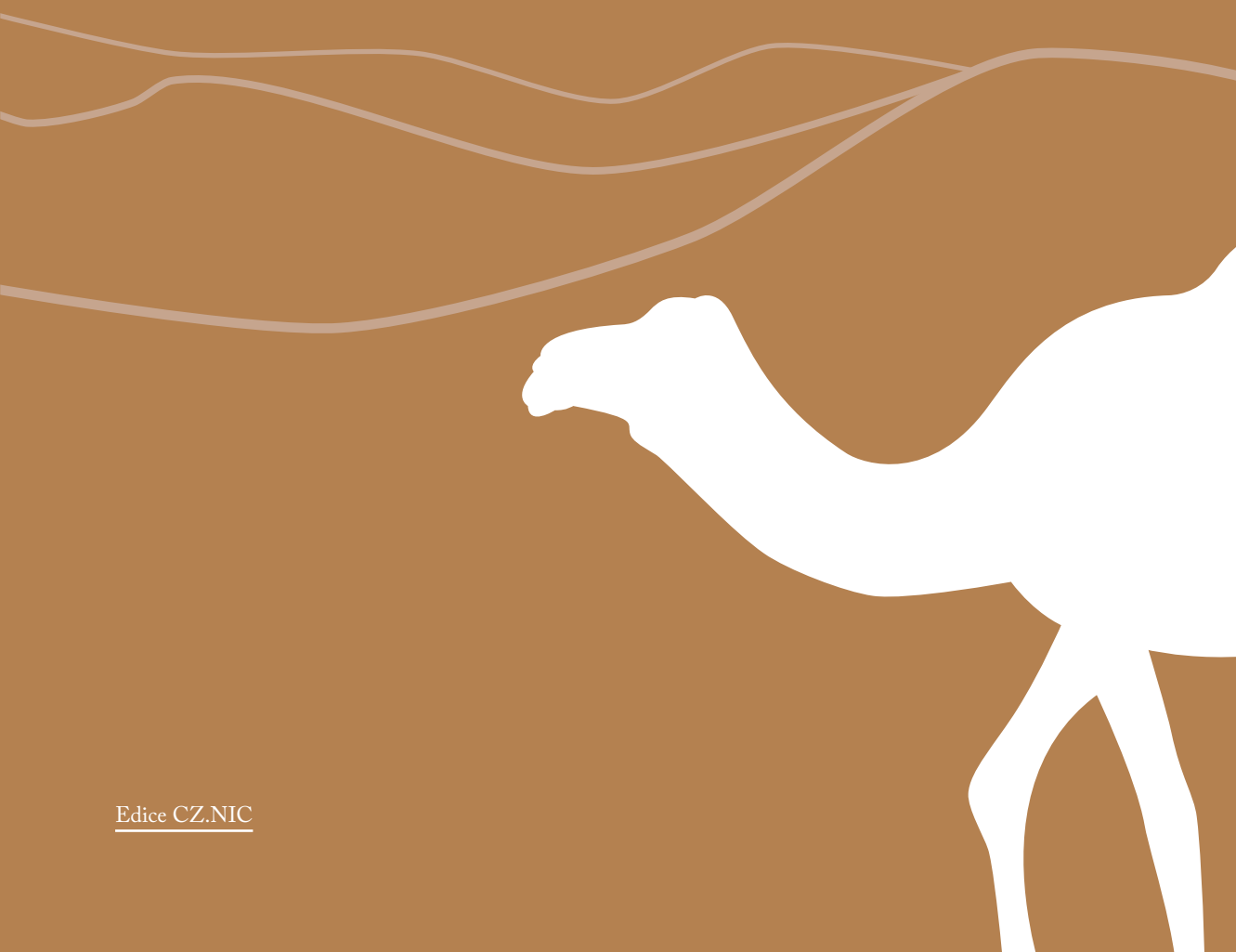
Pavel Satrapa

# Perl
# pro zelenáče



Edice CZ.NIC

**PERL PRO ZELENÁČE**
Pavel Satrapa

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

3. aktualizované a rozšířené vydání, Praha 2018
Kniha vyšla jako 19. publikace v Edici CZ.NIC.
ISBN 978-80-88168-38-6



CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-88168-38-6

— Pavel Satrapa

# Perl pro zelenáče

— Edice CZ.NIC

# Předmluva vydavatele

## Vážení čtenáři,

Pavla Satrapu není třeba příliš představovat, jeho díla o programování nebo Linuxu jsou velmi populární. V roce 2011 jsme v naší edici uvedli třetí vydání jeho knihy o IPv6 a v tuto chvíli od něj držíte v ruce učebnici programování v jazyce Perl 5, který má za sebou již více než třicetiletou historii.

Autor byl k třetímu aktualizovanému vydání knížky „donucen" častými dotazy stále silné komunity uživatelů tohoto skriptovacího jazyka, kteří mají předešlá vydání díky častému používání v salátové podobě a nebo jim v nich schází novinky, které jsou v tomto trvale se rozvíjejícím jazyce k dispozici. Kniha tedy vychází po sedmnácti letech znovu, a pokud jste četli předchozí verze, čekejte v této upravenou kapitolu o objektově orientovaném programování a zcela novou část o funkcionálním programování.

Sám jsem Perl nikdy aktivně nepoužíval. Potkávám jej ale po celou dobu své IT kariéry. Nejčastěji jsem Perl slyšel skloňovat správce unixových systémů, kteří jej vždy hojně využívali pro psaní jednorázových skriptů, například při zpracování statistik z rozsáhlých logů. I oni však na Perlu dokázali postavit složitější modulární projekty. Také proto jsem dnes, když jsem o vydání této knížky mluvil se svými kolegy, narazil na jiskru v oku nejen u administrátorů, ale také u vývojářů.

Je jasné, že co se týče rychlosti, nemůže Perl soutěžit s kompilovanými jazyky (C++) nebo s jazyky kompilovanými za běhu (Java), ale co se týče skriptovacích jazyků, není na tom s výkonem nijak špatně. Oceňovanou vlastností Perlu je i to, že regulární výrazy jsou integrální součástí jazyka. A s oblibou využívaný je i archív CPAN, který obsahuje neuvěřitelné množství rozšiřujících modulů k řešení snad všeho myslitelného.

V Perlu vznikla celá řada velmi kvalitních open source aplikací. Kdo z vás nikdy nepoužil nebo alespoň nezná Request Tracker, AWstats nebo SpamAssassin? A co git-svn, colordiff, autotools? Stále málo důkazů, jak moc je pro nás Perl důležitý, a málo důvodů ponořit se do čtení a studia? Tak snad ještě jeden. Tato knížka vás naučí základům Perlu příjemnou a veselou formou, typickou pro autora.

Kdesi jsem četl komentář o nějakém hutném technickém textu: „… ale není to žádný Satrapa". Tato knížka ale Satrapa je!

Hezkou zábavu při čtení a vlastním perlení!

**Zdeněk Brůna, CZ.NIC**
*Sklenařice, 12. října 2018*

# Předmluva

## Předmluva

Perl je programovací jazyk, který se v současné době těší značné oblibě. Přestože vůči němu lze mít z teoretického hlediska vážné námitky, je to jazyk zajímavý a mimořádně praktický. Dokáže totiž snadno a rychle vyřešit leckteré i dosti složité problémy.

Cílem této knihy je naučit vás programovat v Perlu. Nejedná se o kompletní referenční příručku – ta je k dispozici v češtině v podobě knihy [1] a má téměř 700 stran. Perl pro zelenáče je učebnice, která vás postupně zasvětí do tohoto programovacího jazyka.

Snažil jsem se postupovat po jednotlivých logických celcích, abyste se co nejrychleji naučili vytvářet jednoduché programy a své znalosti postupně prohlubovali. Kniha je rozdělena do čtyř částí. První obsahuje úvod do jazyka, jeho základní datové typy a příkazy. Dozvíte se také něco o ladění programů. Druhá část popisuje největší lákadla Perlu – regulární výrazy a asociativní pole. Kromě toho se věnuje podprogramům a prohloubí vaše znalosti vstupů a výstupů. Třetí část knihy už je věnována programátorské vyšší dívčí, jako je modulární a objektově orientované programování či odkazy. Popisuje také zcela praktické apsekty Perlu pro styk s okolním světem, především databázemi a programování pro WWW. Čtvrtá část obsahuje přílohy. Najdete zde řešení ke cvičením a návod pro instalaci samotného Perlu i jeho modulů.

Mějte na paměti, že programování je do značné míry dovednost. Nestačí přečíst si haldy moudrých knih, potřebujete praxi. Proto jsem do knihy kromě řady příkladů zařadil i cvičení. Dovolím si na vás apelovat, abyste se je pokusili vyřešit. Vlastní zkušenosti totiž podstatným způsobem prohloubí vaše znalosti a dostanou vám Perl „do krve“. Své výsledky pak můžete konfrontovat s mými řešeními, která najdete v příloze 17 na straně 253.

Kniha striktně nepožaduje, abyste již uměli programovat, nicméně považuji to za významné plus. Přes vřelé city, které k tomuto jazyku chovám, totiž Perl nepovažuji za vhodný k tomu, abyste se na něm učili základům programování.

Chtěl bych poděkovat celé své rodině za veškerou podporu, kterou mi při vytváření knihy poskytla. Dále patří dík všem pokusným čtenářům, kteří svými poznámkami a návrhy přispěli k doladění konečné podoby textu. Jmenovitě to jsou Petr Kolář, Jiří Novák, David Kmoch, Jakub Steiner a Petr Titěra.

**Pavel Satrapa**
*Liberec, leden 2000*

## Předmluva ke třetímu vydání

První vydání vyšlo v roce 2000, rychle se rozebralo a bylo o rok později následováno druhým. Po čase se vyprodalo i to a tím jsem věc považoval za uzavřenou. V poslední době mi však dorazilo několik dopisů, jejichž autoři měli o knihu zájem a sháněli, kde by se dala zakoupit. Usoudil jsem proto, že má smysl pokusit se o nové vydání.

Krajina programovacích jazyků se za tu dobu významně změnila. Velká lákadla Perlu, jimiž jsou zejména regulární výrazy a asociativní pole, se mezitím objevila v řadě dalších jazyků. To mu ubralo na atraktivitě, nicméně stále má co nabídnout. Jedná se o dobře dostupný a silný jazyk, ve kterém lze rychle a celkem snadno psát netriviální aplikace. Programátora neotravuje, i když někdy dokáže pořádně kousnout do nohy.

Při práci na třetím vydání jsem se snažil knihu aktualizovat podle současných zvyklostí programování v Perlu. Ty obecně směřují k potlačování divokých konstrukcí a tvorbě slušnějšího kódu, což mi zcela vyhovuje. Jádro textu zůstalo beze změny, obsahuje však mnoho dílčích aktualizací a doplňků. Také v ukázkových programech jsem provedl četné drobné úpravy.

Výrazně jsem změnil kapitolu o objektově orientovaném programování. V původní verzi popisovala nativní prostředky Perlu, postupem času se ovšem pro tento účel prosadilo používání modulu *Moose* jako de facto standard. Proto je najdete i zde. Přidal jsem také novou kapitolu o funkcionálním programování, které se poslední dobou protlačuje do popředí zájmu.

Děkuji všem, jejichž zájem motivoval vznik tohoto vydání. A samozřejmě děkuji své rodině za veškerou podporu, kterou mi setrvale poskytuje.

**Pavel Satrapa**
*Liberec, říjen 2018*

# Obsah

# Typografie a on-line podpora

## Typografie

Kniha je protkána ukázkami programů a jmény různých programových konstrukcí. Abych je odlišil od běžného textu, používám následující konvence:

*Identifikátory* jsou sázeny kurzívou. Pro **klíčová slova** Perlu a také pro jeho **standardní funkce** používám tučné písmo. *»Obecné pojmy«*, které je třeba nahradit konkrétními hodnotami, sázím kurzívou se šipkami po stranách. Pro `ukázky výstupů` jsem zvolil neproporcionální písmo. `Uživatelské vstupy` (tedy vámi zadávané řetězce znaků) jsou navíc podloženy barvou. V některých situacích jsou vyznačeny mezery, aby bylo evidentní přesné složení daného řetězce. V takových případech mezeru symbolizuje znak ␣.

Ukázky programů jsou zpravidla sázeny tímto způsobem:

*ukazka.pl*

```
1   if ( $x >= 0 ) {
2       print "x je nezáporné\n";
3   else {
4       print "x je záporné\n";
5   }
```

V některých případech jsou řádky zdrojového textu číslovány. Čísla řádků *nepatří* do zdrojového textu. Jedná se pouze o podpůrný aparát, který mi umožňuje jednoznačně se odkazovat na jednotlivé řádky. Jméno napravo od prvního řádku udává název souboru, ve kterém je uložen zdrojový text programu. Archiv zdrojových textů najdete na WWW stránkách knihy – viz informace o on-line podpoře.

**Příklad:** Rozsáhlejší příklady jsou zahájeny slovem „Příklad“ a zakončeny grafickou značkou. ■

**Cvičení 0.1:** Analogicky jsou vysázena cvičení. Ta jsou navíc číslována, abyste v příloze 17 na straně 253 snadno našli odpovídající řešení. ■

Žárovka upozorňuje na místa, která si zaslouží zvýšenou pozornost. Zpravidla takto upozorňuji na informace, které pokládám za zvláště důležité.

Blesk signalizuje, že jde do tuhého. Zde se píše o něčem nebezpečném – očekávejte informace, které se týkají rizika vzniku chyb, příkazů, které se snadno vymknou kontrole, a podobně.

Pokud se v textu odkazuji na WWW stránku či distribuční adresu, je zvýrazněna symbolem článků řetězu. Vlastní odkaz je podtržen:

https://www.nic.cz/

## On-line podpora

Jak je mým zvykem, připravil jsem pro knihu doplňkové webové stránky. Kromě obecných informací zde najdete především zdrojové texty mnoha programů z textu. Pokud se rozhodnete s některým experimentovat, nemusíte jej pracně opisovat. Stačí si stáhnout archiv se zdrojovými texty, rozbalit jej a upravovat existující verzi programu. Jména příslušných souborů jsou uvedena vždy vpravo na začátku zdrojového textu.

Stránky najdete na adrese:

http://www.nti.tul.cz/~satrapa/knihy/perl3/

# Část I

# Oťukávání

První část knihy obsahuje základní seznámení s jazykem Perl. Úvodní kapitola vám umožní obhlédnout tvar a vlastnosti jeho programů a věnuje se také problematice spouštění existujících programů.

Následuje výuka základních prvků jazyka. Nejprve jednoduché (skalární) datové typy pro čísla a řetězce znaků, poté strukturované příkazy. Pak přijde na řadu kapitola o ladění programů, čili hledání a odstraňování chyb. Kromě standardních prostředků jazyka vás seznámí i s některými zajímavými programy na toto téma.

Závěrečná kapitola představí první strukturovaný typ Perlu – typ pole.

# 1 Ochutnejte Perl

Jsou programovací jazyky neúspěšné a jsou jazyky úspěšné. Některé prošumí, aniž by po sobě zanechaly výraznější stopu. Jiné se naopak dočkají širokého nasazení, hustě se vyskytují v literatuře a ještě častěji v běžné denní praxi. Perl patří mezi ty druhé. Těší se mimořádné popularitě mezi systémovými správci a svého času si vydobyl pozici de facto standardu na poli CGI programů, které jsou dnes sice již za zenitem, nicméně dosud pohánějí jednodušší webové aplikace.

Autorem jazyka je Larry Wall. Vytvořil jej původně pro svou vlastní potřebu, když jej neuspokojovaly dostupné nástroje pro práci s texty. Pak se rozhodl dát jej k dispozici veřejnosti a nestačil se divit, jaký zájem vyvolal. Kladná odezva okolí vedla k dalšímu vývoji jazyka a jeho postupnému obohacování.

V době vzniku prvního vydání této knihy byla aktuální verzí jazyka verze 5. Přinesla některá významná vylepšení (například objektově orientované programování) a vyřešila pár závažných nedostatků. Nejedná se o žádnou strhující novinku – Perl 5 je k dispozici již od roku 1994.

V roce 2000 začal vznikat Perl 6, jehož specifikace byla zveřejněna koncem roku 2015. Se zpětnou kompatibilitou se při vývoji této verze nikdy nepočítalo. S trochou nadsázky lze prohlásit, že vznikl úplně jiný jazyk, který se sice hlásí k odkazu Perlu, ale zavedl řadu konstrukcí, kterými divoký svět Perlu přibližuje zvyklostem ostatních jazyků.

V roce 2018 je Perl 6 spíše okrajovou záležitostí, existuje jen jedna aktivně vyvíjená implementace. Kniha je proto postavena na verzi 5 a verzí 6 se nadále nebudu zabývat.

## 1.1 Jaký je

Perl je zajímavý a kontroverzní jazyk. Rozhodně se nedá říci, že má všech pět „P“ (například rozhodně není poctivý). Na jedné straně oplývá úžasnými lákadly, na straně druhé pak číhají nechutné záludnosti. Z toho také pramení značně různorodé názory na tento programovací jazyk, které sahají od zbožného uctívání až po naprosté zavržení.

Snad nejtěžším kalibrem mezi zbraněmi Perlu jsou regulární výrazy. Představují neskutečně silný nástroj pro práci s textovými informacemi a jsou jedním z nosných pilířů operačního systému Unix. Zatím se však používaly jen ve speciálních jednoúčelových nástrojích, jako jsou *awk* či různé editory. Perl je spojuje s běžnými konstrukcemi programovacích jazyků a dává vám do rukou velice pružný a výkonný nástroj.

Druhou příjemností Perlu jsou asociativní pole, která řeší problém vyhledávání. Zapomeňte na stromy a podobné datové struktury. V Perlu vám hledání zajistí základní konstrukce jazyka. Třetí

milou vlastností je, že mnohé prvky Perlu jsou totožné s jazykem C. To podstatným způsobem usnadňuje přechod na tento jazyk či jeho paralelní používání s C.

Zmiňované konstrukce vám zásadně zjednoduší život. Řadu i dosti složitých problémů dokážete vyřešit pomocí základních prvků jazyka.

A teď něco o temné straně Síly. Z pohledu teorie programovacích jazyků Perl představuje jednu z největších zpotvořenin, které kdy spatřily světlo světa. Proměnné se nedeklarují. Nemůžete si definovat vlastní datové typy. Syntaxe je velmi proměnlivá a celou řadu konstrukcí lze zapisovat několika různými způsoby. Chování leckterých prvků závisí na kontextu, ve kterém je použijete. A tak dále a tak podobně.

Ve světle těchto nedostatků bych si vám dovolil nedoporučit Perl jako jazyk, na kterém se budete učit programovat. Říká se, že váš první jazyk ovlivní celou vaši programátorskou praxi. Osvojíte-li si dobré návyky, již vás neopustí. Totéž však platí o zlozvycích. Perl rozhodně není pedagogický jazyk, který by vás jaksi sám od sebe vedl k systematickým řešením či používání čistých a srozumitelných konstrukcí. Pokud jste začínajícím programátorem či programátorkou, sáhněte ve svých začátcích raději po „slušnějším" jazyku, jako je třeba Pascal. Perlu se můžete věnovat později, až budete mít za sebou jistou praxi a nedáte se tak snadno zkazit.

Tak proč v něm tolik lidí píše? Perl je mimo jakoukoli pochybnost mimořádně praktický. Vznikl pod heslem „Nechť lze jednoduché věci dělat jednoduše, aniž bychom si znemožnili věci složité." Myslím, že se je podařilo naplnit.

## 1.2 Malinká demonstrace síly

Standardním příkladem, který čtenáři umožní přičichnout k programovacímu jazyku a udělat si obrázek o podobě programů v něm, je pozdrav. Program, který vypíše „Nazdar lidi!", by v Perlu vypadal takto:

```
print "Nazdar lidi!";
```

Vidíte tu stručnost? Žádné deklarace. Žádné úvodní či ukončovací konstrukce. Žádné rituální tanečky, jde se rovnou k jádru věci. Obávám se, že jako milenec by Perl velkou kariéru neudělal. Ovšem na pozici programovacího jazyka má za sebou famózní úspěchy.

Podívejme se schválně na něco náročnějšího, aneb jak jednoduše dělat složité věci. Krátký program:

*kalkul.pl*

```
while ( $radek = <> ) {
        print "Výsledek: ", eval($radek), "\n";
}
```

realizuje pěkně silnou kalkulačku. Má celou řadu aritmetických funkcí, mezivýsledky si můžete ukládat do pomocných proměnných a závorky můžete vnořovat, co hrdlo ráčí. Dokonce má i vstup ze souboru, takže si můžete požadované výpočty předem připravit a pak je nechat zpracovat naráz. Jsem přesvědčen, že standardní kalkulačka vašeho operačního systému dovede mnohem méně. Podívejte se na ukázku praktického použití:

```
3*12                          nejprve něco jednoduchého
Výsledek: 36
sqrt(2)                       vestavěná funkce
Výsledek: 1.4142135623731
$a = 10**2                    uložím do proměnné…
Výsledek: 100
1 / 3 * $a                    … a použiji
Výsledek: 33.3333333333333
Ctrl-D                        ukončím vstup
```

Uživatelské vstupy jsou vyznačeny šedým podkladem. Jediným trikem je zastavení programu – musíte ukončit vstupní soubor. V prostředí Unixu to znamená stisknout kombinaci **Ctrl-D**, v operačních systémech firmy Microsoft k tomu účelu slouží **Ctrl-Z** **Enter**.

Jak to funguje? Základním kamenem je volání funkce **eval**, která zpracuje obsah řetězce, jako by to byla část programu. De facto jsem postavil zjednodušený interpret Perlu a skutečnost, že také dovede počítat, je jen jedním z jeho projevů. Můžete mu vnutit podmínky, cykly, podprogramy…

Celý program je tvořen cyklem **while**. Dokud neskončí vstup, načte vždy do proměnné *$radek* jeden řádek, zavolá **eval** a výsledek vypíše prostřednictvím **print**. Toť vše.

## 1.3 Jak spustit program

Perl je interpretovaný jazyk. To znamená, že program se nijak nepřekládá a vykonává se přímo ze zdrojového kódu. Takový přístup má výhodu ve své jednoduchosti a snadné modifikovatelnosti programů – pokud vám to běží, můžete to i upravovat. Důležitou nevýhodou však je, že programy nejsou soběstačné. Abyste je mohli spustit, musíte mít instalován interpret Perlu.

A teď jednu dobrou zprávu: interpret je volně šiřitelný. Zatoužíte-li po něm, stačí si jej obstarat třeba v Internetu a nainstalovat. Díky rostoucí popularitě Perlu má stále více a více operačních systémů interpret předinstalován. Zda je přítomen si ověříte velmi snadno – zadejte na příkazovém řádku příkaz:

```
perl -v
```

Výsledkem by mělo být cosi jako:

```
This is perl 5, version 26, subversion 1 (v5.26.1) ...
```

Pokud zjistíte, že interpret Perlu nemáte nainstalován nebo oplýváte pouze starší verzí (určitě byste měli mít alespoň verzi 5.20), přečtěte si přílohu 18 na straně 275. V ní se dozvíte, kde si můžete obstarat aktuální verzi Perlu a jak si ji nainstalovat. V dalším textu budu předpokládat, že interpret máte k dispozici.

Program zpravidla máte uložen v souboru. Řekněme, že se bude jmenovat *pokus.pl* (přípona *.pl* je obvyklou konvencí pro programy v Perlu). Nejjednodušším způsobem pro jeho spuštění je zavolat Perl ručně a jméno programu mu předat jako parametr:

```
perl pokus.pl
```

Nejprimitivnější příkazy můžete dokonce psát přímo na příkazový řádek, což ale není příliš praktické.

Jedním ze závažných problémů Perlu je, že programátora nehlídá. Dovolí mu udělat téměř cokoli a iniciativně si vše přizpůsobí tak, aby vaše příkazy něco provedly. Ono něco však může být na hony vzdáleno vašim skutečným tužbám.

Naštěstí však sám nabízí ochranu proti své vlastní benevolenci. Je jí volba `-w`. Pokud ji použijete, bude sice program fungovat úplně stejně, ale obdaří vás varováním za každou podezřelou konstrukci (např. použití proměnné, které nebyla přiřazena hodnota či jediný výskyt proměnné v celém programu, což jsou žhaví kandidáti na překlep). Vzhledem k *silné* užitečnosti této konstrukce si prosím navykněte spouštět programy zásadně s volbou `-w`:

```
perl -w pokus.pl
```

Ještě kratší vodítko si lze nasadit použitím **use** *strict*. Více se o něm dočtete v části 10.4 na straně 171. Používání obou hlídačů rozhodně doporučuji, někdy vás sice budou pěkně štvát, ale odhalí řadu chyb a donutí vás psát slušnější programy.

Příkazový řádek se utěšeně prodlužuje a spuštění programu začíná až příliš připomínat práci… Naštěstí v operačních systémech typu Unix existuje příjemné usnadnění. Říká se mu konvence #! a spočívá v tom, že první řádek souboru se zdrojovým textem sestavíte zcela speciálně. Vypadá takto:

`#!` *název_interpretu*

Soubor pak označíte jako spustitelný a můžete jej rovnou startovat. Operační systém do něj nahlédne, najde tento první řádek, spustí v něm uvedený interpret a jako parametr mu předá název souboru, který jste původně spustili. Dokonce můžete interpretu zadat i jeden (ale ne více) parametr. V našem konkrétním případě by to znamenalo, že soubor se zdrojovým textem programu by obsahoval:

```
#!/usr/bin/perl -w
print "Nazdar lidi!";
```

Uložíte jej pod názvem *pokus.pl* a učiníte spustitelným pomocí:

```
chmod a+x pokus.pl
```

Od tohoto okamžiku jej můžete spustit stejně, jako kterýkoli jiný program pomocí prostého:

```
pokus.pl
```

Případně `./pokus.pl`, pokud máte interpret příkazů konfigurovaný tak, aby nehledal spouštěné příkazy v aktuálním adresáři (vřele doporučuji). Unix se do něj pustí a podle instrukcí z prvního řádku si jej interně převede na volání:

```
/usr/bin/perl -w pokus.pl
```

Toto usnadnění je velmi praktické a používají je snad všichni programátoři v prostředí Unixu. Mechanismus je zcela obecný, takže jej můžete použít pro libovolné interpretované programy, ať už je interpretem Perl, sed, awk, interpret příkazů či kdokoli jiný.

⚡ Za výše popsaným usnadňovadlem se skrývá jedno nemilé úskalí. Interpret v prvním řádku totiž musí být zadán svou *úplnou cestou* (raději absolutní, abyste mohli program bez následků stěhovat). Pokud je špatná, program nefunguje. Jestliže při pokusu o spuštění obdržíte hlášení „Command not found“, přestože se příkazem `ls` přesvědčíte, že program skutečně existuje a nepřeklepli jste se při jeho spuštění, bude problém v chybné cestě k interpretu.

Tato chyba nejčastěji vzniká dvěma způsoby: prostým překlepem (což většinou snadno zjistíte) nebo přenosem programu na jiný počítač. Existují totiž dvě obvyklá umístění Perlu: buď */usr/bin/perl* nebo */usr/local/bin/perl*. Kde se nachází ten váš snadno zjistíte příkazem:

```
which perl
```

Nejjednodušším řešením je vyrobit na druhém místě symbolický odkaz a zajistit tak, že ve vašem Unixu budou fungovat *obě* cesty k interpretu.

V prostředí MS Windows se interpret při své instalaci zpravidla chopí přípony *.pl*, kterou mívají programy v Perlu. Můžete je pak spouštět prostým poklepáním, musíte se však držet jedné přípony. Pokud ji dotyčný soubor nemá, lze sáhnout po příkazovém řádku:

```
perl -w »soubor s programem«
```

## 1.4 Jak rychlý je Perl?

Obecně se traduje, že interpretované jazyky jsou pomalé. Perl je interpretovaný jazyk. Z toho snadno dedukujeme, milý Watsone, že Perl bude pomalý. Také to o něm řada lidí říká.

A co na to praxe? V knize [7] na straně 479 najdete srovnání výkonnosti několika programů při hledání řetězce v hromadě krátkých textových souborů. Vítězem byl *egrep* (spotřeboval 3,37 s procesorového času), druhý o prsa Perl (3,63 s). Zbytek pelotonu (včetně programů *grep* a *fgrep*) se pohyboval kolem 5 s. Přitom *grep* & spol. jsou programy specializované na vyhledávání řetězců, pochopitelně psané v jazyce C, u nějž je rychlost součástí image.

S výkonem Perlu to zjevně nebude tak špatné. Celkově lze prohlásit, že výrok „interpretované jazyky jsou pomalé“ patří spíše mezi programátorské báje a pověsti, než do běžného života. Kdysi dávno, kdy interprety analyzovaly program příkaz po příkazu vždy těsně před provedením a při opakování některého příkazu jej zpracovávaly znovu a znovu, to skutečně byla pravda.

Moderní *interprety* (Perl nevyjímaje) zpravidla mají fázi předkompilace, kdy vstupní textový tvar programu převedou do jakéhosi binárního mezikódu a ten pak provádějí. Tento přístup má dvě podstatné výhody. V úvodu se zpracuje celý program a budou tudíž odhaleny syntaktické chyby i v těch částech, k jejichž vykonání později vůbec nedojde. Druhou předností je, že program pak běží mnohem rychleji.

Současné *překladače* často převádějí program na sadu volání různých knihovních podprogramů. Jinými slovy jejich výsledek se docela podobá binárnímu mezikódu interpretů a v rychlosti běžícího programu proto nejsou zásadní rozdíly mezi interpretovanými a překládanými jazyky.

Jedinou cenou, kterou interpretované jazyky musí platit, je úvodní předkompilace. Ta probíhá před každým spuštěním. Její podíl na celkové době běhu programu však bývá zanedbatelný.

Výkonnostní penalizace programů v Perlu je paradoxně nejhorší na poli, kde dosáhl největších úspěchů – u CGI programů. Zpravidla bývají dost jednoduché a je třeba, aby byly provedeny co nejrychleji. Start interpretu a úvodní předkompilace mohou představovat výrazné zpomalení. Ovšem zde přicházejí ke slovu jiné optimalizační mechanismy na úrovni operačního systému (paralelně

běžící interprety využívají společnou paměť, diskové cache a podobně) či programu realizujícího WWW server (FastCGI, zabudovaný interpret, jako je třeba *mod_perl* pro *Apache*, a podobně).

Sečteno a podtrženo: z hlediska výkonu si Perl stojí velmi dobře a troufám si tvrdit, že tady vás bota tlačit nebude. Navíc se v něm řada věcí programuje velmi snadno a rychle. Pro běžnou praxi zpravidla bývá mnohem významnější, zda program vyvíjíte dva dny nebo týden, než zda jeho běh trvá 5 nebo 15 sekund.

Historicky býval součástí distribuce Perlu i překladač *perlcc*. Fungoval tak, že převedl program do výše zmiňovaného binárního mezikódu a přibalil k němu vše, co je potřeba k jeho provedení. Neměl však příliš dobrou pověst a ve verzi 5.10 byl opuštěn. Jeho roli převzal *Perl Archive Toolkit (PAR)* a jeho *PAR Packager (pp)*, který umožňuje vytvářet spustitelné balíčky perlivých programů s příslušenstvím:

https://metacpan.org/pod/PAR

Reálně se ale perlivé programy příliš často nepřekládají.

## 1.5 Dokumentace a další informace

Jednu věc Perlu rozhodně nelze vyčítat: chabou dokumentaci. Součástí základní distribuce je velmi bohatá a kvalitní sada manuálových stránek. Dohromady čítá přes 40 MB HTML kódu a ve formátech HTML a PDF je ke stažení na adrese:

http://perldoc.perl.org/

Základní vstupní stránkou je:

```
man perl
```

Obsahuje nejzákladnější obecné informace o Perlu, ale především seznam a obsah dalších manuálových stránek, které jsou členěny tematicky. Pokrývají velmi široké spektrum od textů uvádějících do určité problematiky až po stránky charakteru referenční příručky. Jejich kvalita je obecně velmi vysoká. Za pozornost určitě stojí `perlfaq`, obsahující často kladené otázky a odpovědi na ně. Vzhledem ke svému rozsahu je také tato stránka rozdělena do několika dalších.

V implementaci ActivePerl pro MS Windows najdete dokumentaci v podobě sady HTML stránek. Obsahuje pochopitelně i informace specifické pro tento konkrétní interpret.

Ve světě Perlu se hojně používají různé moduly. Když si některý nainstalujete, bývá jeho součástí i manuálová stránka s podrobnějším popisem. Zobrazíte ji pomocí:

```
man »jméno modulu«
```

Perl je dobře zdokumentován také v papírové podobě. Existuje o něm řada knih a dokonce i v češtině vyšlo pár titulů. Přehled těch, které považuji za zajímavé, najdete společně se stručným komentářem v seznamu literatury na konci knihy.

V Internetu je základní adresou pro všechny Perlisty:

🔗 https://www.perl.org/

Najdete tu všechny důležité odkazy: implementace pro různé platformy, archiv modulů a doplňků, dokumenty a návody, diskusní fóra a další.

# 2 Základní kameny, mísy až trámy

V této kapitole se podívám na nejzákladnější konstrukce a příkazy Perlu. Jsou to takové ty Popelky, na nichž není nic moc originálního, najdete je ve většině jazyků, ale jsou pořád potřeba a oddřou valnou většinu práce.

## 2.1 Proměnné

Základním kamenem všech programů bývá obyčejná proměnná. V Perlu do ní můžete uložit číslo, řetězec znaků nebo odkaz na jinou proměnnou či datovou strukturu. Souhrnně se tyto tři typy dat nazývají *skaláry*. Proměnná nesoucí skalární hodnotu se v programu zapisuje ve tvaru *$»jméno_proměnné«*. Například *$pocet* nebo *$x2*.

Jména proměnných, čili *identifikátory*, mohou být dost divoká. Mnohá ze speciálních jmen, jako například proměnné *$_* či *$@* však mají přidělen speciální význam. Proto vřele doporučuji držet se při zemi a při sestavování identifikátorů ctít obvyklé pravidlo: použít libovolně dlouhou posloupnost písmen anglické abecedy, číslic a podtržítek, která nezačíná číslicí.

**Příklad:** Několik vhodně pojmenovaných skalárních proměnných:

```
$radek              $x1
$novy_radek         $pocatek1x
$NovyRadek          $auto_1
```

A teď pro změnu pár chyb:

```
$pozice–x           $otec&syn           $auto=1
```

Chyby spočívají v použití speciálních znaků (–, & a =), jejichž přítomnost v identifikátoru není povolena. Bohužel se mi nepodařilo najít přesnou definici, které znaky Perl připouští ve jménech proměnných a které ne. Musím tedy vystačit s obecným doporučením: vystačíte-li s písmeny anglické abecedy, podtržítky a číslicemi, nebudete mít problémy. ■

Prostřednictvím identifikátorů se neoznačují jen proměnné, ale řada dalších prvků jazyka. Například podprogramy, ovladače souborů či moduly. Aby je navzájem výrazněji odlišili, zavedli si programátoři jisté konvence, které je záhodno dodržovat. Jejich souhrn uvádí tabulka 2.1. Pokud se jméno skládá z několika slov (např. „nový řádek“), oddělte je navzájem podtržítky (*$novy_radek*).

Perl rozlišuje malá písmena od velkých. Proto *$pokus*, *$Pokus*, *$POKUS* a *$pOkus* jsou čtyři různé proměnné. Budete-li se držet doporučení pojmenovávat proměnné malými písmeny, nehrozí vám

| | |
|---|---|
| *malými_písmeny* | lokální proměnné, podprogramy |
| *První_Velká* | globální proměnné, moduly |
| *VELKÝMI_PÍSMENY* | ovladače souborů, konstanty, návěští |
| *BezPodtržítek* | moduly |

Tabulka 2.1: Konvence pro identifikátory

z tohoto směru žádné potíže. Rozhodně si nevytvářejte několik proměnných, jejichž názvy by se lišily jen velikostí písmen. To je zaručený recept na obtížně hledatelné chyby.

Je velmi důležité zvyknout si používat mnemotechnické identifikátory. To znamená, že z názvu proměnné by mělo být patrné, k čemu je určena. Počítáte-li počet, součet a průměr známek, měly by se dotyčné proměnné nazývat *$pocet*, *$soucet* a *$prumer*, nikoli *$a*, *$b*, *$c* či dokonce *$prepona*, *$odvesna*, *$kurnik*. Mnemotechnické identifikátory *velmi výrazně* zvyšují srozumitelnost programu. Ta je jednou z nejdůležitějších vlastností. Snadno se může stát, že někdo[1] bude nucen se ve vašem díle vyznat a provést v něm jisté úpravy. Shledá-li váš program srozumitelným, bude vám líbat ruce, nohy.

Proměnné se v Perlu nemusí nijak deklarovat. Vznikají automaticky při svém prvním použití. Tato vlastnost je pro programátora velmi příjemná a velmi nebezpečná. Stačí se překlepnout v identifikátoru a Perl místo použití stávající proměnné hbitě založí novou. Následky mohou být katastrofální. Proto je důležité vždy a všude spouštět interpret s volbou `-w`, která na takové situace upozorní.

**Příklad:** Na tento chybný program (na druhém řádku chybí „k“):

```
$preklep=1;
print $prelep*2;
```

reaguje `perl -w` následovně:

```
Name "main::prelep" used only once: possible typo at perklep.pl line 2.
Name "main::preklep" used only once: possible typo at perklep.pl line 1.
Use of uninitialized value at perklep.pl line 2.
0
```

1: Pozor, můžete to být vy sami! Pokud svůj program na rok odložíte, garantuji vám, že se v něm budete orientovat stejně špatně, jako kdokoli cizí.

Závěrečná nula je vlastním výstupem programu a kdybyste vynechali volbu `-w`, byla by také výstupem jediným. První dva řádky upozorňují, že identifikátory „prelep“ a „preklep“ jsou použity jen jednou a že to zavání překlepem. Varování na třetím řádku je o tom, že ve druhém programovém řádku byla použita proměnná, které dosud nebyla přiřazena žádná hodnota.

Jinými slovy dostali jste docela cenné informace, s jejichž pomocí snadno odhalíte chybu[2]. ■

Deklarace proměnné není sice povinná, nicméně je považována za slušnost a vřele se doporučuje ji provádět. Není to jen pro obecné blaho, ale jak jsem naznačil, chráníte tím i sami sebe před vlastními chybami. Nejčastějším způsobem je použití klíčového slova **my** následovaného jménem proměnné. Například:

**my** *$prumer*;

deklaruje proměnnou *$prumer*. Chcete-li deklarovat několik proměnných, buď zopakujte tuto konstrukci několikrát, nebo poskytněte jednomu **my** jejich seznam uzavřený do kulatých závorek a vzájemně oddělovaný čárkami:

**my** ( *$prumer*, *$median*, *$maximum* );

Účinek **my** je ve skutečnosti silnější, protože deklaruje proměnnou jako lokální v dané části programu. Na to je ale ještě moc brzy, k tématu se vrátím podrobněji v části 8.2 na straně 112. Zatím si jednoduše zvykněte používané proměnné deklarovat – buď společně na začátku programu, nebo při prvním použití.

## 2.2 Přiřazování hodnot

Proměnná vlastně není nic jiného, než pojmenované místo pro uložení hodnoty. Svou hodnotu nejčastěji získá prostřenictvím přiřazovacího příkazu, který má v Perlu tvar:

*»proměnná«* = *»výraz«*

Levá strana určuje cílovou proměnnou. Perl musí nejprve vyhodnotit *»výraz«* na pravé straně. Výsledek pak uloží do příslušné *»proměnné«*.

2: Kouzlo nechtěného: všimli jste si v chybových hlášeních, jak se jmenuje soubor s tímto příkladem? To skutečně nebyl záměr.

**Příklad:** Přiřazovací příkazy:

```
my $x = 20;
$x = $y + $z;
$x = $b**2 – 4*$a*$c;
```

uloží do proměnné *$x* postupně hodnoty 20, součet proměnných *$y* a *$z* a konečně diskriminant kvadratické rovnice (za předpokladu, že v proměnných *$a*, *$b* a *$c* jsou příslušné koeficienty).

Jak vidíte, přiřazení lze spojit s deklarací. První řádek v příkladu vytvoří proměnnou *$x* a ihned jí přiřadí hodnotu 20. ■

Uložení nové hodnoty pochopitelně znamená, že dosavadní obsah proměnné je nenávratně ztracen a nelze jej nijak obnovit. Pokud bych skutečně použil trojici příkazů z předchozího příkladu tak, jak jsou zapsány, mohl bych první dva klidně vymazat. Na výsledném efektu by se nic nezměnilo.

Proměnná, do níž je přiřazováno, se může vyskytnout i ve výrazu na pravé straně. Vzhledem ke způsobu zpracování přiřazovacího příkazu (nejprve vyhodnotit výraz, pak uložit hodnotu), to znamená, že ve výrazu bude použita hodnota, kterou měla před zahájením provádění příkazu. Teprve v samotném závěru se uloží hodnota nová. Díky tomu příkaz:

```
$x = $x + 1
```

způsobí zvýšení hodnoty *$x* o jedničku: při vyhodnocování se vezme stávající hodnota *$x*, přičte se jedna a výsledek se uloží opět do *$x*. Jelikož jsou podobné případy v programátorské praxi dosti časté, okoukal Perl z jazyka C zkratku. Stejného výsledku dosáhnete i zápisem:

```
$x += 1
```

Tento tvar lze použít pro valnou většinu operací, o nichž se dočtete v nadcházejících kapitolách. Obecně platí, že:

```
»proměnná« »operace«= »výraz«
```

je zkratkou (a tudíž zcela ekvivalentní):

```
»proměnná« = »proměnná« »operace« »výraz«
```

Všimněte si, že ve zkrácené verzi mezi symbolem operace a rovnítkem *není* mezera.

**Cvičení 2.1:** Napište dlouhou a zkrácenou verzi přiřazovacích příkazů, které zdvojnásobí hodnotu proměnné *$zisk* (násobení se zapisuje hvězdičkou) a které od aktuální hodnoty proměnné *$sklad* odečtou hodnotu *$prodano*. ■

Když proměnná vznikne, je její hodnota nedefinována. Perl tento speciální stav označuje jako hodnotu **undef**. Jak předvedl příklad s překlepem, při použití si ji iniciativně převede na nulu nebo prázdný řetězec. Nicméně použití proměnné, které dosud nebyla přidělena hodnota, signalizuje logické klopýtnutí. Snažte se mu vyhýbat a každé proměnné poctivě přidělit úvodní hodnotu, byť by byla nulová. Kdykoli později ji můžete vrátit do nedefinovaného stavu pomocí:

```
$promenna = undef;
```

Jistou specialitou (opět inspirovanou jazykem C) je skutečnost, že celý přiřazovací příkaz je zároveň výrazem. Výsledkem jeho vyhodnocení je hodnota, která byla uložena do proměnné. Díky tomu konstrukce:

```
$a = $b = $c = 8;
```

přiřadí postupně do proměnných *$c*, *$b* a *$a* hodnotu 8. Kdybych chtěl zvýraznit uspořádání jednotlivých operací v ní pomocí závorek, vypadal by příkaz takto:

```
$a = ( $b = ( $c = 8 ) );
```

Přiřazovací příkaz v roli výrazu bývá nejčastěji používán v podmínkách cyklů či podmíněných příkazů, ke kterým se dostanu v příští kapitole. Teď se ještě musím porozhlédnout, co se vlastně dá do proměnných přiřazovat.

## 2.3 Čísla

Jedním ze základních datových typů je typ číselný. Na rozdíl od většiny programovacích jazyků je Perl při práci s čísly velmi demokratický. Nerozlišuje celočíselné hodnoty od reálných. Zde jsou si všechna čísla rovna.

Zápis číselných konstant vychází z obvyklých základů. Nejjednodušší je zápis celého čísla, který tvoří prostá skupina číslic, případně předcházená znaménkem. Zapisujete-li číslo se zlomkovou částí, oddělte ji desetinnou tečkou, nikoli čárkou[3]. Přípustný je i zápis v semilogaritmickém tvaru, který bývá občas nazýván vědeckou notací (ti vědci!). Předveďme si ukázku:

3: Konvence pro zápis čísel se v jednotlivých zemích bohužel značně liší. Snad všechny programovací jazyky vycházejí ze zvyklostí USA, kde se desetinná část čísla odděluje tečkou.

`1.23e4` znamená $1{,}23 \cdot 10^4$ tedy 12 300

Zejména při programování na úrovni blízké operačnímu systému se občas hodí, aby jedna číslice odpovídala určitému počtu bitů paměti. Nejčastěji používanými číselnými soustavami, které vyhovují tomuto požadavku, jsou osmičková (jedna číslice je uložena ve třech bitech) a šestnáctková (čtyři bity na číslici). V Perlu můžete zapisovat celočíselné konstanty v těchto soustavách tak, že jim předřadíte „0b“ (binární soustava), „0“ (osmičková) nebo „0x“ (šestnáctková).

**Příklad:** Všechny následující zápisy představují stejnou hodnotu – číslo 668:

```
668    +668    668.00    6.68e2    01234    0x29c
```

■

Kouzlo čísel spočívá především v tom, že s nimi lze počítat. Sortiment aritmetických schopností Perlu nijak významně nevybočuje z běžné nabídky programovacích jazyků. Dostupné aritmetické operace a funkce shrnuje tabulka 2.2.

Především v oblasti funkcí je patrná snaha o minimalizaci. Například zde najdete jen nejzákladnější goniometrické funkce. Ostatní si musíte doprogramovat[4]. Jak bývá zvykem, úhly se zadávají v radiánech. Převod mezi nimi a pro nás obvyklejšími stupni lze realizovat těmito přiřazovacími příkazy:

```
$radiany = $stupne / 180 * 3.1415926
$stupne = $radiany / 3.1415926 * 180
```

**Cvičení 2.2:** Řekněme, že proměnné *$a* a *$b* obsahují délky odvěsen pravoúhlého trojúhelníka. Napište přiřazovací příkaz, který do proměnné *$c* spočítá délku přepony. Pythagorovu větu vám jistě nemusím představovat. Dokážete vytvořit více variant příkazu? ■

**Cvičení 2.3:** Napište přiřazovací příkaz, který hodnotu proměnné *$x* zaokrouhlí na nejbližší nižší číslo dělitelné deseti. ■

Za povšimnutí stojí ještě operace ++ a −−, které představují další stupeň zkrácení běžných příkazů. Realizují jednu z častých programátorských konstrukcí – zvětšení resp. zmenšení hodnoty proměnné o jedničku. Chcete-li tedy k proměnné *$x* přičíst jedničku, máte na výběr následující možnosti:

4: Nebo použít modul POSIX, ale na to je zatím příliš brzy.

| Základní operace | |
|---|---|
| + | sčítání |
| – | odčítání |
| * | násobení |
| / | dělení |
| **Něco navíc** | |
| ** | mocnina (3**2 znamená $3^2$) |
| % | zbytek po dělení čili modulo (7%3 vydá 1) |
| ++ | zvětšení o 1 |
| –– | zmenšení o 1 |
| **Bitové operace** | |
| & | bitové „and“ (6&3 vydá 2) |
| \| | bitové „or“ (6\|3 vydá 7) |
| << | bitový posun doleva (1<<4 vydá 16) |
| >> | bitový posun doprava (8>>1 vydá 4) |
| **Goniometrické funkce** | |
| **sin**($x$) | $\sin x$ |
| **cos**($x$) | $\cos x$ |
| **atan2**($y$,$x$) | arctg $\frac{y}{x}$ |
| **Ostatní aritmetické funkce** | |
| **abs**($x$) | $\lvert x \rvert$ |
| **sqrt**($x$) | $\sqrt{x}$ |
| **log**($x$) | $\ln x$ (přirozený logaritmus) |
| **exp**($x$) | $e^x$ |
| **Konverzní funkce** | |
| **int**($x$) | celá část $x$ (**int**(5.19) vydá 5) |
| **hex**($s$) | převede šestnáctkový zápis řetězce $s$ na číslo |
| **oct**($s$) | převede osmičkový (a další) zápis řetězce $s$ na číslo |
| **Náhodná čísla** | |
| **rand**($n$) | vydá náhodné číslo z intervalu $\langle 0 \ldots n)$ |
| **srand** | inicializuje generátor náhodných čísel |

Tabulka 2.2: Číselné operace a funkce

```
$x = $x + 1
$x += 1
$x++
```

Varianta *$x++* je nejen nejkratší, ale také nejsnáze použitelná ve výrazech. Máte dokonce na vybranou, zda použijete *$x++* nebo *++$x*. V prvním případě takovýto výraz vydá aktuální hodnotu *$x* a poté zvětší obsah proměnné o jedničku, v případě druhém nejprve dojde ke zvětšení *$x* a výsledkem bude až tato nová hodnota. Pokud potřebujete do proměnné *$y* uložit polovinu hodnoty *$x* a následně *$x* zvětšit o jedničku, zvládne to jediný přiřazovací příkaz:

```
$y = $x++ / 2;
```

Perl umožňuje celou řadu podobných zahuštění, kdy určitá konstrukce kromě své základní činnosti jako bonus zdarma provede ještě něco navíc. Přiznám se, že je příliš nemiluji. Jistě se zde projevuje, že jsem byl odkojen jazykem Pascal, který podobné skopičiny nepovoluje. Jsem tudíž zvyklý dělat věci pěkně postupně.

Zcela objektivně je třeba upozornit na nemalé nepříjemnosti, které s sebou tento kompaktní způsob vyjadřování přináší. Výrazně totiž komplikuje srozumitelnost programu, odvádí myšlenky postranními cestičkami a snadno se může stát, že vám přeroste přes hlavu. Silně to připomíná situaci, kdy pravou rukou mícháte kaši na sporáku, současně levou zatloukáte hřebík a šlapacím ježkem nafukujete matraci. Jsou lidé, kteří to bravurně zvládnou. Ovšem riziko, že skončíte s hřebíkem v matraci, hrncem kaše na hlavě a budete marně přemýšlet, proč jste polykali to kladivo, je podle mne neúnosně vysoké. Proto vám vřele doporučuji se podobným kondenzátům raději vyhýbat.

Já osobně se k nim uchyluji jen málokdy. Nejčastěji v podmínce strukturovaného příkazu (viz následující kapitola), kdy se provede příkaz a výsledná hodnota je zároveň použita pro vyhodnocení podmínky. Například načtete do proměnné další řádek ze vstupu a podle výstupní hodnoty tohoto příkazu se pozná, zda vstup již neskončil.

Dalším příkladem zkracování, kterému se usilovně vyhýbám, je implicitní proměnná *$_*. Celé řadě konstrukcí totiž můžete vynechat parametr, cíl přiřazení a podobně. Pokud to uděláte, použije se místo něj právě zmíněná implicitní proměnná. Její explicitní zápis *$_* vlastně uvidíte jen vzácně, většinou si ji automaticky doplňuje interpret do míst, kde jste vynechali některý z prvků.

Nemám ji rád. Nejsem ochoten hlídat, kdy ji kdo čím změnil či nezměnil, a pamatovat si, jak se přesně chová příkaz, když mu vykastrujete polovinu proměnných. Proto jsem implicitní proměnnou exkomunikoval ze své hlavy a nadále se budu tvářit, jako by vůbec neexistovala.

## 2.4 Řetězce znaků

Dvojici základních jednoduchých typů doplňuje řetězec znaků. Na něm je postavena práce s textovými informacemi.

Chcete-li do zdrojového textu programu zapsat řetězcovou konstantu, uzavřete ji do uvozovek. Přesněji řečeno máte na výběr dva druhy: uvozovky dvojité ("…") a uvozovky jednoduché čili apostrofy ('…'). Jednodušší jsou apostrofy. Řetězec znaků, který jejich nasazením vznikne, bude obsahovat přesně ty znaky, které jste napsali. Jedinými výjimkami jsou \' a \\, umožňující vložit do řetězce apostrof a zpětné lomítko – viz níže.

Daleko zajímavější jsou konstanty uzavřené do dvojitých uvozovek. Ty jsou totiž před použitím zpracovány. Pokud v nich uvedete proměnnou, bude nahrazena svou aktuální hodnotou. Kromě toho můžete prostřednictvím speciálních skupin znaků \»*cosi*« vkládat speciální znaky nebo dokonce měnit část řetězce. Sortiment dostupných konstrukcí shrnuje tabulka 2.3.

Má-li v sobě řetězec obsahovat stejný typ uvozovek, kterými je obklopen, musíte jim předsadit zpětné lomítko. Kdybyste zapomněli, budou uvozovky pochopeny jako ukončení řetězce a následující znaky pravděpodobně způsobí chybové hlášení.

**Příklad:** Po provedení příkazů:

```
$jmeno = "Pepa";
$raz = 'To je přece $jmeno!';
$dva = "To je přece $jmeno!";
$tri = "To je přece \U$jmeno\E!";
```

bude proměnná *$raz* obsahovat řetězec „To je přece $jmeno!“, proměnná *$dva* „To je přece Pepa!“ a proměnná *$tri* „To je přece PEPA!“. Modifikátory ze spodní části tabulky 2.3 se používají především na úpravu hodnot vložených z proměnných, jak jsem právě předvedl. ■

Přidávání zpětných lomítek může být docela otravné. Proto Perl nabízí i funkce **q**{…} a **qq**{…}, které se chovají jako jednoduché a dvojité uvozovky. Řetězec je v nich vymezen složenými závorkami, proto v něm můžete používat uvozovky bez omezení. Následující dvě konstanty jsou ekvivalentní:

```
"$jmeno křičel \"Hej!\" a \"Počkejte!\""
qq{$jmeno křičel "Hej!" a "Počkejte!"}
```

Řetězec může zabírat několik řádků, ale zápis s uvozovkami není v takovém případě ideální. Pro víceřádkové konstanty je vhodnější používat konstrukci nazývanou v angličtině *here documents*.

| Znaky | |
|---|---|
| \n | nový řádek |
| \r | návrat vozíku |
| \t | tabulátor |
| \f | nová stránka |
| \b | couvnutí (znak backspace) |
| \a | varovné pípnutí |
| \e | znak Esc |
| \033 | znak s daným kódem v osmičkové soustavě (zde Esc) |
| \0x7f | znak s daným kódem v šestnáctkové soustavě (zde Del) |
| \cX | Ctrl-X |
| \" | uvozovky (") |
| \\ | zpětné lomítko (\) |
| **Modifikátory** | |
| \u | převede následující písmeno na velké |
| \l | převede následující písmeno na malé |
| \U | převede následující skupinu písmen na velká |
| \L | převede následující skupinu písmen na malá |
| \Q | nealfanumerickým znakům v následující skupině přidá \ |
| \E | ukončuje skupinu pro \U, \L či \Q |

Tabulka 2.3: Speciální znaky ve dvojitých uvozovkách

Začíná dvojicí << následovanou ukončujícím řetězcem. Můžete jej uzavřít do jednoduchých nebo dvojitých (ty se použijí, pokud uvozovky vynecháte) uvozovek a podle toho pak bude řetězcová konstanta interpretována. Patří do ní všechny následující řádky až po řádek obsahující samotný ukončující řetězec. Vypadá to takhle:

```
$zahajeni = <<"KONEC_ZAHAJENI";
Vážený $jmeno,
gratulujeme Vám k Vaší první víceřádkové
textové konstantě v Perlu.
KONEC_ZAHAJENI
```

Operátory a funkce dostupné pro práci s řetězci znaků najdete v tabulce 2.4. Nejzákladnější operací je zřetězení, které prostě spojí dva řetězce v jeden. Nic mezi ně nevkládá. Pokud se jedná o dvě slova, mezi nimiž chcete mít mezeru, musíte ji vložit sami. Zřetězení se zapisuje ve formě tečky mezi spojovanými řetězci.

**Příklad:** Každý z níže uvedených příkazů uloží do proměnné *$napis* řetězec znaků „raz a dva“. Všimněte si mezer uvnitř řetězcových konstant:

```
$napis = "raz␣a␣dva";
$napis = "raz␣a␣" . "dva";
$napis = 'raz' . "␣" . 'a' . '␣' . "dva";
```

Jako pro většinu ostatních operátorů existuje i pro tečku speciální přiřazení .=, které „přiřetězí“ výsledek vyhodnocení výrazu na pravé straně ke stávajícímu obsahu proměnné. Mohl bych proto se stejným výsledkem použít i dvojici příkazů:

```
$napis = "raz";
$napis .= " a dva";
```

■

Speciální odrůdou zřetězení je opakování představované operátorem **x**. Jeho prvním operandem je řetězec a druhým počet výskytů. Například:

```
"pod" x 3
```

vydá řetězec „podpodpod“.

Z funkcí jsou nejzajímavější ty, které se zaměřují na hledání a vydávání podřetězců. Hledání mají na starosti dvě funkce – **index** a **rindex**. Liší se pouze tím, že **index** hledá zleva (tedy první výskyt),

| Operace | |
|---|---|
| . | zřetězení |
| **x** | opakování |
| **Ořezávání** | |
| **chop**(*řetězec*) | odstraní poslední znak v *řetězci* |
| **chomp**(*řetězec*) | odstraní konce řádků |
| **Hledání, podřetězce a spol.** | |
| **length**(*řetězec*) | počet znaků v *řetězci* |
| **index**(*kde*,*co*,*odkud*) | vydá první pozici řetězce *co* v *kde* nebo –1 |
| **rindex**(*kde*,*co*,*odkud*) | vydá poslední pozici *co* v *kde* nebo –1 |
| **substr**(*»řetězec«*,*»start«*,*»kolik«*,*»nahraď«*) | vydá podřetězec |
| **Konverzní funkce** | |
| **uc**(*řetězec*) | převede na velká písmena |
| **lc**(*řetězec*) | převede na malá písmena |
| **ucfirst**(*řetězec*) | převede první písmeno na velké |
| **lcfirst**(*řetězec*) | převede první písmeno na malé |
| **chr**(*číslo*) | vydá znak s daným kódem |
| **ord**(*řetězec*) | vydá kód prvního znaku řetězce |
| **crypt**(*řetězec*,*sůl*) | zašifruje řetězec (dvouznaková *sůl* ovlivňuje výsledek) |

Tabulka 2.4: Řetězcové operace a funkce

zatímco **rindex** zprava (poslední výskyt). Oběma musíte zadat přinejmenším dva parametry: kde se má hledat a co. Navíc smíte přidat parametr třetí, udávající od jaké pozice se má začít s hledáním.

Pokud funkce najde, vydá index (pořadové číslo od počátku prohledávaného řetězce) prvního znaku nalezeného podřetězce. Čísluje se od nuly. Neúspěšné hledání signalizuje návratovou hodnotou –1.

**substr** vám umožňuje vybrat z existujícího řetězce jeho určitou část. Často spolupracuje s hledacími funkcemi – pomocí **index** či **rindex** si najdete hranici, kterou pak použijete při výběru podřetězce.

Parametry funkce **substr** jsou dosti košaté. Povinné jsou opět dva: výchozí řetězec, z nějž vybíráte část, a počáteční pozice (*start*). Je-li nezáporná, počítá se od začátku řetězce, záporná od jeho konce. Třetím, už nepovinným parametrem je délka vybíraného podřetězce (*kolik*). Chybí-li, sahá podřetězec až do konce původního řetězce. Je-li délka zadána záporným číslem, určí se tak, aby do konce původního řetězce zbylo *kolik* znaků. Uf! Honem rychle příklad…

**Příklad:** Předvedu několik výsledků funkce **substr**:

| | | |
|---|---|---|
| **substr**( "Abeceda", 2, 3 ) | vydá | ece |
| **substr**( "Abeceda", 2 ) | vydá | eceda |
| **substr**( "Abeceda", 0, –3 ) | vydá | Abec |
| **substr**( "Abeceda", –4, 3 ) | vydá | ced |
| **substr**( "Abeceda", –4, –3 ) | vydá | c |

■

**Příklad:** Druhý příklad už bude ze života. V proměnné *$cesta* mám cestu k určitému souboru. Rád bych ji rozdělil na vlastní jméno souboru, které uložím do proměnné *$soubor*, a adresář (do proměnné *$adresar*). Jelikož jsou jednotlivé adresáře v cestě oddělovány lomítky, stačí najít v proměnné *$cesta* poslední lomítko, do *$adresar* uložit část cesty před ním a do *$soubor* za ním. Realizace v Perlu by mohla vypadat takto:

*jmsoub.pl*

```
$lomitko = rindex( $cesta, "/" );
$adresar = substr( $cesta, 0, $lomitko+1 );
$soubor = substr( $cesta, $lomitko+1 );
```

Řešení není příliš elegantní, ale zatím se s ním musíte spokojit. Něco lepšího, kdy výše uvedenou trojici příkazů nahradí jediný, poznáte v kapitole o regulárních výrazech (strana 89). ■

Funkce **substr** nemusí jen trpně vydávat podřetězec. Dokáže jej také nahradit jiným – pokud jí zadáte čtvrtý parametr *»nahraď«*. V takovém případě bude určený podřetězec nahrazen obsahem tohoto parametru. Na výsledku funkce se nic nezmění, tím bude stále podřetězec nalezený v původním řetězci.

**Příklad:** Po provedení:

```
$slovo = "Popokatepetl";
$usek = substr( $slovo, 2, 8, "ple" );
```

bude proměnná *$usek* obsahovat řetězec „pokatepe“ a proměnná *$slovo* řetězec „Popletl“. ■

Pokud jste v Perlu očekávali opravdu hodně divoký jazyk, možná se touto dobou cítíte poněkud zklamáni. Snad až na dolary před názvy proměnných zatím vše vypadá víceméně normálně, nic neobvyklého se nekonalo. Dobrá, přitlačíme na pilu.

Funkce **substr** se může vyskytovat i na levé straně přiřazovacího příkazu. Dýchejte zhluboka a přečtěte si ještě jednou, jaký nesmysl jsem právě napsal. Funkce v programovacích jazycích odjakživa slouží k tomu, abychom z jejich parametrů vypočítali určitý výsledek. A tady se najednou má přiřazovat do onoho výsledku!

Skutečnost je taková, že přestože dle oficiálního názvosloví Perlu **substr** je funkce, ve skutečnosti se jedná o syntaktickou konstrukci, která označuje určitou část řetězce. Pokud ji použiji nalevo od znaku pro přiřazení, znamená to, že tato část řetězce bude nahrazena výsledkem vyhodnocení výrazu v pravé části přiřazovacího příkazu. Platí, že:

**substr**(*»řetězec«*, *»start«*, *»kolik«*) = *»výraz«*

je z hlediska účinku na obsah proměnné *»řetězec«* zcela ekvivalentní zápisu:

**substr**(*»řetězec«*, *»start«*, *»kolik«*, *»výraz«*)

Liší se jen výstupní hodnota.

**Cvičení 2.4:** Napište úsek programu, který v řetězci uloženém v proměnné *$slovo* vymění první a poslední znak (z „ahoj“ udělá „jhoa“). ■

V současnosti se pro textové soubory hojně používá kódování znaků UTF-8, se kterým to Perlu notoricky skřípe. Implicitně se k němu nehlásí a zpracování textových dat kódovaných tímto standardem vede k prapodivným výsledkům. Co s tím?

Krátká verze: Přidejte na začátek svého programu příkaz:

**use** *utf8::all*;

Způsobí, že Perl bude korektně zpracovávat řetězce v tomto kódování a také komunikace s okolím – načítání a zapisování hodnot – bude probíhat v UTF-8. Bohužel hodně předbíhám, jedná se

o použití modulu, kterým se budu věnovat až v kapitole 10 na straně 153. Zádrhel spočívá v tom, že modul *utf8::all* nebývá vždy přítomen a musíte jej nainstalovat. Jak na to se píše v příloze 18 na straně 275, obvykle stačí zadat na příkazovém řádku:

```
cpan utf8::all
```

Dlouhá verze: Tom Christiansen podrobně rozebral práci s Unicode a UTF-8 ve svém *Perl Unicode Cookbook*. Text, který ale není zrovna pro zelenáče, najdete na adrese:

https://www.perl.com/pub/2012/04/perlunicook-standard-preamble.html/

## 2.5 Spolupráce řetězců a čísel

Při vzájmeném přiřazování proměnných je Perl velmi benevolentní a vesele převádí čísla na řetězce a opačně, aby vždy něco vyšlo. Pokud do určité proměnné přiřadíte číselnou hodnotu a následně ji použijete jako řetězec, Perl ji automaticky převede na řetězec obsahující zápis dotyčné hodnoty. Například:

```
$cislo = 2 * 11;
$nazev = "Hlava " . $cislo;
```

uloží do proměnné *$nazev* řetězec znaků „Hlava 22“.

Obtížnější je převod opačným směrem – když proměnná obsahuje řetězec znaků a je třeba převést její hodnotu na číslo. Perl se v takovém případě chová velmi podobně, jako při čtení zdrojového textu programu:

- přeskočí v řetězci znaků počáteční prázdné místo (mezery, tabulátory a konce řádků),
- vezme nejdelší souvislou posloupnost znaků, která je platným zápisem čísla a
- převede ji na číselnou hodnotu; pokud řetězec nezačíná zápisem čísla, je výsledkem vyhodnocení nula.

**Příklad:** Několik příkladů výrazů a výsledků jejich vyhodnocení:

```
10 + "19"              dá   29
30 – "    100joj!"     dá  –70
2 * " \n –8"           dá  –16
0 + "3e3e5nazdar"      dá   3000
0 – "pišišvor"         dá   0
```

Všimněte si, že ve čtvrtém řádku je za číslo pokládáno pouze „3e3“, následující znaky už jsou ignorovány, protože netvoří korektní zápis čísla. ■