PB173 Perl

04 Referencie

Roman Lacko xlacko1@fi.muni.cz

2023-03-06

Obsah

Pokročilé zoznamové operátory

Zoznamové operátory: ‹sort›

```
sort LIST
sort { BLOCK } LIST
```

- Zoradí prvky zoznamu podľa výsledku porovnania v prvom argumente. Očakáva sa ‹< 0›, ‹0› alebo ‹> 0› (ako ‹strcmp()› alebo ‹qsort()› v C).
- ‹BLOCK› dostane prvky v lexikálnych premenných ‹$a› a ‹$b›.

```
sort { $a <=> $b } (11, 10, 1, 5);
```

- Ak ‹BLOCK› neuvedieme, ‹sort› zoradí prvky lexikograficky (‹cmp›).

Aký algoritmus ‹sort› používa? Dá sa zmeniť?
‹perldoc sort›, ‹perldoc perl58delta›

Zoznamové operátory: ‹sort›

```
sort FUNCTION LIST
```

- ‹FUNCTION› je názov funkcie bez ‹()›.
- Prvky dostane v *globálnych* premenných ‹$a›, ‹$b›.

‹sort› podá funkcii parametre civilizovane, ak má *prototyp* ‹($$)›.
(Prototyp ≠ signatúra!)

Preferujte radšej ‹sort { BLOCK } LIST›.

```
sort { book_cmp($a, $b) } $books->@*
```

Zoznamové operátory: ‹grep›

```
grep { BLOCK } LIST
grep EXPR, LIST
```

- V zoznamovom kontexte vráti hodnoty, pre ktoré je ‹BLOCK› pravdivé, alebo vyhovujú regulárnemu výrazu v ‹EXPR›.
- V skalárnom kontexte vráti počet vyhovujúcich hodnôt.
- Hodnotu testovanej premennej dostaneme v ‹$_›.

```
grep { defined } @numbers;      # Same as ‹defined $_›
grep !/^A/i, @strings;          # Strings not starting with ‹A›.
```

Regulárne výrazy budú podrobne vysvetlené neskôr.

Zoznamové operátory: ‹map›

```
map { BLOCK } LIS
map EXPR, LIST
```

- Podobné vlastnosti ako ‹`grep`›, ale aplikuje na prvky transformáciu popísanú v ‹`BLOCK`› alebo ‹`EXPR`›, a vráti nový zoznam.

```
map { $_ ** 2 } @numbers;
map s/(\w)\1/#/gr, @strings;
```

Referencie

Referencie

Opakovanie z minulého seminára.

Referencie sú skalárne hodnoty, ktoré odkazujú na iné hodnoty.

```
my $scalar = \5;
say $scalar->$*;

my $array = [1 .. 10];
say $array->[1];
say $array->@[2, 7];
```

Referencie: Autovivifikácia

Použitie ‹undef› hodnoty pri dereferencii v kontexte, ktorý predpokladá jej existenciu, túto hodnotu vytvorí.

```
my $books;

$books->[42]{title} = "A Hitchiker's Guide to the Galaxy";
$books->[666]{chapters}[0]{demons}++;
```

Takto je možné ušetriť si vytváranie medziľahlých štruktúr:

```
# $books //= [];                                  # Unnecessary.
# $books->[37] //= {};                            # Unnecessary.
$books->[37]->{title} = "A Game of Thrones";      # This is enough.
```

Referencie: Autovivifikácia

Niekedy však so sebou nesie nečakané problémy:

```
my $books;

delete $books->[42]->{title};
say scalar $books->@*;          # 43!
```

Podobné správanie spôsobí aj ‹exists› alebo ‹defined›.

Referencie: Autovivifikácia

Prekvapenie môže priniesť aj oživenie parametra vo funkcii:

```
sub autovivify($ref) {
    exists $ref->[0]->[0];
}

my $a;
autovivify($a);
say defined $a->[0];        # No.

my $b = [];
autovivify($b);
say defined $b->[0];        # Yes.
```

Odbočka: ‹undef› v číselnom kontexte

Niektoré iné aspekty Perlu sa chovajú podobne ako autovivifikácia, aj keď ňou striktne nie sú.

Napríklad operátory ‹+=›, ‹-=›, ‹++› a ‹--›.

```
my ($a, $b, ...);

say $a += 1;        # 1, no warning
say $b -= 5;        # -5, no warning
say ++$c, --$d;     # 1 -1, no warning

say $e *= 1;        # 0, but with a warning
```

Referencie: Type Glob

V Perli existuje ešte jeden *sigil*, ktorý sme dosiaľ nepoužívali, `*`.
Volá sa *type glob* a umožňuje pristupovať k tabuľke symbolov.

Čo sú symboly?
V kontexte Perlu, *symboly* sú funkcie a tzv. *package variables*
(v podstate globálne premenné pre menný priestor).

Referencie: Menné priestory (úvod)

Menné priestory budú podrobne vysvetlené neskôr.

Premenné uvedené pomocou `my` sú **lexikálne premenné**:

- Ich hodnota je zviazaná s názvom premennej len v rozsahu bloku.
- Nemajú záznam v tabuľke symbolov.

Perl má však aj **globálne premenné** (*package variables*):

- Sú zviazané s nejakým menným priestorom.
- Existujú v tabuľke symbolov daného menného priestoru.

Referencie: Menné priestory (úvod)

Všetky premenné, ktoré sme doteraz videli, boli *lexikálne*.

Globálne premenné vytvoríme priradením do premennej s plným menom:

```
my $PI = 3.14159265;        # Lexical variable

$::PI = 3.14159265;         # Package variable
our $PI = 3.14159265;       # Lexical alias for package variable
```

Čo je `::`?

`::` je oddeľovač názvov v mennom priestore.
Východzí menný priestor sa volá `main` a môže sa vynechať:

```
@main::PRIMES = (2, 3, 5, 7, …);
say @::PRIMES;
```

Referencie: Menné priestory (úvod)

Aké menné priestory sme doteraz videli?

- ‹`main`› (východzí menný priestor)
- ‹`builtin`› (‹`builtin::true`›)
- ‹`List::Util`› (‹`hw03a`›)

Nepriamo sme tiež používali

- ‹`CORE`› (štandardné funkcie Perlu)

Referencie: Tabuľka symbolov

Tabuľku symbolov získame ako (pseudo)hash syntaxou `%NS::`:

```
say foreach sort keys %main::;
```

K jednotlivým typom v tabuľke pre symbol potom môžeme pristúpiť práve pomocou *type glob*:

```
$::PI = 3.14159265;

say *::PI{SCALAR};       # SCALAR(…)
say *::PI{SCALAR}->$*;   # Value of the scalar.
```

Referencie: Tabuľka symbolov

Takto môžeme v Perli vytvoriť *globálne aliasy*:

```
$main::a = 42;
*main::b = *main::a;

say $main::b;           # 42
```

... a tie sú nám na čo?
Globálne aliasy sú základom modulového systému.

Ako Perl vytvorí alias v mennom priestore, ktorý nepozná, za behu?

Symbolické referencie

Bežné referencie sú skaláry, ktoré odkazujú na **hodnotu** inej premennej.
Symbolické referencie odkazujú na **meno** inej premennej.

```
{                               # Scope where symbolic refs will be allowed
    no strict 'refs';           # ‹use strict› is default since ‹v5.12›.

    $::PI = 3.14159;
    my $symref = "::PI";
    say $symref->$*;            # 3.14159
}                               # Disallow symbolic refs again
```

Symbolické referencie: `no strict 'refs'`

Symbolické referencie sú **nebezpečné**.

- V modernom Perli sú automaticky zakázané vďaka `use v5.36`, ktoré automaticky zapne `use strict`.

Zároveň sú však občas **užitočné**.

- Priamy prístup k modulovému systému, obrovská flexibilita.
- `use strict` je možné vypnúť pragmou `no strict LIST`.
- Túto pragmu používajte **lexikálne**, nikdy nie **globálne**!

Symbolické referencie

Príklad: Vytváranie pomenovaných funkcií za behu.

```
sub spawn($name) {
    no strict 'refs';           # Limited only to the scope of ‹spawn()›.
    *$name = sub ($arg) {       # Note the type glob
        say "$name says $arg!";
    };
}

spawn("foo");           # ‹foo()› is a valid function from now on.
foo("bar");             # foo says bar!
```

Moduly

Moduly

Jednou z veľkých výhod Perlu je veľké množstvo rozširujúcich modulov.

MetaCPAN je vyhľadávací nástroj pre moduly:
https://metacpan.org/

Niektoré moduly sú zabudované priamo v Perli:
https://perldoc.perl.org/modules

Prítomnosť modulu v jadre Perlu je možné zistiť programom `corelist`:

```
$ corelist List::Util
```

Moduly: Inštalácia

Najprv sa pokúste zistiť, či požadovanú funkcionalitu nemá Perl už v jadre.

1. Preferujte inštaláciu modulov zo systémového správcu (`apt`, `yum`, ...).
2. Inak ho musíte inštalovať sami:

```
$ perl -MCPAN -e 'CPAN::shell'
cpan[1]> install Some::Module
```

```
$ perl -MCPAN -e 'CPAN::Shell->install("Module")';
$ cpan -I Some::Module
$ cpanm Some::Module
```

3. Ako najhoršiu možnosť si môžete preložiť zdrojový kód z MetaCPAN.
   Good luck and have fun.

Moduly: Použitie

```
require FILENAME
require MODULE
```

- Sprístupní modul z uvedeného súboru.
- Neimportuje z neho žiadne symboly, musíme použiť `‹NAMESPACE::SYMBOL›`.
- Ak je parametrom slovo bez úvodzoviek, predpokladá sa názov modulu a cestu k súboru si odvodí sám.

```
require 'List/Util.pm';          # Perl Modules have ‹pm› suffix usually.
require List::Util;              # Basically the same thing.
```

Moduly: Použitie

```
use MODULE
use MODULE LIST
```

- Ako ‹`require MODULE`›, ale navyše umožní importovať symboly modulu.
- Zoznam symbolov je u niektorých modulov možné zmeniť parametrom.
  Ako presne sa ‹`LIST`› interpretuje záleží od modulu!

```
use List::Util;             # Imports no symbols by default.
use List::Util qw{shuffle}  # Now we can say ‹shuffle› without namespace.

use Data::Dumper;           # Imports ‹Dumper()› by default.
use Data::Dumper ();        # Imports nothing.

use Log::Any '$log';        # Creates ‹$log› variable in our namespace.
```

Moduly: ‹CORE›

V mennom priestore ‹CORE› žijú všetky zabudované Perl funkcie.

Používa sa napríklad v prípade, že nejaký modul omylom prekryje zabudovanú funkciu.

```
use Some::Bad::Module;  # ‹reverse› gets shadowed.

reverse @list;          # ‹Some::Bad::Module::reverse› gets called.
CORE::reverse @list;    # Built-in ‹reverse› gets called.
```

Moduly: ‹Data::Dumper›

Vráti reťazcovú reprezentáciu zložitej štruktúry.
Veľmi užitočné na ladenie.

```
use Data::Dumper;       # Imports ‹Dumper()› by default.

say Dumper($something);
```

Moduly: ‹FindBin›

Definuje *package variables*, ktoré obsahujú cestu k skriptu alebo adresára, v ktorom sa skript nachádza.

```
use FindBin qw{$Bin $RealBin $Script $RealScript};

say "Hello, this is $RealScript in $RealBin.";
```

‹$Real*› premenné dereferencujú symbolické odkazy.

Moduly: ‹Getopt::Long›

Spracovanie prepínačov programu.

```
use Getopt::Long;       # Imports ‹GetOptions()› by default.

my $::OPTIONS = {};
GetOptions($::OPTIONS, qw{
    h           help
    output|o=s  set|S=s%
}) or exit 1;

# $ perl PROGRAM --output file.xml -S x=1 -S y=2 source1.xml source2.xml
# $::OPTIONS->{output}: file.xml
# $::OPTIONS->{set}:    x => 1, y => 2
# @ARGV:                source1.xml source2.xml
```

Moduly: ‹Scalar::Util›, ‹List::Util›

```
use Scalar::Util qw{blessed readonly refaddr …};
use List::Util qw{any all …};
```

Veľká sada rozširujúcich funkcií na prácu so skalármi a zoznamami:
https://metacpan.org/pod/Scalar::Util
https://metacpan.org/pod/List::Util

Niektoré funkcie z ‹Scalar::Utils› sú od ‹v5.36› dostupné v ‹builtin›.

Moduly: Honourable Mentions

`Archive::Tar`
: Práca s TAR archívmi

`Encode`
: Kódovanie a dekódovanie reťazcov

`Devel::Peek`
: Nástroj na pomoc pri ladení hodnôt premenných

`Fcntl`, `POSIX`
: Sprístupní niektoré POSIXové funkcie

`JSON::PP`
: Serializácia a deserializácia JSON

Moduly: Honourable mentions

`IO::*`
: Moduly na prácu so vstupom a výstupom

`IPC::*`
: Komunikácia medzi procesmi

`Test::*`
: Testovacie nástroje

`Pod::Usage`
: Generátor nápovedy z dokumentácie skriptu alebo modulu (POD).

Špeciálne premenné

Špeciálne premenné

Niekoľkokrát sme narazili na premennú `$_`.
Perl má niekoľko podobných *špeciálnych* premenných.

Celý zoznam viď https://perldoc.perl.org/perlvar.

English, Perl, do you speak it?!

```
use English;
```

Modul, ktorý vytvorí anglické **aliasy** pre špeciálne premenné.

Špeciálne premenné: ‹$_›, (‹$ARG›)

Implicitná premenná

- Použitie v cykloch, kde premennú pomenovať nemôžeme:

```
$sum += $_ foreach @numbers;
```

- Dostupná v bloku ‹grep›, ‹map›, ‹given›.
- Niektoré zabudované funkcie ju použijú, ak im nedáme žiadny parameter:

```
say;            # ≈ say $_;
length;         # ≈ length $_;
```

Používajte ju radšej len tam, kde musíte.

Špeciálne premenné: ‹@_›, (‹@ARG›)

Argumenty funkcie

- Implicitný argument pre ‹pop› a ‹shift›.
- Pred ‹v5.20› (‹v5.36›) to bol jediný spôsob, ako pristúpiť k argumentom:

```
sub inspect {
    my ($what, @values) = @_;
}

sub inspect {
    my $what = shift;   # ‹shift› uses ‹@_› by default.
}
```

Signatúry sú dnes pohodlnejšie a bezpečnejšie (kontrolujú aritu).

Špeciálne premenné: ‹$"› (‹$LIST_SEPARATOR›)

Oddeľovač pri interpolácii poľa

- Východzia hodnota je ‹' '› (medzera).

```
$" = ':';
my @values = qw(a b c);
say "@values";              # a:b:c
```

Argumenty programu

- ‹($0, @ARGV)› je ekvivalent ‹argv› argumentu ‹main()› v C.

```
say "Arguments: [$0] + [@ARGV]";

# $ perl script.pl --test arg
# Arguments: [script.pl] + [--test arg]
```

Špeciálne premenné: ‹%ENV›

Premenné prostredia

```
say sort keys %ENV;
say $ENV{HOME};          # /home/pazuzu
```

Špeciálne premenné: ‹%SIG›

Spracovanie signálov

```
$SIG{HUP} = sub ($) {
    $log->info("Reloading daemon");
    $::DAEMON->reload;
};
```

Špeciálne premenné: ‹$!› (‹$ERRNO›)

Chybový kód

- Ekvivalent ‹errno› v C.
- *Magická* premenná:
  - ako číslo vráti kód,
  - ako reťazec vráti popis chyby (‹strerror()› v C)

```
$! = 130;
say 0 + $!;      # 130
say "$!";        # Owner died
```


Magic variables je v Perli polo-oficiálny názov pre premenné, ktoré sa za určitých okolností správajú inak, než bežné premenné.

Špeciálne premenné: Identifikácia procesu

`$$`, `$PID`
: Číslo procesu, podobne ako v shelli.

`$<`, `$UID`; `$>`, `$EUID`
: Používateľské ID procesu

`$(`, `$GID`; `$)`, `$EGID`
: Skupinové ID procesu

Špeciality

Špecialiy: Schwartzian Transformation

Návrhový vzor, ktorý umožní obmedziť výpočty pre `sort`:

- `decorate()` vráti `[ ITEM, EXTRA… ]`,
  `EXTRA` sú nejaké predpočítané informácie pre radenie.
- `undecorate()` vráti `ITEM` po triedení.

```
map { undecorate($_) }
sort { ... }
map { decorate ($_) }
LIST
```

Špeciality: Lexikálne funkcie

Tak, ako `my $var` vytvorí lexikálnu premennú, je možné vytvoriť aj lexikálnu funkciu.

```
my sub foo($x) {
    ...
}
```

Tieto funkcie majú podobné vlastnosti, ako lexikálne premenné:

- Sú viditeľné len v rozsahu, v ktorom vznikli.
- Nemajú záznam v tabuľke symbolov.

Navyše, rekurzívne lexikálne funkcie musia použiť `SUB`.

Špeciality: Prototypy

Stručne, **prototyp** urobí z funkcie operátor:

- Pridá parseru informácie o tom, v akom kontexte sa očakávajú argumenty.
- Umožní skrátiť niektoré zápisy.
- Prototyp je atribút symbolu, neprenáša sa na aliasy.

```
sub filter :prototype(&@) ($p, @values) { … }
filter { PREDICATE } 1, 2, 3;

sub punch :prototype(&\@) ($p, $array) { … }
punch { PREDICATE } @array;
```

Používať veľmi opatrne, občas to robí kúzelné veci.

Špeciality: Prototypy

Historicky (pred ‹v5.36›) boli prototypy na mieste signatúr:

```
no feature qw{signatures};
sub foo($$) {}                      # Prototype

use feature qw{signatures};         # Default for ‹v5.36›
sub foo($$) {}                      # (Invalid) signature
sub foo :prototype($$) ($x, $y) {}  # Prototype and signature
```

Viac info vid' https://perldoc.perl.org/perlsub#Prototypes