Úvod do programovacího jazyka Perl

Patrick M. Ryan

patrick.m.ryan@gsfc.nasa.gov

listopad 1993

Obsah

1 Co je Perl?

Perl je programovacím jazykem primárně určeným jako pomůcka pro správu systému. Kombinuje v sobě prvky jazyka C, `awk`, `grep`u, `sed`u, a Bourne shellu. Perl je výtečným nástrojem pro zpracování textu. Ačkoli je Perl nazýván jazykem pro správu systému, lze jej bez problémů užívat i jinde, například pro prototypy složitěějších programů a rozhodně všude tak, kde byl až dosud užíván především skript psaný v shellu.

Slovo „Perl“ je zkratkou odvozenou z anglického *Practical Extraction and Report Language*. Perl vytvořil a dále jej rozvýjí Larry Wall. Perl je svobodně šířen pod GNU licencí a lze jej provozovat na většině užívaných architektur a operačních sytémů; sem patří především celé spektrum užívaných variant OS UNIX[1] dále VMS a dokonce i DOS a jeho nástupci.

Jak již bylo zmíněno, Perl se v některých rysech podobá Bourne shellu, `awk`, `sed`u, `grep`u a navíc umožňuje přistupovat k systémovým voláním a ke knihovním funkcím poskytovaných standardní knihovnou jazyka C a tím zaplňuje prostor mezi shelovskými scripty a programy v jazyce C.

Perl není kompilovaným jazykem, ale je rychlejší něž většina interpretovaných jazyků. Před spuštěním je Perlovský skript nejprve načten programem `perl` do paměti a jaksi předkompilován do rychlého vnitřního formátu. Ve vetšině případů je perlovský skript rychleší než jeho analogie naprogramovaná v Bourne shellu. V následujícím se na jazyk budeme odkazovat slovem *Perl* s velkým P a na program `perl`, či na skript napsaný v Perlu, slovem *perl* s malým p.

Tento dokument neobsahuje vyčerpávající popis jazyka Perl. Daleko obsažnější refence naleznete na konci tohoto článku.

2 Základy syntaxe Perlu

Perl je stejně jako jeho autor a celá myšlenka GNU velice volným a svobodomyslným jazykem co se syntaxe týče. Jeho syntax sice vychází z jazyka C, ale obsahuje i některé rysy jiných programovacích jazyků.

Perlovské programy (skripty) jsou většinou ukládány v souborech zakončených `.pl`. Avšak tato přípona není povinná a většina scriptů na UNIXech je spouštěna pomocí konstrukce `#!`. Proto první řádek skript psaného v Perlu bývá (alepoň ve světě OS UNIX):

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

V Perlu musí být každý příkaz zakončen středníkem (`;`). Text, který začíná znakem mřížka (`#`) je považován za komentář a je ignorován.

Bloky kódu v Perlu, např. v podmíněných výrazech nebo smyčkách, je, narozdíl od jazkyka C, *vždy* potřeba uzavírat do složených závorek (`{...}`).

3 Základní datové typy

Perl rozlišuje tři základní datové typy proměnných:

1. skalár

[1] UNIX je ochranná známka svého současného vlastníka

2. pole

3. asocialtivní pole (známé jako hashe)

3.1 Skaláry

Skalár je základním datovým typem Perlu. Skalár může být celé číslo, číslo s pohyblivou řádovou čárkou nebo znakový řetězec. To, jaký druh skaláru Perl užije, se určuje na základě kontextu, ve kterém je ta která proměnná uvedena. Jméno skalární proměnné začíná znakem dolar (`$`). Přiřazení řetězce do proměnné může vypadat takto:

```
$str = "hello world!";
```

avšak v žádném případě takto

```
str = "hello world!";
```

V Perlu je alfanumerický řetězec bez prefixu je obezně pokládán za řetězcový literál. Proto druhý uvedený výraz značí přiřazení řetězcového literálu `"hello world!"` do řetězcového literálu `"str"`.

Perlovský mechanismus užití uvozovek je obdobný mechanismu užívanému v Bourne shellu. Řetězce uzavřené do uvozovek (`"..."`) umožňují expanzi proměných na jejich hodnoty a interpretují známým escape znak zpětné lomítko ((\).V uvozovkách jsou hodnoty skalárních proměnných převáděny na řetězec. Řetězce uzavřené do apostrofů (`'...'`) neprovádějí žádnou expanzi proměýých a znaků začínajících zpětným lomítkem, kromě `\'` a `\\`.

Perlovské proměnné není nutné definovat předem. Proměnné jsou alokovány dynamicky v čase jejich použití. Je dokoce možné se odkazovat na neexistující proměnné. V numerickém kontextu neexistující proměnná hodnotu 0 a v řetězcovém kontextu hodnotu prázdného řetězce. Perl nabízí prostředky, kterými lze rozlišit, zda se jedná o neexistující proměnnou či existující proměnnou s nulovou hodnotou.

Perlovské proměnné jsou i tištěny a zpracovávány na základě kontextu. Proto řětězcová proměnná, která obsahuje numerické znaky je interpolována v numerickém kontextu na numerickou hodnotu. Mějme následující fragmen kódu:

```
$x = 4;          # celé číslo
$y = "11";       # řetězec
$z = $x+$y;
print $z,"\n";
```

Poté co je tento kód zpuštěn proměnná `$z`nabývá hodnoty 15.

Taková interpolace m;že probíhat i v opačném směru. Číselné hodnoty jsou v řetězcovém kontextu formátovány na řetězce. Proto není třeba provádět nějaké implicitní formátování jako v jazyce C či FORTRAN. Tento typ konverze je prováděn zejména při psaní na standardní výstup. Například:

```
$answer = 65;
print "the answer is $answer";
```

Výstupem tohoto kódu bude `the answer is 65`. Proměnná `$answer` je konvertována na řetězec `"65"` a nikoli na znak s ordinální hodnotou 65 (v ASCII znak `"A"`).

Číselné literály mohou být sopecifikovány některým z běžných formátů pro celá čísla nebo čísla s pohyblivou řádovou čárkou. Celočíselné konstanty lze uvádět i v oktalově či hexadecimálně:

```
$a = 12345          # celé číslo
$b = 12345.65       # číslo s pohyblivou řádovou čárkou
$c = 3.14E11        # scientific notation
$d = 0xABFF         # hexadecimálně
$e = 0377           # osmičkově
```

Řetězcové konstanty mohou být navíc užívány ve formě *here document* obdobně jako v shellu. Řetězcové konstanty tohoto typu začínají specifickým řetězcem a pokračují až do chvíle dalšího výskytu tohoto řetězce. Například v následujícím příkladu se do proměnné `$msg` uloží předformátovaná víceřádková řetězcová zpráva:

```
$msg = <<_EOM_;
               *======================================*
               | The system is going down. Log off now.|
               *======================================*
_EOM_
```

3.2 Skalární pole

V Perlovském scriptu lze používat pole (či seznamy) složené se skalárů. Jméno proměnné odkazující na pole skalárů obsahuje první znak zavináč (`@`). Kromě toho, že lze pole plnit prvek po prvku, lze pole naplnit naráz celé seznamem jeho jednotlivých prvků, přičemž seznam je uzavřen do kulatých závorek (`(...)`):

```
@numbers = (3,1,4,1,5,9);
@letters = ("this","is","a","test");
($word,$another.word) = ("one","two");
```

Na jednotlivé prvky pole se odkazujeme pomocí indexů v hranatých závorkách (`[...]`), přičemž první prvek pole ma defaultně v Perlu index 0 (tuto hodnotu lze změnit):

```
$blah[2] = 2.718281828;
$message[12] = "core dumped\n";
```

Všiměme si, že na prvek pole se odkazujeme jménem pole avšak s prefixem `$` (jde o skalár) následovaným indexem v hranatých závorkách. Proto `@pole` je jméno proměnné typu pole a `$pole[0]` je odkaz na první prvek tohoto pole.

Konstrukce `$#` následovaná jménem proměnné se užívá k nalezení posledního platného indexu pole. Proměnná `$[` obsahuje, případně nastavuje základní (nejnižší) možný použitelný index proměnné typu pole. Defaultně ja tato proměnná nastavena, jak již bylo řečeno, na hodnotu 0. Následující fragment kódu slouží pro zjištění délky pole:

```
# předpokládáme, že  @pole je pole nějakých zajímavých prvků
$n = $#pole - $[ + 1;
print "pole pole má $n prvků\n";
```

Pole stejně jako proměnné jsou alokovány dynamicky prvek po prvku, čili jakmile přiřadíme hodnotu do dalšího prvku pole. Pole lze alokovat předem tak, že přiřadíme do proměnné `$#`*jméno_pole* příslušnou hodnotu.

```
$#months = 11; # pole @months má prvky o indexech  0..11
```

Perl nabízí velké množství nástrojů pro práci s poli; nabízí prostředky pro vkládání, vyjímání, přidání, rozdělení a slučování polí.

Perl umožňuje pracovat pouze s jednorozměrnými poli. Vícerozměrná pole lze však pomocí prostředků Perlu jednoduše napodobit.

3.3 Asociativní skalární pole

Asociativní pole (hash) je v Perlu implementováno pomocí rozptýlených tabulek. Asociativní pole jsou asi nejužitečnějším risem jazyka Perl. Běžné aplikace užívající asociativních polí zahrnují například vytváření tabulek uživatelů klíčované logovacím jménem nebo vytváření tabulek jmen souborů. Prefixem pro označení proměnné typu hash je znak procento (`%`)

K položkám asociativního pole se nepřistupuje přes číselné indexy jako u pole běžného, ale přes indexy (klíče) řetězcové (numerické klíče jsou převáděny na řetězce). Asiciativní pole lze expicitně naplnit seznamem dvojic *položka–hodnota*. Například:

```
%quota = ("root",100000,
          "pat",256,
          "fiona",4000);
```

K položkám pole se přistupuje následujícím způsobem:

```
$quota{dave} = 3000;
```

V tomto příkladě je `dave` klíč (index) a `3000` hodnota. Všiměme si, že se v tomto případě odkazujeme na položku, která je skalárem a proto jméno proměnné pole začíná znakem `$`.

Dalším příkladem může být v Perlu předdefinované pole `%ENV`, které obsahuje hodnoty proměnných prostředí. Klíčem pole je jméno proměnné prostředí. Následuje kousek kódu, který zjistí zda pracujeme v prostředí X Window:

```
if ($ENV{DISPLAY}) {
  print  "Pravděpodobně pracujete v X Window\n";
}
```

V Perlu mámě k dispozici rutiny pro procházení asociativními poli a pro mazání prvků pole. Rutiny each, keys, values a delete.

Na tomto místě je vhodné poznamenat, že v Perlu můžete pracovat bez konfliktu se skalární proměnnou, polem, asiciativním polem, subrutinou a balíkem, které budou mít všechny stejné jméno.

4 Operátory

Množina operátorů užívaných Perlem obsahuje téměř všechny operátory užívané v jazyce C. Všechny obviklé aritmetické operátory užívané v C lze užít i v Perlu. V následujícím seznamu jsou uvedeny operátory, které zná Perl oproti C navíc. Popis jednotlivých operátorů je parafrázování textu z excelentních manuálových stránek Perlu:

`**`	umocnění (x^3 v Perlu `$x**3`)
`**=`	umocnění s přiřazením
`()`	prázdný seznam. Slouží k vynulování pole.
`.`	spojení dvou řetězců
`.=`	spojení s přiřazením
`eq`	srovnání shodnosti řetězců (`==` je numerická rovnost).
`=~`	vyhledání, substituce nebo nahrazení (defaultně v řetězci `$_`)
`x`	operátor opakování. Počet opakování uvádí pravý operand
`..`	operátor rozsahu
`-f, -x, -l,...`	unární operátor pro testování souborů, obdoba příkazu `test(1)`

Podrobnější popis operátorů naleznete v manuálové stránce `perlop(1)`.

5 Implicitní argument

Mnoho funkcí a syntaktických struktur v Perlu užívá implicitních argumentů. Ve většině případů je implicitním argumentem proměnná `$_`.Tato vlastnost Perlu je užitečná zejména pro pokročilé perlovské programátory avšak činí kód perlu pro začátečníka téměř nečitelným.

Proto raději všem nováčkům, kteří ještě nechápou jakým způsobem Perl pracuje s proměnnou `$_`, doporučujeme, aby všude uváděli argumenty funkcí a operátorů explicitně. V různých případech Perl zpracovává proměnnou `$_` jiným způsobem a většinou vše dosti podstatně závisí na kontextu.

Až budete mít za sebou několik obsáhlejších prográmků napsaných v Perlu tak se naopak neostýchejte imlicitních proměnných používat. Pak teprve budete psát rychlé (a tajemně zašifrované) Perlovské kódy.

6 Regulární výrazy

Kdokoli jednou použil program `grep` nebo `expr` k vykledání nebo porovnání řetězců bude jistě potěšen, že takové věci lze v Perlu implikovat přímo do kódu a tvoří tak další podstatný rys jazyka Perl.

6.1 Speciální znaky v regulárních výrazech

Regulární výrazy v Perlu mají obdobnou formu jako v jazyce `vi`.

`.`	libovolný znak, kromě znaku nový řádek.
`+`	alespoň jeden výskyt předchozího znaku.
`?`	žádný nebo jeden výskyt předchozího znaku.
`*`	žádný nebo více výskytů předchozího znaku.
`[...]`	skupina znaků, vyhovat musí právě jeden.
`[^...]`	všechny znaky kromě uvedených v závorkách.
`{N,M}`	minimálně `N` krát a maximálně `M` krát opakování znaků.
`(...)`	skupina znaků později použitelná jako jeden prvek (proměnná nebo `\1 - \9`).
`(..|..|..)`	jedna z alternativ.
`\d`	tj. `[0-9]`
`\D`	tj. `[^0-9]`
`\w`	tj. `[a-zA-Z0-9_]`
`\W`	tj. `[^a-zA-Z0-9_]`
`\s`	tj. `[ \r\t\n\f]` (mezera,CR,tabelátor,LF,)
`\S`	tj. `[^ \r\t\n\f]`
`\1 - \9`	řetězec dříve nalezený uzavřený v `()`.
`\b`	hranice slova `\B` není hranice slova
`^`	začátek řetězce
`$`	konec řetězce
`\n,\r,\f,\t`	mají svůj obvyklý význam

Syntaxe výrazu pro vyhledávání vzoru v řetězci je `m/`*vzor*`/gio`, kde modifikátory mají nsáledující význam: `g` (global) globální prohledávání celého řetězce, `i` (ignore case) ignoruje se velikost znaků, `o` (only once) tento regulární výraz se kompiluje pouze jednou. Spolu s příkazem `m` lze užít k ohraničení argumentů libovolného páru nealfanumerických znaků. Toto je velice užitečné, zejména tehdy, hledáme-li v řetězci jména souboru, které obsahuje znak `/`, například:

```
if (m!^/tmp_mnt!) {
    print "$_ is an automounted file system\n";
}
```

V případě, že *vzor* je ohraničen znaky `/` je uvedení počátečního `m` nepovinné.

Perl umožňuje i vyhledávání na více řádkách, vice se dočtete v obsáhlejší dokumentaci vyhledáte-li si odhaz na proměnnou `$*`.

6.2 Získání vyhledaného řetězce

Obdobně jako ve `vi`, `grep`u či `sed`u umožňuje Perl další zpracování řetězce, který splňuje daný vzor regulárního výrazu. Například následující kód emuluje UNIXový příkaz `basename(1)`:

```
$file = "/auto/home/pat/c/utmpdmp.c";
($base) = ($file =~ m|.*/([^/]+)$|);
```

Výsledkem této části programu je hodnota `utmpdmp.c` v proměnné `$base`. Kulaté závorky v regulárním výrazu vymezují tu část vzoru, který chceme extrahovat.

Návratová hodnota regulárního výrazu závisí na kontextu. V kontextu pole, výraz vrací pole řetězců, které splňují daný vzor. Ve skalárním kontextu (většinou v testu zda řetězec splňuje daný regulární výraz) výraz vrací hodnotu 0 (ne) nebo 1 (ano). V následujícím příkladu uvádíme

použití skalárního kontextu. Konstrukce `<STDIN>` načte jeden řádek ze standardního vstupu (detaily viz níže):

```
$response = <STDIN>;
if ($response =~ /^\s*y/i) {
      print "you said yes\n";
}
```

Znovu připomínáme, že pro mnohé konstrukce v Perlu je třeba rozlišovat skalární kontext a kontext pole (seznamový kontext), jelikož jejich návratová hodnota podstatnou měrou na tomto kontextu závisí.

7 Řídící struktury

Perl poskytuje všechny řídící struktury poskytované běžným procedurálním jazykem a jak je u Peerlu zvykem nabízí ještě některé další.

7.1 `if-else`

Perlovský příkaz `if` má stejnou strukturu jako v jazyce C. Perl používá i stejné booleovské operátory jako C:

`&&`	logický součin (and)
`||`	logický součet (or)
`!`	negace (not)

Narozdíl od jazyka C neumožňuje Perl za podmínkovým výrazem[2] uvézt pouze jeden příkaz, ale vyžaduje vždy užít příkazový blok. Tedy jinak za podmínkovým výrazem musí být následné tělo vždy uzavřeno do složených závorek a to i v případě, že se jedná pouze o jediný příkaz. Například v tomuto kódu v jazyce C:

```
if (error < 0)
  fprintf(stderr,"error code %d received"n",error);
```

odpovídá v Perlu:

```
if ($error < 0)
 { print STDERR "error code $error received\n"; }
```

Perlovské analogie céčkovského `if-else` jsou `else` a `elsif` a jejich použití je zřejmé.

Perl nabízí příkaz `unless`, které obrací smysl podmínkového výrazu. Například:

```
unless ($#ARGV > 0) # jestliže nemá script argument
  { print "error; no arguments specified\n"; exit 1; } # skonči
```

Perlovské pojetí pravdu je shodné s pojetím v jazyce C. V numerickém kontextu je nulová hodnoto považována za "false" a cokoli nenulového za "true". Prázdný řetězec je považován za

[2] a to nejen u příkazu `if`, ale i u příkazů `unless, while, foreach`...

“false” a řetězec délky větší než 1 za “true”. Pole a asociativní pole jsou považována za “true” majíli alepoň jeden prvek a za “false” v případě. že jsou prázdná. Neexistující proměnná je vždy “false”.

Perl nemá obdobu řídící struktury `case`, jelikož tato struktura lze jednoduše nahradit jinými prostředky poskytovanými Perlem.

7.2 Příkaz `while`

Perlovský příkaz `while` je velice universální. Jelikož Perlovské nazírání na pravdu je poměrně pružné může být jako podmínka užito téměř cokoli. Například lze namísto podmínky užít, stejně jako v C, libovolný aritmnetický výraz, přiřazení či funkci.

Příkaz `<STDIN>` bez argumentů znamená přiřazení jednoho řádku ze standardního vstupu universální proměnné `$_`.

```
while (<STDIN>) {
  print "na vstupu bylo: ",$_;
}
```

Budeme-li se držet doporučené začátečnické praxe ohledně implicitních parametrů bude kód vipadat takto:

```
while ( $_ = <STDIN>) {
  print "na vstupu bylo: ",$_;
}
```

Jak jsme uvedli výše, pole je „pravdivé“ v případě, že mu zůstáva ještě nějaký prvek. Například:

```
@users = ("nigel","david","derek","viv");

while (@users) {
   $user = shift @users;
   print "Uživatel $user má na tomto počítači uživatelský účet\n";
}
```

Tato smyčka bude probíhat dokut bude v poli `@users` alespoň jeden prvek. Rutina `shift` vyjme první prvek z pole a navrátí jeho hodnotu.

Pro předčasné ukončení smyčky užívá Perl dvou klíčových slov `next` a `last`. Příkaz `next` je obdoba příkazu `continue` v jazyce C, tj. smyčka ihned v tomto místě programu pokračuje další iterací. Příkaz `last` je analogií příkazu `break` v jazyce C, tj. smyčka se ukončí.

7.3 Příkaz `for` a `foreach`

Perlovské příkazy `for` a `foreach` jsou identické, tj. lze je oba užít ve stejném významu v libovolném kontextu.

Aby se věc ještě více zamotala, existují dva povolené způsoby syntaxe těchto příkazů. První způsob je převzatý z jazyka C:

```
@disks = ("/data1","/data2","/usr","/home");
for ($i=0; $i != $#disks; ++$i) {
     print $disks[$i],"\n";
}
```

Jakmile však jednou porozumíte způsobu jakým se v Perlu pracuje s poli nebudete používat tuto céčkovskou, tříargumetovou, syntax.

Perl užívá syntax jednoargumentovou obdobně jako se užívá, například, v C-shellu. V příkazu `foreach` se jako argument uvede pole (seznam) a příkaz postupně prochází toto pole prvek po prvku, avšak narozdíl od předchozí demonstrace pomocí příkazu `shift` nedojde k destrukci pole.

Předchozí příklad můžeme přepsat, například, takto:

```
@disks = ("/data1","/data2","/usr","/home");
foreach(@disks) {
    print $_,"\n";
}
```

Ještě jenou připomeneme, že toto řešení je daleko elegantnější a nevede k destrukci pole `@disks`. Je tomu tak proto, že v tomto případě proměnná `$_` není neobsahuje kopii hodnoty prvky pole, ale je jakýmsi ukazatelem na daný prvek, proto veškeré změny prvedené na `$_` se projeví i na daném prvku pole.

7.4 Příkaz `goto`

Ano, je to tak i Perl obsahuje tolik zatracovaný příkaz `goto`. Příkaz `goto` *label* přesměruje zpracovávání programu na místo označené značkou *label*. Avšak v Perlu stejně jako všude jinde platí: *„Pokud skutečně nemusíte příkaz“ `goto` nepoužívejte!*

8 Vestavěné funkce Perlu; knihovní a systémové funkce

Perl obsahuje bohatou sadu vestavěných funkcí a umožňuje přístup k nejsužívanějším funkcím standardní knihovny jazyka C. Manuálové stránky k Perlu popisují dopodrobna všechny tyto funkce. My se v tomto úvodu omezíme pouze na několik nejčastěji používaných funkcí. Většina těchto funkcí užívá implicitní proměnnou `$_` Kulaté závorky kolem argumentů funkcí, jsou většinou nepovinné.

8.1 Vestavěné funkce

`chop` *výraz* Odsekne poslední znak z řetězce a tento znak vrátí. Taková funkce se může jevit ne moc zajímavá a užitečná dokud neporozumíme jak Perl pracuje se souborovým vstupem a výstupem. Při načítání řádku do proměnné Perl zachová i znak pro konec řádku `\n`, což je velice výhodné při implementaci filterů, ale například pro interaktivní programy je tato vlastnost nežádoucí. Zde dobře poslouží funkce `chop`.

`defined` *výraz* Určuje zda výraz má či nemá hodnotu.

`die` *výrazy* Vytiskne závěrečnou zprávu a ukončí skript. Tato funkce se užívá v případě, kdy se objeví nějaká zásadní chyba za běhu programu např., když nelze otevřít soubor.

`each` *pole* Navrací iterativním způsobem dvojice klíč-hodnota asociativního pole.

`join` *výraz,pole* Spojí jednotlivé prvky *pole* do jednoho řetězce, v němž budou jednotlivé prvky odděleny hodnotami *výraz*u.

`pop` *pole* Vyjme a vrátí poslední hodnotu pole, přičemž zkrátí pole.

`print` *výraz* Vytiskne své argumenty. Více o této funkci později.

`push` *pole,seznam* Pokládá pole za zásobník a uloží hodnoty uvedené v seznamu na konec pole.

`shift` Vyjme první hodnotu s pole, délku pole zkrátí o 1 a posune všechny prvky spměrem k počátku pole. `shift` a `unshift` pracují obdobně jako `push` a `pop` s tím rozdílem, že `shift` a `unshift` pracují nad počátkem pole.

`split(/`*vzor*`/,`*výraz*`,`*limit*`)` Rozdělí řetězec na jednotlivé prvky a tyto prvky navrátí jako pole. *Vzor* určuje regulárním výrazem řetězec, který uděluje jednotlivé prvky. Běžným užitím této funkce je rozebrání řádků v UNIXovém `/etc/passwd` na jednotlivé komponenty. Parametr *limit* určuje maximum prvků, na které lze řetězec rozdělit.

`substr` *výraz,offset,délka* Vyjme podřetězec z řetězce daného *výraz*em a tento podřetězec vrátí. Podřetězec je vyňat z řetězce počínaje na znaku *offset* od počátku řetězce. Zaporný *offset* určuje offset od konce řetězce. Parametr *délka* určuje délku podřetězce.

8.2 Funkce podobné UNIXovým příkazům

`chmod` Změní přístupová práva k uvedenám souborům.

`chown` Změní vlastníka a skupinu uvedených souborů.

`mkdir` Vytvoří adresáře.

`unlink` Smaže soubor.

`rename` Přejmenuje soubor.

`rmdir` Smaže adresář.

8.3 Knihovní funkce

Perl umožňuje jednoduše používat velkou část knihovních funkcí jazyka C.

`getpw, getgr, ...` Perl má přístup ke všem funkcím pracujících s přístupovými právy, uživately, skupinamy, informacemi o systému atd..

`bind, connect, socket, ...` Funkce pro meziprocesovou komunikaci (IPC); pro práci s BSD sockety, FIFO, rourami atd.

`stat` Informace o přístupových právech k danému souboru získané knihovní funkcí `stat(2)`.

`exit` Ukončí skript s určením návratového stavu.

9 Spolupráce s operačním systémem

Systémové příkazy lze v Perlu spouštět několika způsoby.

Perl buď může užít systémového volání `system(3)`. Daný řetězec je předán k spracování příkazovému interpretru (shellu). Výstup prováděného příkazu je poslán na standardní výstup. Návratový status provedého příkazu je uložen v proměnné `$?`.

Dalším způsobem, který Perl umožňuje, je uzavřít systémový příkaz do zpětných uvozovek (‘) obdobně jako je tomu v shellu. Tento způsob se užívá tehdy, kdý chceme odchytit výstup prováděného příkazu. Například v následujícím příkladě spustíme systémový příkaz `hostname(1)`, jehož výstup odchytíme do proměnné `$host` a funkcí `chop` odřízneme nechtěný znak pro konec řádku:

```
$host = ‘hostname‘; chop($host);
```

I v tomto případě je návratový status uložen do proměnné `$?`.

10 Práce se soubory

Perl poskytuje I/O funkce pro čtení a zápis prováděných nad textovými a nad „neformátovanými“ soubory.

10.1 Textový vstup a výstup

Perl přistupoje k souborů pomocí speciální proměnné ovladače souboru (filehandle). Tyto proměnné je zvykem označovat velkými písmeny.

Soubory se otevírají pomocí funkce `open`. Tato funkce má dva parametry: ovladač souboru a jméno souboru. Jméno souboru obvykle předcházejí modifikátory. Jeden řádek souboru lze získat pomocí operátoru `<>`, jehož parametrem je obladač souboru: `<`*ovladač_souboru*`>`. Například:

```
open(F,"data.txt");
while($line = <F>) {
   # dělej ze vstupem něco zajímavého
}
close F;
```

Jménu souboru může předcházet modifikátor. Je-li modifikátorem znak `<` je soubor otevřen pro čtení (to je implicitní nastavení). Předchází-li znak `>` je soubor otevřen pro zápis. V případě, že soubor již existuje je jeho obsah přepsán novým obsahem. Nakonec je-li prefixem znak `>>` je soubor otevřen pro zápis, přičemž nově zapsaný text se připojuje existujícímu obsahu souboru. Následuje několik příkladů:

```
# užití souboru /etc/passwd
open(PASSWD,"</etc/passwd");
while ($p = <PASSWD>) {
    chop $p;
    @fields= split(/:/,$p);
```

```
    print "Uživatel $fields[0] má domácí adresář $fields[5] \n";
}
close PASSWD;

# přidání informací do logového souboru
open(LOG,">>user.log");
print LOG "user $user logged in as root\n";

# přečte jeden řádek zadaný uživatelem
$response = <STDIN>;
```

Perl má předdefinovány ovladače stadardního vstupu, výstupu a chybového výstupu: `STDIN`, `STDOUT` a `STDERR`.

Pokud se operátor `<>` objeví v kontextu pole jeho chování se změní. Nevrátí pouze jeden řádek, ale pole řádků celého souboru. Například:

```
$file = "nejaky.soubor";
open(F,$file);
@lines = <F>; # Nasaji celý soubor .. mňam, mňam,...
close F;
```

Ačkoli je tato vlastnost užitečná, je třeba užívat jí velice opatrně, jelikož celý soubor je načten do operační paměti, která může být menší něž načítaný soubor. Jelikož Perl užívá bufferovaných I/O operací není moc výhodné načítat celé soubory kvůli urychlení perlovských skriptů.

10.2 Roury

V Perlu lze užít funkci `open` nejen k načítání souborů, ale i ke čtení výstupů jiných procesů (systémových příkazů) nebo k zápisu na standardní vstup jiného programu, obdobně jako u funkce jazyka C `popen(3S)`.

Začíná-li v argumentu funkce `open` jméno souboru znakem roury (`|`) je jméno souboru považováno za systémový příkaz. Tento příkaz je spuštěn a na jeho vstup lze posílat výstup ze skriptu, např. příkazem `print`.

Je-li znak `|` uveden na konci jména souboru, je tento řetězec považovaán za systémový příkaz a je spuštěn. Jeho výstup lze poté číst operátorem `<>`.

```
open(MAIL," | mail root"); #  zaslání e-mailu administrátorovi
print MAIL "Tady se děje něco nekalého\n";
close MAIL; # a nyni je dopis odeslán

open(WHO,"who|"); # podíváme se, kdo je nalogován
while ($who = !WHO?) {
      chop $who;
      ($user,$tty,$junk) = split(/"s+/,$who,3);
      print "Uživatel $user je napojen  na  terminál $tty\n";
}
close(WHO);
```

10.3 Neformátované soubory

Perlovské funkce `sysread` a `syswrite` umožňují přímé čtení a zápis do souborů po bytech. Podprobnosti naleznete v dokumentaci.

10.4 Funkce `print`

Již jsem měli možnost vidět jak pracuje funkce `print`. Popišme nyní tento příkaz podrobněji.

Obecně, příkaz `print` vezme řadu řetězců oddělených čárkou, provede potřebné substituce hodnot proměnných a vytiskne výsledek. Spolu s příkazem `print` se často užívá spojovací operátor (`.`). Všechny náseldující řádky kódu dávají stejný výsledek.

```
print "But these go to 11.\n";
$level = 11;
print "But these go to ",$level,".\n";
print "But these go to $level.\n";
printf "But these go to %d.\n",$level;
print "But these " . "go to " . $level . ".\n";
print join(' ',("But","these","go","to","$level.\n"));
```

Jak jste jistě postřehli, máme v Perlu k dispozici i funkci `printf`, který se chová obdobně jako `printf()` v jazyce C.

Jak jsem viděli v některém z předchozích příkazů, funkce `print` může pracovat volotelným parametrem ovladače souboru, který se od dalších argumentů *neodděluje* čárkou.

11 Poznámky o kontextu pole

V jazyce C každý výraz vrací hodnotu. Tato hodnota může být vstupem další funkce, aniž by byla potřeba nějaká dočasné proměnná. Lze tedy psát, například, takovéto zápisy: `chdir(getenv("HOME"))`

V Perlu navrací mnohé funkce pole. Tato výsledná pole se mohou stát vstupem další funkce podobně tak, jak bylo popsáno výše. Tento přístup snižuje nutnost zbytečných dočasných proměnných.

Následuje několik příkladů. První z nich užívá funkci `sort`, která vrátí setříděné pole, které je uvedeno jako argument.

```
@names = ("bill","hillary","chelsea","socks");
@sorted = sort @names;
foreach $name (@sorted) {
  print $name,"\n";
}
```

Iteraci však můžeme provádět přímo na setříděném poli.

```
foreach $name (sort @names) {
  print $name,"\n";
}
```

Další příklad ukazuje, že lze indexovat přímo navrácené pole:

```
$name = (getpwuid($<))[6];
print "Mé shutečné jméno je: ",$name,"\n";
```

Funkce `getpwuid` vrací pole, a jelikož z tohoto pole chceme pouze položku označující skutečné jméno (GECOS) v `/etc/passwd` odkazujeme se přímo na sedmý prvek navráceného pole (jsme v kontextu pole) a tento prvek ukládáme do proměnné `$name`.

12 Podprogramy a balíky (package)

Perl umožňuje modulární programování tím, že poskytuje prostředky na tvorbu podporgramů a knihoven.

12.1 Podprogramy

Perlovské skripty mohou užívat i uživatelem definované funkce, jež mají své parametry a navracejí hondoty. Níže je uvedena kostra definice takového podprogramu `podprogram`.

```
sub podprogram {
  local($param1,$param2) @_;
  # dělej něco užitečného
  $hodnota
}
```

Tento podprogram lze poté zavolat takto:

```
$return_val = do sub1("this is","a test");
```

klíčové slovo `do` lze nahradit znakem `&`

```
$return_val = &sub1("this is","a test");
```

Při psaní podporgramů je třeba mít na paměti několik věcí. Parametry jsou uvnitř podporgramu uloženy v poli `@_`. Jelikož všechny proměnné jsou implicitně globální, užili jsme funkci `local()`, která specifikovala dané proměnné jako lokální.

Perl obsahuje příkaz `return`, kterým explicite definujeme návratovou kondnotu funkce. Většinou se však `return` neužívá, jelikož za návratovou hodnotu podprogramu bere Perl hodnotu posledního provedeného výrazu. Proto, chceme-li, aby podprogram navráti hodnotu 0, uvedeme jako poslední řádek podprogramu `0;`.

12.2 Packages

Perl obsahuje knohovnu všech možných užitečných podprogramů a funkcí, které lze do skriptu. Perlovská analogie k céčkovskému `#include` je `require`.

Například, Perl má knihovnu pro prohledávání argumentů uvedených na příkazovém řádku podobnou funkci `getopt(3)` v jazyce C.

```
require 'getopts.pl';
&Getopts('vhi:');
if ($opt_v) {
  print "verbose mode is turned on\n";
}
```

Samozřejmě, že je možné psát vlastní knihovny a moduly a tyto pak vkládat do svých skriptů.

13 Předdefinované proměnné

Perl ovsahuje mnoho předdefinovaných proměnných, které jsou podorbně dokumentovány v manuálvých stránkách. Zde uvedeme jen ty nejužívanější.

`S_` Implicitní proměnná pro většinu funkcí a syntatických struktur.

`$?` Stavové slovo navrácené posledním voláním systémového příkazu. Dolní bity obsahují informaci o sygnálu, kterým byl příkaz ukončen. Horní bity obsahují informaci o návratovém kódu.

`$$` Číslo procesu běžícího skryptu.

`$<` UID uživatele, který skript spustil.

`@ARGV` Paramtry skryptu z příkazové řádky. Pozor, `$ARGV[0]` je první skutečný argument a nikoli jméno skriptu, to jlze nalézt v proměnné `$0`;

`%ENV` Asociativní pole, které obsahuje proměnné prostředí.

Reference

Knihy:

1. Larry Wall *Programming Perl*, O'Reilly (v češtině Computer Press)
2. Randal Schwartz *Learning Perl*, O'Reilly

Online help a internet:

1. manuálové stránky
2. htt://www.perl.com/CPAN/ — CPAN – FAQ, Perl, příklady, moduly, knihovny,...
3. http://www.perl.com/
4. newsgroup `comp.lang.perl`