Perl

Practical Extraction and Report Language

Jazyk pro snadné zpracování textových dat; problém mu nečiní binární data ani jejich velikost; formátovaný výstup; regulární výrazy; asociativní pole; přístup k systémovým funkcím; funkce pro práci se sítí; objektové moduly a knihovny funkcí, existují pro celou řadu úkolů; nástroje na volání kódu psaného v jiných jazycích.

Zdroje informací:

```
man perl
http://www.perl.com/
comp.lang.perl.misc
```

CPAN:

```
http://www.fi.muni.cz/ftp/pub/cpan/
```

V lokální instalaci na FI je perl uložen v adresářích:

```
/packages/(run|share)/perl-5.004_04
```

Manuálové stránky zpřístupníme příkazem:

```
module add perl5
```

Manová stránka s vyhledáváním je na URL:

```
http://www.fi.muni.cz/tech/perldoc/
```

Aktuální verze perlu:

```
perl-5.005_03
```

Text k těmto slajdům:

```
http://www.fi.muni.cz/~adelton/perl/
```

Program Hello world

Přímo na příkazové řádce zadáme za parametr `-e` kus kódu:

```
$ perl -e 'print "Hello, world\n";'
```

V souboru píšeme příkazy skriptu a na prvním řádku můžeme uvést cestu k perlu:

```
$ cat test.pl
#!/usr/bin/perl
print "Hello, world\n";
```

Skript by měl mít práva pro provádění a také pro čtení,

```
chmod +x test.pl
```

spustíme ho buď `./test.pl` či `test.pl`, nebo `perl test.pl`, velmi užitečná je volba `-w`.

Umíme tedy tisknout. Nyní načteme řádek ze standardního vstupu do proměnné a použijeme ji:

```
$ perl -e 'print "What is your name? "; $name = <>;
                        print "Hello, ", $name;'
What is your name? adelton
Hello, adelton
```

Zajímavost, které si můžeme všimnout, je fakt, že Perl samovolně neodřezává na vstupu znak konce řádku. To můžeme udělat například funkcí `chomp`.

Jednoduchý grep

Do proměnné `$search` uložíme první argument skriptu, v cyklu `while` čteme operátorem `<>` ze vstupu (`STDIN` nebo ze souborů uvedených jako argumenty skriptu), implicitně do proměnné `$_`, a řádek (obsah `$_`) vytiskneme, pokud uspěje regulární výraz hledání v řetezci.

```
#!/usr/bin/perl
$search = shift @ARGV;
while (<>) {
        print if /$search/;
        }
```

Práva pro provádění:

```
$ chmod +x grep.pl
$ grep.pl if < grep.pl
```

výsledek

```
$search = shift @ARGV;
        print if /$search/;
```

Jiná varianta, pro céčkové programátory:

```
#!/usr/bin/perl -w
$search = $ARGV[0];
while (defined($line = <>)) {
        if ($line =~ /$search/) {
                print $line;
                }
        }
```

Regulární výraz můžeme samozřejmě vložit i přímo do příkazu:

```
$ perl -ne 'print if /a.*b/;'
```

Cykly, regulární výrazy, substituce

Cyklem přes `while`

```
$ perl -e 'while (<>)
              { s/:M012:3:/:M012:4:/; print; }'
```

čteme jednotlivé řádky a nahrazujeme trojku čtverkou, stejně jako příkaz `sed 's/:M012:3:/:M012:4:/'` .

Cyklus pomocí voleb `-n` a `-p`, automatické cyklení nad standardním vstupem, příp. ještě tisk:

```
$ perl -ne 's/:M012:3:/:M012:4:/; print;'
$ perl -pe 's/:M012:3:/:M012:4:/;'
```

Můžeme použít další podmínky, např. proměnná `$.` obsahuje vždy číslo řádku posledně čteného filehandlu:

```
$ perl -ne 's/:M012:3:/:M012:4:/ if /::/;
                  print if $. > 5 and $. < 20;'
```

In-place náhrada — uloží výsledek do souborů, odkud četl, `-i.bak` by navíc zajistilo uchování starých verzí souborů:

```
$ perl -i -pe 's/:M012:3:/:M012:4:/;' *
```

Číslování řádků vstupu:

```
$ perl -ne 'BEGIN { $i = 1; } print $i++, ": $_";'

$ perl -ne 'printf "%d: %s", ++$i, $_;'

$ perl -pe 'print $., ": ";'
```

Poslední příklad ukazuje, že Perl samovolně nepřidává znak nového řádku. (Ale jde to nastavit, proměnná `$\` nebo volba `-l`.) My příkazem `print` tiskneme číslo a dvojtečku, volba `-p` zajistí tisk `$_`.

Backreference — odkazuje na nalezený podřetězec. Proměnná `$&` obsahuje, co bylo nalezeno v posledním hledání, v `$+` je poslední podřetězec, který byl nalezen regulárním výrazem uzavřeným do závorek, krátce obsah namatchovaných závorek.

```
$ perl -e '$_ = <>; s/\w/$&:/g; print;'
General protection
G:e:n:e:r:a:l: p:r:o:t:e:c:t:i:o:n:
```

```
$ perl -e '$_ = <>; s/e(\w)/e$+$+/g; print;'
Letter of agreement
Lettterr of agreeemennt
```

V regulárních výrazech můžeme kromě konstrukcí `.`, `.*`, `.?` nebo `[a-h]` použít i `\w`, `\d`, `\s`, `\S`, `\B` a mnohé další.

```
$ echo $LC_CTYPE
cs
$ perl -e 'use locale; $_ = <>; s/(\w)\B/$1:/g; print;'
maličký ježeček
m:a:l:i:č:k:ý j:e:ž:e:č:e:k
```

`\B` znamená nehranici mezi slovy a `$1` obsahuje řetězec z prvních závorek v regulárním výrazu.

Perl se chová správně podle nastavených locales, musíme mu to ale říci direktivou `use locale`:

```
$ ( LC_CTYPE=C; perl -e 'use locale;
                $_ = <>; s/(\w)\B/$1:/g; print;' )
maličký ježeček
m:a:l:ičký j:ežeče:k
```

V US-ASCII nejsou akcentované znaky považovány za součást slova, proto tento výsledek.

Lookahead — dovoluje podívat se, co je v řetězci dále vpravo, ale nebere tuto část jako součást nalezeného podřetězce:

```
s/(\w)(\w)/$1:$2/g;
m:al:ič:ký j:ež:eč:ek
```

po nalezení `ma` a vložení dvojtečky pokračuje hledání od `l`, což zde ale nechceme. Kód

```
s/(\w)(?=\w)/$1:/g;
m:a:l:i:č:k:ý j:e:ž:e:č:e:k
```

již pracuje správně, pokračuje od druhého nalezeného písmene.

Význam hvězdičky a plus, které znamenají maximální možný počet opakování, je možné otočit přidáním otazníku. Například tento rušič poznámek

```
#!/usr/bin/perl -w
while (<>) {
        $longer and s!^.*?\*/!!
                      and $longer = 0;
        $longer and $_ = '';
        s!/\*.*?\*/!!g;
        s!/\*.*!! and $longer = 1;
        print;
        }
```

pracuje správně i v situaci

```
    /* Poznámka */   i = 3; /*/ Další poznámka
                    */
```

protože hledání `.*?\*/` se zastaví hned, jak je to jen možné, není hladové.

Vidíme také, že regulární výrazy nemusíme uvozovat jen lomítky, čehož využijeme např. při zpracování jmen souborů a adresářů.

Pole, seznamy, hashe

Slova ve vstupním souboru můžeme spočítat třeba

```
use locale;
while (<>) {
        @a = split /\W+/, $_;
        $total += scalar @a;
        }
print "Slov celkem: $total\n";
```

Funkce `split` rozdělí řetězec podle zadaného regulárního výrazu a vrátí seznam podřetězců mezi takovými dělítky. Tento seznam zde přiřazujeme do pole (začíná zavináčem). Pole ve skalárním kontextu (kontext je určený typem vlevo od rovnítka) vrací počet svých prvků a ten přičítáme k celkovému součtu `$total`.

Rozdělování pomocí `splitu` se často používá, pokud zpracováváme soubor s údaji ve sloupcích. Soubor s daty

```
Brouk   Pytlík          157
Ferda   Mravenec        156
Víla    Amálka          152
```

bychom četli například cyklem `while` a jednotlivé položky dostaneme příkazem

```
($jmeno, $prijmeni, $tel) = split /\s+/, $line;
```

Vlevo je seznam skalárů, `split` vrací seznam, takže toto přiřazení je v pořádku.

Pokud bychom ale chtěli použít `split` na zpracování logu http daemona, narazili bychom na fakt, že mezery jsou zde i uvnitř jednotlivých položek:

```
nemesis.fi.muni.cz - - [02/Nov/1997:15:57:32 +0100] \
                "GET /~adelton/perl/ HTTP/1.0" 200 3831
```

Operátor vyhledání regulárního výrazu, který ve skalárním kontextu vratí true/false, vrací v seznamovém kontextu seznam podřetězců ze závorek regulárního výrazu. Tedy na http log můžeme jít příkazem

```
($host, $ident, $user, $date, $command, $status, $length) =
    /^(.+?) (.+?) (.+?) \[(.*?)\] "(.*?)" (\d+|-) (\d+|-)/;
print "Host:\t$host\nCom:\t$command\nStatus:\t$status\n";
```

Prvky seznamu na levé straně naplníme podřetězci vzešlými z regulárního výrazu.

Takto bychom mohli získat z cesty k souboru jeho jméno, bez adresáře:

```
$filename = "/home/adelton/perl/XBase/XBase.pm";
($basename) = $filename =~ m!([^/]+$)!;
```

Následující příklad rozdělí vstup podle dvojteček, spojí ho zpět pomocí znaků nového řádku a výsledný řetězec vytiskne. Všimněme si, jak se z řetězce stává seznam a z něho zpět řetězec.

```
$ perl -e 'print join "\n",
                split /:/, "34:Record:č:3\n";'
34
Record
č
3
```

Pokud máme vstup, ze kterého chceme získat jen čísla, můžeme se inspirovat tímto skriptem:

```
perl -ne 'print "#Next line\n";
        for (/(\d+)/g) { print ++$i, ": $_\n"; }'
```

Vyhledání regulárního vzorku s volbou `/g` prohledá postupně celý řetězec. Podčásti ze závorek se stanou prvky jednoho dlouhého seznamu. Tento seznam projdeme pomocí `for`, jednotlivé prvky se dočasně objeví v proměnné `$_`.

K prvkům seznamů i polí můžeme přistupovat nejen iterativně přes `for`, ale například i pomocí `shift`, `pop`, na konec přidáváme pomocí `push`. Tyto funkce mění obsah pole. Jednotlivé prvky můžeme odkázat i přímo, indexem (defaultně od nuly):

```
@a = (3, 5, 6, 2, 4);
print $a[2], "\n";
```

a je možná i interpolace uvnitř řetězce: `print "$a[2]\n"`.

Funkce `localtime` vrací seznam hodnot s významem:

```
($sec, $min, $hour, $mday, $mon, $year,
        $wday, $yday, $isdst) = localtime(time);
```

Na vytištění data můžeme použít třeba

```
($mday, $mon, $year) = (localtime(time))[3 .. 5];
print $mday, '. ', $mon+1, '. ', $year+1900, "\n";
```

kde nejprve přiřadíme část seznamu z `localtime` do seznamu na levé straně a pak vytiskneme správně posunuté hodnoty proměnných. Nebo použijeme pole:

```
@time = localtime(time);
print $time[3], '. ', $time[4]+1, '. ',
                        $time[5]+1900, "\n";
```

Jiná variace:

```
@time = localtime(time);
$time[4]++;
$time[5] += 1900;
print join(". ", @time[3,4,5]), "\n";
```

Zajímavá úloha, při které použijeme asociativní pole nebo-li hash, je spočítat, kolik je na vstupu kterých slov:

```
use locale;
while (<>) {
        for $word (split /\W+/) {
                $counts{$word}++ if $word;
                }
        }
for (sort keys %counts) {
        print "$_: $counts{$_}\n";
        }
```

Hash `%counts` obsahuje prvky, které nejsou indexovány pořadím, ale řetězcem — ke každému slovu je jeho hodnota počet výskytů. Cyklus `for $word (...` projde seznam vrácený `split`em, jeho prvky přiřazuje do proměnné `$word` a k danému slovu vždy zvýší odpovídající hodnotu v asociativním poli. Funkce `sort` setřídí seznam klíčů (`keys`) z hashe a vytiskne statistiku.

`Sort`u můžeme změnit způsob třídění:

```
for (sort {$counts{$b} <=> $counts{$a}}
                                keys %counts) {
        printf "%5d: %s\n", $counts{$_}, $_;
        }
```

— zadáváme zde alternativní kousek kódu, který se bude při třídění volat na porovnání dvou hodnot (předaných v `$a` a `$b`).

I proměnné prostředí jsou vlastně hash a v Perlu jsou přístupné jako hash `%ENV`:

```
for (sort keys %ENV) {
        print "$_=$ENV{$_}\n";
        }
```

Toto je jednoduchá verze `whereis`, která vychází z obsahu `%ENV`:

```
(@ARGV) or die "Usage: whereis substr\n";
$string = shift;
@path = ();
@path = split /:/, $ENV{PATH} if defined $ENV{PATH};
push @path, split /:/, $ENV{MANPATH}
                             if defined $ENV{MANPATH};
unless (@path) {
        die "No PATH or MANPATH found\n";
        }
for $dir (@path) {
        next unless -d $dir;
        for (<$dir/$string*>) {
                print "$_\n";
                }
        }
```

Operátor `<>` zde nečte ze souboru (žádný jsme nespecifikovali), ale vrací seznam souborů po rozexpandování shellem.

Hash můžeme využít například na vyhledání duplicitních záznamů v `/etc/passwd`:

```
#!/usr/bin/perl -w
open FILE, "/etc/passwd" or die "Error reading /etc/passwd: $!";
while (<FILE>) {
    ($login, undef, $uid) = split /:/;
    if (defined $passwd{$login}) {
        print "Login $login has uids $passwd{$login} and $uid\n";
        }
    else { $passwd{$login} = $uid; }
    }
close FILE;
```

`Die` ukončí skript s chybovým hlášením na `STDERR`, magická proměnná `$!` obsahuje hlášení od systému, například `No such file or directory`. Poznamenáme si první a třetí položku na řádku. Pokud se login objevil v souboru už dříve, najdeme ho v hashi `%passwd` a vytiskneme hlášení.

Soubory, externí programy

Již jsme viděli, jak otevřít soubor pro čtení; při zápisu do souboru použijeme notaci známou z shellu:

```
open OUT, "> /tmp/process.log.$$" or
        die "Error writing logfile: $!\n";
print OUT $line;
```

a samozřejmě pro připojování na konec `>>`. Podobně funguje také spouštění externích programů, zápis do roury by byl

```
open MAIL, "| mail $user"
print MAIL "Vážený uživateli $user, ...";
close MAIL;
```

Zde poznamenejme, že takovýto příkaz je zpracován shellem, proto bychom měli velice přesně vědět, co je v proměnné `$user` uloženo. Čtení z roury je naopak

```
open PASSWD, "ypcat |" or die ...
```

Stejný význam jako v shellu má i operátor obráceného apostrofu.

Funkce

Funkce dovolují definovat opakovaně používaný kód. Obecně pracují nad seznamem a vracejí seznam. Uvnitř funkce se parametry objeví v poli `@_`. Definujme funkci `sum`, která sečte prvky pole:

```
sub sum {
        my ($total, $elem) = (0, 0);
        for $elem (@_) { $total += $elem; }
        $total;
        }
print sum(3, 7, 8), "\n";
```

Návratová hodnota funkce je hodnota posledního provedeného výrazu, funkci ale můžeme ukončit také pomocí `return`. Pomocí `my` definujeme lokální proměnnou s lexikálním rozsahem platnosti (jako auto proměnné v C).

Funkcím jsou parametry standardně předány odkazem:

```
@a = (2, 4, 6);
sub inc {
        for (@_) { $_++; }
        }
inc(@a);
print "@a\n";
```

lokální kopii si zajistíme pomocí `my`, případně `local` (dynamický scoping, uloží kopii a při návratu hodnotu obnoví):

```
sub inc {
        my @data = @_;
        for (@data) { $_++; }
        @data;
        }
@a = inc @a;
```

Reference

Skalární proměnné (a tedy i prvky polí a hashů) mohou mít jako hodnoty kromě řetězců a čísel i reference (adresy, ukazatele).

```
$a = 56; $b = \$a;
print "$a, $$b, ";
$$b++;
print "$$b\n";
56, 56, 57
```

Podobně pomocí backslashe získáme referenci na pole či na hash: `$arref = \@array`, `$hashref = \%hash`. Anonymní pole vytvoříme pomocí `[ ]`, anonymní hash pomocí `{ }`. Tyto operátory vrací referenci na pole, resp. hash. Nyní nám již nic nebrání vytvářet složité struktury jako pole hashů (ve skutečnosti pole referencí na hash), hash polí, hash hashů polí referencí na funkci, atp.

```
my $friends = { 'Peter' => { 'height' => 179, 'weight' => 69 },
        'David' => { 'height' => 188, 'weight' => 85 },
        'Ludvík' => { 'height' => 183, 'weight' => 72,
                        'children' => [ 'Ludvíček', 'Martina' ],
                        'wife' => 'yes', }
                };
print $friends->{'Peter'}{'height'}, "\n";
print "@{$friends->{'Ludvík'}{'children'}}\n";
```

a můžeme se hned také ptát

```
if (defined $friends->{'Peter'}{'children'})
        { print "good\n"; }
```

Šipka `=>` v definici hashe je syntaktickým cukrem místo čárky, šipka `->` znamená dereferenci, protože `$friends` není hash, ale reference na něho. Takovéto složené objekty se v Perlu používají místo struktur, unií a recordů, které známe z typovaných jazyků. Jsou také základním kamenem při tvorbě Perlovských objektů, protože jsou schopny nést jejich datové atributy.

Package, moduly, třídy, objekty

Package oddělují prostory jmen, takže nedochází ke kolizím mezi programy a knihovnami, napsanými různými lidmi. Package, který je definován ve vlastním souboru odpovídajícího jména, se nazývá modul. V jednom skriptu můžeme použít zároveň modul na zpracování CGI požadavků, modul pro práci s MIME formáty a třeba modul pro práci s databází, a díky odlišení prostoru jmen se vzájemně negativně neovlivňují. Package funguje jako třída, pokud jeho funkce jsou schopny pracovat jako metody a funkce funguje jako metoda, pokud jako svůj první parametr akceptuje referenci na objekt. Objekt je pak něco příslušné k nějakému package.

```
use Cz::Cstocs;                     # použijeme modul
my $il2_ascii = new Cz::Cstocs 'il2', 'ascii';
                                    # vytvoříme objekt
while (<>) { print $il2_ascii->conv($_); }
                                    # zavoláme metodu
```

Tento příklad ukazuje použití modulu `Cz::Cstocs`. Objekt, který vytvoříme voláním konstruktoru `new` dává k dispozici metodu, která konvertuje řetězce.

Další příklad ukazuje načítání MIME zpráv:

```
use MIME::Parser;
my $parser = new MIME::Parser;
my $entity;
while ($entity = $parser->read(\*STDIN))
        {
        my $head = $entity->head;
        $head->unfold(); $head->decode();
        print 'From: ', $head->get('From'),
                'Subject: ', $head->get('Subject');
...
```

Konstruktor se obvykle jmenuje `new`, ale je zcela na autorovi modulu, jaké jméno zvolí (Perl v tomto směru nic nepředepisuje). Vidíme také, že objekt (zde typu `MIME::Entity`) může vzniknout voláním metody jiméno objektu. I když to na první pohled z kódu nemusí být zřejmé, jak `$entity`, tak `$head` jsou reference na objekty (pokud jde tedy vše dobře, také mohou mít nedefinovanou hodnotu). Operátorem `ref` typ objektu zjistíme, vrací `undef`, pokud skalár referenci neobsahuje.

Na závěr příklad zpracování http požadavku:

```
use CGI;
$q = new CGI;
print $q->header('text/html; charset=iso-8859-2');
print $q->start_html("Automobily");
print "Vítáme příznivce škodovky.<P>\n"
                    if $q->param('auto') eq 'Škoda';
print $q->startform, "Vaše jméno", "<BR>",
                       $q->textfield('name'), "<P>";
print $q->radio_group('-name' => 'auto',
     '-values' => ['žádné', "Škoda", "VW", "lepší"],
     '-default' => "žádné");
print $q->submit('Action', 'Odeslat');
print "<P> Proměnné: <P>", $q->dump;
print $q->endform;
print $q->end_html;
```

Konstruktor načte vstup a parametry a dále můžeme objekt použít jednak k testování parametrů, jednak k vytvoření výstupního html (či jiného) souboru.