Версия 2.1.0 - 31.12.2001
Може да изтеглите последната версия от
perlbg.tripod.com/index.html

Начални сведения

Какво трябва да знаете и какво е нужно да имате?

Не ви е нужен опит в програмирането за да започнете изучаването на Perl. Нужно е да разбирате основните компютърни операции. Ако не разбирате какво са директории и файлове, тогава изучаването на езика ще ви е трудно.
За да започнете да програмирате на Perl ви е нужно PC, което може да стартира Win32 операционна система. Това са Windows NT 3.5, 3.51, 4.0 или по-голяма версия, или Windows 95, Windows 98 или Windows 2000. Сигурно няма да имате проблеми ако използвате Windows 3.1 (но не ми се вярва да работите с тази версия). Нужно е да си намерите копие на Perl, и затова може би ви е нужен интернет. Отидете на http://www.activestate.com/ и намерете откъде да си свалите ActivePerl. Това ще е самостоятелен файл с име нещо като - ActivePerl-5.6.1.626-MSWin32-x86-multi-thread.msi. Става въпрос за операционна система Win32 и Intel процесор, не Alpha. 5.6.1 в случая е версията на Perl. Инсталация - следвайте инсталацията на програмата (има се предвид натискайте "Yes" и "Next" навсякъде (казано най-просто)). Може да намерите Perl и без internet, ако си го качите от някое CD.

Не е нужно да имате Win32 PC, може да използвате Perl под други операционни системи като Linux, но тогава информацията, която ще намерите в това ръководство няма да е съвсем вярна и пълна. Не се нуждаете от компилатор. Perl е интерпретаторен език, което означава, че може да стартирате кода директно, не е нужно да го компилирате първо.
След инсталирането,трябва да се направи асоциация с пърловските файлове (за да се отварят автоматично при натискането им). Нужно е, ако при инсталирането Perl не го е направил. Операциите са следните:
Start -> Settings -> Folder Options ->File Types ->New Type
Попълнете полетата:
Descriptions of type: PL file
Associated extension: .pl
Content Type (MIME) : text/plain

Actions: Open
Aplication used to perform action:
c:\perl\bin\perl.exe "%1" %* - или друго място, където сте инсталирали интерпретатора на Perl

Малко повече информация за Perl

Създаден е от Лари Уол. Езика Perl е най-популярния метод за осигуряване на динамика на Web страници. Простото обяснение е, че до последните няколко години на практика всеки съществуващ Web сървър работеше върху UNIX платформа, а Perl е сред най-полезните инструменти на UNIX. И това е една малка част от възможностите за използване на този език. Силата на езика Perl почива основно върху вградените възможности за обработване на текст, чрез създаване на шаблони за търсене и заместване на низове във файлове или цели групи файлове. Друго основно предимство, е че програмите на Perl са със значително по-малък код, отколкото ако ги напишете на друг език. Perl е интерпретаторен език, а не компилаторен като C и C++. Основната разлика, е че при компилирането цялата програма се транслира на машинен език на компютъра, на който ще се изпълнява, от друга програма наречена компилатор. Компилираните файлове се изпълняват самостоятелно. От друга страна интерпретираните програми се транслират в процеса на изпълнението си от програма наречена интерпретатор. Perl програмите не се компилират, поради което ги наричаме скриптове. Когато говорим за интерпретатора на Perl, ще го записваме с малко 'p' ето така - perl.

Как да използваме това ръководство ...

Просто го прочетете от началото до края. Основно, обясненията следват кода. Преди да прочетете обясненията опитайте се да разберете какво прави кода и после проверете дали сте били прави. По този начин ще получите максимални знания, а и ще напрегнете сивите си клетки. Когато свършите изпратете ми мнението си за дизайна, съдържанието и ако откриете грешки в ръководство. Ще отговоря на всички писма! Моля отбележете -не ми изпращайте въпроси за проблемите си с Perl - не съм техническа поддръжка (ако някой иска трябва да плаща). Ако търсите решения на проблемите си пишете на Usenet или на ActiveState mailing lists. Във това ръководство кодът ще е е със син цвят, а кодът които се записва за да се стартира скрипта със зелен цвят.

```
print "You must teach hard Perl";
```

```
C:>perl myfirst.pl
```

Интернет ресурси свързани с Perl

Ако прочетете цялото ръководство написано от мен (заедно с предстоящите допълнения) за езика Perl, вие ще добиете добра представа за него. Следващите интернет адреси, ще ви изведат на едно по-високо ниво в работата с Perl. Препоръчвам първо да прочетете информацията в това ръководство и след това да търсите отговори на възникналите въпроси.
Първо може да прочетете за Usenet, услуга която предоставя информация. Вие може да обилаляте из различните нюз групи и да събирате различна информация. Всеки път когато имате въпроси за Perl или CGI програмирането може да си задавате въпросите на дадена нюз група. Отговорите обикновенно идват бързо, ако добре сте обяснили проблема си. Ако сте нов в CGI програмирането с Perl, вие ще искате да обиколите всички сайтове описани по-долу. Ако го направите, това ще ви даде добра представа какво може да намерите в интернет. Ако намерите нещо полезно сваляйте го на компютъра си като запазвате адреса от където сте си свалили файла, датата и версията.

Usenet нюз групи

Usenet е Internet услуга, която разпределя съобщения между сървъри. Всека стая си има специфична група. Нуждаете се от програма която чете новини и да ги запазвате на компютъра си.

Нюз група	Обяснение
comp.lang.perl.misc	Покрива общите въпроси свързани с Perl.
comp.lang.perl.announce	Покрива Perl-свързани съобщения.
comp.lang.perl.modules	Нюз група е много полезна, давайки отговори какви модули са налични, как да ги използваме и ако има някакви проблеми с използването им, винаги може да си зададете въпроса
comp.lang.perl.tk	Perl/Tk интеграцията и използването им. Това е форум където се дискотира Tk и Perl. Tk е интерфейс разработен от Sun, основно за да се използва с Tcl, внедряващ скриптов езика.
comp.infosystems.www.announce	Не е свързан с Perl, но е много полезен за да научавате новите разработки в web пространството.
comp.internet.net-happenings	Друг нюзгрупа даваща добра представа за интернет разработките.

Най-използваната нюз група свързана с Perl е comp.lang.perl.misc. Когато имате въпроси или проблеми пишете на нея. Не задавайте въпросите си в повече от една нюз група, защото хората, които отговарят проверяват няколко нюз групи им става неприятно когато видят един и същи въпрос зададен в няколко нюз групи. Преди да зададете въпросите си прочетете Perl FAQ. FAQ са документи

съдържащи най-често задаваните въпроси. Ако зададете въпрос, който вече е зададен преди вас, ще ви се скарат другите, които четат списъка с въпросите. Запомнете, че хората които отговарят не са длъжни да го правя. Ако вие се изразявате неясно и не описвате добре проблема си или се правите на многознайковци, в замяна няма да получите нищо. Може да намерите FAQ документа на http://www.perl.com/perl/faq/ уеб страница.

Web Сайтове

Следващите сайтове са добро място за посещение за да си създадете собствена Perl и CGI скриптова библиотека. Сайтовете ще ви дадат представа какво вече съществува, да намерите информация която може да използвате и скриптове които да използвате или модифицирате. Много от скриптовете на посочените сайтове са freeware или shareware.
езика Perl
ftp://convex.com/pub/perl/info/lwall-quotes - сайт за Larry Wall. Почитателите му са направили този сайт за да покажат някои от коментарите написани за него.
http://www.yahoo.com - Едно от най-добрите места за търсене на информация или файлове - една от най-добре организираните и просторни търсачки в нета. Yahoo има отделни категории за Perl и CGI.
http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Perl/
http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI___Common_Gateway_Interface/
http://www-genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.cfml - Сайт за CGI.pm Модула. CGI.pm е модул,който дава много възможности за писане на HTML форми и CGI програмиране с.
http://www.stars.com - Сайта съдържа много ръководства за HTML, CGI, HTTP, Databases, CSS и др.. Съдържа много богата колекция от линкове свързани с уеб програмирането.
http://www.activestate.com/ - Сайт е за Perl и Rython за Win32 основно. Всеки които възнамерява да се занимава с Perl на Windows операционна система, трябва да посети този адрес.
http://www.teleport.com/~merlyn/ - Randal L. Schwartz е един от гурутата в Perl програмирането. Сайта съдържа интересна и полезна информация.
http://www.engr.iupui.edu/~dbewley/perl/ - Perl информация. Съдържа ръководства, книги, скриптове.
http://www.engr.iupui.edu/~dbewley/cgi/ - CGI информация
http://www.worldwidemart.com/scripts/ Matt Wright's scripts е много добър сайт с добра организация, интуитивен, съдържащ добре документирани скриптове. Препоръчвам го ако искате да си изтеглите CGI скриптове, които да разгледате и прочетете упътванията за работата с тях.
http://www.hermetica.com/technologia/DBI/index.cfml - Сайта е за Perl и база данни. С DBperl може да осъществявате връзка с такива бази данни като: Oracle, Sybase, mSQL, Informix и Quickbase.
http://www.virtualville.com/library/cgi.cfml Дава начални познания за CGI и линкове към ресурси свързани с този интерфейс.
http://www.selah.net/cgi.cfml - сайта съдържа колекция от CGI скриптове, създадени с Perl или C.
http://kufacts.cc.ukans.edu/info/forms/forms-intro.cfml - Сайта дава представа за CGI и формите. Сайта не само обяснява, но и има графична демонстрация как формите чрез CGI осъществяват контакт.
http://hoohoo.ncsa.uiuc.edu/docs/cgi/ - CGI документация от NCSA.Този сайт дава представа за CGI. Също така включва скриптове, сигурност и др. свързани CGI проблеми.
http://www.atmos.washington.edu/perl/perl.cfml - предлага ръководство по Perl
http://www0.cise.ufl.edu/perl/ - Страницата за Perl на университета във Флорида.

Internet Relay Chat или IRC

Internet Relay Chat усуга е много добро средство за намиране на информация. Ако имате късмет може да се свържете с хора, които имат големи познания по въпросите, които ви интересуват и могат да ви отговорят. Предимството на IRC е, че контакта става в реално време. Задавате въпрос и ви се отговарят веднага. Има няколко мрежи поддържащи IRC: EfNet, Undernet и DALnet. Perl гурутата са в EfNet. #perl IRC канала е добро място за задаване на въпроси. Ако имате по-прости въпроси, за вас е канала #perl-basics. CGI въпросите трябва да се задават в #cgi.

Нека започнем с изучаването на Perl

Вашия първи скрипт

Създайте нова директория, където ще съхранявате вашите perl скриптове например c:\scripts\. Стартирайте текстовия редактор с който ще пишете скриптовете се. Аз използвам EditPlus, който може да изтеглите от сайта www.editplus.com. Напишете:

```
print "My first Perl script\n";
```

Запишете файла като myfirst.pl. Внимавайте с какво разширение ще запишете скрипта. Не затваряйте Editplus - дръжте го отворен за да правите промени по скрипта в движение. Стартирайте вашия command prompt (За тези които съвсем нищо не знаят това е ДОС. Стартирането става като натиснете Start->Programs->MS-DOS Prompt). Напишете doskey. Това стартира досовската програма doskey, която служи да запомня командите които пишете. Например написвате perl myfirst.pl, за да стартирате скрипта. Скрипта се изпълнява. След неговото изпълнение искате пак да го изпълните, не е нужно отново да пишете perl myfirst.pl, а просто със стрелките нагоре и надолу може да извиквате всичко което сте писали в конзолата (промпта, ДОС режима). Стигнете до директорията съдържаща скрипта. Напишете следващото за да стартирате скрипта:

```
perl myfirst.pl
```

В тази кратка програма използвахме само една функция на Perl, която е print. Тук print получава низ (заграден с кавички текст), като свой аргумент. В случая казваме на print, че искаме да отпечатаме на екрана My first Perl script.

Shebang

Почти всяка книга за Perl е написана за UN*X, което е проблем за Win32. Това води до скриптове като:

```
#!c:/perl/perl.exe
print "I'm a cool Perl hacker\n";
```

Функцията на 'shebang' линията (първия ред на програмата започващ със символите #!) е да каже как да се изпълни файла. Под UNIX, това води до резултат. Под Win32, системата трябва вече да знае как да изпълни файла преди файла да бъде изпълнен, така че линията не е нужна. Както и да е, линията не се обезмисля напълно, тъй като могат да бъдат зададени първоначални опции на изпълнение на този ред (за пример -w е флаг, който кара интерпретатора на Perl да изпълни скрипта в предупредителен режим. Вие може да напишете линията, така че скрипта да може да се изпълнява директно под UNIX без модификации.

Типове данни

Променливи

Всеки програмен език разполага с типове данни върху които извършва операции. В Perl те могат да бъдат от една страна число или низ, константа или променлива.
Число: едно- или многоцифрено бройно число.
Низ: сбор от символи (числа, букви, всъщност всеки ASCII код от 0 до 255). Единичните кавички (') оказват начало и край на низа. Двойните кавички имат по специално предназначение. Чрез тях се включват някой специални символи.Низ може да бъде заграден от единични или двойни кавички според ефекта, които търсим.

Константа: това е число или низ, което предварително е известно и зададено в скрипта, не се изменя с изпълнението на скрипта.

Променлива: това е място в паметта, където може временно да се съхранява информация, по време на изпълнение на скрипта тази информация може да се променя. В Perl всяка променлива може да бъде с "произволна" дължина. т.е променливата освен че може да променя съдържанието си, но може и да променя своят размер - всичко това става автоматично. За разлика от други програмни езици, променливите не е необходимо да бъдат декларирани, те могат да се обявяват направо като се използват.

Променливите се означават чрез поставянето на специален символ, последван от име на променливата. Специалните символи могат да бъдат: $, @, %- всеки от тях определя типът на променливата.Името може да бъде произволно, с много малко ограничения, едно от които е че ако името започва с число, то в него не могад да се съдържат и букви, само числа са разрешени.

Скалари

А сега да преминем към нещо по интересно от принтването на екрана. Скалара е самостоятелна група данни. Скалара е единична променлива. В $var=10, скалар е $var, която задава на променливата с името $var стойността 10. Означаването на скалари става посредством знакът: $, който много прилича на "S"(Scalar). По-късно,ние ще се запознаем с масивите и хешовете, където @var се обръща към повече от една стойност.

```
$string="perl";
$num1=20;
$num2=10.75;
print "The string is $string, number 1 is $num1 and number 2 is $num2\n";
```

$ % @ са полезни. Ако вие сте запознат с други програмни езици ще се изненадате от кода $var=10. С повечето езици, ако искате да определите стойността 10 на променливата с име var вие трябва предварително да определите какъв тип ще е var integer, boolean или др.. Не е така в Perl. Това е отличителна черта. Всички променливи имат за представка символ като $ @ % . Които ги определят какъв тип променливи са. За разлика от други езици, променливата може да се зададе и в самата операция в която се използва.

Интерполация на променливи

Ние все още не сме приключили с кода. Забележете начина на използване на променливите в стринга. Когато искаме променливата да се интерполира - да покажем стойността, използваме "", а когато не искаме интерполация - да покажем името на променливата - използваме ''.

```
$string="perl";
$num=20;
print "Doubles: The string is $string and the number is $num\n";
print 'Singles: The string is $string and the number is $num\n';
```

Оператори

Оператора изпълнява функция или операция над някакви данни. Например оператора + събира две числа. Ако искате да прибавите 1 към променливата, вие може да направите следното; $num=$num+1 .Има и кратък начин да направите това, който е $num++. Това е автоинкрементация. Познаите какво е това: $num-- . Да, това е автодекрементация.

```
$num=10;
print "\$num is $num\n";
$num++;
print "\$num is $num\n";
$num--;
```

```
print "\$num is $num\n";
$num+=3;
print "\$num is $num\n";
```

При последния пример се извършва ето това - $num=$num+3.

Escaping

Има нещо ново в горния код. Символът \ . Може да видите какво прави той - 'премахва' специалното значение на символа $. Символа \ има и по-широко използване - поставен пред който и да е специален знак в Perl му премахва значението и го показва като обикновен символ. Също така, прави някои не специали символи специални. Прибавянето на \ прави от простия символ 'n' указание за нов ред. Той може да премахва и собственото си значение, например \\ . Така че ако искате да изпечатате единична \ пробвайте:

```
print "the MS-DOS path is c:\\scripts\\";
```

'\' се използва също така и при референциите. Perl използва доста символи които не са еднозначни по своето значение в различни случай. В някои случай един символ има две или повече значения в зависимост от контекста в които се употребяват. Коректорния знак ^ има различно значение в [^abc] и [a^bc] .

Променливи, нарастването и принтирането им

```
$string="perl";
$num=20;
$mx=3;
print "The string is $string and the number is $num\n";
$num*=$mx;
$string++;
print "The string is $string and the number is $num\n";
```

Забележете, че *= означава 'умножи $num по $mx' или, $num=$num*$mx . Разбира се Perl поддържа обикновенните оператори + - * / ** % . Последните две са степенуване и делене, връщащо остатъка като отговор.
Print е функция за работа със списъци. Това означава, че приема списък от стойности, разделени със запетайки, както във следващия пример:

```
print "a doublequoted string ", $var, 'that was a variable called var', $num," and a newline \n";
```

Разбира се, вие може да сложите всичко това между "":

```
print "a doublequoted string $var that was a variable called var $num and a newline \n";
```

за да постигнете същия резултат. Предимството от използването на print в списъчен контекст, е че изразите се изчисляват преди да бъдат принтирани. За пример пробвайте това:

```
$var="Perl";
$num=10;
print "Two \$nums are $num * 2 and adding one to \$var makes $var++\n";
print "Two \$nums are ", $num * 2," and adding one to \$var makes ", $var++,"\n";
```

Ако искаме към $var да се прибави единица преди да се принтира то тогава трябва да използваме следния записване ++$var (ако си спомняте).

Подпрограми - общо представяне

Нека погледнем по друг начин на кода, който вече използвахме. Забелязваме, че много пъти ни се налага да пишем едно и също. Може да се сложи кода в блок (блок е всичко което е записано между {}). Блокът се изпълнява само ако му се каже да се изпълни. Подпрограмата е блок с име. Чрез извикване на името на подпрограмата, ние ще изпълним блока и. Чрез подпрограмите се намалява писането на код, увеличава се прегледността на програмата и се намалява големината на файла и т.н.. А ето и примера:

```
$num=10;
&print_results;

$num++;
&print_results;

$num*=3;
&print_results;

$num/=3;
&print_results;

sub print_results {
print "\$num is $num\n";
}
```

Подпрограмите могат да се слагат навсякъде в скрипта, в началото, края, по средата ... няма значение. Добра практика е всички подпрограми да се слагат накрая на файла. По-прегледно и удобно е. Подпрограмите са код, който искате да се изпълни повече от веднъж в скрипта. Подпрограмите се задават чрез sub и след него името на подпрограмата. След което лява голяма скоба { , кода на подпрограмата и затваряща дясна голяма скоба }. Подпрограмите обикновенно се извикват чрез импеданс '&' и името на подпрограмата, като &print_results; . Може да се пропусне импеданса, но тогава възниква възможност за грешки със файловите манипулатори (за това по-късно), затова по-добре не го правете.

Коментари

Виждате ли знака # в кода на скрипта. Това е знака за коментар. Всичко след знака # се игнорира при изпълнението на скрипта. Вие не може да продължавате коментара от един ред на друг. Затова ако искате да продължите коментара на следващия ред поставете знака # в началото му. Може да документирате програмата си , така че другите хора (или вие, след като месеци не сте я докосвали) да могат да разберат какво се постига с този код.

Сравнения

IF

Работата с функцията if е проста. Ако деня е Sunday, тогава ще си лежа в леглото. Прост тест, с два изхода:

```
if ($day eq "sunday") {
&lie_in_bed;
}
```

Вие вече знаете, че &lie_in_bed извиква подпрограма със същото име. Допускаме, че променливата $day е зададена по рано в скрипта. Ако $day не е еднаква с 'Sunday', &lie_in_bed няма да се изпълни. Пробвайте това:

```
$day="sunday";
if ($day eq "sunday") {
```

```
print "Zzzzz....\n";
}
```

Забележете синтаксиса. if се нуждае от нещо, което да се тества и да върне като резултат истина (1) или лъжа (0). Този израз трябва да е в () - скоби. следвани от {} - където трябва да се постави блока за изпълнение, ако израза в () върне като отговор истина.

Истина според Perl

Има много функции в Perl които тестват за Истина. Някои от тях са if, while, unless. Така че е важно да знаете какво е истина, като дефиниция в Perl. Има три основни правила:

Всеки стринг е истина с изключение на "" и "0".
Всяко число е истина с изключение на 0. Истина са и отрицателните числа.
Всяка недифинирана (not defined) стойност е лъжа. Недефинираната стойност е тази, която няма стойност, не е определена.
Код който илюстрира по-горните твърдения:

```
&isit; # $test1 в този момент е недефинирана - тя не съществува

$test1="hello"; # стринг който е различен от "" и "0"
&isit;

$test1=0.0; # $test1 е число, но реално е 0
&isit;

$test1="0.0"; # $test1 е стринг, но НЕ е 0 !
&isit;

sub isit {
if ($test1) { # тества $test1 за истина или не
print "$test1 is true\n";
} else { # else ако не е истина
print "$test1 is false\n";
}
}
```

Първия тест пропада, защото $test1 е недефинирана. Това означава, че не е била създадена за да приеме някаква стойност. Последните два теста са интересни. Разбира се 0.0 е едно и също с 0 в числен контекст. Но 0 в контекста на стринг е истина. Тук ние тествахме единични променливи. Сега ще тестваме резултатите от изрази.

```
$x=5;
$y=5;

if ($x - $y) {
print '$x - $y is ',$x-$y," which is true\n";
} else {
print '$x - $y is ',$x-$y," which is false\n";
}
```

Теста пропада защото 5-5 е 0, което е грешка. print може да изглежда малко странно. Имаме списък, в който първата стойност е в ". Това е защото не искаме да интерполираме стойностите. Следващата стойност е израза който трябва да се пресметне, затова е извън всякакви кавички. И накрая стойността е в "", защото искаме след изпълнение на кода интервала да отиде на нов ред - \n, а без "" това няма да стане. Друг тест е да сравним две стойности за истина.

```
$lucky=15;
$drawnum=15;

if ($lucky == $drawnum) {
print "Congratulations!\n";
} else {
print "Guess who hasn't won!\n";
}
```

Важна роля в горния пример има оператора за равенство == .

Сравненията и Perl

Сега трябва да внимавате във следващите обяснения, защото после ще се връщате тук отново. Символът = е оператор за присвояване на стойност, не е оператор за сравнение. Следователно:
if ($x = 10) е винаги истина, защото на $x успешно се присвоява стойността 10.
if ($x == 10) сравнява две стойности, които може да не са равни.
И така ние сравнявахме до сега числа, но ние можем и сигурно ще се налага да сравняваме променливи.

```
$name = 'Mark';
$goodguy = 'Tony';
if ($name == $goodguy) {
print "Hello, Sir.\n";
} else {
print "Begone, evil peon!\n";
}
```

Нещо грешно има в кода. Нещо изглежда не е както трябва. Очевидно Mark е различно от Tony, тогава защо perl ги намира за равни? Те са равни -- числено. Ние трябва да ги тестваме като стрингове, а не като числа. За да направим това, просто заместете == със eq и евевнтуално горния код ще работи както трябва. Има два типа оператори за сравнение - за числа и за стрингове. Видяхте вече два такива, == и eq. Пробвайте този код:

```
$foo=291;
$bar=30;
if ($foo < $bar) {
print "$foo is less than $bar (numeric)\n";
}
if ($foo lt $bar) {
print "$foo is less than $bar (string)\n";
}
```

Операторът lt сравнява в контекста на стринг, и разбира се < сравнява в контекста на числа. Азбучно, това е в контекста на стринг, 291 е след 30. Това е по кодовата таблица ASCII. Мотото на Perl е "Има повече има от един начин да се направи" или TIMTOWTDI. Произнася се 'Tim-Toady'. Това ръководство не се опитва да ви покаже всички начини за решаването на даден проблем. При писане на вашите програми ще ги откривате.
В следващата таблица са показани операторите за сравнение в Perl:

	Числен контекст	В контекста на стринг
Равно	==	eq
Не равно	!=	ne
По-голямо от	>	gt
По малко от	<	lt
По-голямо или равно	>=	ge
По-малко или равно	<=	le

Повече за if

Пробвайте този код:

```
$age=25;
$max=30;

if ($age > $max) {
print "Too old !\n";
} else {
print "Young person !\n";
}
```

Лесно е да видите какво прави той. Ако израза е грешен, тогава все едно какво има в else - блока се изпълнява. Просто е. Но ако искате още един тест? Perl може и това.

```
elsif
$age=25;
$max=30;
$min=18;

if ($age > $max) {
print "Too old !\n";
} elsif ($age < $min) {
print "Too young !\n";
} else {
print "Just right !\n";
}
```

Ако първия тест пропадне, вторият се преценява. Това се извършва докато има elsif, или else е достигната. Има голяма разлика между горния код и следващия по-долу:

```
if ($age > $max) {
print "Too old !\n";
}

if ($age < $min) {
print "Too young !\n";
}
```

Ако го стартирате ще ви върне същия резултат - в този случай. Както и да е, но това е Лоша Практика за Програмиране. В този случай ние тестваме число, но представете си че тестваме стринг за да видим дали съдържат R или S. Възможно е стринга да съдържа и двете R ид S. Така ще се преминат и двата 'if' теста. С използването на elsif ще избегнете този недостатък. При първото срещане на истина в elsif изявление то се изпълнява, като следващите не се изпълняват.

Въвеждане на данни

STDIN и другите filehandles

Понякога вие трябва да контактувате с потребителя за да получите информация и да вършите действие с нея. Пробвайте това:

```
print "Please tell me your name: ";
$name=<STDIN>;
print "Thanks for making me happy, $name !\n";
```

Има нови елементи, които трябва да научите тук. Първо <STDIN> . STDIN е filehandle. Filehandles са неща, които служат за да се взаймодейства с обекти като файлове, сокети, входни данни и др. Може да се каже, че STDIN е стандартната функция за вход. В този случай STDIN чете от клавиатурата (входните данни въведени от потребителя). Този тип скоби - <> чете данни от filehandle. И така ние прочитаме въведените данни от STDIN. Зададената стойност се установява на провенливата $name и принтирани. Някаква идея защо има още един ред. Като натиснете Enter, вие включвате нов ред във вашите данни. Лесния начин да премахнете новия ред е чрез chop:

```
chop
print "Please tell me your name: ";
$name=<STDIN>;
chop $name
print "Thanks for making me happy, $name !\n"
```

и този код пропада поради синтактична грешка. Можете ли да познаете защо? Погледнете изведената информация, вижте номера на реда, където е открита грешката и вижте къде в синтаксиса сте сбъркали. Отговорът е в липсващите точка и запетая (;) в края на последните два реда. Ако вие добавите ; в края на трети ред, но не и на последния ред, тогава програматаще работи както трябва. Това е така защото Perl не се нуждае от ; на последния ред от блок. Препоръчвам винаги да се слага, най-малкото защото е едно натискане на клавиш, а и след дадения код винаги може да добавите нещо без ако сте пропуснали ; да търсите къде има синтактична грешка. Функцията chop премахва последния символ от всичко каквото и е дадено. В този случай премахва символа за нов ред. Кодът може да се съкрати, ето как:

```
print "Please tell me your name: ";
chop ($name=<STDIN>);
print "Thanks for making me happy, $name !";
```

Скобите ( ) принуждават chop да действа на резултата от това което е вътре в скобите. И така $name=<STDIN> се изпълнява първо, после резултата от това, който е $name със зададена стойност, после от тази стойност се премахва последния символ. Пробвайте без него. Може да прочетете от STDIN колкото пъти поискате. Вижте този код.

```
print "Please tell me your name: ";
chop ($name=<STDIN>);

print "Please tell me your nationality: ";
chop ($nation=<STDIN>);

if ($nation eq "British" or $nation eq "New Zealand") {
print "Hallo $name, pleased to meet you!\n";

} elsif ($nation eq "Dutch" or $nation eq "Flemish") {
print "Hoi $name, hoe gaat het met u vandaag?!\n";

} else {
```

```
print "HELLO!!! SPEAKEEE ENGLIEESH???\n";
}
```

Премахването е опасно, в период когато най-важното нещо е сигурността при програмирането.

Безопасно премахване чрез chomp

Ние искаме да премахваме само символа за нов ред, а не безогледно които и да е последен символ. Това може да стане с chomp - който премахва последния символ само ако той е за нов ред.

```
chomp ($name=<STDIN>);
```

И тук Perl гурутата извикват "Открих грешка!". Е добре, chomp не винаги премахва последния символ, ако той е за нов ред, но ако не го премахне, вие имате специалната променлива, наречена $/ .Като зададете нов символ на тази променлива, тази стойност става символа за нов ред, на която стойността по подразбиране е \n.

Масиви

Какво са масивите?

В Perl има два типа масиви, асоциативни масиви (хешове) и масиви. И двата масива са списъци. Списъка с сбор от променливи. Може да мислити за списъците в Perl като за стадо от животни. Списъка се обръща към цялото стадо, а скалара се обръща само към едно животно (Много МУУУУУУ стана,А?). Списъка е стадо от променливи. Не е задължително променливите да бъдат от един и същи тип. Може да имате три скалара, два масива и три хеша в един масив. Всеки тип от списък си има свое име в Perl. Масивите представляват съвкупност от елементи, подредени един след друг. Дефиницията на масив в другите езици е: съвкупност от еднотипни елементи (с еднаква големина). В Perl обаче всеки елемент може да е различен и с произволна дължина, следователно масивът е съставен от разнородни по състав елементи. За това често масивът е наричан списък. Масивите се означават с @ - която наподобява "a"(Array-масив).

Основи на работата с масиви

За пример, масива е списък от стойности. Списъка може да се обръща към целия списък или към отделни елементи от него. Скрипта по долу задава масива наречен @names . Зададени са 5 стойности в масива.

```
@names=("Muriel","Gavin","Susanne","Sarah","Anna");

print "The elements of \@names are @names\n";
print "The first element is $names[0] \n";
print "The third element is $names[2] \n";
print 'There are ',scalar(@names)," elements in the array\n";
```

Първо, забележете как се декларира масива - @names . Всяка стойност е заградена в кавички "",а запетайката е подразбиращия се разделител за стойности в масива. След това забележете как се извежда. Първо се обръща към себе си като цяло в контекста на списък. Това означава че се обръща към повече от една стойност. Вижте следващия код

```
@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Trish","Simon");

print @names;
print "\n";
print "@names";
```

Когато списъка е между "" стойностите в него се извеждат с интервал между тях. Ако искаме да се

обърнем към повече от една стойност от масива използвайте префикса @ . Когато се обръщате към повече от една стойност от масива, но не към целия масив, това се нарича дял, резен (slice) . Аналогията с кекс е подходяща. Парче от кекс е подходящо сравнение за дял.

Елементите на масива

Масивите не са много полезно нещо ако не можем да се обръщаме към стойностите в него. Първо ако извикваме единичен елемент от масив не трябва да използваме префикса @ който се използва за обръщение към няколко стойности. Ако ни трябва единична стойност, то това ще е скалар, а префикса за скалар е $. Второ ние трябва да укажем, кой точно елемент искаме. Това е лесно - $array[0] за първия елемент, $array[1] за втория и т.н.. Индекса на списъците започва от 0, докато не направите нещо което да промени това (нещо което е твърде неодобрително и противопоказно за добро и пълноценно програмиране). И последно, използвахме масива в скаларен контекст за да видим колко елемента има, чрез scalar(@names) в по-горния пример.

Как да се обръщаме към елементите на масив

```
$myvar="scalar variable";
@myvar=("one","element","of","an","array","called","myvar");

print $myvar; # обръща се към скалар с името myvar
print $myvar[1]; # обръща се към втория елемент на масива myvar
print @myvar; # обръща се към всички елементи на масива myvar
```

Двете променливи нямат нищо общо помежду си. Ако се върнем към аналогията с животните, това е като да си имаш куче с име 'Myvar' и златна рибка 'Myvar'. Вие никога няма да ги сбъркате, нали? Номера на елемента от масив, който извикваме може да бъде променлива.

```
print "Enter a number :";
chomp ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah","Anna");

print "You requested element $x who is $names[$x]\n";

print "The index number of the last element is $#names \n";
```

Забележете последния ред на примера. Връща индекса на последния елемент на масива. Същото може да го направите и с $last=scalar(@names)-1; но това не е толкова ефективно. Има по-лесен начин да получите последния елемент на масива, както ще видите:

```
print "Enter the number of the element you wish to view :";
chomp ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Trish","Simon");

print "The first two elements are @names[0,1]\n";
print "The first three elements are @names[0..2]\n";
print "You requested element $x who is $names[$x-1]\n"; # започва от 0
print "The elements before and after are : @names[$x-2,$x]\n";
print "The first, second, third and fifth elements are @names[0..2,4]\n";

print "a) The last element is $names[$#names]\n"; # първи начин
print "b) The last element is @names[-1]\n"; # втори начин
```

Забележете, че имате две стойности разделени от многоточие. Това дава всички стойности от първата до последната между двете числа. [0..3] означава 0, 1, 2, 3 .Когато за индекс се използва отрицателно

число, стойностите на масива се взимат отзад напред, -1 е за последната стойност, -2 за предпоследната и т.н.

За циклите

Всичко дотук е добре, но как ние ще изведем всички елементи на масив един след друг? Може да го направим чрез повторение ето така:

```
@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Trish","Simon");

for ($x=0; $x <= $#names; $x++) {
print "$names[$x]\n";
}
```

което задава на $x стойност 0, прави един цикъл, прибавя единица към $x , проверява дали е по-малка от $#names , и т.н.. Това е пример за цикъл. За да бъде по-детайлно, цикъла има три части:

- Инициализация
- Проверка на условието
- Модификация

В този случай, променливата $x е инициализирана на 0. След това веднага е проверена дали е по-малка или равна на $#names . Ако е истина, блокът се изпълнява веднъж. Ако условието не е изпълнено и блока не се изпълнява.
Един път след като блока се изпълни, променливата се модифицира. Това е $x++ . След това теста за изпълнение на условието се изпълнява и т.н..

За циклите с .. - оператор за поредица

Има и друг начин за написването на по-горния код:

```
for $x (0 .. $#names) {
print "$names[$x]\n";
}
```

който използва оператора за поредица .. (две точки една до друга). Това просто задава на $x стойност 0, после увеличава $x с 1 докато стане равно на $#names . През цялото това време изпълнява блока по веднъж на всяко увеличаване с единица.

foreach

За добър код трябва да се използва foreach .

```
foreach $person (@names) {
print "$person";
}
```

Цикъла минава през всеки елемент ('многократно повторение', друго техническо описание на процеса) на масива @names , и всеки елемент се връща като стойност на променливата $person . След това може да извършвате всякакви действия с променливата. Вие може да използвате и:

```
for $person (@names) {
print "$person";
}
```

ако искате. Няма разлика като цяло, но кода изглежда малко по-неясно.

Невероятното $_

За да намалим кода , ще ви преставя $_ , които е входа по подразбиране и шаблонна променлива.

```
foreach (@names) {
print "$_";
}
```

Ако вие не зададете променлива в която да се присвояват като стойности, елементите на масива , $_ се използва по подразбиране за тази и много, много други операции в Perl. Още един пример с функцията print:

```
foreach (@names) {
print ;
}
```

Ние не определяме кой елемент да се принтира, затова $_ се изпозва по подразбиране. По дефиниция ако на print не е зададена променлива, тя използва стойността която се намира в специалната променлива $_.

Преждевременен край на повторения

Процесът на повторенията завършва при постигането на някакво условие зададено първоначално или въобще не завършва. Следващия скрипт е пример за непрекъсваем процес, защото 1 - е истина, и условието е винаги истана.

```
while (1) {
$x++;
print "$x: Знаете ли че може чрез натискане на CTRL-C да прекъснете изпълнението на perl скрипт?
\n";
}
```

Друг начин да излезете от процеса, ако при изреждането на елементите на масива намерим този, който ни трябва и продължаването на процеса се обезмисля.

```
@names=('Mrs Smith','Mr Jones','Ms Samuel','Dr Jansen','Sir Philip');

foreach $person (@names) {
print "$person\n";
last if $person=~/Dr /;
}
```

Оператора last изпълнява функцията да прекъсне процеса. Не се тревожете за /Dr / това е регулярен израз който ще бъде обяснен по-късно. Всичко което трябва да знаете засега е, че връща истина ако намери съвпадение между това в скобите и подадената променлива. В случая когато открие съвпадение подава истина и функцията last се изпълнява. Контрол върху функцията last се извършва чрез етикети.
Засега е просто, но почакайте? Неи се нуждаем от лекар, който го има и в списъка с имена, не просто само лекар. Следващия пример е за това:

```
@names =('Mrs Smith','Mr Jones','Ms Samuel','Dr Jansen','Sir Philip');
@medics =('Dr Black','Dr Waymour','Dr Jansen','Dr Pettle');

foreach $person (@names) {
print "$person\n";
if ($person=~/Dr /) {
foreach $doc (@medics) {
```

```
print "\t$doc\n";
last if $doc eq $person;
}
}
}
```

Първо се взима първия елемент на масива @names, изписва се на екрана, и след това се проверява за съвпадение с Dr, ако се открие съвпадение се изпълнява следващия блок. В него се изреждат всички елементи на масива @medics, и ако се открие, че съдържанието на променливата $doc съвпада със съдържанието на променливата $person се прекъсва изпълнението на този блок и се преминава към външния цикъл с нова променлива $person.

Но има малък проблем, ние искаме при намиране на съвпадение да прекъснем изцяло цикъла, а не само изпълнението на вътрешния цикъл. Трябва да укажем на функцията last, кое точно искаме да прекъснем. Това става чрез етикети, като във следващия пример:

```
@names =('Mrs Smith','Mr Jones','Ms Samuel','Dr Jansen','Sir Philip');
@medics =('Dr Black','Dr Waymour','Dr Jansen','Dr Pettle');

LBL: foreach $person (@names) {
print "$person\n";
if ($person=~/Dr /) {
foreach $doc (@medics) {
print "\t$doc\n";
last LBL if $doc eq $person;
}
}
}
```

Има само две промени тук. Дефинирали сме етикет с името LBL. Когато укажем на last даден етикет, той прекъсва функцията на която е зададен първоначално този етикет.

Има още две функции за работа с цикли redo и goto. При посочване на първата, връща цикъла отначало. Goto изпраща към друг цикъл, подпрограма или израз за изчисляване. Не се препоръчва използването му.

Промяна на елементите на масив

И така имаме масива @names. Искаме да го променим. Пробвайте следващия код:

```
print "Enter a name :";
chomp ($x=<STDIN>);

@names=("Muriel","Gavin","Susanne","Sarah");

print "@names\n";

push (@names, $x);

print "@names\n";
```

Push функцията прибавя нова стойност към края на масива. Разбира се може да не е само една стойност:

```
print "Enter a name :";
chop ($x=<STDIN>);
```

```
@names=("Muriel","Gavin","Susanne","Sarah");
@cities=("Brussels","Hamburg","London","Breda");

print "@names\n";

push (@names, $x, 10, @cities[1..3]);

print "@names\n";
```

Още нови функции за работа с масив:

```
@names=("Muriel","Gavin","Susanne","Sarah");
@cities=("Brussels","Hamburg","London","Breda");

&look;

$last=pop(@names);
unshift (@cities, $last);

&look;

sub look {
print "Names : @names\n";
print "Cities: @cities\n";
}
```

Сега имаме два масива. pop функцията премахва последния елемент на масив и го връща, което означава че може да извършите някакви действия с върнатата стойност. Функцията unshift прибавя стойност в началото на масив. В таблицата по долу са изборени функциите които се използват за работа с масив.

push	Прибавя стойност към края на масив.
pop	Премахва и връща последната стойност на масив.
shift	Премахва и връща първата стойност на масив.
unshift	Прибавя стойност към началото на масива.

Сега за достъпа до другите елементи на масив. Ще ви представя функцията splice.

```
@names=("Muriel","Sarah","Susanne","Gavin");

&look;

@middle=splice (@names, 1, 2);

&look;

sub look {
print "Names : @names\n";
print "The Splice Girls are: @middle\n";
}
```

Първия аргумент на функцията splice е масив. Втората стойност е индекс (число) показващо, коя стойност от списъка (масива) да се използва за начало. В този случай тя е 1 (ще се започне от втория

елемент). После идва елемента който показва колко елементи да се премахнат от масива и да се прибавят към втория.
Ако използвате резултата от splice като скалар:

```
@names=("Muriel","Sarah","Susanne","Gavin");

&look;

$middle=splice (@names, 1, 2);

&look;

sub look {
print "Names : @names\n";
print "The Splice Girls are: $middle\n";
}
```

тогава скалара взема като стойност последната му подадена стойност. Функцията splice се използва и за триене на елементи от масив.

Триене на променливи от масив

Искаме да изтрием Hamburg от масива. Как да го направим? Може би ето така:

```
@cities=("Brussels","Hamburg","London","Breda");

&look;

$cities[1]="";

&look;

sub look {
print "Cities: ",scalar(@cities), ": @cities\n";
}
```

Разбира се Hamburg е премахнат, но забележете, броят на елементите остана един и същ. Все още има 4 елемента в масива. Резултата от горния код е, че $cities[1], съществува но няма стойност. И така ние трябва да използваме splice функцията за да премахнем елемента изцяло.

```
splice (@cities, 1, 1);
```

Сега нека пробваме друго:

```
$car ="Porsche 911";
$aircraft="G-BBNX";

&look;

$car="";

&look;

sub look {
print "Car :$car: Aircraft:$aircraft:\n";
print "Aircraft exists !\n" if $aircraft;
print "Car exists !\n" if $car;
```

```
}
```

Изглежда че сме изтрили променливата $car. Но не точно. Ние сме изтрили стойността, която тя има, но не и самата променлива. В момента тя има стойност null и затова теста с if пропада. Само ако нещо дава като отговор грешка, то това не означава, че не съществува. Перуката е фалшива коса, но перуката съществува. Вашата променлива е все още там. Perl има функция, която да тества дали нещо съществува. Съществувам, в превод на Perl означава дефиниран (defined):

```
print "Car is defined !\n" if defined $car;
```

Този код ще върне истина. Примерите дотук поставят въпроса, всъщност как да премахнем невъзвратимо една променлива. Много просто, ето как:

```
$car ="Porsche 911";
$aircraft="G-BBNX";

&look;

undef $car;

&look;

sub look {
print "Car :$car: Aircraft:$aircraft:\n";
print "Aircraft exists !\n" if $aircraft;
print "Car exists !\n" if $car;
}
```

Променливата $car е унищожена, тя липсва.

Shebang

Флагове

Вие знаете, че може да пуснете режима с повече предупреждения и обяснения при грешки с -w от командния ред. Всяка опция на perl, която може да се зададе от командния ред, може да се зададе и в кода на скрипта.

```
perl script.pl hello
```

за да изпълни този код:

```
#!perl -w

@input=@ARGV;

$outfile='outfile.txt';
open OUT, ">$outfile" or die "Can't open $outfile for write:$!\n";

$input2++;
$delay=2 if $input[0] eq 'sleep';

sleep $delay;

print "The first element of \@input is $input[0]\n";
```

```
print OUY "Slept $delay!\n";
```

който ще се стартира по-същия начин , ако го стартирате така:

```
perl -w script.pl hello
```

Много по-удобно е да се слагат флагове вътре в скриптовете. Не е задължително да е -w , може да е всеки аргумент който Perl поддържа. Стартирайте

```
perl -h
```

за пълния списък на аргументите.

```
use strict;
```

Какво е това строгост (strict) и как да я използваме? Модулът strict ограничава 'несигурните, опасните конструкции', съгласно perl документацията. Няма смисъл да се притеснявате за опасни конструкции, ако сте прекарали часове наред в дебъгване на кода си. Когато включите модула strict , има три неща които Perl държи да са зададени точно:

- Променливите 'vars'
- Референциите 'refs'
- Подпрограмите 'subs'

При използването на този модул трябва променливите да се декларират преди използването им, като всяка променлива се дефинира с my . Това е пример за програма която не използва стриктен режим:

```
perl script.pl "Alain James Smith";
```

където "" затваря стринга като единичен параметър, иначе той ще се разглежда като три параметъра.

```
#use strict; # махнете '#' след като стартирате кода няколко пъти

$name=shift; # дава първия аргумент от масива @ARGV

print "The name is $name\n";
$inis=&initials($name);

$luck=int(rand(10)) if $inis=~/^(?:[a-d]|[n-p]|[x-z])/i;

print "The initials are $inis, lucky number: $luck\n";

sub initials {
my $name=shift;
$initials.=$1 while $name=~/(\w)\w+\s?/g;
return $initials;
}
```

Вие вече трябва да разбирате какво върши по-горния код. Когато премахнете '#' пред use strict; прагмата и стартирате кода отново, вие ще получите изход подобен на този:

```
Global symbol "$name" requires explicit package name at n1.pl line 3.
Global symbol "$inis" requires explicit package name at n1.pl line 6.
Global symbol "$luck" requires explicit package name at n1.pl line 8.
Global symbol "$initials" requires explicit package name at n1.pl line 14.
Execution of n1.pl aborted due to compilation errors.
```

Тези предупреждения означават, че на Perl не му е ясно какъв е обсега на действие на тези променливи. Това означава, че вие трябва да декларирате всяка една от тях с my, за да ограничите изпълнението им в дадения блок в които се използват, или да им се зададе референцията с тяхното пълно име. В долния пример се използват и двата метода:

```
use strict;

$MAIN::name=shift; # shifts @ARGV if no arguments supplied

print "The name is ",$MAIN::name,"\n";
my $inis='';
my $luck='';

$inis=&initials($MAIN::name);

$luck=int(rand(10)) if $inis=~/^(?:[a-d]|[n-p]|[x-z])/i;

print "The initials are $inis, lucky number: $luck\n";

sub initials {
my $name=shift;
my $initials;
$initials.=$1 while $name=~/(\w)\w+\s?/g;
return $initials;
}
```

Използването на my в подпрограма не е нищо ново за вас, а my извън подпрограма вече е. Ако се замислите цялата програма е един блок, така че може да се зададе променливите да се виждат само в този блок. Друга интересна част от кода е $MAIN::name. Това както може би очаквате е пълното име на променливата. Първата част е името на пакета, в случая MAIN (главния, основния). Втората част е името на променливата.

Логически оператори

Логически оператори са OR, NOT, AND и др.. Те всичките оценяват изрази. Оценените изрази връщат true (истина - 1) или false (лъжа - 0), в зависимост какъв критерии се използва за оценка от операторите.

or

or оператора работи както следва:

```
open STUFF, $stuff or die "Cannot open $stuff for read :$!";
```

Този код означава - ако операцията по отварянето на файла STUFF пропадне, тогава направи нещо друго. Друг пример:

```
$_=shift;
/^R/ or print "Doesn't start with R\n";
```

Ако регулярния израз върне лъжа (не е намерил ред започващ с R), тогава се изпълнява каквото е от лявата страна на or . Както знаете, shift изпълнява действието се върху @ARGV, ако не е зададена стойност, или върху @_ във подпрограма. Perl има два OR оператора. Единия ни е известния or, а

другия е ||.

Приоритет: Кое се изпълнява първо

За да разберете разликата между двата оператора, трябва да поговорим за приоритета при изпълнението. Добър пример е следния:

```
perl -e"print 2+8
```

което както знаем ще изпечата 10. Но ако ние направим така:

```
perl -e"print 2+8/2
```

Сега, това 2+8 == 10, разделено на 2 ли е или може би 8/2 == 4, плюс 2 == 6?

Приоритет е кое ще се изпълни първо. В примера горе може да видите, че делението се извършва преди събирането. Следователно, делението има предимство пред събирането.

Може да използвате скоби за да е по-чисто и чсно, коя след коя команда да се изпълнява:

```
perl -e"print ((2+8)/2)
```

И така основната разлика между or и || е приоритета на изпълнение. В примера по-долу, ние се опитваме да присвоим на две променливи стойностите на два несъществуващи елемента на масив. Това ще пропадне:

```
@list=qw(a b c);

$name1 = $list[4] or "1-Unknown";

$name2 = $list[4] || "2-Unknown";

print "Name1 is $name1, Name2 is $name2\n";

print "Name1 exists\n" if defined $name1;
print "Name2 exists\n" if defined $name2;
```

Изходът е интересен. Променливата $name2 е създадена с фалшива стойност. Обаче, $name1 не съществува. Причината е в приоритетите на операциите. or оператор има по малък приоритет от || .

Това означава, че or поглежда на входа от лявата страна. В този случай, това е $name1 = $list[4] . Ако той е истина, всичко е наред. Ако е лъжа, дясната страна се пресмята, и лявата страна се игнорира, все едно никога не е съществувала. В примера горе, когато лявата страна се открива че е лъжа, дясната се пресмята, която е "1-Unknown" която може да е истина но води до някакво действие (изход).

В случая на || , който има по-висок приоритет, израза от лявата страна веднага се пресмята. В случая е $list[4]. Той е лъжа, така че веднага се пресмята кода от дясната страна на оператора. Но кода от лявата страна които не е изчислен, $name2 = не е забравен. Следователно израза се изчислява не $name2 = "2-Unknown".

Примера по-долу ще ви помогне да си доизясните нещата:

```
@list=qw(a b c);

$ele1 = $list[4] or print "1 Failed\n";
$ele2 = $list[4] || print "2 Failed\n";
```

```
print <<PRT;
ele1 :$ele1:

ele2 :$ele2:

PRT
```

Друг пример:

```
@list=qw(a b c);

$name1 = $list[4] or "1-Unknown";

($name2 = $list[4]) || "2-Unknown";

print "Name1 is $name1, Name2 is $name2\n";

print "Name1 exists\n" if defined $name1;
print "Name2 exists\n" if defined $name2;
```

Сега, ($name2 = $list[4]) е самостоятелен изразкойто се изчислява, а не само $list[4] така че ние получаваме същия резултат ако бяхме използвали or .

Трети пример:

```
# $a е равно на $b ако $b е истина, иначе $c
$a = $b || $c;

# $x е равно на $y ако $x не е истина
$x ||= $y

# $a е равно на $b ако $b е дефинирана, иначе $c
$a = defined($b) ? $b : $c;
```

And

Логическия оператор AND оценява два израза, и връща true само ако и двата са истина. Контраста е с OR, който връща истина само ако един или повече от два израза са истина. Perl има няколко AND оператора.

Първия вид AND е && :

```
@list=qw(a b c);

print "List is:@list\n";

if ($list[0] eq 'x' && $list[2]++ eq 'd') {
print "True\n";
} else {
print "False\n";
}

print "List is:@list\n";
```

Изхода тук е лъжа. Ясно е, че $list[0] не съдържа x . AND оператора може да върне истина само ако и двата израза са истина, и така след като първия е лъжа perl решава, че няма смисъл да прсмята втория израз, защото няма смисъл.

Втория вид AND е & . Подобен е на && . Помъчете се да разберете разликата в кода по-долу:

```
@list=qw(a b c);

print "List is:@list\n";

if ($list[0] eq 'x' & $list[2]++ eq 'd') {
print "True\n";
} else {
print "False\n";
}

print "List is:@list\n";
```

Разликата е, че се пресмята дясната част на израза независимо от резултата на лявата част. Въпреки факта, че AND оператора не може да върне истина, perl продължева напред и пресмята втория израз при всички положения.

Третия вид AND е познатото ни and . Има същата чуствителност, като && но е с малко по-малък приоритет. Но всички указания за || и or могат да се приложат и тук.

Други логически оператори

Perl има not ,който работи като ! като изключим че има малко предимство. Ако се чудите къде сте виждали ! преди, ето два пример:

```
$x !~/match/;

if ($t != 5) {
This means:
}
```

Има и едно особенно OR, или XOR. Ако единия израз е верен, XOR връща истина
Ако и двата израза са лъжа, XOR връща лъжа (неистина)
Ако и двата израз са истина, XOR връща лъжа (основната разлика със OR)
Това се нуждае от примери. Имате две момичета Jane и Sonia, които не искате да дойдат заедно на вашето парти. Вие ще направите малък скрипт като този по-долу, който ще изпълнява ролята на фейс контрол.

```
($name1,$name2)=@ARGV;

if ($name1 eq 'Jane' xor $name2 eq 'Sonia') {
print "OK, allowed\n";
} else {
print "Sorry, not allowed\n";
}
```

Предлагам да изпробвате скрипта със следните стойности:

perl script.pl Jane Karen
(една истина, една лъжа)

perl script.pl Jim Sonia
(една истина, една лъжа)

perl script.pl Jane Sonia
(и двете са истина)

perl script.pl Jim Sam
(и двете са лъжа)

Свързване

Когато имате данни които трябва да се свържат, например в скалар използвайте '.' . Например:

```
$x="Hello";
$y=" World";
$z="\n";

print "$x\n";
$prt=$x.$y.$z;
print $prt;
$x.=$y." again ".$z;
print $x;
```

Файлове

Отваряне

```
$stuff="c:/scripts/stuff.txt";

open (STUFF, $stuff) or die $!;

while (&ltSTUFF>) {
print "Line number $. is : $_";
}
```

Задаваме във един скалар пътя до даден файл. Функцията open има две променливи който трябва да се зададат. Първата е името което ще се даде на отворения файл, което ще се използва за достъп до него наричано filehandle. Втората променлива е името на файла, в случая скалава $stuff го съдържа. Функцията die прекъсва програмата и изкарва последната системна грешка ($!), ако отварянето на файла не е било успешно.

Предполагам вече се досещате какво е това - while (&ltSTUFF>). <> - този оператор наричан диамант служи за четене от файлове. В него се поставя filehandle за да се прочита файла. Функцията while прочита един ред от файла изпълнява нещо с нея зададено в блока и, и после пак докато се стигне до последния и ред. Специалната променлива $. - съдържа номера на реда от който се чете в момента.

Запис

```
$out="c:/scripts/out.txt";

open OUT, ">$out" or die "Cannot open $out for write :$!";
for $i (1..10) {
print OUT "$i : The time is now : ",scalar(localtime),"\n";
}
```

В този скрипт първо отваряме файл и го означаваме с filehandle OUT. Функцията for се изпълнява 10 пъти, като всеки път записва във файла out.txt текущата дата и час. Когато на print и се посочи даден filehandle, в случая OUT, тя знае че записите трябва да се правят в този filehandle. Ако не е зададен filehandle, използва по подразбиране STDOUT, което е монитора на компютъра.

@ARGV: Аргументи от командния ред

Perl притежава специален масив наречен @ARGV. Това е списък от аргументи следващи името на скрипта на командния ред. Стартирайте следния код:

```
perl myscript.pl hello world how are you

foreach (@ARGV) {
print "$_\n";
}
```

Друг начин да направите същото е :

```
while (<>) {
print ;
}
```

или:

```
print <> ;
```

Директории

glob

```
@fa=&ltC:/*.*>; # Масива се попълва със всички файлове на C:/
@fa=&ltC:/*.html>; # Попълва се само от файлове с разширение html
@ga=glob ("C:/*.com"); # Прави същото, но тук може да се задава и със скалар, допуска интерполация
@ga=glob ("$aaa/*.com"); # например в $aaa е зададен пътя


open(SESAME, "filename"); # прочита от съществуващ файл
open(SESAME, "&ltfilename"); # същото
open(SESAME, ">filename"); # създава файл и записва в него
open(SESAME, ">>filename"); # допълва съществуващ файл
open(SESAME, "| output-pipe-command"); # изпраща информация
open(SESAME, "input-pipe-command |"); # приема информация


$dir=shift; # shifts @ARGV

chdir $dir or die "Can't chdir to $dir:$!\n" if $dir;

while (<*>) {
print "Found a file: $_\n" if -T;
}
```

Функцията chdir променя текущата директория от която може да се чете, на друга подадена на chdir, в случая която е зададена от скалара $dir, който пък си взима стойността от командния ред. След това while прочита всички файлове в текущата директория, преминава ги през теста -f, и ако са файлове се изпечатват на екрана.

```
$dir =shift;
$type='txt';

chdir $dir or die "Can't chdir to $dir:$!\n" if $dir;

while (<*.$type>) {
print "Found a file: $_\n";
}
```

Тук в директорията ще се търсят файлове само с разширение зададено от $type, което в случая е txt.

readdir - как да четем от директория

Има и фукция readdir с която може да направим по друг начин горния пример:

```
$dir= shift || '.';

opendir DIR, $dir or die "Can't open directory $dir: $!\n";

while ($file= readdir DIR) {
print "Found a file: $file\n";
}
```

Първата разлика с горния пример, е че ако няма стойност в @ARGV то $dir = ., което ще е текущата директория. След това отваряме директорията, анаголично на командата open използвана за файлове, но тук е opendir и задаваме възможност ако директорията не може да бъде отворена то да нои се съобщи. Функцията readdir прочита съдържание на един ред и го подава на скалара $file.

```
@files=readdir(DIR);
```

По този начин се прочита цялото съдържание на директорията и попълваме с нея масива @files.

За да затворим един filehandle на файл или директория използваме close или closedir.

grep

```
@files=grep !/^P/, readdir(DIR);
```

Тук срещаме новата функция grep. При нея се задава един списък (всички файлове в директорията), обработват се - търсят се всички файлове и директории в директорията които не започват с P и така обработения списък се задава на @files. Ако търсите във списък и създавате друг списък с нещата които сте открили, тогава grep е отличното средство.

```
@stuff=qw(flying gliding skiing dancing parties racing);

@new = grep /ing/, @stuff;
```

Имаме масива @stuff, който се претърсва за елементи, които съдържат в себе си ing и списъка с намерените думи се задава на @new.

```
@new = grep s/ing//, @stuff;
```

Тук всички намерени думи съдържащи ing, се обработват като им се маха ing и се връщат на масива @new.

```
@new = grep { s/ing// if /^[gsp]/ } @stuff;
```

Тук grep изпълнява блок от команди дададен с {}. Ако думата не започва (^) с g,s или p ([gsp]), то само тогава се премахва ing.

Map

Map работи по същия начин както grep, и двете функции обработват масив и връщат масив. Но има две основни разлики:

- grep връща стойностите на всичко което е обработено и е върнало стойност истина;
- map връща стойностите на всичко което е обработено независимо дали е изчършена дадената операция или не

```
@stuff=qw(flying gliding skiing dancing parties racing);

@mapped = map /ing/, @stuff;
@grepped = grep /ing/, @stuff;

print "There are ",scalar(@stuff)," elements in \@stuff\n";
print join ":",@stuff,"\n";

print "There are ",scalar(@mapped)," elements in \@mapped\n";
print join ":",@mapped,"\n";

print "There are ",scalar(@grepped)," elements in \@grepped\n";
print join ":",@grepped,"\n";
```

От резултата виждате, че @mapped съдържа списък от единици. ЗАбележете че са 5 за разлика от оригиналния масив който има 6 стойности. Това е защото @mapped съдържа стойностите истина резултат от map.

```
@letters=(a,b,c,d,e);

@ords=map ord, @letters;
print join ":",@ords,"\n";

@chrs=map chr, @ords;
print join ":",@chrs,"\n";
```

Функцията ord променя всеки символ на ASCII еквивалента му, след това chr функцията променя ASCII цифрите на символи. Ако промените map на grep в примера по-горе, ще видите, че нищо няма да се случи. grep ще се опитва да обработи подадената стойност с израза, и ако успее ще върне подадената стойност а не резултата от обработка. Като израз за проверка или изпълнение на map или grep може да му се зададе подпрограма.

Външни команди

Exec

Exec спира работата на скрипта ви и стартира каквото сте му посочили. Ако не може да стартира външния процес, връща код за грешка. Не работи както трябва под Perl за Win32.

System

Стартира външна команда, която се изпълнява заедно със скрипта. Винаги връща отговор, който се попълва във $?. Това означава, че вие може да тествате, дали програмата ви работи. Всъщност вие тествате да видите дали тя може да бъде стартирана, какво прави когато е стартирана и каквото прави е извън ваш контрол, ако използвате system.
С примера по-долу ще илюстрирам system вдействие. Стартирайте 'vol' от командния ред първо ако не сте запознат с нея за да фидите резултата от нея. След това стартирайте 'vole' командата.

```
system("vole");

print "\n\nResult: $?\n\n";

system("vol");

print "\n\nResult: $?\n\n";
```

Както виждате, успешно системно извикване връща като резултат 0. Неуспешното връща число което е необходимо да разделите на 256 за да намерите истинската върната стойност. Също така забележете, че може да видите и изхода от командата.

Backticks

Между използването на ` и system и exec има разлики. Те също стартират външни процеси, но връщат изхода от процеса. След това вие може да правите каквото си поизкате с изходната информация.

```
$volume=`vol`;

print "The contents of the variable \$volume are:\n\n";

print $volume;

print "\nWe shall regexise this variable thus :\n\n";

$volume=~m#Volume in drive \w is (.*)#;

print "$1\n";
```

Както виждате, командата vol се изпълнява. След това извеждаме на екрана изхода на командата. След това с малък регулярен израз извеждаме името на устройстото на което те.

Кога да се използват системни извиквания

Преди да почнете да използвате чъншни команди в скрипта си за извикване на net команди, обърнете внимание, че вече има написани отлични модули които вършат отлична работа, и че всяко стартиране на външна команда забавя скрипта ви. Също така по-добре използвайте readdir, а не `dir` - по-бързо и ефикасно е, това се отнася за всички команди които вътрешни за Perl. Също така се гарантира, че при използването само на вътрешни команди, скрипра ще се изпълнява на 99,99% от всички съществуващи ОС, докато ако използвате външни команди това няма да е така.

Стартиране на процес

Проблема с backticks е че трябва да се изчака целия процес да завърши , и тогава да анализирате и

обработвате информацията. Това е голям проблем ако кодът който се връща е голям или процеса е бавен. можем да стартираме процес и през канал (pipe) да обработваме информацията чрез filehandle, също както го правим с файл. Кодат по-долу е същия като отварянето на файл с две разлики:

- Използваме външна команда, а не име на файл.
- Канала е |, който следва името на командата.

```
open TRIN, "dir |";
while (&ltTRIN>) {
print "$. $_";
}
```

Забележете, че | означава че информацията ще буде получена от външен процес. Също така вие може да подавате информация на външни команди ако | е първия символ.

Oneliners (Скриптове от командния ред)

Малък пример

Perl код може да се изпълнява директно и от командния ред. За пример:

```
perl -e"for (55..75) { print chr($_) }"
```

Флагът -e показва на Perl, че следват команди. Командите трябва да са в двойни кавички, не единични като в Unix. Командата в този пример извежда ASCII стойностите за номерата от 55 до 75.

Достъп до файлове

Следващия пример е установена практика за писане на команда за търсене на стринг в даден файл.

```
perl -e"while (<>) {print if /^[bv]/i}" shop.txt
```

while (<>) конструкцията ще отвори, всичко което е в @ARGV . В този случай, имаме файла shop.txt , който ще бъде отворен и ще бъдат принтирани редовете започващи с 'b' или 'v'. Същия резултат може да се постигне и по по-кратък начин. Стартирайте perl -h и ще видите списък с опции за командния ред. Една от тях ще използваме сега, тя е -n, която слага - while (<>) { } - около целия код между "", посочен с опцията -e. Така че:

```
perl -ne"print if /^[bv]/i" shop.txt
```

прави абсолютно същото като по-горния пример.

Регулярни изрази

Общо представяне

Регулярните изрази ни позволяват да търсим шаблони в данните си. Повечето букви и символи просто ще съвпадат със самите себе си. Например, регулярният израз "test" просто и точно ще съвпада със символния низ "test". Можете да включите режим, нечувствителен към разликата между малки и големи букви, който ще позволи да съвпадне също така и с "Test" или "TEST". Има изключения от това правило, някои символи са особени и не съвпадат със самите себе си. Вместо това те сигнализират, че имат специално значение и трябва да се разглеждат със различно значение от това просто как изглеждат. Например \w не е ескейпвана w, а нещо друго - всяка една буква или цифра от 0 до 9. Голяма част от този документ е посветена на обсъждането на различни метасимволи и тяхното действие. Ето един пълен списък на метасимволите:

```
\ | ( ) [] {} ^ $ * + ? .
```

Може да се изключат специалните значения, използвайки ескейп последователността \Q - след нея 14 специални знака по-горе автоматично приемат своите обикновенни буквални значения. Това е дотогава, докато Perl не види \E или края на шаблона. Например /\Q$future\E/. Не спира интерпретирането на променливите.
Първият метасимвол, на който ще обърнем внимание е "[]". Използват се използва за определяне на клас от символи, представляващ набора от символи, които искате да използвате за съвпадение. Символите могат да бъдат изброявани индивидуално или един диапазон от символи може да бъде обозначен чрез два символа и разделител "-". Например, [abc] ще пасне с всички символи "a", "b", или "c"; това е същото както [a-c], където се използва диапазон, за да изрази същия набор от символи. Ако искате да пасне с която и да е малка буква от латинската азбука, то би трябвало да бъде [a-z].
Метасимволите не са активни вътре в класовете. Например [akm$] ще пасне с всеки от символите "a", "k", "m", или "$"; "$" обикновено е метасимвол, но вътре в класа от символи той е лишен от особенната си природа. Допълвайки набора, можете да пасвате символи, които не са в дадения диапазон. Това се посочва чрез добавянето на "^" като първи символ от класа. Поставен където и да е другаде, "^" просто ще съвпада със символа "^". Например, [^5] ще пасне с всеки символ, с изключение на "5".
Може би най-важният метасимвол е обратно наклонената черта, "\". Обратно наклонената черта може да бъде последвана от различни символи за да се обозначат различни специални последователности. Тя също се използва и за да се избегнат всички метасимволи така че все пак да можете да ги използвате за съпоставка в образци. Например, ако искате да съвпаднете "[" или "\", те трябва бъдат предшествани от обратно наклонена черта за да се премахне специалното им значение: \[или \\. Някои от тези специални последователности представят предварително дефинирани набори от символи, които много често влизат в употреба, като например наборът от цифри, или наборът от латински букви, или наборът от всички символи, които не са празни (whitespace). На разположение са следните предварително дефинирани специални последователности:

\d - Пасва с всяка десетична цифра; това е еквивалент на класа [0-9].
\D - Пасва с всеки символ, който не е цифра; това е еквивалент на класа [^0-9].
\s - Пасва с всеки празен символ; това е еквивалент на класа [\t\n\r\f\v].
\S - Пасва с всеки не-празен символ; това е еквивалент на класа [^ \t\n\r\f\v].
\w - Пасва с всеки буквеноцифров символ; това е еквивалент на класа [a-zA-Z0-9_].
\W - Пасва с всеки не-буквеноцифров символ; това е еквивалент на класа [^a-zA-Z0-9_].

Тези последователности могат от своя страна да бъдат включвани в класове от символи. Например, [\s,.] е клас от символи, който пасва с всеки празен символ, или "," или ".".

Първият метасимвол за повторения, който ще разгледаме, е *. * не съвпада буквално със символа "*"; вместо това, той указва, че предшестващият символ трябва да се среща нула или повече пъти, вместо точно веднъж. Например, ca*t ще пасне с "ct" (0 символа "a"), "cat"(1 "a"), "caaat" (3 символа "a"), и тъй нататък.
Да разгледаме израза a[bcd]*b. В началото той пасва с буквата "a", после с нула или повече букви от класа [bcd], и накрая завършва с "b".
Друг метасимвол за повторение е +, който пасва един или повече пъти. Обърнете внимание на разликата между * и +; * пасва нула или повече пъти, така че това дето трябва да се повтаря може изобщо да не се среща, докато + изисква минимум едно срещане. За да използваме подобен пример, ca+t ще пасне с "cat" (1 символ "a"), "caaat" (3 символа "a"), но няма да пасне с "ct".
Съществуват още два квалификатора за повторение. Въпросителният знак ? пасва с едно или с нула повторения. Можете да мислите за него като за белег за нещо незадължително. Например, пиво-?варна пасва хем с "пивоварна", хем с "пиво-варна".
Друг квалификатор е {m,n}, където m и n са десетични числа. Той обозначава, че трябва да има поне m на брой повторения, но най-много n. Например, a/{1,3}b ще пасне с "a/b", "a//b", и "a///b". Но няма да пасне с "ab", защото не съдържа наклонени черти, или с "a////b", защото съдържа четири.

Можете да изпуснете някоя от стойностите на m или n - например {,4} или {2,}. В такъв случай, на нейно място се приема някаква разумна стойност. Изпускането на m се тълкува като 0 за долна граница, докато изпускането на n води до установяването на безкрайността като горна граница.

Читателите от редукционалисткия лагер може би са забелязали, че останалите 3 квалификатора също могат да бъдат изразени чрез тази система за означаване. {0,} е същото като *, {1,} е еквивалент на +, и {0,1} е същото като ?. Но е по-добре да използвате *, +, или ?, просто защото те са по-кратки и лесни за четене.

Регулярния израз се задава така $http=~m/^http://([^/]+)(.*)/. В края на него може да се окажат модификатори които повлияват на интерпретирането на самия рефулярен израз - променя неговото поведение. Значението на модификаторите е следното:

i	Игнорира големината на буквите, които се срещат в изследвания стринг.
m	Третира низ като съставен от много редове. Например "one\ntwo".
s	Третира стринга като съставен от един ред. Позволява използването на "." за намирането на символа за нов ред.
x	Игнорира празните места и символа за нов ред в регулярния израз. Позволява писането на коментари.
o	Компилира регулярния израз само веднъж.
g	Открива всички съвпадения в дадения стринг (не спира при първото съвпадение). Ако се постави котвата /G в началото на регулярния израз ще го закотви в крайната точка на последното съвпадение.

Друг начин на писане на коментари в регулярните изрази е чрез (?#). Например:

```
/Mitko (?# This is me) e pich/
```

Ако имате модификатор от който искате да се оттървете временно меже да използвате (?-x)

```
/Mitko ((?-i) e pich)/i;
```

За да погледнем напред, за търсена дума и само тогава стринга да има съвпадение използваме:

```
/Mitko (?=e pich)/i;
```

За негативно използваме:

```
/Mitko (?!e pich)/i;
```

За поглед назад:

```
/Mitko (?<=e pich)/i;
```

Мета символи

.	Съвпада с всеки символ с изключение на нов ред
[..]	Съвпада с всеки символ в скобите.
[^...]	Съвпада с всеки символ, освен с тези в скобите.

Количествени определители

Служат за определение колко пъти може да съвпада един символ.

?	Съвпада с предишния елемент 0 или 1 пъти.	0 < ? < 1
*	Съвпада с предишния елемент 0 или повече пъти.	0 > * > max
+	Съвпада с предишния елемент 1 или повече пъти.	1 > + > max
{число}	Съвпада точно определен брой пъти от числото.	
{min, max}	Съвпада от минимум пъти до максимум, зададени от числата.	

{min,} Съвпада с минимума числа или повече. {,max} Съвпада с максимума числа или по-малко.

За пример:

a.*e

Означава, че се търси стринг започващ с "a" следван от всеки символ 0 или повече пъти следвани от "e". Ще намери думи като "alpine" и "apple". Ако искате да укажете колко цхисла да има след "alpina" :

apple[0-9]{2}

ще намери съвпадение с "alpina" следвана от две числа от 0 до 9, например с "apple34". Ако изкате да укажете някакъв обхват, например да намери "alpina" следвана от три до пет числа от 0 до 9, тогава :

apple[0-9]{3,5}

Съвпаденията ще са например: "apple123", "apple4321", "apple15243", но не и "apple21".

Котви

Специфицират позицията на съвпадението

^	Съвпада с началото на реда (низа).
$	Съвпада с края на реда (низа).
\<	Съвпада с началото на дума.
\>	Съвпада с края на дума.
\b	Търси съвпадение с място между нещо, което не е знак за дума и нещо което е. Граница между знак \w и \w.
\B	Съвпада с всеки символ които не е в началото или в края на думата.

Примери:

Имената на месеците.
Този алгоритъм намира намира англиискиите имена на дванадесетте месеца в пълната или абревиатурна форма.

```
$foo =~ m/
  (?:
    # Абревиатурна форма:
    (?:jan|feb|mar|apr|may|jun|
      jul|aug|sep|oct|nov|dec\.?
    )
    |  # Или...
    # Не абревиатурна:
    (?:january|february|march|april|may|june|july|
              august|september|october|november|december
            )
          )/xi;
```

```
$http = 'http://www.perl.org/index.html';
if ($http=~m#^http://([^/]+)(.*)#) {
print "host = $1\n"; # host = www.perl.org
print "path = $2\n"; # path = /index.html
}
```

$1, $2, $3, ... - скаларни променливи, които съдържат само цифри в името си се използват в

регулярните изрази, при намиране на съвпадение, съвпадението се предава на такава скаларна променлива за да може да се обръщаме към нея и да я извикваме.

```
$ftp = 'ftp://ftp.uu.net/pub/systems';
if ($ftp=~m#^ftp://([^/]+)(/[^/]*)+#) {
print "host = $1\n"; # host = ftp.uu.net
print "fragment = $2\n"; # fragment = /systems
}
```

В горния пример ' (/[^/]*)+ ' - съвпада с /pub и /systems, нищо, че има + и съвпада с /pub/systems, но в скобите за референция е само едно /xxx и се предава само на $2 се предава последното съвпадение.

```
if ($ftp=~m#^ftp://([^/]+)((/[^/]*)+)#) {
print "host = $1\n"; # host = ftp.uu.net
print "path = $2\n"; # path = /pub/systems
print "fragment = $3\n"; # fragment = /systems
}
```

В този случай второто съвпадение е целия път, а третото само последното съвпадение. Външната двойка () за ' (/[^/]*)+ ' е разликата с горния пример, която позволява да се маркира за обратна референция '/pub/systems'. Предаването на съвпаденията започват от външните скоби към вътрешните.

```
if ($ftp=~m#^((http)|(ftp)|(file)):#) {
print "protocol = $1\n"; # protocol = ftp
print "http = $2\n"; # http =
print "ftp = $3\n"; # ftp = ftp
print "file = $4\n"; # file =
print "\$+ = $+\n"; # $+ = ftp
};
```

Намира съвпадението с зададения протокол на URL. Специалната променлива $+ съдържа стойността на последното не празно съвпадение.

```
$a='I am sleepy....snore....DING ! Wake Up!';
if ($a=~/snore/) {
print "Postmatch: $'\n";
print "Prematch: $`\n";
print "Match: $&\n";
}
```

Референции

Основи

Референцията е скалар който посочва мястото в паметта, което съдържа някакъв тип информация. Има 6 типа референции. Дадени са по-долу в таблицата.

Създаване на референция	Дереференция
$refScalar = \$scalar;	${$refScalar}
$refArray = \@array;	@{$refArray}
$refHash = \%hash;	%{$refHash}
$refFunction = \&function;	&{$refFunction}
$refGlob = *FILE;	$refGlob
$refRef = \$refScalar;	${${$refScalar}}

Пример: Използване на DATA

DATA позволява да съхранявате информация само за четене в изпълним файл, която може да ви потрябва и да имате достъп до нея. Позволява да не се отваря друг файл, а да се използва работния, като се използва оператора за четене . Следващият пример показва:

- Прочитане на всички линии които са след __END__
- Преминаване през всички елементи на масива @lines и принтирането им.
- Всичко над линията __END__ е код, всичко под нея е дата.

```
@lines = <DATA>;

foreach (@lines) {
   print("$_");
}

__END__
Line one
Line two
Line three
```

Тази програма извежда като резултат следното:

```
Line one
Line two
Line three
```

модулът Benchmark

Както много други неща в Perl, един от най-добрите начини да решите как да направите скрипта Ви да работи по-бързо е да напишете няколко алтернативни кода и да видите кой от тях се изпълнява най-бързо с модула Benchmark.

```
#!perl
use Benchmark;

open (AA,"access.log");
@data = <AA>;

my $host;
timethese (100,
{ mem => q{
for (@data) {
($host) = m/(\w+(\.\w+)+)/; }
},

memfree => q{
for (@data) {
($host) = m/(\w+(?:\.\w+)+)/; }
}
}
);
```

И резултата:
Benchmark: timing 100 iterations of mem, memfree...
mem: 3 wallclock secs (3.24 usr 0.00 sys = 3.24 cpu)
memfree: 3 wallclock secs (2.63 usr 0.00 sys = 2.43 cpu)
Не лош резултат: вторият код който не иска попълване за обратна референция е с близо 25 % по-бърз

от първия, в който съвпаденията се запомнят за обратна референция.

Синтаксиса е следния:

```
timethis ($count, "code");
# Use Perl code in strings...
timethese($count, {
   'Name1' => '...code1...',
   'Name2' => '...code2...',
});
# ... or use subroutine references.
timethese($count, {
   'Name1' => sub { ...code1... },
   'Name2' => sub { ...code2... },
});
            # cmpthese can be used both ways as well
cmpthese($count, {
   'Name1' => '...code1...',
   'Name2' => '...code2...',
});
cmpthese($count, {
   'Name1' => sub { ...code1... },
   'Name2' => sub { ...code2... },
});
# ...or in two stages
$results = timethese($count,
   {
      'Name1' => sub { ...code1... },
      'Name2' => sub { ...code2... },
   },
   'none'
);
cmpthese( $results ) ;
$t = timeit($count, '...other code...')
print "$count loops of other code took:",timestr($t),"\n";
$t = countit($time, '...other code...')
$count = $t->iters ;
print "$count loops of other code took:",timestr($t),"\n";
```

timethis - изпълнява даден код няколко пъти
timethis (брой изпълнение, "код")
timethese - изпълнява няколко парчета код няколко пъти
cmpthese - изкарва резултата от изпълнението на timethese като сравнителна графика
timeit - изпълнява даден код и определя колко време продължава това
countit - дава колко пъти се е изпълнил даден код за зададено време

модулът Win32::ODBC, SQL и базите данни

По-долу съм дал напълно работещ пример за работа с бази данни. Постарал съм се, кода по-долу да е ясно написан и описан. Предполагам, че имате познания по SQL или сте прочели ръководството ми за SQL. Тук ще намерите допълнение към него. Може да копирате кода и да го стартирате така като е.

```
#!perl

use warnings;
use CGI::Carp qw(fatalsToBrowser);              #Ако има грешки в скрипта, съобщенията за тях от
# командния ред се извеждат директно на браузъра, а не се записват в error.log на Apache, като се
# изпълняват  запитвания на  сървъра
use Win32::ODBC;            #Включване на модула за работа с бази данни на Win32  операционна
# система

print "Content-type: text/html\n\n";
print <<EOF;
<head><title>Perl from $HeadHunter_Sid</title>
            <META NAME="Author" CONTENT="$HeadHunter_Sid - Dimitar Mihailov">
            <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=windows-1251"><style>
BODY { BACKGROUND: #ffffff;FONT-FAMILY: Verdana, Arial;          FONT-WEIGHT:
normal;font-size : 12px;}
A:link {  COLOR: #b82619; TEXT-DECORATION: underline}
A:visited {             COLOR: #80764f; TEXT-DECORATION: underline}
A:hover {               COLOR: #000000; TEXT-DECORATION: underline}
A:active {              COLOR: #b82619; TEXT-DECORATION: underline}</style></head><body>
EOF

%drivers=Win32::ODBC::Drivers();                #Създава хеш с инсталираните драйвъри на машината

# 1. Има два начина на създаване на връзка към БД, Конфигурация без Perl (външно) - 1.1
# и чрез Perl - 1.2
# 1.1. Създаване на нов обект за връзка с бази данни - new(bon) - в скобите трябва да се
# попълни със създаден  драйвер за база данни, това става от Settings -> Control Panel -> ODBC32
# $base=Win32::ODBC->new(bon);
#
# 1.2.
$                                       |         1.2.1            |                 1.2.2                        |
# Win32::ODBC::ConfigDSN(ODBC_ADD_DSN,"Microsoft Access Driver (*.mdb)",
# |    1.2.3       |          1.2.4             |          1.2.5              | 1.2.6  |   1.2.7   |
# ("DSN=bon","Description=Tables","DBQ=C:/bons.mdb","UID=","PWD="));
#
# $base=Win32::ODBC->new(bon); # Създаването на новия обект
#
# 1.2.1. За конфигуриране на DSN (името под което ще се вика БД
# 1.2.2. Двайвера за БД - Access; Oracle или др.
# 1.2.3. DSN  името се задава
# 1.2.4. Малко обяснение на БД
# 1.2.5. Локалния път до самата БД
# 1.2.6 и 1.2.7 ако има user name и password за достъп до БД
#



Win32::ODBC::ConfigDSN(ODBC_ADD_DSN,"Microsoft Access Driver
(*.mdb)",("DSN=bon","Description=Tables","DBQ=C:/db.mdb"));
$base=Win32::ODBC->new(bon);
if (!$base) {
            Win32::ODBC::DumpError();           # Задължително трябва да се добавя този блок за да
```

```
#  съобщава ако има  грешки (в случая ако няма създаден достъп до БД)
        die;
}

$base->Connection();                    # Свързване с БД

if ($base->Sql("Create table bonbions (edno char (22),dve char (123))")) {# Между
# "" се пише Sql заявката която се обработва от БД и се връща резултата; В случая създавам таблица с
# една колона
        Win32::ODBC::DumpError();       # Ако заявката не се изпълни показва се грешката
        # и се прекратява скрипта
        die;                            # Забележи че няма ! пред $base, в случаягрешното изпълнение е
        # вярно и се изпълнява if конструкцията
}

if ($base->Sql("Create Table employees (firstname char, age integer not null unique primary key)"))
        {       Win32::ODBC::DumpError();       die;}

if ($base->Sql("Delete from shipping where CarrierID>1 "))
        {       Win32::ODBC::DumpError();       die;}

if ($base->Sql("Update bonbons set [Chocolate Type]='Yellow' where [nut type]='none'"))
        {       Win32::ODBC::DumpError();       die;}

if ($base->Sql("Select [bonbon name],[nut type],[filling type] from bonbons order by [nut type]"))
        {       Win32::ODBC::DumpError();       die;}
print "<table>";
while ($base->FetchRow()) {
        ($a,$b,$c)=$base->Data('bonbon name','nut type','filling type');
        print "<tr><td>$a</td><td>$b</td><td>$c</td></tr>";
}
print "</table>";

while ($base->FetchRow()) {
        %col=$base->DataHash();
        foreach  (keys %col) {
                print "$_  -- $col{$_}<br>";
        }
        print "<hr>";
}

@mass_danni=qw/sad dsf ds fdg dfh g t trg df gh ty ht y hrsgsrt htr h df g trd htr g dsv fd gsf h/;
foreach  (@mass_danni) {
        if ($base->Sql("Insert into bonbions (edno) Values ('$_')")) {          # Попълва полето edno на
# таблицата bonibons със стойностите на масива; Задължително подаденията стойности на sql заявката
# трябва да са в кавички



        Win32::ODBC::DumpError();       # Ако заявката не се изпълни показва се
        # грешката и се прекратява скрипта
        die;
        }
}
```

```
if ($base->Sql("drop table bb")) {      Win32::ODBC::DumpError();  die;}    # Изтрива таблицата bb

@TableList=$base->TableList;          # Присвоява на @TableList списък с таблиците в БД

$base->Close();                       # Задължително трябва накрая да се прекъсне
# връзката с БД иначе БД може да отиде на кино или някъде другаде
```

Функции използвани върху скалари или стрингове

chomp

```
chomp VARIABLE
chomp LIST
chomp
```

Тази безопасна форма на chop премахва само стойността зададена от специалната променлива $/ (още позната като $INPUT_RECORD_SEPARATOR в Англииския модул), съдържаща символа, който се приема за знак означаващ последен ред, който по подразбиране е '\n'. Връща броя на всички премахнати $/ . Използва се за премахване на новия ред от края на въведена информация. Ако променливата не е зададена, chomp действа на $_. Пример:

```
while (<>) {
chomp; # премахва \n от всеки ред
@array = split(/:/);
# ...
}
```

Може да премахвате $/ от всяка въведена стойност:

```
chomp($cwd = `pwd`); chomp($answer = );
```

Ако прилагате chomp върху списък, ще бъде върнат като отговор общия брой на премахнатите $/ .

chop

```
chop VARIABLE
chop LIST
chop
```

Премахва последния символ от стринг и го връща. Може да се използва и за премахване на нов ред, но е много по-ефективен от s/\n//, защото не сканира стринга. Ако променливата не е зададена, chop действа на $_.

```
$text="Long text";
while ($text) { # докато има текст ще се изпечатва буква по буква
chop; # премахва последния символ
print $_,"\n" # изпечатва го;
}
```

Друг пример:

```
chop($cwd = `pwd`);
chop($answer = );
```

chr

chr NUMBER
chr

Връща символа представен от число в ASCII таблица. За пример chr(65) е "A" в ASCII. За обратната операция, използваите функцията ord. Ако променливата не е зададена, chr действа на $_.

hex

hex EXPR
hex

Интерпретира EXPR като шестнадесетично число и връща кореспондиращата му стойност. За преобразуване на стрингове започващи с 0, 0x или 0b, вижте функцията oct . Ако EXPR не е зададена, hex действа на $_.

```
print hex '0xAf'; # показва на екрана '175'
print hex 'aF'; # същото
```

index

index string, substring, position
index string, substring

Връща нулево-базираната позиция на подниза substring в низа string, която се намира след зададената в position позиция. Ако не е зададена се приема, че е равна на нула. Връща -1 ако не е открито съвпадение.

```
$text="That Perl is an easy";
# 012345678.....
$pos=index $text,"Perl";
print $pos
```

Резултат: 5

```
$pos=index $text,"Perl",12;
print $pos
```

Резултат: -1. Започва търсенето от интервала пред an и затова не намира съвпадение

lc

lc EXPR
lc

Връща EXPR но с малки букви. Тази функция е аналогична на \L ескейпа в стринг затворен с двойни кавички- "\L string".

```
chomp ($a=<STDIN>);
print lc $a;
```

При вход Perl, на екрана ще бъде изписано perl.

lcfirst

lcfirst EXPR
lcfirst

Връща EXPR но с първа малка буква. Тази функция е аналогична на \l ескейпа в стринг затворен с двойни кавички- "\l string".

```
chomp ($a=<STDIN>);
print lcfirst $a;
```

При вход PERL, на екрана ще бъде изписано pERL.

length

length EXPR
length

Връща броя на символите в EXPR. Ако EXPR е изпуснат, връща броят на символите в $_. Тази функция не може да се използва за да се намери от колко елемента се състой даден масив или хеш. За това трябва да се ползва scalar @array и scalar keys %hesh.

```
print length "asdas";
```

Ще изведе като отговор 5, защото в стринга има 5 символа.

reverse

reverse LIST

В списъчен контекст, връща елементите на списъка, но от отзад напред. В скаларен контекст, връща символите на стринга в обратен ред.

```
$a="perl"
print reverse $a; #изпечатва: lrep
```

Тази функция може да се използва и върху масив и хеш. При използването му върху хеш, специфичното е, че ако има две еднакви стойности който при прилагането на функцията вече стават ключове, поради факта, че всеки ключ трябва да е уникален, едната стойност ще се загуби.

```
%by_name = reverse %by_address;
```

substr

substr EXPR,POSITION,LENGHT,REPLACEMENT
substr EXPR,POSITION,LENGHT
substr EXPR,POSITION

Връща подниз на низа EXPR с дължина length знака, като започва от знака с номер position. Първата позиция е 0 (от нея започва броенето) освен ако не сте я променили чрез $[(не се препоръчва да го правите).

uc

uc EXPR
uc

Връща EXPR, но с големи букви. Тази функция е аналогична на \U ескейпа в стринг затворен с двойни кавички- "\U string".

```
chomp ($a=<STDIN>);
print uc $a;
```

ucfirst

ucfirst EXPR
ucfirst

Връща EXPR но с първа голяма буква. Тази функция е аналогична на \u ескейпа в стринг затворен с двойни кавички- "\u string".

```
chomp ($a=<STDIN>);
print ucfirst $a;
```

При вход perl, на екрана ще бъде изписано Perl.

rand

rand EXPR
rand

Връща случайно дробно число по-голямо или равно от 0 и по-малко от стойността зададена с EXPR. (EXPR трябва да е положително число). Ако EXPR е пропуснат, използва се стойност 1. Автоматично извиква функцията srand() освен ако srand() не е била извикана вече.

```
$a=rand; # $a ще е със стойност от 0 до 1
$a=rand 100; # $a ще е със стойност от 0 до 100, за стойност може да се задава масив, скалар
```

int

int EXPR
int

Не се използва за закръгляване. Подадено дробно число го превръща в цяло. Ако му се подаде число 2.85464 ще върне 2.

```
$a=int rand 100; # rand ще върне число от 0 до 100, например 43.966723532, а int ще върне 43
```

. . .

Copyright 2001 - $HeadHunter_Sid - Dimitar Mihailov

mitko_ddt@yahoo.com