grafix/spacer.gif!../../spacer.gif!igs;
if($page=~m!<TABLE WIDTH="400" BORDER="0" CELLPADDING="0" CELLSPACING="0">(.*?)</TD>
</TR></TABLE>!igsm){
  $file=$1;
  &getlink($page);
  foreach $names(@res){
    $names=~s|.*/||ig;
    $file=~s|src="http://(.*?)$names"|src=$imgurl/$names|igs;
```

```
    }
    $html=qq~
    <TABLE BORDER="0" CELLPADDING="0"
    CELLSPACING="0">
    $file
    </TD></TR></TABLE>~;
  }
  open F, ">$dir/news.txt";
  print F $html or die "\n\n\n ERROR: $!\n\n\n";
  close F;
  sub getimg{
    &getlink($_[0]);
    foreach $img(@res){
      my $res = LWP::UserAgent->new->request(new HTTP::Request GET => $img);
      if ($res->is_success) {
        $img=~s|.*/||;
        open (ABC, ">$dir/$img") or die "\n\n\nERROR: $!\n\n\n";
        binmode(ABC);
        print ABC $res->content; close ABC or die "\n\n\nERROR: $!\n\n\n";
      } else {
        print $res->status_line;
      }
    }
    return @res;
  }
  sub getlink{
    local $_=$_[0];
    push(@res, "http://$2")
      while m{SRC\s*=\s*(["'])http://(.*?)\1\s*(.*?)WIDTH="100" HEIGHT="100"(.*?)>}igs;
    return @res;
  }
```

Вывод результатов поиска

Предположим есть необходимость подсветить результаты поиска в файлах, подобно тому как это делает поисковик aport. Данное регулярное выражение позволяет влоб реализовать эту красивую функцию для поисковика. но оно имеет очень большой минус, при обработке текста машина начинает неимоверно подтормаживать, но мы рассмотрим этот регексп из общих соображений:

```
  $sn=4;
  {
    local $_=$description1;
    print "...$1<font color=red>$3</font>$4..."
      while(m/(([\s,\.\n^]*\w*){$sn})(\s*$query\s*)(([\s,\.\n^]*\w+){$sn})/ig);
```

```
}
$_="";
```

Исходная задача состоит в следующем: вывести по 4 слова спереди и сзади результата поиска, причем так, чтобы если слово находится первым, то будет видно 4 слова позади него. В точности такое-же условие и для последнего слова.

Соответственно из вида регекспа понятно, что разделителями слов могут быть символы `[\s,\.\n^]*`, в том числе и символ перевода каретки `^`. Комбинация `(\d\d\d){$sn}` значит что нужно начти 3 цифры три раза.