Perl — Practical Extraction and Report Language

Perl je programovací jazyk, který byl původně určen pro zpracování textových souborů, provádění textových manipulací nad nimi a tisk složitých výstupů. Jeho schopnosti jsou ale nyní mnohonásobně širší. Je vhodný pro nejrůznější typy úkolů při správě a používání UNIXového systému, protože zde najdeme množství systémových funkcí, můžeme volat externí programy, manipulovat s adresářovými strukturami i psát složité filtry, problémem nejsou pochopitelně ani binární data.

Datové struktury perlu jsou velmi flexibilní a snadno použitelné, nechybí zde pole či asociativní pole (hash). Od verze 5 navíc perl podporuje objektový způsob práce s daty, což dále zjednodušilo jeho použití. Pro mnoho úkolů najdeme již hotové moduly, ke kterým přistupujeme definovaným objektovým rozhraním bez starosti o vnitřní implementaci.

Syntaxe v sobě sdružuje to nejlepší z jazyka C, `sed`, `awk` či `sh`. Většinu věcí je možno psát intuitivně bez dlouhého rozmýšlení, protože konstrukce, na kterou jsme zvyklí, je s velkou pravděpodobností podporována.

Autorem perlu je Larry Wall. Domovskou stránku jazyka perl najdeme na Webu na adrese `http://www.perl.com/perl`. Zde jsou informace o tom, jak perl získat, jsou zde FAQ i například přehled dostupných perlovských modulů. Na Webu existuje i CPAN (Comprehensive Perl Archive Network), jehož k nám nejbližší mirror je na Karlově univerzitě (URL `http://sunsite.mff.cuni.cz/ftp/Languages/Perl/CPAN/`) a najdeme zde spoustu software, příkladů, programů a odkazů.

Tento text má sloužit jako stručný průvodce po možnostech jazyka perl, nikoli jako referenční příručka či manuálová stránka. Snažil jsem se zahrnout co možná nejvíce příkladů, na kterých bych mohl popisované vlastnosti ilustrovat. Pokud hledáte uspořádanější výklad, zkuste `man perl`, `http://www.perl.com/perl`, `comp.lang.perl.*` či některou z knih nakladatelství O'Reilly.

Všechny příklady nechť jsou prosím považovány za příklady. Jsou zde uvedeny pro demonstraci vlastností perlu, nikoli jako produkční verze programů. Mnoho z nich by bylo možno napsat elegantněji či efektivněji. Většinou také nemají ošetřeny chybové stavy, takže je velmi pravděpodobné, že by se v nestandardních podmínkách chovaly jinak, než jak je popsáno. Vždy je dobré konzultovat nejasnosti s `man perl`.

V celém textu tiše předpokládám perl verze 5 (a více), a proto nebudu upozorňovat na věci, které se objevily až zde. Pokud pracujete stále ještě s verzí 4, je nejvyšší čas navštívit nejbližší CPAN.

Jednoduché skripty

Začneme tradičně:

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

na příkazovém řádku shellu dá očekávaný výstup. Dolar na začátku je prompt shellu, potom spouštíme program perl. Říkáme mu, že má provést program, uvedený na příkazovém řádku (`-e` = expression). Program (skript) můžeme také uložit to souboru:

```
$ cat test.pl
print "Hello, world\n";
```

a potom `perl test.pl` bude provádět obsah tohoto souboru. Pokud na prvním řádku našeho perlového skriptu uvedeme cestu k programu perl (`#!/usr/bin/perl`) a přidělíme skriptu práva pro provádění (`chmod +x test.pl`), pak příkazem `test.pl` spustíme `/usr/bin/perl` se jménem našeho skriptu jako prvním parametrem. Perl může být v lokální instalaci uložen i jinde, pak je potřeba cestu k němu odpovídajícím způsobem změnit. Pokud spustíme pouze perl bez uvedení jména skriptu, bude očekávat řádky skriptu na standardním vstupu.

Následující skript funguje jako jednoduchá verze programu `grep`:

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

Na řádku 2 jsme do proměnné `$search` uložili nultý prvek pole `@ARGV`, které jsme zároveň posunuli o jeden prvek doleva. Z toho vyplývá, že jména skalárních proměnných začínají znakem dolar. Skalární proměnné mohou obsahovat čísla či řetězce (a několik dalších typů), numerická či textová interpretace hodnoty závisí na kontextu. Jednorozměrné pole se označuje znakem zavináč (`@`), jeho prvky indexujeme pomocí hranatých závorek `[`, `]` a pokud neřekneme jinak, indexuje se od nuly. U jmen proměnných se rozlišují velká a malá písmena a stejné jméno je možné použít zároveň jak pro skalár, tak pro pole, tak pro jiný typ.

Speciální pole `@ARGV` obsahuje parametry z příkazového řádku, tedy nyní je v proměnné `$search` uložen první parametr a v `@ARGV` zbývající. Příkaz `while` je cyklus a konstrukce `<>` říká čti řádek ze souborů, které jsou uvedeny jako parametry, pokud žádné nejsou, čti ze standardního vstupu. Načtený řádek je v příkazu `while` uložen do speciální proměnné `$_` a čtení `<>` vrací true, dokud jsou nějaká data na vstupu. Přesněji, čtení vrací načtený řádek, který je považován za pravdivý, pokud je nenulový. Perl nemá booleovský typ, číslo různé od nuly případně neprázdný řetězec jsou chápany jako hodnota true, nula a prázdný řetězec jako false.

Příkaz print bez parametrů vytiskne obsah `$_`, ale v našem případě pouze tehdy (`if`), pokud `/$search/` vrátí true. Konstrukce `/.../` znamená vyhledávání regulárního výrazu a pokud není uvedeno jinak, prohledává se retězec v proměnné `$_`.

Když teď náš skript spustíme s parametry „if“ a „grep.pl“:

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

bude výsledek:

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

Vypíše ty řádky v našem skriptu, ve kterých se vyskytuje podřetězec „if". Skript jsme také mohli volat `grep.pl if < grep.pl`, pak by konstrukce `<>` četla ze standardního vstupu.

Skript můžeme pochopitelně přepsat do tvaru, který bude méně mást programátory zvyklé na jazyk C:

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

Zde si nejprve uložíme do proměnné `$search` nultý prvek pole `@ARGV`. Když přistupujeme k prvkům pole, jsou to již skaláry, a proto je potřeba psát `$ARGV[0]` s dolarem na začátku. Poté provedeme posun pole parametrů. Následuje cyklus, zde ale explicitně přiřazujeme do proměnné `$line`. A v testu `if` operátorem `=~` říkáme, který řetězec (`$line`) se má prohledávat.

Vedle příkazu `/` existuje i `s` jako substitute, tedy náhrada podřetězce. Příkaz

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

provede ve všech řádcích vstupu změnu trojky na čtverku (v daném kontextu `:M012:3:`).

Perl akceptuje množství parametrů na příkazovém řádku, například

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

je ekvivalentní předchozímu příkladu, stejně tak jako

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

Parametry `-p` (print) a `-n` (no print) obalí skript cyklem `while (<>)`, `-p` navíc ještě na konci cyklu automaticky provede tisk `$_`. Výše uvedený `grep.pl` můžeme jednorázově zapsat i jako

```
$ perl -ne 'print if /reg_výraz/;'
```

kde regulární výraz uvedeme přímo do těla příkazu, nikoli jako argument skriptu.

Posledním parametrem, který v této části zmíníme, je `-i` (in place). Vztahuje se na vstupní soubory zpracovávané konstrukcí `<>`, každý z nich nejprve přejmenuje a výstup směruje do původního souboru. Můžeme také zadat příponu pro vytváření záložních kopií. Pokud bychom chtěli provést výše uvedenou substituci ve všech souborech v aktuálním adresáři, použili bychom v shellu příkaz jako

```
$ while i in * ; do i sed 's/:M012:3:/:M012:4:/' $i > $i.new ;
        mv $i.new $i ; done
```

zatímco v perlu nám stačí


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

a navíc máme zachována přístupová práva a startovali jsme na to celé pouze jeden proces.

Ukažme si nyní, jak můžeme očíslovat řádky ze vstupního souboru:

```
$ perl -ne 'print ++$i, ": $_";'
```

Pro každý řádek vstupu (to říká volba `-n`) se provede příkaz v apostrofech (volba `-e`). Ten obsahuje pouze `print`, který má dva argumenty: proměnnou `$i`, kterou nejprve zvýší o jedna, a text, který se má vytisknout. Proměnná `$i` je na začátku nedefinovaná, což je v numerickém kontextu to samé jako nula. Text je uzavřen v uvozovkách, proto se v něm expandují proměnné, v našem případě `$_`. Ta obsahuje načtený řádek vstupu. Všimněme si, že za ni při tisku nepřidáváme znak `\n`. Perl totiž standardně znak oddělující řádky neořezává, proto ho také nemusíme dávat zpět.

Následuje variace na to samé téma:

```
$ perl -ne 'BEGIN { $i = 1; } printf "%d: %s", $i++, $_;'
```

Zde jsme použili konstrukci `BEGIN`. Ta se provede vždy před zbytkem skriptu, a i u volby `-n` nebo `-p` se provádí ještě před první iterací.

Poslední možnost využívá parametr `-p`:

```
$ perl -pe 'print ++$i, ": ";'
```

Námi dodaná funkce `print` vytiskne číslo řádku a dvojtečku, o tisk zbytku řádku se postará perl sám.

Regulární výrazy

Perl kromě základních regulárních výrazů definuje i celou řadu rozšíření. Například `\w` označuje jakýkoli symbol, který může být součástí slova. Tedy například

```
$ perl -e '$_ = <>; s/(\w)/$1:/g; print;'
```

nejprve načte jeden řádek ze vstupu do proměnné `$_` a potom za každý znak, který může být součástí slova (písmeno, číslice, podtržítko), přidá dvojtečku. Symbol `$1` je naplněn hodnotou, která byla nalezena mezi párem kulatých závorek. Modifikátor `/g` říká, že touto náhradou má postupně projít celý řetězec. Pokud není uvedena konstrukce `=~`, pracuje operátor `s//` standardně nad proměnnou `$_`:

```
$ perl -e '$_ = <>; s/(\w)/$1:/g; print;'
General protection
G:e:n:e:r:a:l: p:r:o:t:e:c:t:i:o:n:
```

Za každým písmenem je dvojtečka, za mezerou nikoli. Výhoda `\w` oproti zdánlivě ekvivalentní konstrukci `s/([A-Za-z0-9_])/$1:/g;` spočívá v tom, že interpretuje nastavené locales. Takže správné výsledky dává i

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

a můžeme si ověřit, že pokud nastavíme locales na defaultní hodnotu, výstup se patřičně změní:

```
$ ( LC_CTYPE=C; perl -e '$_ = <>; s/(\w)/$1:/g; print;' )
maličký ježeček
m:a:l:i:čk:ý j:e:že:če:k:
```

V US-ASCII nejsou česká písmena s diakritikou považována za písmena, proto se za nimi dvojtečka neobjeví.

Nyní zadání změníme: chceme vložit dvojtečku pouze dovnitř slov, nikoli i za ně. Zkusíme-li dát hledání dvou písmen vedle sebe `s/(\w)(\w)/$1:$2/g;`, není výsledek příliš uspokojivý: `m:al:ič:ký j:ež:eč:ek`. Při vyhledávání jsou totiž nalezeny dva znaky `(\w)(\w)`, ty jsou správně uloženy do proměnných `$1` a `$2` a mezi ně vložena dvojtečka, ale při dalším průchodu prohledávání začíná za až druhým znakem. Použijeme-li perlovou rozšířenou syntaxi `s/(\w)(?=\w)/$1:/g;`, bude už vše správně: `m:a:l:i:č:k:ý j:e:ž:e:č:e:k`. Zde `(?=reg_výraz)` znamená dopředné prohledávání s nulovou šířkou, které neovlivní další průchod. Jiná možnost je `s/(\w)\B/$1:/g;` se symbolem `\B`, který označuje bezrozměrnou nehranici mezi slovy (hranice slova je pak `\b`).

Ukažme si nyní program, který ze zdrojového textu jazyka C odstraní komentáře.

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



```
        print;
        }
```

Parametrem `-w` říkáme perlu, že chceme, aby nás informoval o podezřelých konstrukcích v našem programu a možných problémech. U delších programů se velmi doporučuje ho používat a význam má i u takto krátkého skriptu. Povšimněme si, že zde nahrazujeme pomocí `s!...!!`. Pokud bychom použili místo vykřičníku lomítko, museli bychom skutečné lomítko céčkového komentáře opentlit backslashem. Perl proto dovoluje jako oddělovač v substituci cokoli jiného. Je možno použít i závorky a pak píšeme intuitivně levou a pravou, například `s(...)[...]g;` je naprosto v pořádku. Pro úplnost dodejme, že i při vyhledávání pomocí `/reg_výraz/` tuto možnost máme také, pak je ale nutné uvést písmeno `m` (match), které se jinak při použití lomítek může vynechat: `m#reg_výraz#`.

Za zmínku stojí i zápis `.*?`. Běžné `.*` totiž vyhledá nejdelší odpovídající řetězec, což my zde nepotřebujeme. Pokud by totiž v našem C-programu byly řádky

```
/* Poznámka */   /*/ Další poznámka
        */
```

a my použili normální hladový přístup, hledání prvního řetězce by se zastavilo až na druhém `*/` a my bychom našli celý řetězec `/* Poznámka */ /*/` a druhý komentář bychom už neodstranili. Přidáme-li otazník, je vždy nalezen nejkratší odpovídající podřetězec, což se nám velmi často velmi hodí.

V úvodu jsem sliboval podobnost se syntaxí jazyka C a nyní zde vidíme Pascalské `and`. To není omyl, `&&` můžeme pochopitelně použít také, s jedinou výhradou: `and` (a ostatní slovní logické spojky) mají velmi nízkou prioritu. Proto při jejich používání neriskujeme, že by se vyhodnocovaly v rámci příkazů, které spojují. Pokud bychom například první dvě `and` nahradili `&&`, perl by si postěžoval `Can't modify logical and in scalar assignment`, protože přiřazení má nižší prioritu než logické `&&` a my bychom se tímto snažili do `&&` přiřadit nulu, což nejde. Samozřejmě, že se tomu můžeme vyhnout i tím, že výraz patřičným způsobem ozávorkujeme nebo ho přepíšeme do tvaru `if {...} elsif {...} else {...}`.

Skript neošetřuje výskyt komentářů v řetězcích — to ponechávám za domácí cvičení.

Perl má velmi málo omezení, co se týče velikosti zpracovávaných dat. Obsah celého dvoumegabajtového UNIXového jádra můžeme načíst do jednoho řetězce a jsme při tom omezeni v podstatě jen dostupnou pamětí a swapem. Výše uvedený comment stripper může mít i takovouto podobu:

```
#!/usr/bin/perl -w
undef $/;
$a = <>;
$a =~ s!/\*.*?\*/!!gs;
print $a;
```

Proměnná `$/` je ekvivalent `RS` v `awk`, určuje, čím jsou odděleny vstupní řádky. Defaultně je nastavena na `\n`, můžeme ji libovolně změnit (třeba na velké A, pokud bychom to potřebovali), a můžeme jí také dát nedefinovanou hodnotu. Perl potom nemá jak vstup do řádků rozdělit a proto ho načte jako jeden dlouhý řetězec. Nad ním nám potom stačí jediný příkaz a pak zbývá už jen tisk. Při pozornějším zkoumání objevíme znak `s` na konci substituce, ten říká, že se má celý řetězec brát jako jeden řádek.

Pole a hashe

Následující příklad ukazuje využití polí a asociativních polí. Chceme-li spočítat slova ve vstupním textu, můžeme použít tento cyklus:

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

Na druhém řádku přiřazujeme do pole `@a` seznam, vrácený funkcí `split`. Ta rozdělí obsah řetězce `$line`, přičemž oddělovači jsou nenulové sekvence nepísmen (opak `\w`). V `@a` jsou potom slova ze vstupního řádku. Pole `@a` nyní přičítáme k proměnné `$total`. To, že sčítáme proměnné různých typů (pole a skalár) vůbec nevadí, naopak. Perl zjistí, že nalevo je skalár a převede pole na počet jeho prvků, tedy počet slov v řádku.

Při používání možná objevíme jednu vadu na kráse: pokud řádek začíná mezerou (či jiným neznakem), `split` považuje prázný podřetězec před ní za slovo, které (byť má nulovou délku) se promítne do počtu slov. Můžeme to opravit například tak, že k `$total` přičteme jen počet těch prvků, které jsou nenulové. Třetí řádek přepíšeme na

```
$total += grep /./, @a;
```

kde `grep` vezme na vstupu pole `@a` a vrátí z něho jen ty prvky, které obsahují aspoň jeden znak. Tento nový seznam je potom převeden na skalár (počet prvků) a tato hodnota je přičtena k proměnné `$total`.

Nyní bychom chtěli vědět, kolikrát se které slovo v textu vyskytuje. K tomu využijeme slibovaný hash. Hash je pole, jehož prvky nejsou indexovány svou pozicí (od nuly do počtu prvků jako běžné pole), ale řetězcem. Celý hash se označuje procentem (například `%names`), jeho jednotlivé prvky jsou potom skaláry, tedy začínají znakem dolar, indexy jsou uzavřeny ve složených závorkách: `$firstname{"login"}` je prvek asociativního pole indexovaný řetězcem `login`, `$firstname{$login}` je prvek indexovaný hodnotou proměnné `$login`.

Nyní můžeme přistoupit k vlastnímu programu. V cyklu `while` čteme řádky do proměnné `$_`, následný `split` nemá uvedenou vstupní proměnnou, takže defaultně předpokládá `$_`, a vrací seznam slov jako v předchozím příkladu. Příkazem `for` nyní projdeme tímto seznamem, postupně jsou jeho všechny prvky přiřazeny do proměnné `$word`. Na dalším řádku testujeme, jestli je obsah `$word` neprázdný, pokud ano, zvýšíme počet v příslušném prvku asociativního pole o jedna.

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

Jsme tedy v polovině skriptu a máme polovinu práce hotovou: asociativní pole `%counts` obsahuje pro každé slovo, které se v textu objevilo, počet jeho výskytů. Nyní tento hash

projdeme a vytiskneme vždy slovo (index v hashi) a počet (příslušnou hodnotu). Seznam všech indexů dostaneme funkcí `keys`, ten ještě setřídíme `sort`em. Opět použijeme cyklus `for`, nyní neuvádíme proměnnou, takže se předpokládá `$_`.

Můžeme ale chtít víc. Například setřídit slova od nejčetnějších po nejméně častá. Funkci `sort` můžeme říci, podle jakého kritéria (nevyhovuje-li nám defaultní) má třídit. Pokud změníme druhou část programu na

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

udělá `sort` následující: do proměnných `$a` a `$b` dosadí vždy porovnávané hodnoty. Standardně by provedl ekvivalent `$a cmp $b`, tedy textové porovnání. My chceme porovnávat numericky (`<=>`), a neporovnáváme indexy do pole (prvky `keys`), ale odkazované hodnoty (`$counts{$a}`). Navíc jsme prohodili pořadí, takže nejvyšší budou první. A na závěr jsme použili formátování pomocí `printf` místo obyčejného `print`, takže počty jsou pěkně zarovnány doprava. (Návyky z céčka se zde rozhodně neztratí.)

Jiný příklad: chceme zjistit, zda se v souboru `/etc/passwd` nevyskytují uživatelé se stejným loginem vícekrát.

```
#!/usr/bin/perl
open FILE, "/etc/passwd" or die;
while (<FILE>) {
        chomp;
        ($login, $pw, $uid, $gid, $name, $home, $sh) = split /:/;
        if (defined $passwd{$login}) {
                print "$passwd{$login} x ",
                        "$login: $name, uid $uid, shell $sh";
                }
        else {
                $passwd{$login} = "$name, uid $uid, shell $sh";
                }
        }
close FILE;
```

Na začátku otevíráme soubor `/etc/passwd` pro čtení. `FILE` je filehandle, kterým se dále budeme na takto otevřený soubor odkazovat. Pokud bychom chtěli naopak do nějakého souboru zapisovat, dali bychom na začátek jeho jména znak `>`, podobně jako v `shell`u, pomocí `|` můžeme nastartovat další process a zapisovat do roury:

```
open MAIL, "| mail $user"
```

Pokud `open` neuspěje, zavolá se `die`, předčasné vyskočení z programu. V cyklu čteme z otevřeného souboru, vstupy se ukládají jak jsme zvyklí do `$_`. Funkce `chomp` odřízne z konce řádku znak `\n`, případně znak nastavený v proměnné `$/`. Často se můžeme setkat i s variantou `chop`, která při odřezávání posledního znaku netestuje, zda se jedná o oddělovač vstupních řádků.

Dále provedeme `split` vstupu podle dvojteček, výsledný seznam přiřadíme do seznamu proměnných uvedených vlevo. Podíváme se, je-li již daný login v hashi `%passwd` zařazen (`defined $passwd{$login}`), pokud ano, vytiskneme hlášení o nalezené shodě, pokud ne, uložíme jméno, číslo uživatele a jeho shell do hashe pod jeho loginem.

Pokud bychom tento skript pustili s paramatrem `-w`, dostali bychom několik hlášení o proměnných použitých jen jednou (např. `$pw`).

Následující skript je jednoduchým ekvivalentem UNIXového příkazu `pwconv`. Ten má za úkol odstranit ze souboru `/etc/passwd` zakódovaná hesla a přesunout je do souboru `/etc/shadow`, který je pro svět nečitelný. Pozor: tento skript přepisuje základní systémové soubory! Snažil jsem se, aby v něm nebyly chyby, ale radši ho podrobte pečlivé kontrole, než ho spustíte s rootovskými právy.

```
#!/usr/bin/perl -w
$PASSWD = "/etc/passwd"; $SHADOW = "/etc/shadow";
open(SHADOW, $SHADOW) || print STDERR "Error reading $SHADOW";
while (<SHADOW>) {
        ($login) = split /:/;
        $shadow{$login} = $_;
        }
close SHADOW;
open(PASSWD, $PASSWD) || die "Error reading $PASSWD";
umask 0133; unlink "$PASSWD.$$";
open(OUT, "> $PASSWD.$$") || die "Error writing $PASSWD.$$";
umask 0377; unlink "$SHADOW.$$";
open(SHADOWOUT, "> $SHADOW.$$") || die "Error writing $SHADOW.$$";
while (<PASSWD>) {
        ($login, $passwd, $rest) = split /:/, $_, 3;
        print OUT "$login:x:$rest";
        $sh = "$login:${passwd}::::::\n";
        $sh = delete $shadow{$login} if (defined $shadow{$login});
        print SHADOWOUT $sh;
        }
close SHADOWOUT; close OUT; close PASSWD;
rename "$PASSWD.$$", "$PASSWD"; rename "$SHADOW.$$", "$SHADOW";
if (%shadow) {
        print STDERR "Removed from shadow ",
                join (", ", keys %shadow), "\n";
        }
```

Nebudeme zde rozebírat program krok za krokem, zaměříme se jen na věci které jsou nové a mohou být zajímavé.

Proč je `$login` na řádku `($login) = split /:/;` v závorkách? Napravo od rovnítka je seznam, který vrátí příkaz `split`. Pokud bychom vlevo měli skalární proměnnou bez závorek, byl by tento seznam převeden na skalár a do proměnné `$login` by se dostal počet prvků, nikoli první prvek pole. Tím, že jsme jej obalili závorkami, jsme nalevo vytvořili seznam, a seznam do seznamu můžeme přiřadit. Seznam vlevo má jen jeden prvek, přiřadí se tedy jen první.

Druhý split voláme se třemi parametry, ten poslední je počet, na kolik chceme řádek rozdělit. Kdybychom zde tu trojku neměli, vytvořil by se napravo seznam o osmi prvcích, z nichž první tři by se přiřadili a o ostatní bychom přišli.

V cyklu při čtení z `/etc/passwd` jsme nepoužili `chomp`. V tomto případě by to bylo zbytečné, protože bychom znak nového řádku stejně přidávali zpět. Proměnná `$$` obsahuje stejně jako v `shellu` číslo procesu. Funkce `delete` odstraní a vrátí prvek z asociativního pole. Funkce `join` spojí prvky seznamu uvedeným řetězcem, je to opak funkce `split`. Funkce `umask`, `unlink` a `rename` jsou ekvivalenty systémových funkcí.

Ve speciálním hashi `%ENV` jsou uloženy proměnné prostředí procesu. Funkci příkazu `set` můžeme napodobit skriptem:

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

V cyklu `for` se do proměnné `$_` přiřadí jméno proměnné (klíč v asociativním poli `%ENV`), `print`em pak tiskneme nejprve tento klíč a za rovnítkem odpovídající hodnotu z hashe.

Následujcící skript je jednoduchou verzí příkazu `whereis`. V adresářích, uvedených v proměnných `$PATH` a `$MANPATH`, najde všechny soubory, které ve svém jméně obsahují podřetězec uvedený jako první argument.

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

Na prvním řádku testujeme, jestli jsou uvedeny nejaké argumenty. Pokud ano, uložíme první z nich příkazem `shift` do proměnné `$string`. V následujících třech řádcích programu uložíme do pole `@path` adresáře posbírané z proměnných `$PATH` a `$MANPATH`. Test `if defined $ENV{PATH};` by zde být nemusel, protože `split` na nedefinované hodnotě vrátí prázdný seznam. Zajímavý je `push`, který přidá seznam, vrácený `split`em na konec uvedeného pole. Existuje i jeho opak `pop`, a také opak `shift`u `unshift`.

Pokud jsme ze žádné z obou proměnných nebyli schopni získat ani jedno jméno adresáře, skončíme se smutným hlášením. Jinak tento seznam adresářů projdeme cyklem `for`. Pokud adresář zadaného jména neexistuje (test `-d` jako v `shellu`, existuje i `-f`, `-r` `-M` a mnoho dalších), provede se `next`, skok na další iteraci cyklu. Jinak se provede konstrukce `<$dir/*$string*>`, která rozexpanduje jména souborů. Nalezené soubory vytiskneme.

Funkce

V perlu můžeme definovat svoje uživatelské funkce. Typický postup je:

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

Definice začíná klíčovým slovem `sub`, za kterým následuje jméno naší proměnné a ve složených závorkách tělo funkce. V něm deklarujeme lokální proměnnou `$total` pomocí `my`. Taková proměnná se chová stejně jako auto proměnná v jazyce C. Existuje i starší způsob pro označení lokální proměnné, `local`. Rozdíl je v tom, že `my` proměnná je viditelná pouze ve funkci, kde byla takto deklarována. Konstrukce `local` pouze uloží starou hodnotu globální proměnné a při návratu z naší proměnné ji zase obnoví, proměnná je ale vidět ve všech funkcích z naší funkce volaných.

Cyklem `for` projdeme pole `@_`, ve kterém se funkcím předávají argumenty. Návratovou hodnotou funkce je hodnota posledního výrazu, v našem případě součet `$total`. Funkci můžeme také ukončit příkazem `return`.

Parametry se funkcím předávají odkazem. Změníme-li tedy hodnotu v `@_`, změní se tím i hodnota ve volající funkci:

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

výsledek bude `3,5,7`. Pokud bychom si ale argumenty funkce uložili do pomocného lokálního pole, budeme už pak pracovat s jejich kopiemi.

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

a proměnné v globálním `@a` zůstanou nezměněny. Návratovou hodnotou naší funkce je seznam z pole `@data`, a s tímto seznamem můžeme ve volající funkci libovolně manipulovat, například přiřadit do pole:

```
@b = inc (@a);
print join(",", @b), "\n";
```


Volby příkazové řádky perlu

Následuje seznam parametrů, které perl akceptuje na příkazové řádce. Je možné je spojovat, například místo `-l -a -n -e` můžeme psát `-lane`.

`-0`číslice

: Oktalové číslo začínající nulou, které určuje znak pro oddělovač řádků (`$/`). 00 znamená odstavcový mód (jeden či více prázdných řádků), 0777 znamená čtení celého vstupu naráz.

`-a` (autosplit)

: Při použití s `-n` nebo `-p` dělá automaticky `split` vstupu do pole `@F`. Defaultně rozděluje na mezeře, je možno změnit pomocí `-F`.

`-c` (check)

: Perl načte skript a provede pouze syntaktickou kontrolu, skript nespouští až na nezbytné `BEGIN`, `END` a `use`.

`-d` (debug)

: Spustí perl debugger.

`-D` číslo nebo `-D` seznam (debugging flags)

: Číslo nebo písmena v seznamu říkají, jaké ladící informace se budou zobrazovat. Perl musí být přeložený s volbou `-DDEBUGGING`.

`-e` příkaz (execute)

: Na místě příkazu je řádka perlového skriptu. Musí správně obsahovat středníky. Více řádek se zadá dalšími volbami `-e`. Je vhodné obalit příkaz apostrofy, abychom předešli expanzi metaznaků shellem. Je-li na příkazovém řádku volba `-e`, perl pak už nehledá jméno skriptu.

`-F`regulární výraz

: Regulární výraz pro autosplit pomocí `-a`.

`-i`přípona (in place)

: Soubory zpracovávané konstrukcí `<>` budou změněny a zůstanou na místě. Pokud uvedeme příponu, dostanou ji staré verze souborů.

`-I` adresář (include files)

: Říká C preprocesoru (volba `-P`), kde má hledat include soubory.

`-l`oktalové číslo (line-end processing)

: Při použití s `-n` či `-p` dělá automaticky `chomp` a nastavuje `$\` na uvedené oktalové číslo, takže tento znak je při tisku `print`em přidán zpět. Pokud číslo není uvedeno, nastavuje `$\` na `$/`.

`-n` (no print)

: Obalí skript cyklem `while (<>) { ... }`, tedy skript se provede pro každý vstupní řádek. Bloky `BEGIN` a `END` jsou provedeny mimo tento cyklus.

`-p` (print)

: Obalí skript cyklem `while (<>) { ... } continue { print; }`, tedy jako `-n` a navíc se automaticky ještě tiskne `$_`.

`-P` (preprocesor)

Před tím, než je zpracován perlem, je skript prohnán C preprocesorem.

`-s` (switch parsing)

Vyhledá na příkazové řádce přepínače, odstraní je z `@ARGV` a nastaví proměnné s odpovídajícími jmény.

`-S` (search through script)

Bude hledat skript podle proměnné prostředí `PATH`. Je možné použít na systémech, které perlu předají jen jméno skriptu bez cesty.

`-T` (taint checks)

Zapíná přísnější kontrolu podezřelých nebo nečistých programátorských konstrukcí. Všechny vstupy z venku jsou považovány za ne bezpečné a s takovými perl nedovolí udělat ne bezpečné výstupní operace.

`-u` (undump)

Poté, co perl náš skript zkompiluje, vypíše core. Ten můžeme příkazem `undump` převést zpět do binární executable podoby.

`-U` (unsafe)

Povoluje perlu, aby dělal ne bezpečné operace.

`-v` (version)

Vytiskne verzi a patchlevel našeho perl programu.

`-w` (warnings)

Tiskne varování o některých syntakticky méně čistých věcech ve skriptu, které mohly vzniknout překlepnutím. U rozsáhlejších programů je to základní volba.

`-x` adresář

Odstraní vše až po první řádek začínající `#!` a obsahující slovo `perl`. Adresář (je-li uveden) určuje, kam se má perl před spuštěním skriptu přepnout. Užitečné, pokud chceme spustit skript uložený v delší zprávě.

Předdefinované perlovské proměnné

Některé proměnné v perlu mají speciální význam, někdy i dost dalekosáhlý, ovlivňující jeho celkové chování při běhu našeho programu. Pomocí některých z nich můžeme snadno dosáhnout věcí, které bychom jinak museli dělat opisem a složitými konstrukcemi. Proto je dobré alespoň tušit, jaké možnosti se v nich skrývají.

`$_`

Proměnná asi nejzákladnější. Je to defaultní místo, kde se vyhledává, defaultní hodnota, která se tiskne, pokud neřekneme našim funkcím jinak.

`$?`

Status vrácený posledním voláním funkce `system`, zpětnými apostrofy (“) či uzavřením roury.

`$!`

V numerickém kontextu aktuální hodnota proměnné `errno`, v textovém kontextu aktuální systémové chybové hlášení.

`$@`

Chybové hlášení perlu z posledního příkazu `eval`, nuluje se, pokud `eval` proběhl bez chyby.

`$]`

Verze perlu, například 5.003.

`$$`

Číslo procesu.

`$<`

Reálné UID procesu.

`$>`

Efektivní UID procesu.

`$(`

Reálné GID procesu.

`$)`

Efektivní GID procesu.

`$[`

Index, od kterého se indexují normální pole a znaky v řetezcích, defaultně 0.

`$0`

Název našeho programu. Pokud ji změníme, změní se text například ve výpisu příkazem `ps`.

`$1`, `$2`, ... `$číslo`

Obsahuje vyhledané podřetězce z posledního úspěšného hledání či substituce, odpovídající částem regulárního výrazu uzavřeným v závorkách.

`$&`

Ten řetězec, který byl nalezen posledním úspěšným hledáním.

$` a $'

Podřetězec, který předcházel a následoval v posledním úspěšně prohledávaném řetězci před a za nalezeným vzorkem.

$+

Text v poslední nalezené závorce posledního hledání.

$*

Je-li tato proměnná nastavena na 1, předpokládá se při prohledávání řetězce, že obsahuje více řádků, 0 říká, že řetězce obsahuje jediný řádek. Je lepší používat volbu /m u jednotlivých hledání.

$.

Číslo aktuálního vstupního řádku posledně čteného filehandlu. Konstrukce <> nezavírá jednotlivé soubory, a proto se tato hodnota při čtení nového souboru pomocí <> nenuluje.

$/

Oddělovač vstupních řádků, defaultně \n. Nastavení na '' znamená čtení po odstavcích (oddělovačem je jeden či více prázdných řádků), pokud chceme číst celý vstup naráz, dáme undef $/.

$\

Oddělovač výstupních řádků. Defaltně není nastaven, takže print tiskne přesně to, co mu napíšeme.

$|

Má-li hodnotu 1, vypíná bufferování výstupu. Je vhodné např. při zápisu do roury, kdy výstup chceme dále zpracovávat.

$,

Oddělovač v seznamu pro příkaz print, tedy řetězec, kterým se nahradí každá čárka v seznamu. Defaultně není nastaven.

$"

Oddělovač prvků pole, které zapíšeme do řetězce uzavřeného uvozovkami. Defaultně mezera.

$;

Oddělovač, kterým se nahradí čárka v indexu do hashe. Takto můžeme emulovat vícerozměný hash v jednorozměrném.

$#

Výstupní formát pro tisk čísel, defaultně nastaven na %.20g.

$%

Aktuální číslo stránky aktuálního výstupu.

$=

Aktuální délka stránky aktuálního výstupu.

$-

Kolik řádků zbývá na stránce aktuálního výstupu.

$~

Název aktuálního výstupního formátu.

$^

Název aktuálního výstupního formátu pro hlavičku stránky.

$:

Seznam znaků, na nichž je možné zalomit řádek při použití výstupního formátu, defaultně ,\n-.

$^A

Aktuální hodnota akumulátoru funkce write, nastavovaná funkcí formline.

$^L

Určuje, jak formáty dělají formfeed, defaultně \f.

$^D

Aktuální hodnota debuggovacích příznaků.

$^F

Nejvyšší hodnota systémového file descriptoru, běžně 2.

$^I

Aktuální přípona pro in-place změny souborů.

$^P

Interní příznak perl debuggeru.

$^T

Čas startu běhu skriptu, v sekundách od 1. ledna 1970.

$^W

Aktuální hodnota přepínače -w.

$^X

Jméno perl binárního souboru.

$ARGV

Jméno aktuálního souboru čteného konstrukcí <>.

@ARGV

Obsahuje argumenty z příkazového řádku skriptu.

@INC

Seznam míst, kde perl bude hledat soubory načítané pomocí do, require nebo use.

%INC

Pro každý soubor načtený pomocí do nebo require obsahuje k jeho jménu jeho místo uložení v systému.

%ENV

Obsahuje aktuální nastavení proměnných prostředí.

%SIG

Používá se pro nastavení signal handlerů.

Perlovské funkce a klíčová slova

Funkce pro práce s poli či skaláry

chomp, chop, chr, crypt, hex, index, lc, lcfirst, length, oct, ord, pack, q/STRING/, qq/STRING/, reverse, rindex, sprintf, substr, tr///, uc, ucfirst, y///

Regulární výrazy, prohledávání textových řetězců

m//, pos, quotemeta, s///, split, study

Numerické funkce

abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand

Pole

pop, push, shift, splice, unshift

Seznamy

grep, join, map, qw/STRING/, reverse, sort, unpack

Hashe

delete, each, exists, keys, values

Vstupně/výstupní funkce

binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, rewinddir, seek, seekdir, select, syscall, sysread, syswrite, tell, telldir, truncate, warn, write

Funkce pro práci s daty s udanou délkou

pack, read, syscall, sysread, syswrite, unpack, vec

Soubory, adresáře, filehandly

`-X`, chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, stat, symlink, umask, unlink, utime

Klíčová slova pro řízení běhu programu

caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray

Klíčová slova pro definování rozsahu platnosti

caller, import, local, my, package, use

Další (nezatříděné) funkce

defined, dump, eval, formline, local, my, reset, scalar, undef, wantarray

Procesy a skupiny procesů

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx/STRING/, setpgrp, setpriority, sleep, system, times, wait, waitpid

Moduly

do, import, no, package, require, use

Objekty

bless, dbmclose, dbmopen, package, ref, tie, untie, use

Práce se sockety

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send,

setsockopt, shutdown, socket, socketpair

Funkce pro komunikaci (System V)

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Informace o uživatelích a skupinách

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Informace o síti

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Práce s časem

gmtime, localtime, time, times

Zajímavosti

Časopis The Perl Journal vyhlásil soutěž o co nejnečitelnější perlovský skript. Kategorii o nejlepší signaturu vyhrál Robert Klep s následujícím programem:

```
$Y=-1.2;for(0..24){$X=-2;for(0..79){($r,$i)=(0,0);for(0..15){$n=$_;$r=($x=$
r)*$x-($y=$i)*$y+$X;$i=2*$x*$y+$Y;$x*$x+$y*$y>4&&last}print unpack("\@$n a"
,".,:;=+itIYVXRBM ");$X+=3/80}$Y+=2.4/25}
```

Program spočítá a na obrazovku vytiskne Mandelbrotovu množinu.

Tento fragment kódu vrátí seznam souborů, setříděný podle data poslední změny:

```
sort { ($m{$a} ||= -M $a) <=> ($m{$b} ||= -M $b) } <*>;
```

Operátor `||=` je zde použit pro vytváření pomocného hashe `%m`. Tímto způsobem dosáhneme lineárního počtu volání `stat`.

`FMTEYEWTK` znamená Far More Than Everything You've Ever Wanted to Know a má URL `http://www.perl.com/perl/everything_to_know/index.html`.