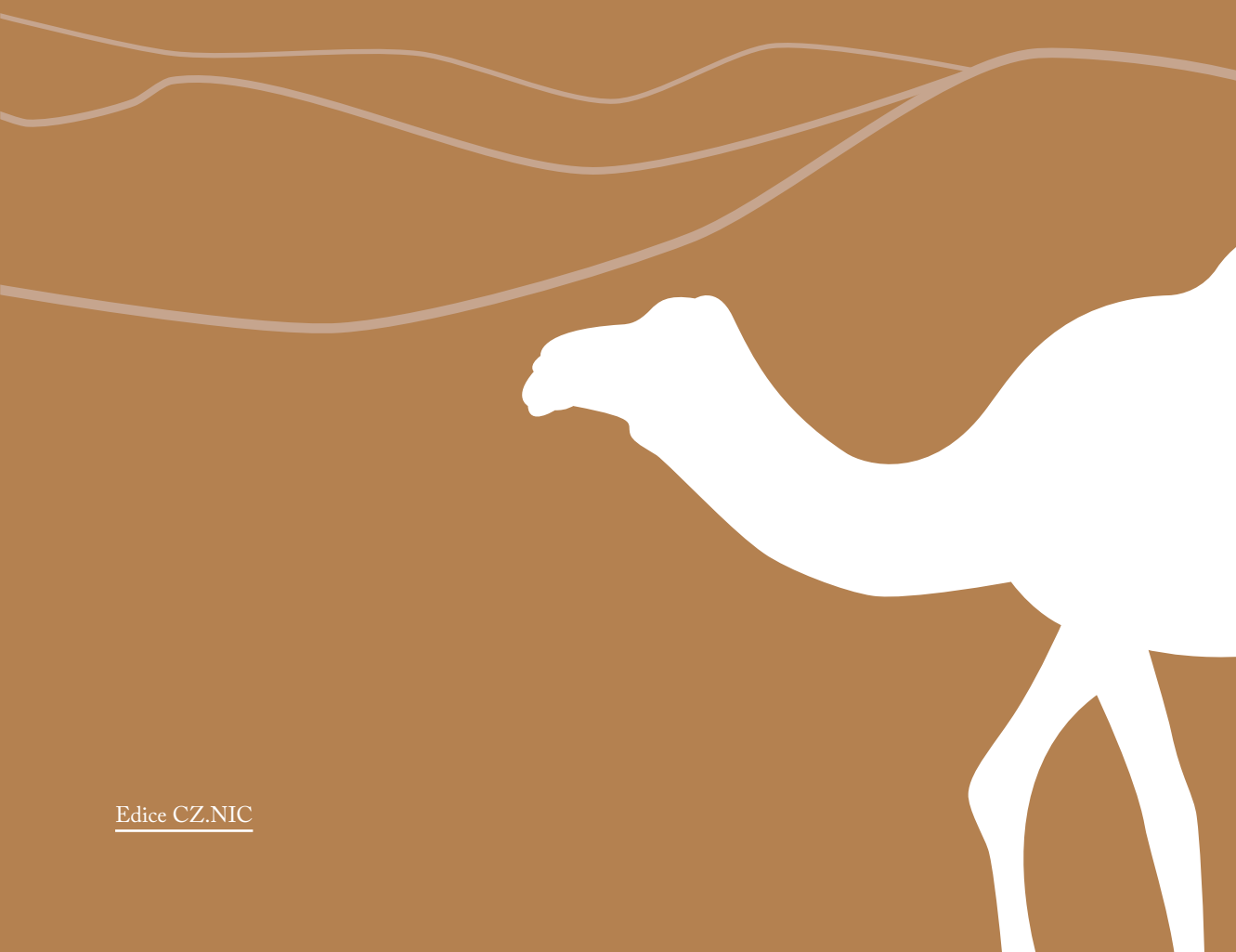
Pavel Satrapa

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. ■

Jediným nedostatkem automatické konverze v porovnání s analýzou číselných konstant ve zdrojovém textu programu je, že nerozpoznává čísla v osmičkové a šestnáctkové soustavě. Pokud je chcete používat, musíte pro převod takového řetězce na číselnou hodnotu zavolat funkci **oct**.

2.6 Úvod do vstupů a výstupů

Mezi programy jen vzácně potkáte naprosté introverty. Takové, které nemají žádný vstup ani výstup. Zpravidla je třeba nějakým způsobem převzít data ke zpracování a vydat výsledky. Na podrobnější diskusi je zatím příliš brzy, nicméně alespoň nejzákladnější konstrukce budu zanedlouho potřebovat.

Začnu tím jednodušším – výstupem. Je realizován funkcí **print**, která jako parametr dostane seznam vystupujících hodnot. Například:

```
$cosi = "stříkaček";
print 333, " stříbných $cosi\n";
```

vypíše „333 stříbrných stříkaček“ a znak konce řádku.

Vstup se nejčastěji odehrává prostřednictvím konstrukce <>, které se říká diamantový operátor[5]. Ten načte ze vstupu jeden řádek a vydá jej jako svůj výsledek. Jelikož je součástí řádku také jeho ukončující znak "\n" (v operačních systémech firmy Microsoft dvojice znaků "\r\n"), zpravidla následuje volání funkce **chomp**, která jej odstraní:

```
$radek = <>;
chomp( $radek );
```

Po provedení této dvojice příkazů obsahuje proměnná *$radek* jeden řádek načtený ze vstupu. A co je oním vstupem? Odpověď na tuto otázku je poněkud komplikovaná. Diamantový operátor dává programům stejné chování, jaké nabízí řada filtrů v Unixu. Pokud programu zadáte jako parametry příkazového řádku jména souborů, bude <> postupně načítat řádky nejprve z prvního souboru, po jeho skončení z druhého atd. Jestliže parametry chybí, bude číst ze standardního vstupu.

Pokud vás zajímá jen a pouze standardní vstup, použijte diamantový operátor v tomto tvaru:

```
$radek = <STDIN>;
```

Když vstup skončí, bude výsledkem diamantového operátoru hodnota **undef**.

5: Z toho je vidět, jak jsou ti angloameričané slušní. Oni když vidí kosočtverec, napadne je diamant.

3 Strukturované příkazy

Do této skupiny patří příkazy, které v sobě obsahují příkazy jiné. Slouží k podmínečnému či opakovanému provádění jednotlivých částí programu. Jelikož hrají klíčovou roli při vytváření struktury programu, dostaly název „strukturované“.

3.1 Blok

Nejjednodušším strukturovaným příkazem je blok. Jedná se o konstrukci, která skupinu příkazů shrne do jednoho celku. Zapisuje se jako dvojice složených závorek:

```
{
   »příkaz1«;
   »příkaz2«;
        ⋮
   »příkazN«;
}
```

Jednou z funkcí bloku je, že umožňuje vytvářet lokální proměnné, jak se dozvíte později. Jeho hlavní role však spočívá jinde. Většina ostatních strukturovaných příkazů, kterými jsou různé podmínky a cykly, totiž jako své tělo nese právě blok. Než se však do nich pustím, seznámím vás s druhým nosným pilířem.

3.2 Podmínky

Činnost řady strukturovaných příkazů je řízena podmínkou. Její splnění či nesplnění rozhoduje o tom, kudy se bude program dále ubírat. Perl bohužel nemá specializovaný pravdivostní typ, který by k tomuto účelu sloužil. Při rozhodování platí/neplatí musí vycházet z čísel a řetězců.

Filozoficky dosti náročnou otázku „Co je pravda?“ si vyřešil jednoduše. Bylo stanoveno, že jako *nepravdivé* budou brány tyto hodnoty:

undef
číslo 0
prázdný řetězec (””)
řetězec obsahující nulu (”0”)

Všechny ostatní hodnoty jsou interpretovány jako pravdivé. Z toho lze usuzovat, že autor Perlu je idealista.

Příklad: Ukážu, jak by dopadlo pravdivostní posouzení některých číselných a řetězcových hodnot:

28	pravda
25 – (3 + 2)*5	nepravda (výsledkem je 0)
"" . 0	nepravda (výsledkem je "0")
"+0"	pravda (řetězec se liší od "0")
1 * "+0"	nepravda (zde se "+0" převede na číslo a výsledkem násobení je pak 0)

■

Z výše uvedeného je patrné, že může být problém odlišit případ, kdy je proměnná nedefinována, od případu, kdy obsahuje hodnotu vyhodnocovanou jako nepravdivá. Pokud takové rozlišení potřebujete, máte k dispozici funkci **defined**. Jako parametr dostane proměnnou a vydá hodnotu „pravda" pouze v případě, že dotyčná proměnná má definovanou (jakoukoli) hodnotu.

Cvičení 3.1: Pokud proměnná *$pokus* bude obsahovat řetězec „nazdar" a jako podmínku použiji funkci:

substr(*$pokus*, 0, 6, "")

Bude výsledkem pravda nebo ne? ■

Nejčastější podmínkou bývá vzájemné porovnání dvou hodnot typu „je *$x* větší než *$y*?" Jeho perlivou podobu najdete v tabulce 3.1.

Význam porovnání číselných hodnot je evidentní a odpovídá tomu, co znáte z matematiky. V případě řetězců Perl používá tak zvané lexikografické porovnání. To znamená, že začne hezky od začátku a porovná první znak levého řetězce s prvním znakem pravého, druhý s druhým a tak dále. Dokud jsou porovnávané znaky shodné, pokračuje dál. Jakmile narazí na první rozdíl (nebo konec řetězců), ten rozhodne.

Znaky porovnává podle jejich ASCII kódu. Takže platí, že „c" je menší než „x", ale také, že libovolné velké písmeno je menší než libovolné malé. Další problém hrozí s českými znaky. Jak na něj se dozvíte na straně 82.

⚡ Perl tady připravil ošklivou pastičku. Nevím, jak vy, ale já jsem se s odlišnými operátory pro porovnání čísel a řetězců sžíval dlouho a bolestně. Dnes již nespočítám, kolikrát jsem si naběhl a srovnával řetězce pomocí == či jiného číselného operátoru.

	čísla	řetězce
je rovno	==	**eq**
není rovno	!=	**ne**
menší než	<	**lt**
menší nebo rovno	<=	**le**
větší než	>	**gt**
větší nebo rovno	>=	**ge**

Tabulka 3.1: Operátory porovnání hodnot

Pokud to uděláte, výsledkem je nesmysl. Řetězce se totiž nejprve převedou na čísla a ta se potom porovnají. Takže například platí:

```
"houba" == "Popokatepetl"
```

protože vyhodnocením obou řetězců vznikne nula (což je ostatně osudem většiny řetězců obsahujících text). Naproti tomu neplatí:

```
"1houba" == "houba"
```

protože výsledkem vyhodnocení levého řetězce je číslo 1. Takže ještě jednou připomínám: nezapomeňte, že řetězce se porovnávají pomocí slůvek **eq**, **ne** a spol. Rovnítka a nerovnítka jsou jen pro čísla.

Jednoduché podmínky člověku občas nestačí. Tu a tam je potřeba říci „pokud je $a>b$ a zároveň je $b>0$“ nebo něco podobného. K tomu slouží logické operace, jejichž nabídku najdete v tabulce 3.2. Konjunkce je splněna pouze tehdy, pokud jsou pravdivé oba její operandy. Naproti tomu disjunkce vydá hodnotu pravda, pokud touto hodnotou skončí vyhodnocení alespoň jednoho z jejích operandů. Negace má jen jeden operand a dává hodnotu přesně opačnou, než on.

konjunkce (logické „a“)	**and** nebo &&
disjunkce (logické „nebo“)	**or** nebo ||
negace	**not** nebo !

Tabulka 3.2: Logické operace

Příklad: Zmíněnou podmínku „pokud je *a*>*b* a zároveň je *b*>0“ bychom v Perlu zapsali:

$a*>*$b **and** *$b*>0

■

Na detailní vysvětlení priorit (která operace má přednost) je zatím příliš brzy. Dočkáte se ho v příloze na straně 252. Zde pouze řeknu, že vše je nastaveno přirozeným způsobem, aby se složitější logické výrazy chovaly tak, jak by člověk očekával. Mocnina má tedy přednost před násobením a dělením, to má vyšší prioritu než sčítání a odčítání, teprve pak následuje porovnávání, ještě níže na žebříčku si stojí konjunkce a pod ní je disjunkce. Za pozornost stojí dost vysoká priorita negace, která se nachází mezi mocninou a násobením. Pokud si nejste jisti nebo chcete změnit implicitní pořadí vyhodnocení, použijte závorky.

⚡ Je třeba dávat si pozor na priority logických operací. Každá z nich má dva povolené tvary – slovní (např. **and**) a znakový (&&). Vykonávají stejnou činnost, ale podstatně se liší v prioritě. Doporučuji vám navyknout si na slovní tvar **and**, **or**, **not**, jehož nízké priority zpravidla odpovídají programátorovým tužbám.

Cvičení 3.2: Napište v Perlu následující podmínky:

1. Řetězec v proměnné *$r* je neprázdný, ale je kratší, než řetězec v *$s*.
2. Číslo v proměnné *$a* má opačné znaménko, než číslo v *$b*.

■

Perl jakožto úsporný jazyk používá tak zvané zkrácené vyhodnocování logických výrazů. Znamená to, že vyhodnocuje jen tak dlouho, dokud není výsledek jistý. Pokud vyhodnocení prvního operandu konjunkce skončí závěrem „nepravda“, druhému operandu se vůbec nebude věnovat. Ten už na celkovém výsledku konjunkce nic nemůže změnit (jelikož konjunkce ke svému splnění potřebuje, aby byly pravdivé oba operandy, bude její výsledek jistě „nepravda“). Stejný osud potká druhý operand disjunkce, pokud byl první shledán pravdivým.

U „normálních“ podmínek vám může být chování interpretu lhostejné (až na výslednou rychlost). Pokud však mají vaše podmínky nějaké vedlejší efekty (například mění hodnoty proměnných), musíte se zkráceným vyhodnocením počítat.

3.3 Podmíněný příkaz

Jedním z nejčastějších programátorských obratů je podmínit provedení části programu splněním určité podmínky. Slouží k tomu podmíněný příkaz, jehož klasická varianta má v Perlu tvar:

```
if ( »podmínka« )
    »blok1«
else
    »blok2«
```

Zpracovává se tak, že nejprve je vyhodnocena *»podmínka«*, kterou může být libovolný výraz. Je-li výsledkem hodnota „pravda“, je podmínka splněna a provedou se příkazy z *»bloku1«*. V opačném případě bude proveden *»blok2«*.

Příklad: Zaměstnavatel se rozhodl plošně navýšit všem svým zaměstnancům plat o 10 %, nejvýše však o 1 500 Kč. Proměnná *$plat* obsahuje mzdu zaměstnance. Její zvýšení podle uvedeného pravidla by zajistil následující programový úsek:

plat.pl

```
if ( $plat < 15000 ) {
    $plat *= 1.1;
} else {
    $plat += 1500;
}
print "Zvýšený plat: $plat\n";
```

■

Dosti častou odrůdou je neúplný podmíněný příkaz. V něm chybí klíčové slovo **else** a druhý blok. Pokud podmínka není splněna, neprovede se prostě nic a vykonávání programu pokračuje za pravou složenou závorkou uzavírající podmíněný blok.

Příklad: V předchozím příkladu na zvýšení platu jsem trochu přemýšlel za počítač, když jsem v podmínce porovnával plat s hranicí 15 000, nad kterou 10% navýšení překročí danou mez. O něco přímočařejší verze by s použitím neúplného podmíněného příkazu (přesáhne-li navýšení limit, omezí se) vypadala třeba takto:

plat2.pl

```
$navyseni = $plat * 0.1;
if ( $navyseni > 1500 ) {
    $navyseni = 1500;
}
$plat += $navyseni;
print "Zvýšený plat: $plat\n";
```

■

Ale co když potřebujete rozvětvit program do více než dvou alternativ? I pro takovou situaci má Perl řešení. Nabízí klíčové slovo **elsif**, které se chová stejně, jako kdyby za **else** bezprostředně následoval další **if**. Tyto dva úseky programu jsou tudíž zcela ekvivalentní:

```
if ( $x <= 0 ) {
   print "x je nanejvýš 0\n";
} elsif ( $x <= 5 ) {
   print "0 < x <= 5\n";
} elsif ( $x <= 100 ) {
   print "5 < x <= 100\n";
} else {
   print "x je větší než 100\n"
}
```

```
if ( $x <= 0 ) {
   print "x je nanejvýš 0\n";
} else { if ( $x <= 5 ) {
   print "0 < x <= 5\n";
} else { if ( $x <= 100 ) {
   print "5 < x <= 100\n";
} else {
   print "x je větší než 100\n"
} } } }
```

Perl je pověstný tím, že stejnou konstrukci můžete zapsat řadou různých způsobů. V případě podmíněného příkazu propukly hotové orgie různorodosti. Už jsem ukázal, že základní podmíněný příkaz má několik variant. Ovšem zdaleka nejsme u konce. Podmínku můžete připsat také za příkaz. Výsledná konstrukce má tvar:

»příkaz« **if** (*»podmínka«*);

Tento druh podmínění se nazývá modifikátor příkazu. *»příkaz«* bude proveden, pouze pokud platí *»podmínka«*. Na rozdíl od podmíněného příkazu zde nemáte k dispozici **else** ani **elsif** a před modifikátorem smí stát jen jednoduchý příkaz, nikoli blok. Závorky kolem *»podmínky«* jsou zde nepovinné.

Já osobně tuto odrůdu podmínek příliš nemiluji. Ženaté čtenáře snadno přesvědčím následujícím příkladem: Je propastný rozdíl, zda řeknete manželce „Jestli na ně máš, běž si koupit nové šaty." nebo „Běž si koupit nové šaty, jestli na ně máš." Ve druhém případě totiž podmínku říkáte do prázdna, protože žena už vyrazila.

S podmiňováním jsem ještě zdaleka neskončil. Perl nabízí také klíčové slovo **unless**, které můžete použít všude tam, kde **if** (včetně modifikátoru). Význam podmínky se v tom případě obrací:

unless (*»podmínka«*)

se chová stejně jako:

if (**not** (*»podmínka«*))

A ještě do poslední nohy: Podmiňování lze zajistit i použitím logických operátorů **and** a **or** mezi příkazy. Jedná se o neobvyklou aplikaci zkráceného vyhodnocování logických výrazů. Provede se první příkaz, čímž vznikne jakási výsledná hodnota. K provedení druhého příkazu dojde jen když

a) příkazy jsou spojeny **and** a výsledkem prvního byla „pravda“ nebo
b) příkazy jsou spojeny **or** a výsledkem prvního byla „nepravda“.

Příklad: S využitím popsaných alternativ lze zkonstruovat celkem šest způsobů, jak převést číslo v proměnné *$x* na jeho absolutní hodnotu:

```
if ( $x < 0 ) { $x = -$x; }
unless ( $x >= 0 ) { $x = -$x; }
$x = -$x if ( $x < 0 );
$x = -$x unless ( $x >= 0 );
$x < 0 and $x = -$x;
$x >= 0 or $x = -$x;
```

■

Jak se z toho nezbláznit? Vybrat si jednu, možná dvě alternativy a ostatní víceméně ignorovat. Například já až na výjimky používám klasický podmíněný příkaz. K ostatním variantám sahám jen výjimečně a uvedl jsem je spíše pro ilustraci mnohotvárnosti Perlu. U následujících příkazů se většinou soustředím jen na jejich vybrané tvary a alternativy buď zcela zatajím nebo se o nich zmíním jen okrajově.

Cvičení 3.3: Přiřaďte do proměnné *$max* větší z hodnot uložených v proměnných *$a* a *$b*. Ve druhé fázi pak zkuste do *$max* přiřadit největší z trojice *$a*, *$b*, *$c*. ■

Cvičení 3.4: A teď něco praktického. Napište program (skutečně celý program, včetně vstupů a výstupů), kterému uživatel zadá koeficienty kvadratické rovnice $ax^2 + bx + c = 0$ a výsledkem budou kořeny dané rovnice. Příslušné vzorečky a různé počty kořenů v závislosti na hodnotě diskriminantu považuji s dovolením za notoricky známé. ■

Lehce odbočím, nicméně v souvislosti s podmíněnými příkazy stojí za zmínku i podmíněné výrazy, které se jim trochu podobají a v jednoduchých případech ušetří práci. Podmíněný výraz je skutečně výraz a vyskytuje se nejčastěji na pravé straně přiřazovacího příkazu. Vypadá takto:

(*»podmínka«*) ? *»hodnota1«* : *»hodnota2«*

Jeho vyhodnocení začne vyhodnocením *»podmínky«*. Pokud je pravdivá, vyhodnotí se *»hodnota1«* a stane se výsledkem celého výrazu. Není-li *»podmínka«* splněna, bude jako výsledek podmíněného

výrazu vyhodnocena »*hodnota2*«. Například do proměnné *$absx* můžete uložit absolutní hodnotu proměnné *$x* příkazem:

```
$absx = ( $x >= 0 ) ? $x : -$x
```

Je to kompaktnější (i když o něco méně přehledné) než tradiční:

```
if ( $x >= 0 ) { $absx = $x } else { $absx = -$x }
```

Blízkým příbuzným je výraz označovaný „definováno–nebo“:

```
»hodnota1« // »hodnota2«
```

Začne vyhodnocením »*hodnoty1*«. Je-li definována, stane se výsledkem. V opačném případě bude výsledkem »*hodnota2*«. Toto se velmi hodí pro výchozí hodnoty, které čekají za dvojicí lomítek a pokud nebyla určena explicitní hodnota, přijdou ke slovu:

```
$posledni = $kapacita // 100;
```

Pokud má proměnná *$kapacita* definovanou hodnotu, bude přiřazena do proměnné *$posledni*. Není-li definována, použije se jako výchozí hodnota 100. Konec odbočky, vraťme se ke strukturovaným příkazům.

3.4 while cyklus

Cyklus je konstrukce, která umožňuje opakované provádění určité části programu. Perl jich nabízí několik druhů. Zde se podívám na cykly řízené podmínkou. Další odrůdy, které se vyskytují nejčastěji v souvislosti s poli, si pošetřím až na kapitolu o polích.

Asi nejběžnějším perlivým cyklem je **while**. Zapisuje se ve tvaru:

```
while ( »podmínka« )
  »blok«
```

Jeho účinek by se dal popsat větou „dokud platí »*podmínka*«, opakuj »*blok*«“. Přesněji řečeno funguje takto:

- Vyhodnotí »*podmínku*«.
- Je-li výsledkem „pravda“, provede »*blok*« a opakuje postup znovu od začátku (vyhodnocením podmínky).
- Pokud podmínka není splněna, pokračuje se v provádění programu za cyklem.

Příklad: Kdyby neexistovala funkce **length**, mohl bych si ji díky **while** cyklu naprogramovat sám. Řekněme, že proměnná *$retezec* obsahuje řetězec znaků, jehož délku chci změřit:

delka.pl

```
$delka = 0;
while ( $retezec ne "" ) {
    $delka++;
    substr( $retezec, 0, 1, "" ); #odstraní 1 znak
}
print "Délka je $delka znaků.\n";
```

Dokud je řetězec neprázdný, odeberu vždy jeden znak a přičtu jedničku k dosavadní délce. Jakmile tento řetězec vyprázdním, podmínka přestane platit, cyklus skončí a ke slovu přijde závěrečné ohlášení výsledků.

Pokud podmínka není splněna hned při vstupu do cyklu, tělo se neprovede ani jednou. Díky tomu program funguje správně i pro prázdné řetězce – ohlásí nulovou délku. ■

Také cyklus má alternativní způsoby zápisu. Tím prvním je:

```
do
  »blok«
while ( »podmínka« );
```

Tentokrát se nejprve provede *»blok«* a až po něm následuje vyhodnocení *»podmínky«*. Pokud platí, opakuje se vše znovu. V opačném případě cyklus končí a provádění programu pokračuje následujícím příkazem.

Nejvýznamnějším rozdílem obou variant je, že při testování podmínky na začátku se cyklus nemusí vůbec provést, jestliže hned při vstupu do cyklu podmínka neplatí. Naproti tomu při testování na konci se tělo cyklu vždy alespoň jednou provede – k prvnímu vyhodnocení podmínky dojde až poté, co byl poprvé proveden *»blok«*. V řadě případů je použití cyklu s podmínkou před a za tělem zcela ekvivalentní. Například pro tabulku druhých mocnin čísel 1 až 10 je jedno, po které variantě sáhnete:

```
my $i=1;
while ( $i <= 10 ) {
  print "$i * $i = ", $i*$i, "\n";
  $i++;
}
```

```
my $i=1;
do {
  print "$i * $i = ", $i*$i, "\n";
  $i++;
} while ( $i <= 10 );
```

Tu a tam narazíte na situaci, kdy jedna z variant vede k elegantnějšímu řešení. Například počítání délky řetězce z posledního příkladu by s podmínkou za tělem bylo poněkud krkolomné (prázdný řetězec číhá!). Já osobně podmínku za tělem cyklu prakticky nepoužívám.

Letem světem se zmíním o dalších cyklistických možnostech. **while** lze použít jako modifikátor za jednoduchým příkazem. Kromě toho existuje klíčové slovo **until**, které se podobá **while**, ovšem obrací význam podmínky. Při jeho použití se cyklus opakuje, dokud podmínka neplatí – je ekvivalentní **while** (**not** (…)).

Cvičení 3.5: Napište úsek programu, který do proměnné *$delitel* uloží největšího společného dělitele čísel *$a* a *$b*. Pokud neznáte dotyčný algoritmus, je celkem jednoduchý: dokud se *$a* liší od *$b*, odečtete vždy od větší z hodnot tu menší. Až budou stejné, jsou rovny největšímu společnému děliteli. ■

3.5 Řízení cyklů

Základním ovládacím prvkem cyklu je jeho podmínka. Perl nabízí navíc několik možností, jak jej řídit i zevnitř. Konkrétně se jedná o příkazy **last**, **next** a **redo**. Aby měly rozumný smysl, bývají obaleny podmíněným příkazem. Při splnění jeho podmínky pak dojde ke změně ve zpracování cyklu.

last způsobí jeho okamžité ukončení. Tělo cyklu se přestane zpracovávat a program pokračuje příkazem následujícím za cyklem. **next** přeskočí na novou obrátku cyklu. Zbytek těla se vynechá a zpracování pokračuje vyhodnocením podmínky cyklu, která rozhodne o tom, jak bude zpracování pokračovat. **redo** zopakuje provádění těla cyklu od začátku. K vyhodnocení podmínky zde nedochází a v provádění programu se pokračuje od prvního příkazu v těle cyklu.

Prostředky pro řízení cyklů se zpravidla používají k ošetření výjimečných situací. V zásadě nejsou nezbytné – stejného výsledku by se dalo dosáhnout vhodnými podmíněnými příkazy. Někdy však program výrazně zjednoduší a zpřehlední.

Příklad: Představte si, že máte zpracovávat jistý vstupní soubor. Bylo stanoveno, že řádky začínající znakem # jsou komentáře a mají být při zpracování ignorovány. Pokud řádek začíná řetězcem ##END, jedná se o ukončení vstupních dat a vše, co následuje za ním, má být ignorováno. Cyklus pro zpracování takového souboru by mohl vypadat následovně:

```
while ( $radek = »načti_další_řádek« ) {
  if ( index ($radek, "##END") == 0 ) { last; }
  if ( index ($radek, "#") == 0 ) { next; }
  »zpracování_řádku«
}
```

■

Chcete-li si vychutnat tyto nástroje až do dna, nemohu před vámi dále tajit blok **continue**. Jedná se o nepovinný doplňující blok cyklu, který se provádí vždy po provedení těla cyklu. Úplné zpracování cyklu tedy je:

podmínka tělo continue podmínka tělo continue…

Blok **continue** zpravidla chybí. Smysl má pouze ve spojitosti s příkazem **next**. Ten totiž nepokračuje podmínkou cyklu, jak jsem před chvilkou tvrdil, ale ještě před ní bude proveden blok **continue**. Teprve pak přijde ke slovu podmínka. U **last** ani **redo** se blok **continue** neprovádí. Pokud jej použijete, zpravidla se do něj zařazují příkazy přidělující řídící proměnné cyklu novou hodnotu.

Uvedené řídící příkazy se vždy týkají toho cyklu, kterým jsou bezprostředně obklopeny. Oficiálně se tomu říká „nejvnitřnější vnořený cyklus“. Pokud chcete, aby se činnost příkazu týkala jiného cyklu, použijte návěští.

V podstatě to znamená, že příslušný cyklus označíte jménem a toto jméno pak použijete jako parametr příkazu **last**, **next** či **redo**. Návěští je běžný identifikátor, který byste však měli psát velkými písmeny, aby nehrozil konflikt s jiným prvkem programu a také aby se návěští dostatečně odlišilo od běžného textu. Od vlastního příkazu je odděleno dvojtečkou.

Příklad: A ještě jednou komentáře, tentokrát trochu jinak. Sestrojím program, který opisuje svůj vstup do výstupu, ale ignoruje komentáře zahájené znakem „#“. Skládá se z dvojice navzájem vnořených cyklů. Vnější načítá řádky ze vstupu. Vnitřní pak zpracovává řádek po jednotlivých znacích a když narazí na „#“, poskočí rovnou na další řádek:

koment.pl

```
1   RADKY: while ( my $radek = <> ) {
2               chomp($radek);
3               while ( my $znak = substr($radek,0,1,"") ) {
4                   if ( $znak eq "#" ) {
5                       next RADKY;
6                   } else {
7                       print $znak;
8                   }
9               }
10          } continue { print "\n"; }
```

Dovoluji si upozornit, že program vznikl z myšlenky „jak najít slespoň trochu smysluplný příklad na návěští a **continue**“, nikoli „jak optimálně vypouštět komentáře“. Daleko efektivnější by samozřejmě bylo prostě v řádku najít znak „#“ a vymazat jej i vše za ním. ■

Příklad zároveň ilustruje dost častou situaci, kdy k řízení cyklu slouží tak zvaná řídicí proměnná. Ta postupně prochází jednotlivé hodnoty (zde řádky ze vstupu, resp. písmena v řádku) a dokud splňuje podmínku, pro každou z hodnot se provede tělo cyklu.

Řízení cyklu je jediným smyslem existence této proměnné, mimo něj se vůbec nepoužívá a nemusí tam ani existovat. Řídicí proměnné se proto obvykle deklarují pomocí **my** přímo v podmínce, jak vidíte v příkladu na řádcích 1 a 3.

3.6 Zápis programu

Z dosavadních ukázek už jste si jistě udělali poměrně dobrou představu o tvaru, ve kterém se zapisují perlivé programy. Považuji však za slušnost učinit na toto téma oficiální prohlášení.

Programy v Perlu mají volný zápis. To znamená, že mezi jednotlivé prvky jazyka můžete podle libosti vkládat mezery a konce řádků. Díky tomu jsou následující tři přiřazovací příkazy zcela ekvivalentní:

```
$a=-sqrt($b);
$a = - sqrt ( $b );
$a =
-       sqrt (    $b
);
```

Třetí z nich považuji s dovolením za zvěrstvo. Jestliže vás Perl nenutí do pevně daného tvaru, snažte se uspořádat zdrojový text tak, aby pro vás byl co nejsrozumitelnější.

Jedním z nejzákladnějších prostředků pro zvýšení přehlednosti je zvýrazňování bloků pomocí odsazení od levého okraje. Základní pravidla jsou jednoduchá: kdykoli začnete nový blok (tělo strukturovaného příkazu), prodlužte odsazení jeho příkazů od levého okraje. Příkazy, které jsou na stejné blokové úrovni, by měly mít stejné odsazení. Ukončující složená závorka by pak měla být na stejné úrovni, jako klíčové slovo, ke kterému blok patří.

Ilustrativní příklad jste mohli vidět v programu *koment* na předchozí straně. V něm jsou celkem tři úrovně odsazení: tělo vnějšího **while** cyklu, tělo vnitřního **while** cyklu a tělo podmíněného příkazu. Už z uspořádání programu je jasně patrné, co je v čem vnořeno. Porovnejte si jeho srozumitelnost s následujícím tvarem (program je shodný, změnilo se jen uspořádání):

koment2.pl

```
RADKY: while ( $radek = <> ) { chomp($radek); while ( $znak =
substr($radek,0,1,"") ) { if ( $znak eq "#" ) { next RADKY; } else
{ print $znak; } } } continue { print "\n"; }
```

Jako optimální krok při odsazování se doporučuje používat čtyři mezery (resp. tabulátor s čtyřmezerovým krokem).

Jedinou striktní povinností z hlediska zápisu jsou středníky, kterými od sebe musíte oddělovat jednotlivé příkazy. Středník je nepovinný jen za posledním příkazem v bloku, nicméně doporučuji vám, abyste jej psali i tam. Pokud se později rozhodnete přidat do bloku ještě pár příkazů, nehrozí vám, že jej zapomenete připsat a vyloudíte tak zbytečnou chybu.

Středník navíc (jako ten před koncem bloku) ničemu nevadí. Pouze vytvoří prázdný příkaz, který nic nedělá a interpret jej beztak vypustí již při předkompilaci.

Poněkud obskurní je přístup Perlu k závorkám při volání funkcí. Na rozdíl od většiny jazyků jsou zde nepovinné. Chcete-li do proměnné *$x* přiřadit odmocninu ze dvou, máte na výběr dva tvary:

$x = **sqrt**(2);
$x = **sqrt** 2;

Výsledkem této nejednoznačnosti je, že se ustálily jisté konvence, ve kterých lze jen těžko vysledovat nějakou logiku. Zpravidla vycházejí z prostého „většina lidí to tak píše“. Typickou funkcí, která se píše bez závorek, je **print**. Naproti tomu aritmetické funkce se nejčastěji píší s nimi.

Pokud se rozhodnete závorky vynechat, musíte přinejmenším otevírací nahradit mezerou, abyste oddělili první parametr funkce od jejího názvu. Když závorky píšete, doporučuje se, abyste mezi názvem funkce a levou závorkou zahajující seznam parametrů nedělali žádnou mezeru.

Pokud si chcete svůj styl dále pilovat, zkuste:

```
man perlstyle
```

4 Ladění programů

Pojmem „ladění“ se v programátorském světě označuje hledání a odstraňování chyb. Jedná se o bolestný proces, během nějž vyvracíte mýtus o vlastní neomylnosti. Murphyho zákony jsou však neúprosné. Říkají:

- V každém netriviálním programu je alespoň jedna chyba.
- Odstraněním chyby způsobíte alespoň jednu novou.

Boj s chybami tedy lehce připomíná pověstný boj s větrnými mlýny, který svedl don Quijote. Podívejme se, jakou zbroj k němu dostanete.

4.1 Ladicí tisky

Nejzdlouhavější fází odstraňování chyby bývá její hledání. Program chrlí bláboly a vy vůbec netušíte, kde a proč k tomu dochází. Nejčastějšími zdroji chyb bývají nezvládnuté regulární výrazy či přiřazovací příkazy, kdy se v proměnné ocitne něco, co jste nečekali.

Primitivním, ale vždy dostupným prostředkem pro sledování činnosti programu jsou ladicí tisky. Jednoduše tu a tam na klíčové místo programu vložíte příkaz **print**, který se netýká jeho vlastní činnosti. Má za cíl informovat vás o tom, kterou větví se program vydal, kolikrát byl zopakován cyklus či jakou hodnotu má některá proměnná.

Neexistuje univerzálně platná rada kam umisťovat ladicí tisky. Je potřeba vytipovat problematické místo a jeho funkci pak vhodně prověřit.

Z hlediska vlastní konstrukce tiskových příkazů bych si dovolil dvě doporučení. Jejich výsledky směrujte do STDERR, abyste měli šanci je oddělit od „regulérních“ výstupů programu. Na konec vždy připojte určitý konstantní komentář (např. já používám #!!!), abyste je mohli později snadno vyhledat a odstranit.

Příklad: Kdyby se řekněme vzpěčoval program pro výpočet největšího společného dělitele ze strany 255, vložil bych jako první příkaz **while** cyklu ladicí tisk ohlašující hodnoty proměnných *$a* a *$b*:

```
print STDERR "a=$a, b=$b\n"; #!!!
```

Tak bych mohl sledovat jejich vývoj. Pokud by neodpovídal očekávání, vložil bych další ladicí tisky do obou větví podmíněného příkazu v těle cyklu, abych zjistil, kudy se ubírá chod programu a co tam dělá. ■

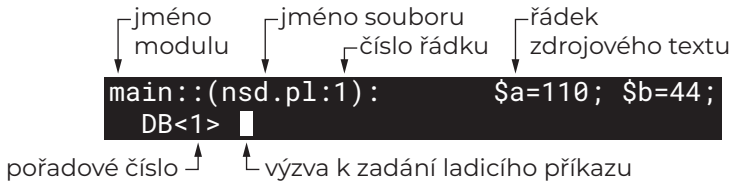
Základní výhodou ladicích tisků je, že se jedná o zcela standardní příkazy a nemusíte se tudíž učit nic nového. Platíte za ni omezenými možnostmi, které vám nabízejí. Co takhle něco silnějšího?

4.2 Vestavěný debugger

Anglicky se programová chyba označuje bug (veš), proces odstraňování chyb debugging (odvšivování) a nástroj, kterým jej provádíte, je debugger (Biolit). Perl má takový ladič přímo vestavěn sám v sobě. Stačí interpret spustit s volbou `-d`, například:

```
perl -d nsd.pl
```

Objeví se výzva, jakou vidíte na obrázku 4.1. Ladič čeká na vaše příkazy. Princip jeho činnosti spočívá v tom, že vám umožňuje pohybovat se v rozpracovaném programu a řídit postup jeho vykonávání. Můžete jej pozastavit v kritickém místě a zjišťovat, jakou má co hodnotu a jak ji ovlivní následující běh programu.



Obrázek 4.1: Příkazový řádek vestavěného ladiče

Ladicí příkazy se zadávají zpravidla jedním znakem. Lze je rozdělit do několika skupin. V přehledné formě vám je nabídne nápověda vyvolaná pomocí `h h`. Chcete-li se dozvědět větší podrobnosti, použijte `h` *příkaz* nebo samotné `h`.

Základní podmínkou úspěšného ladění je zorientovat se. K tomu potřebujete příkazy, které vám vypíší text laděného programu či aktuální hodnoty některých proměnných. Nejzákladnějším příkazem pro zobrazování zdrojového kódu je `l`. Vypíše několik řádků programu počínaje aktuálním. Zároveň si zapamatuje, kde skončil. Při příštím použití `l` bude výpis pokračovat za naposledy zobrazeným řádkem. Chcete-li vidět určitý konkrétní řádek, použijte `l` *číslo_řádku*.

Alternativou je příkaz `w`. Jeho činnost je podobná, ale svůj výpis zahájí vždy několik řádků před aktuálním. Díky němu vidíte nejen to, co následuje, ale i co předcházelo. Také `w` můžete jako parametr zadat číslo řádku. Ve zdrojovém textu si můžete nechat vyhledat určitý řetězec znaků. Slouží k tomu obvyklé `/`*vzor* pro hledání směrem ke konci souboru a `?`*vzor* pro obrácený směr.

Pro inspekci současného stavu proměnných se nejčastěji používá příkaz `p`. Chová se velmi podobně jako **print** v Perlu. Například hodnoty proměnných *$a* a *$b* zobrazíte příkazem:

```
p "$a $b"
```

Jste-li zorientováni, můžete se pustit do díla. Základní ladicí technikou je tak zvané *krokování*. Spočívá v tom, že necháte provést vždy jen jeden příkaz a průběžně sledujete, jak se vyvíjí chod programu a hodnoty proměnných.

Hlavními krokovacími příkazy jsou `n` a `s`. Ve většině případů se jejich činnost neliší: provedou jeden příkaz. Rozdíl nastane, pokud je oním příkazem volání podprogramu[1]. Zde `n` chápe volání jako jeden příkaz, provede celý podprogram a poskočí na příkaz následující za voláním. Naproti tomu `s` vstoupí do podprogramu a pokračuje v krokování na jeho prvním řádku.

Ku prospěchu vašich prstů nabízí ladič jednoduchou zkratku: stisknutím samotné klávesy **Enter** zopakujete poslední příkaz `n` nebo `s`.

Abyste se snadněji orientovali, vypíše ladič při každém zastavení příkaz, který bude proveden jako následující. Postupuje skutečně po příkazech. Pokud jich jeden řádek obsahuje několik, nenechte se zmást skutečností, že se objeví opakovaně – pro každý svůj příkaz jednou.

Krokování připomíná činnost, které na vojně říkají „plížením plazením vpřed“. Je to celkem bezpečné, ale pomalé. V rozlehlých programech byste nejspíš usnuli, než byste dokráčeli do míst, která způsobují problémy. Proto existuje několik urychlovačů.

Jedním z nich je `c` *číslo_řádku*. Říká „Pokračuj sám v provádění programu a až dojdeš na řádek daného čísla, zastav!“ Pokud podezíráte ze zlých úmyslů podprogram začínající na řádku 132, zadejte `c 132`. Tím se jedním mocným skokem ocitnete na jeho začátku a dále můžete pokračovat krokováním této podezřelé části. Jestliže nenarazíte na nic zajímavého, můžete si za chvíli dalším příkazem `c` poskočit jinam.

Chcete-li vidět, kudy Perl kráčí, zapněte si trasovací režim. V něm interpret vypisuje na obrazovku příkazy, které provádí. Zapínání a vypínání trasovacího režimu má na starosti příkaz `t`.

V některých situacích je stanovení čísla řádku pro zastavení obtížné (předem nevíte, kterou větví se program bude ubírat) nebo nepohodlné (zadáváte pořád dokola stejné číslo). Řešením je tak zvaný *bod přerušení*, anglicky breakpoint. Vložíte-li na některý řádek bod přerušení, kdykoli k němu ladič dorazí, přeruší vykonávání programu a vyzve vás k zadávání příkazů.

1: Dobrá, zatím jsem o podprogramech nemluvil, ale časem na ně dojde.

Bod přerušení zavedete příkazem `b` *`číslo_řádku`*. Navíc jej můžete doplnit podmínkou, která se zapisuje i vyhodnocuje jako v Perlu. Chybí pouze **if** a okolní závorky. V takovém případě se chod programu přeruší jen když dorazí na daný řádek a podmínka je splněna.

Příklad: Pokud chcete vložit na řádek číslo 8 bod přerušení, který pozastaví program jen když bude *$a* menší než nula, použijte příkaz:

```
b 8 $a<0
```

■

Místo čísla řádku můžete použít název podprogramu. Bod přerušení bude vložen na jeho první řádek. Bodů přerušení můžete mít, co hrdlo ráčí. Příkaz `L` vám zobrazí jejich seznam. Pomocí `d` *`číslo_řádku`* dotyčný bod přerušení odstraníte. Příkaz `D` pak zlikviduje všechny naráz.

Ve spojení s body přerušení se často používá příkaz `c` bez parametrů. Dáváte jím příkaz, aby chod programu pokračoval, dokud nenarazí na některý bod přerušení nebo na konec.

Některé hodnoty jsou natolik zajímavé, že si o nich chcete udržovat trvalý přehled. K tomu slouží tak zvané hlídané výrazy (watch expressions) nebo akce. Hlídané výrazy jsou velmi intenzivní. K jejich vyhodnocení dochází před provedením *každého* příkazu ze zdrojového textu. Zavádějí se příkazem `W` *`výraz`*, kde *výraz* bývá nejčastěji perlovský příkaz **print**. Vymazat lze pouze všechny najednou použitím samotného `W`.

Akce se jim podobají, ale jsou přiřazeny určitému konkrétnímu řádku. Zavádějí se pomocí `a` *`číslo_řádku příkaz`*. Nejčastěji používaným příkazem opět bývá **print**. K jejich provedení dochází vždy před provedením příkazu na daném řádku. O likvidaci všech akcí se postará příkaz `A`.

Příklad: Nadešel čas na komplexnější ukázku. Budu ladit hledání největšího společného dělitele ze strany 255. Na začátek přidám řádek, kde nastavím proměnným *$a* a *$b* úvodní hodnoty 110 a 44. Klíčovou informací je právě vývoj jejich hodnot uvnitř cyklu, zajímavý bude především řádek 8 – první příkaz v těle cyklu. Na něm si nastavím bod přerušení a přidám akci, která vždy vypíše aktuální hodnoty těchto proměnných:

```
DB<1> b 8
DB<2> a 8 print "a=$a b=$b\n"
```

Nyní mohu klidně pozorovat dění:

```
DB<3> c
main::(nsd.pl:8):               if ( $a < $b ) {
a=110 b=44
DB<3> c
main::(nsd.pl:8):               if ( $a < $b ) {
a=66 b=44
DB<3> c
main::(nsd.pl:8):               if ( $a < $b ) {
a=22 b=44
```

Čuchám, čuchám člověčinu! Hodnoty by se měly v příštím kroku vyrovnat. Nasadím tedy trvalé sledování a navíc trasování programu, ať vidím do detailu, co se děje:

```
DB<3> W print "a=$a b=$b\n"
a=22 b=44
DB<4> t
Trace = on
DB<4> c
a=22 b=44
main::(nsd.pl:9):                   $b -= $a;
a=22 b=22
main::(nsd.pl:7):           while ( $a != $b ) {
a=22 b=22
main::(nsd.pl:14):          $nsd = $a;
a=22 b=22
main::(nsd.pl:16):      print "$nsd\ n";
22
a= b=
Debugged program terminated...
```

Závěrečné hlášení (které jsem si dovolil poněkud zkrátit) oznamuje, že program byl dokončen. Chcete-li jej spustit znovu, restartujte ladič příkazem `R` a začněte zase od začátku. Všechny body přerušení i akce zůstaly nastaveny. ■

Při ladění podprogramů vám přijde k duhu příkaz `T`. Vypíše tak zvaný zásobník volání, čili informace o tom, kdo zavolal právě zpracovávaný podprogram, kdo zavolal tohoto volajícího atd. atd. až po úroveň hlavního programu. Ladíte-li složitější program s řadou podprogramů, je tato informace velice cenná.

Pro zobrazování složitějších datových struktur používajících odkazy je určen příkaz `x`. Interpretuje svůj parametr jako pole, vypíše jeho hodnoty a pokud je některá z nich odkazem, vypíše i to, co

najde na jeho konci. Ke zobrazování dostupných metod objektů slouží `m`, ale to už opravdu hodně předbíhám.

4.3 Data Display Debugger

Uživatele operačních systémů typu Unix by mohl zajímat *Data Display Debugger (DDD)*. Jedná se o zajímavý volně použitelný program, jehož autory jsou Dorothea Lütkehaus a Andreas Zeller. Jejich cílem bylo vytvořit grafickou nadstavbu nad existujícími textovými ladiči. Takovéto nadstavby nejsou nic nového pod sluncem a jejich kvalita často bývá nevalná. DDD je výjimkou potvrzující pravidlo. Je přehledný a snadno použitelný. Získáte jej v Internetu na adrese:

https://www.gnu.org/software/ddd/

nebo nainstalujete z balíčku. Jeho spuštění zajistí příkaz:

```
ddd »program« &
```

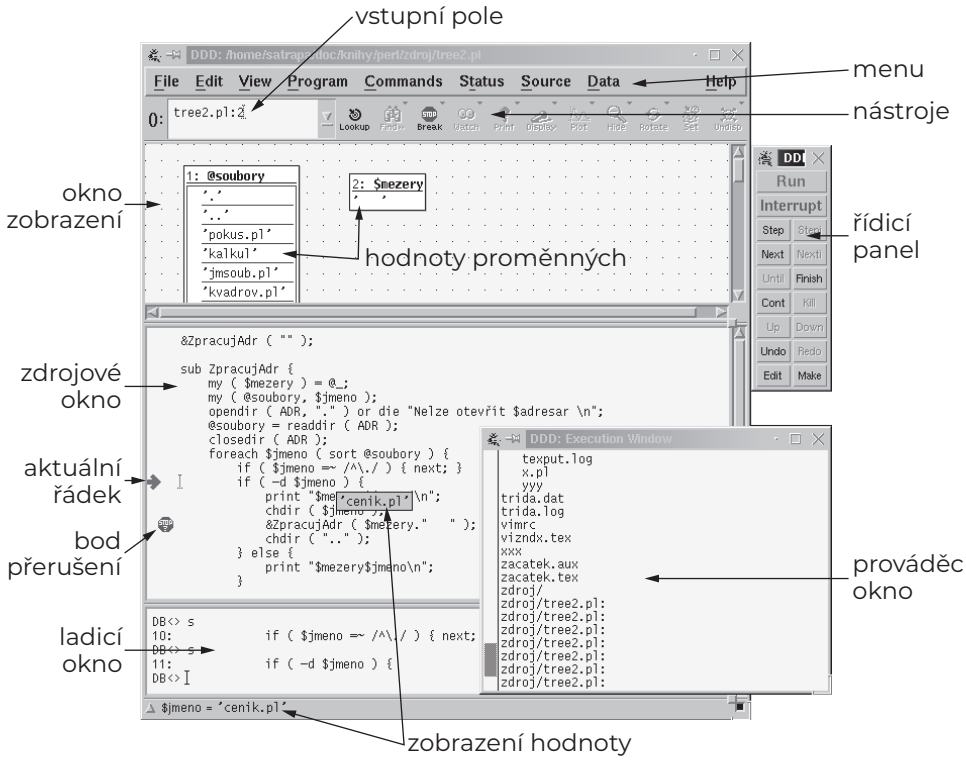
DDD je chytrý a sám si určí, v jakém jazyce je *»program«* zapsán a podle toho si zvolí odpovídající ladič. Jedná se o grafický program, takže jej musíte spouštět z grafického uživatelského prostředí. Pokud má *»program«* dostat nějaké parametry, uveďte je za jeho názvem, jako obvykle. Kdyby snad DDD nepoznal, že se jedná o program v Perlu, sdělte mu to sami pomocí:

```
ddd --perl »program« …
```

DDD používá několik oken, která buď mohou být umístěna samostatně nebo vnořena do jednoho velkého (což je implicitní nastavení). Program uprostřed práce vidíte na obrázku 4.2.

Pro orientaci v programu a sledování postupu je rozhodující zdrojové okno. Obsahuje zdrojový text programu a informace, které se k němu vztahují. Řádek, který se má provést, je označen zelenou šipkou vlevo. Tamtéž najdete i červené značky „Stop“ signalizující body přerušení.

Postup ladění budete ovládat z řídicího panelu. Pro Perl řada jeho tlačítek nemá smysl, takže aktivní je přibližně polovina. Ta nejzajímavější najdete v levém sloupci – **Step** provede jeden příkaz (vstupuje do podprogramů), **Next** udělá totéž, ale podprogramy bere jako jeden příkaz, a **Cont** nechá program pokračovat až do konce (resp. po následující bod přerušení). Poněkud nešťastně je pojmenováno tlačítko **Run**. Název hlasitě našeptává, že se jím bude něco spouštět. Ve skutečnosti pouze restartuje provádění programu – zlikviduje všechny stávající proměnné a hodnoty a převede laděný program do téhož stavu, ve kterém byl před provedením svého prvního příkazu. Použijte je, kdykoli chcete začít s laděním zase od začátku.



Obrázek 4.2: Data Display Debugger

Ladicí okno obsahuje výstupy a příkazy standardního perlovského ladiče. Pokud si neotevřete samostatné prováděcí okno (což zajistí příkaz **View / Execution Window**), najdete zde také výstupy samotného programu a zadáváte vstupy.

Pro zobrazení hodnoty proměnné máte hned několik alternativ. Nejjednodušší je prostě na ni přesunout kurzor myši. Vyskočí bublinová „nápověda“ obsahující aktuální hodnotu. Jednoduché, pohotové, geniální.

Pokud vás zajímá trvaleji, můžete si proměnnou zařadit do zobrazovacího okna. Stiskněte na ní pravé tlačítko myši a z menu vyberte **Display**. Jestliže zobrazovací okno dosud neexistovalo, automaticky se otevře. DDD si dokáže inteligentně poradit ve všemi typy proměnných Perlu a zobrazuje je rozumným způsobem.

Zobrazovací okno se chová jako nástěnka, na které máte přilepeny lístečky s hodnotami jednotlivých proměnných. Můžete je myší libovolně přesouvat. Pokud vás některý omrzí, stiskněte na něm pravé tlačítko a vyberte **Undisplay**. Volbou **Set value** z téhož menu lze změnit hodnotu libovolné z proměnných.

Pravé tlačítko se nepochybně stane vaším přítelem, protože jeho prostřednictvím budete zadávat řadu pokynů. Většina z nich má své ekvivalenty i v menu či liště nástrojů, nicméně pravé tlačítko bývá rychlejší.

Tak například body přerušení. Stiskěte pravé tlačítko myši na řádku, kde chcete vytvořit bod přerušení. Musíte je stisknou *mimo* jeho zdrojový text, obecně se doporučuje klepnout nalevo od něj. Z menu vyberte **Set Breakpoint** a je hotovo. Když později klepnete pravým tlačítkem na značce bodu přerušení, nabídnou se vám věci jako **Properties** (upravit vlastnosti – např. k němu přidat podmínku), **Disable Breakpoint** (dočasně jej vypnout) či **Delete Breakpoint** (zcela jej odstranit). Můžete jej také popadnout levým tlačítkem myši a přetáhnout na jiný řádek.

Zajímavou možností je uložení stavu DDD. Z menu **File** vyberte položku **Save Session As** a zadejte jméno souboru. Později můžete stav obnovit pomocí **File / Open Session**. Upřímně řečeno tady autoři mohli jít o kousek dál. Uložení se totiž týká především bodů přerušení. Neukládá se informace o zdrojovém kódu (ten musíte načíst před obnovením stavu ladění) ani obsahu okna zobrazení. Doufám, že v budoucích verzích (v době psaní byla aktuální 3.1.3) tahle vlastnost povyroste.

Některé položky menu a snad všechny nástroje obsahují tajemný symbol **()**. Představuje parametr, který potřebují ke své činnosti (například text, který se má vyhledat ve zdrojovém kódu). Jeho hodnotu zadáte ve vstupním poli, které najdete v levé části nástrojové lišty. Když klepnete levým tlačítkem myši na některé slovo nebo vyberete určitou část zdrojového textu, automaticky se do tohoto pole vloží. Díky tomu nemusíte vše vypisovat ručně.

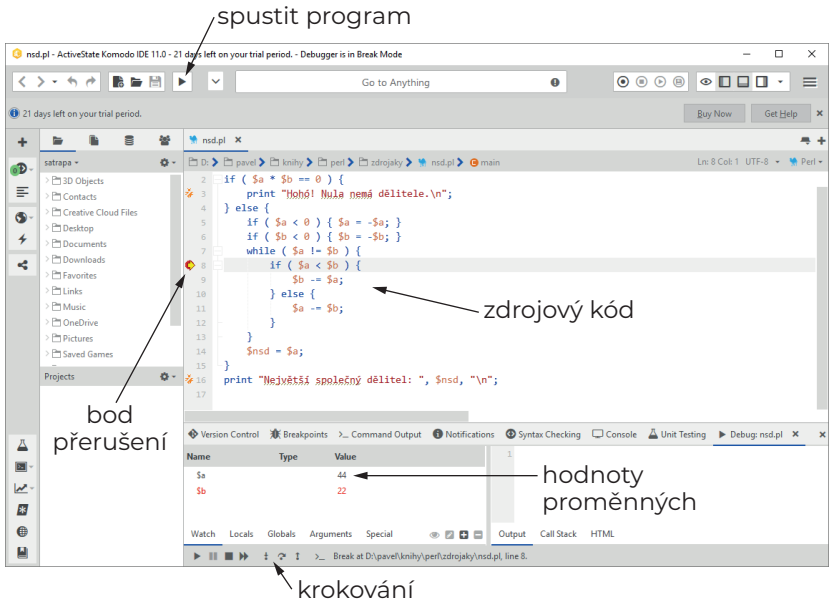
Pro DDD existuje velmi podrobný a pěkně provedený manuál o rozsahu přes 200 stran. Distribuuje se v podobě PDF či PostScriptového souboru společně s programem a najdete jej i na výše odkazovaných stránkách.

4.4 Komodo IDE

Ještě o kus dál než DDD jsou z hlediska schopností a uživatelského pohodlí integrovaná vývojová prostředí, která se snaží pod jednou střechu nabídnout vše, co k vývoji aplikací potřebujete. Velmi dobou pověst má například komerční *Komodo IDE*:

https://www.activestate.com/komodo-ide

Jedná se o integrované vývojové prostředí pro širokou škálu jazyků, Perl nevyjímaje. Jeho podobu představuje obrázek 4.3.



Obrázek 4.3: Komodo IDE

Hlavní část okna je věnována zdrojovému textu programu. Nalevo od něj jsou zařazeny ikony usnadňující vaši orientaci. Příkaz, který se bude provádět, je označen žlutou šipkou. Aktuální řádek (kterého se bude týkat např. nastavení bodu přerušení) má šedý podklad.

Pod zdrojovým kódem vlevo vidíte informační okénko s hodnotami proměnných. Lze se v něm přepínat mezi vámi sledovanými výrazy (**Watch**), lokálními či globálními proměnnými a dalšími možnostmi. Napravo od něj se standardně zobrazuje výstup programu, můžete ale místo něj nechat ukázat zásobník volání podprogramů.

Kontextové menu ve zdrojovém kódu programu kromě běžných editačních možností zpřístupňuje i několik příkazů pro ladění. Můžete si například přidat proměnnou ke sledovaným – přesuňte na ni myš, pravým tlačítkem otevřete kontextové menu a zvolte **Debug / Add to Watch**. Můžete zde také přidat, odebrat či upravit bod přerušení. Touto cestou ale obvykle půjdete, jen když u něj chcete nastavit podmínku. Nejjednodušší cestou k vytvoření a zrušení bodu přerušení je jednoduše kliknout myší nalevo od čísla příslušného řádku. Objeví se červené kolečko a bod přerušení je nastaven. Stejným způsobem jej lze i zrušit.

Akční příkazy zpravidla zadáváte prostřednictvím lišty nástrojů u spodního okraje okna nebo rovnou horkými klávesami. Najdete zde obvyklý sortiment: spuštění programu **F5**, provedení jednoho příkazu se vstupem do podprogramu **F11** i jeho překročením **F10** a ještě něco navíc.

5 Pole, lány, seznamy a seznamky

Doposud jsem se zabýval pouze jednoduchými proměnnými. Dala se do nich uložit vždy jen jediná hodnota – číslo nebo řetězec znaků. Často však pracujete s celou skupinou navzájem souvisejících údajů. Například byste si rádi někam uložili počty prodaných exemplářů vašeho skvělého programu (pochopitelně napsaného v Perlu) v jednotlivých měsících. Jistě to lze udělat skupinou proměnných:

```
$prodej_leden = 25;
$prodej_unor = 186;    #úspěch!
      ⋮
```

Ovšem to byste museli být blázni. Daleko elegantnější je zavést si jednu proměnnou – pole *@prodej* – a jednotlivé měsíce v něm vyznačovat indexy:

```
$prodej[1] = 25;
$prodej[2] = 186;    #úspěch!
      ⋮
```

Tím si zachováte možnost přístupu k údajům za jednotlivé měsíce (prostřednictvím indexů) a navíc získáte tyto výhody:

1. Můžete s polem pracovat jako s celkem – například je jediným přiřazovacím příkazem zkopírovat do jiného pole.
2. Pole lze automaticky zpracovávat – například projít indexem od 1 do 12, sečíst tyto položky a vypočítat tak celoroční prodej.
3. Téměř neomezený počet položek – kdybyste měli pomocí individuálních proměnných realizovat skladovou evidenci pro 10 000 různých druhů zboží, psali byste ten program roky…

5.1 Pole v Perlu

Z předchozích řádků už jste si mohli udělat nejzákladnější představu o tom, jak se v Perlu pracuje s polem. Pokud je používáte jako celek, předřazuje se jeho identifikátoru znak „@“. Tím se liší od běžných proměnných, které jsou zahájeny dolarem.

Zajímavostí je, že jmenné prostory proměnných a polí jsou navzájem nezávislé. To znamená, že klidně můžete mít v jednom programu proměnnou *$pom* a pole *@pom*, aniž by spolu tyto dva prvky měly co společného nebo se nějak ovlivňovaly. Je to zkrátka pouhá shoda jmen, která nic neznamená.

Nedělejte to. Budete-li používat stejnojmenné identifikátory, které se liší jen úvodním znakem, a přitom představují zcela rozdílné proměnné, zaděláváte si na nepříjemné chyby. Vřele doporučuji, abyste ve svých programech používali jednoznačné identifikátory.

Jako indexy pro pole používá Perl zásadně celá čísla. První prvek má vždy index 0. Pokud pole *@pokus* obsahuje deset prvků, budou to následující:

$pokus*[0], *$pokus[1], … *$pokus*[9]

Možná jste se právě zaradovali, že jste mne přistihli při překlepu. Před chvilkou jsem tvrdil, že se před identifikátorem pole píše „*@*“ a najednou tu mám *$pole*[0]. Ne, že bych nedělal překlepy, ale tentokrát jsem nechybil. *@* se používá jen v případech, kdy pracujete s polem jako s celkem. Pokud používáte jeho jednotlivé prvky, zapisujete je ve tvaru:

$»pole«[*»index«*]

Je to proto, že pole je v podstatě organizovaná skupina jednoduchých proměnných. Jakmile prostřednictvím indexu vyberete jednu z nich, přistupujete vlastně k běžné proměnné, a proto se používá obvyklý dolar. Takže příkaz:

@dva = *@raz*;

vezme pole *@raz* a naráz je celé přiřadí do pole *@dva*. Naproti tomu:

$dva*[3] = *$raz[3];

přiřadí pouze čtvrtý prvek pole *@raz* (indexy začínají nulou, takže čtvrtý prvek má index 3) do čtvrtého prvku pole *@dva*.

Indexy mohou být i záporné, v tom případě se počítají od konce. Například pomocí *$raz*[–1] získáte poslední prvek pole *@raz*.

Příklad: Vraťme se k motivačnímu příkladu ze začátku kapitoly. Řekněme, že bych skutečně měl pole *@prodej*, ve kterém jsou s indexy 1 až 12 uloženy dílčí prodeje v jednotlivých měsících. Celkový počet prodaných kusů za rok by vypsal následující programový úsek:

prodej.pl

```
my $celkem = 0;
my $mesic = 1;
while ( $mesic <= 12 ) {
    $celkem += $prodej[$mesic];
    $mesic++;
}
print "Prodej celkem: $celkem\n";
```

Všimněte si jisté neefektivity. Indexy pole začínají od nuly, zatímco já je využívám až od čísla 1. První prvek je zhola zbytečný. Pokud bych se tomuto lehkému plýtvání chtěl vyhnout, mohl bych od čísla měsíce odečítat jedničku. Leden by vystupoval pod indexem 0 a prosinec jako 11. To je však podstatně méně intuitivní. ■

Pole nemusíte nijak deklarovat, automaticky se založí při svém prvním použití. Zrovna tak jeho jednotlivé položky vznikají, když jim poprvé přiřadíte hodnotu. Ovšem stejně jako u skalárních proměnných, je slušností i pole deklarovat pomocí **my**.

Perl vám umožňuje přiřazovat i celé skupiny hodnot naráz. Slouží k tomu tak zvané *seznamy*, kdy jednotlivé hodnoty oddělíte čárkami a celý seznam uzavřete do kulatých závorek. Například:

```
my @pole = ( 35, "nazdar", -9 );
```

je ekvivalentní čtveřici příkazů:

```
my @pole = ();
$pole[0] = 35;
$pole[1] = "nazdar";
$pole[2] = -9;
```

Zároveň jsem tím demonstroval, že do jednoho pole můžete ukládat hodnoty různých typů (což je v konzervativnějších programovacích jazycích zakázáno).

Docela často se vyskytují seznamy zahrnující všechny hodnoty v určitém rozmezí. Pro jejich snadnou tvorbu Perl nabízí operátor řada, který se zapisuje v tvaru *»od«..»do«*. Naplnění pole hodnotami 1 až 100 zařídí:

```
my @pole = ( 1 .. 100 );
```

Pokud je počáteční hodnota větší než koncová, bude výsledkem prázdný seznam. Operátor se dá používat i pro znaky a řetězce, pak po „z“ následuje „aa“ atd. Doporučuji to nepřehánět a držet se jednoduchých situací:

```
my @pismena = ( 'a'..'z', 'A'..'Z' );
```

Při inicializaci polí se může hodit také opakovací operátor **x**, který kromě řetězců znaků dovede opakovat i seznamy. Je-li jeho prvním operandem seznam, zopakuje jej uvedený počet krát a vše spojí do jednoho výsledného seznamu. Díky tomu lze snadno založit pole a vytvořit mu tisíc prvků naplněných nulami:

```
my @pole = ( 0 ) x 1000;
```

Nezapomeňte na závorky, bez nich by nulu vzal jako řetězec. Vzniklo by pole s jedním prvkem obsahujícím řetězec s tisíci nulami.

Dost běžnou konstrukcí je pole řetězců. Abyste se při jeho inicializaci samými uvozovkami neupsali, dává Perl k dispozici funkci **qw** (quote word). Svůj parametr interpretuje jako řetězec, který prázdná místa (mezery, tabulátory, konce řádků) rozdělí na seznam řetězců. Ten je výstupem funkce. Následující dva přiřazovací příkazy jsou proto zcela ekvivalentní:

```
@pole = ("raz", "dva", "tři", "čtyři");
@pole = qw(raz dva tři čtyři);
```

Cvičení 5.1: Napište program, který načte ze standardního vstupu číslo od 1 do 12 a na oplátku vypíše jméno měsíce s daným číslem. Rozmyslete si, jak byste tuto úlohu řešili bez použití pole a jak s ním. Porovnejte vlastnosti obou variant. ■

Mezi pokročilejší prvky patří výřez pole (anglicky array slice). Jedná se o konstrukci, kdy z určitého pole vyberete pouze některé prvky. Ty můžete zadávat buď jednotlivými indexy oddělenými čárkami nebo celým rozmezím od..do. Horní hranice rozmezí je od dolní oddělena dvěma tečkami. Jelikož výřez představuje opět skupinu hodnot, píše se před identifikátorem znak „@“:

```
@pole[3,5]    znamená    ( $pole[3], $pole[5] )
@pole[3..5]   znamená    ( $pole[3], $pole[4], $pole[5])
```

Příklad: Příkaz:

```
@pole = @lan[5..8];
```

je ekvivalentní:

```
@pole = ( $lan[5], $lan[6], $lan[7], $lan[8] );
```

a také:

```
@pole = ();
$pole[0] = $lan[5];
$pole[1] = $lan[6];
$pole[2] = $lan[7];
$pole[3] = $lan[8];
```

A proč bych si vlastně nedovolil něco opravdu odvážného? Výřez se může nacházet i na *levé* straně přiřazovacího příkazu (zrovna tak, jako seznam). Pak se jednotlivé položky seznamu vpravo postupně přiřazují odpovídajícím položkám seznamu vlevo. Takže:

```
@pole[2,3] = @lan[9,10];
```

udělá totéž, co:

```
$pole[2] = $lan[9];
$pole[3] = $lan[10];
```

Je-li seznam na pravé straně přiřazovacího příkazu delší, než ten vlevo, jsou přebytečné hodnoty ignorovány. Pokud je naopak kratší, budou mít přebývající prvky levého seznamu nedefinovanou hodnotu (jako kdyby jim nikdy nikdo nic nepřiřadil). ■

Předchozí příklady mohly vyvolat dojem, že přiřazování seznamu hodnot je ekvivalentní skupině příkazů přiřazujících jednotlivé hodnoty. Ve většině případů to skutečně platí, existují však výjimky. Je třeba mít na paměti, že přiřazení seznamu je jediným přiřazovacím příkazem. Ten se zpracovává obvyklým postupem – vyhodnotí se pravá strana, pak se vyhodnotí levá strana a na závěr se provede vlastní přiřazení. To znamená, že jednotlivá dílčí přiřazení v rámci seznamu se navzájem neovlivňují, protože všechny přiřazované hodnoty byly stanoveny již na začátku.

Cvičení 5.2: Řekněme, že pole *@test* bylo inicializováno příkazem:

```
@test = ( 0, 1, 2, 3, 4, 5 );
```

Jaké budou hodnoty jeho prvků pro provedení příkazů:

```
@test[0,1] = @test[1,0];
$i = 2;
@test[$i..$i+2] = @test[$i+1..$i+3];
```

■

5.2 Cykly for a foreach

Se seznamy hodnot jsou těsně svázány cykly **for** a **foreach**. Nejčastěji bývají používány právě k procházení seznamů a **foreach** se ani k ničemu jinému nehodí. Jeho tvar je prostý:

foreach *»proměnná«* (*»seznam«*)
  »blok«

Funguje takto: do *»proměnné«* přiřadí první hodnotu ze *»seznamu«* a provede *»blok«*. Poté do *»proměnné«* přiřadí druhou hodnotu a opět provede blok. Tak pokračuje, dokud nedojde na konec *»seznamu«*.

Příklad: V poli *@vyroba* jsou s indexy 1 až 12 uloženy objemy výroby v jednotlivých měsících. Následující úsek programu je přehledně vypíše a přidá informaci o celkové výrobě:

vyroba1.pl

```
my $celkem = 0;
foreach my $mesic ( 1..12 ) {
    print "Výroba v měsíci $mesic: $vyroba[$mesic]\n";
    $celkem += $vyroba[$mesic];
}
print "========================\n";
print "Výroba celkem: $celkem\n";
```

Pokud by mne zajímal jen celkový údaj, vystačil bych s jednodušší konstrukcí:

vyroba2.pl

```
my $celkem = 0;
foreach my $vyrobeno ( @vyroba ) {
    $celkem += $vyrobeno;
}
print "Výroba celkem: $celkem\n";
```

Situace, kdy roli seznamu v cyklu **foreach** hraje pole, je celkem obvyklá. V takovém případě je seznam tvořen jednotlivými položkami pole. Proměnné *$vyrobeno* jsou tudíž postupně přiřazovány objemy výroby v jednotlivých měsících. ■

foreach je jednoduchý, šikovný, ale jednoúčelový nástroj. Cyklus **for** nabízí o něco větší pružnost. Zapisuje se takto:

for (*»inicializace«*; *»podmínka pokračování«*; *»přírůstek«*)
  »blok«

Při vstupu do cyklu se provede *»inicializace«*. Pokud platí *»podmínka pokračování«*, vykoná se *»blok«* a po něm následuje *»přírůstek«*. Zpracování pokračuje opětovným vyhodnocením *»podmínky pokračování«* a dokud platí, celý cyklus se opakuje.

Příklad: Typickou aplikací cyklu **for** je zpracování pole – průchod všemi jeho položkami a provedení určitých akcí pro každou z nich. V tomto směru je konkurencí pro **foreach**. Zkusím s použitím cyklu **for** vypočítat průměrnou výrobu:

vyroba3.pl

```
my $celkem = 0;
for ( my $mesic = 1; $mesic <= 12; $mesic++ ) {
    $celkem += $vyroba[$mesic];
}
print "Průměrná výroba: ", $celkem/12, "\n";
```

Cyklus **for** zde vystupuje ve své nejklasičtější roli. Celý jeho běh závisí na řídicí proměnné (zde *$mesic*). Té je na počátku přiřazena vstupní hodnota (1), která se při každé obrátce zvětší o jedničku a porovná s hodnotou cílovou (12). ■

Dlužno přiznat, že **for** cyklus je vlastně zbytečný. Stejného efektu dosáhnete pomocí **while**:

```
»inicializace«
while ( »podmínka pokračování« ) {
      »blok«
} continue {
      »přírůstek«
}
```

Nicméně je kompaktnější a programátoři jsou na něj zvyklí z jiných jazyků, kde bývá standardní konstrukcí pro práci s polem. V tomto směru mu v Perlu ostře konkuruje **foreach**, takže se s ním setkáte podstatně vzácněji, než jinde. Zcela v duchu mnohotvárnosti jazyka můžete **for** používat místo **foreach**:

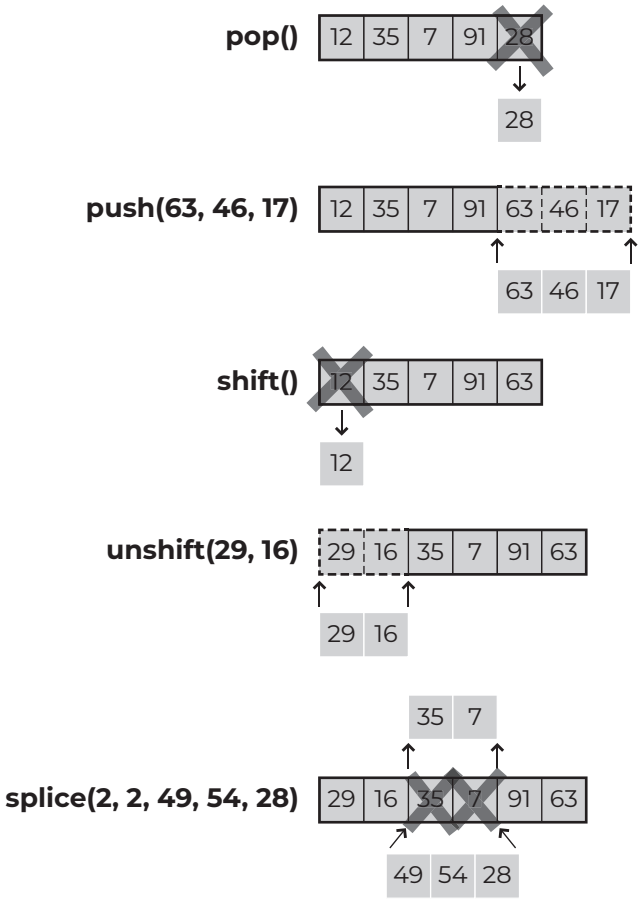
```
for »proměnná« ( »seznam« )
   »blok«
```

Vypadá a chová se stejně, jen klíčové slovo je jiné. Nicméně připadá mi vhodnější ty dva rozlišovat, pro průchod seznamem hodnot se držet **foreach** a pro cyklus s podmínkou a přírůstkem **for**.

Cvičení 5.3: Zjistěte v poli *@vyroba* měsíc, ve kterém bylo dosaženo největšího objemu výroby. ■

5.3 Funkce pro pole a seznamy

Funkce, které Perl nabízí pro práci s polem, se zabývají přidáváním a odebíráním jednotlivých prvků. Tvoří je dvě dvojice (**push**, **pop** a **shift**, **unshift**) a specialitka navíc (**splice**). Jejich činnost představuje obrázek 5.1.



Obrázek 5.1: Funkce pro pole

pop(*pole*) je velmi jednoduchá funkce, která odstraní z pole jeho poslední prvek a vydá jej jako svůj výsledek. Pole se odpovídajícím způsobem zkrátí.

push(*pole*, *seznam*) představuje protiváhu předchozí funkce. Hodnoty, které dostane v seznamu, naopak připojí na konec pole.

Analogicky se chovají **shift**(*pole*) a **unshift**(*pole*, *seznam*). Rozdíl je pouze v tom, že tyto funkce pracují se začátkem pole namísto s koncem. Při jejich použití se mění indexy všech prvků v poli (odeberete-li první prvek, indexy zbývajících o jedničku poklesnou).

Nejkomplikovanější je funkce **splice**, která může mít až čtyři parametry: **splice**(*pole*, *začátek*, *kolik*, *seznam*). Říká: z uvedeného *pole* bude odstraněno *kolik* položek počínaje indexem *začátek*. Je-li uveden *seznam*, vloží se jeho prvky do pole místo odstraněných položek. Chybí-li *kolik* položek se má odstranit, budou zlikvidovány všechny od pozice *začátek* až do konce *pole*. Jako svůj výsledek funkce vydá seznam odstraněných hodnot.

Příklad: Předvedu skupinu příkazů a jejich účinky. V levé části najdete úsek programu. Napravo je vždy uvedena hodnota pole *@test* po provedení příkazu vlevo:

my *@test* = (10, 15, 30);	(10, 15, 30)
pop(*@test*);	(10, 15)
push(*@test*, 23, 71);	(10, 15, 23, 71)
shift(*@test*);	(15, 23, 71)
unshift(*@test*, 48, 6);	(48, 6, 15, 23, 71)
splice(*@test*, 1, 3, 88, 99);	(48, 88, 99, 71)

■

Cvičení 5.4: Myslím, že bych vám už mohl dát něco těžšího. Postfixový zápis zvaný též polská reverzní notace je matematická podivnost, kdy se operátor píše až za operandy. Takže místo „10 + 5“ se píše „10 5 +“. Pro člověka je to nepřehledné, ale snadno se to implementuje a navíc nepotřebujete závorky – případné priority se dají vyjádřit pomocí pořadí.

Tento zápis používaly některé kalkulačky. Pokuste se jednu vytvořit v Perlu. Uživatel bude zadávat čísla a operátory, kalkulačka bude vypisovat výsledky prováděných operací. Postačí základní čtveřice: sčítání, odečítání, násobení a dělení. ■

Nejběžněji používané funkce pro práci se seznamy hodnot slouží ke změně pořadí. **reverse**(*seznam*) vydá jako svůj výsledek seznam obsahující stejné hodnoty, ovšem v obráceném pořadí.

Náročnější **sort**(*seznam*) vám vrátí seznam hodnot ze vstupního *seznamu* uspořádaný od nejmenší k největší. Hodnoty se porovnávají jako řetězce znaků, čili lexikograficky. 5 je zde větší než 20, protože je větší jeho první číslice.

Příklad: V poli *@zaci* jsou uložena jména žáků ve třídě. Jejich abecedně uspořádaný seznam vám poskytne příkaz:

```
foreach my $jmeno ( sort @zaci ) {
  print "$jmeno\n";
}
```

■

Již podruhé narážím na problém s češtinou. Porovnávání standardně vychází z ASCII kódů jednotlivých znaků. Ten však produkuje nesmysly. Malá písmena jsou podle něj větší než velká (Praha < auto) a písmena s diakritickými znaky větší než bez nich (slon < čečetka).

Tento problém trvá již dlouho a přinejmenším ve světě Unixu pro něj existuje řešení. Nazývá se locales a jedná se o kompletní popis vlastností jazyka. Je v něm uloženo, jaké znaky připouští a jaké je jejich uspořádání, jak se nazývají jednotlivé měsíce atd. atp. Podrobnější popis do této knihy nepatří, ale jistě vás potěším sdělením, že pokud váš systém podporuje locales[1], máte vystaráno.

Ke štěstí vám stačí dva kručky. Jednak musíte sdělit systému, jaký je váš jazyk. To provedete nastavením proměnné prostředí `LANG` na hodnotu `cs_CZ`, doprovázenou používaným kódováním. Jelikož z tohoto nastavení těží řada programů, bývá v současných systémech nastavena na rozumnou výchozí hodnotu. Pokud ne, je vhodné ji inicializovat automaticky při startu interpretu příkazů. Například já jsem míval ve svém souboru *~/.zshrc* příkaz:

```
setenv LANG cs_CZ.UTF-8
```

Druhým krokem je aktivovat locales ve svém perlivém programu. To je zcela triviální – stačí na jeho začátek vložit příkaz:

```
use locale;
```

Od tohoto okamžiku bude veškeré porovnávání znaků a řetězců fungovat zcela v souladu s našimi zvyklostmi. Moc užitečná věcička – zvykněte si psát tento příkaz zcela automaticky na začátek všech svých programů.

Funkce **sort** je ještě daleko větší mazlíček, který se dokáže přizpůsobit vašemu přání. **sort** je vlastně jakýsi obecný mechanismus pro uspořádání seznamu podle velikosti. Vy mu jako první parametr můžete předložit kritérium pro posouzení, kdo z porovnávané dvojice je větší.

Můžete porovnávající kód buď vepsat přímo do volání **sort** nebo zde uvést jen název podprogramu, který rozhodne (o podprogramech se dočtete v kapitole 8 na straně 109). Dvojice porovnávaných hodnot zde vystupuje jako proměnné *$a* a *$b*. Výsledkem porovnání by měla být hodnota 1, pokud

1: Jako třeba můj Linux.

je *$a* větší, 0 pokud jsou si rovny a –1 jestliže je větší *$b*. Perl dokonce nabízí operátor, který porovná dvě čísla a vydá výsledek podle popsané konvence. Zapisuje se pomocí <=>. Seřazení seznamu čísel je tudíž zcela snadné:

sort {*$a*<=>*$b*} »*seznam*«

Kdybyste chtěli sestupné pořadí a nechtěli se obtěžovat s funkcí **reverse**, jednoduše prohoďte proměnné v porovnání:

sort {*$b*<=>*$a*} »*seznam*«

⚡ Váš kód (ani jako podprogram) nesmí hodnoty proměnných *$a* a *$b* měnit, jinak se dočkáte nepěkných efektů. Pokud je potřebujete modifikovat (např. z nich odstranit nežádoucí části), zkopírujte je do pomocných proměnných a dále pracujte na nich. Tuto techniku používá například podprogram *srovnej_prospech* v programu *trida* na straně 138.

5.4 Nauka o kontextech

Nebudu to dlouho skrývat: kontexty jsou nepříjemná věc. Způsobují, že se stejná konstrukce chová odlišně v závislosti na prostředí, v němž ji použijete. Každý výraz je totiž v Perlu vyhodnocován v určitém kontextu. Ten závisí na okolí výrazu.

Základní dva kontexty jsou *skalární*, kdy je očekávána jediná hodnota, a *seznamový*, kdy se předpokládá, že výsledkem bude seznam hodnot. Vezměme si jako příklad běžný přiřazovací příkaz. Pokud se na levé straně nachází jednoduchá proměnná, je pravá strana vyhodnocována ve skalárním kontextu. Je-li levou stranou přiřazovacího příkazu pole či jeho výřez, bude pravá strana zpracována v seznamovém kontextu.

Kontext si můžete i poručit. Klíčové slovo **scalar** před výrazem zajistí, že bude vyhodnocen ve skalárním kontextu. Naopak výraz uzavřený do uvozovek bude vyhodnocen v kontextu seznamovém.

Jednou z konstrukcí, jejichž chování se v závislosti na kontextu radikálně mění, je pole. Pokud je použijete v seznamovém kontextu, převede se na seznam hodnot svých prvků. Přiřazovací příkaz:

@leve = @prave

způsobí, že se pole *@prave* rozvine na seznam svých hodnot a ten se přiřadí do pole *@leve*. Pole použité ve skalárním kontextu vydá jako svou hodnotu počet prvků. Délku pole lze proto získat následovně:

$delka = @prave

Příklad: Obecný způsob pro zobrazení prvků v poli vypadá takto:

```
for ( my $i=0; $i<@pole; $i++ ) {
  print "$i ... $pole[$i]\n";
}
```

V podmínce pro pokračování cyklu se porovnává proměnná *$i* s polem *@pole*, což by jakémkoli normálním jazyce vyvolalo významné poklepávání na hlavu. V Perlu je to běžné. *@pole* je zde použito ve skalárním kontextu, takže se nahradí počtem svých prvků. Smyslem podmínky tedy je, zda proměnná *$i* už překročila index posledního prvku (který je o jedničku menší než počet prvků v poli). ■

Také chování seznamu hodnot silně závisí na kontextu. Jeho přirozeným prostředím je seznamový kontext – zde se od něj očekává očekávatelné: seznam jeho hodnot. Za pozornost stojí, že jednotlivé prvky seznamu jsou také vyhodnocovány v seznamovém kontextu. Pokud tedy do seznamu zařadíte pole či vnořený seznam, budou převedeny na seznam svých hodnot a ten se zařadí mezi okolní prvky. Důsledkem je, že tato vnořená konstrukce zcela ztratí svou identitu. Je jedno, zda použijete:

```
@pom = ( 3, 19, 15 );
@pole = ( "raz", ( "dva", ((@pom))) , "tři", "čtyři" );
```

nebo:

```
@pole = ( "raz", "dva", 3, 19, 15, "tři", "čtyři" )
```

Výsledek je v obou případech totožný.

Chování seznamu ve skalárním kontextu se bohužel liší od pole. Výsledkem zde není počet jeho prvků, ale hodnota posledního z nich.

Cvičení 5.5: Jaké budou hodnoty proměnných *$a* a *$b* po provedení následujících příkazů:

```
@pole = ( "červená", "žlutá", "zelená" );
$a = ( "červená", "žlutá", "zelená" );
$b = @pole;
```

■

I diamantový operátor mění své chování. Zatím jsem jej používal ve skalárním kontextu, kdy postupně vydává jeden řádek za druhým. Pokud jej použijete v seznamovém kontextu, načte rovnou celý zbytek vstupu a vydá jej jako výsledný seznam. Každý řádek v něm bude tvořit jeden prvek.

Příklad: Naprosto stupidní způsob pro zjištění počtu řádků v souboru představuje tento program:

```
my @soubor = <>;
my $radky = @soubor;
print "Počet řádků vstupu: $radky\n";
```

Sice funguje a zabere jen malý počet řádků, ale jeho paměťové nároky jsou obludné a zcela neadekvátní řešené úloze. ■

Stále znovu a znovu se setkávám s programátory, kteří se nesmyslně snaží načíst celý soubor do paměti a pak jej řádek po řádku zpracovávat. Ovšem v takovém případě stačí načíst vždy jen jediný řádek, ten zpracovat a načíst další. Je zhola zbytečné spotřebovat potenciálně značné množství paměti na data, ze kterých vás v daném okamžiku beztak zajímá jen zlomek. Situace, kdy skutečně potřebujete mít v paměti celý soubor najednou, jsou velice vzácné. Nevzpomínám si na žádný svůj program, který by načítal celé soubory naráz.

Kromě popsaných dvou základních kontextů existuje i několik speciálních. Například logický kontext je speciálním případem skalárního. Vzniká tam, kde se vyhodnocuje podmínka. Jeho cílem je vydat hodnotu pravda či nepravda. Výraz je nejprve vyhodnocen ve skalárním kontextu. Výsledná hodnota se pak posoudí podle pravidel, která jsem uvedl na straně 49.

Druhou specialitou je prázdný kontext – pokud použijete výraz někde, kde není požadována žádná výsledná hodnota. Při použití volby `-w` může vést k varování typu:

```
Useless use of a constant in void context at x.pl line 2.
```

A do třetice všeho speciálního: Vkládací kontext panuje uvnitř dvojitých uvozovek a v některých dalších případech. Dochází v něm ke vkládání hodnot proměnných a speciálních znaků prostřednictvím zpětného lomítka. Podrobnosti najdete v části o řetězcích (strana 41).

Část II

Přicházejí těžké váhy

Ve druhé části vás seznámím s těmi prvky jazyka, díky kterým jsem se do Perlu zamiloval. V první řadě to jsou regulární výrazy, které vám umožní provádět s texty takové věci, o jakých se vám ani nesnilo. Této problematice jsem věnoval značný prostor, a to z několika důvodů: je vysoce přínosná, není úplně jednoduchá a neomezuje se pouze na Perl. Znalost regulárních výrazů totiž můžete uplatnit i v řadě jiných programů pocházejících z prostředí Unixu. Jejich podoba se sice program od programu poněkud liší, ale mají rozumný společný základ.

Druhou lahůdkou jsou asociativní pole. V nich vám Perl přináší ďábelsky rychlý vyhledávací mechanismus. Následuje kapitola o podprogramech, jejich vzájemné komunikaci a dekompozici programu. Závěrečná kapitola se věnuje vstupům a výstupům. Dozvíte se zde něco o formátování výstupů a především o práci se soubory.

6 Regulární výrazy

Smyslem regulárních výrazů je vyhledávat určité části textu. Jejich prostřednictvím zapisujete tak zvané *vzory*. Perl je pak porovnává se zadaným textem a snaží se určit, které jeho partie odpovídají vzoru.

Hlavní kouzlo spočívá v tom, že tyto vzory jsou neskutečně silné a dokáží s textem pravé divy. Regulární výrazy nejsou vynálezem Perlu. V operačním systému Unix si jejich dobrodiní můžete užívat už nějakých čtyřicet let.

Ovšem Perl je lahodným způsobem spojil s běžným programovacím jazykem. Díky téhle dvojici budete po zpracovávaných textech klouzat s elegancí krasobruslaře. Tu si vykroutíte piruetku, tam poskočíte trojitého Rittbergera… Ale dost už poezie a vzhůru do práce.

6.1 Jednoduché vzory

Tím nejjednodušším vzorem je prostý znak – písmeno, číslice a podobně. Pokud jej použijete jako regulární výraz, Perl bude hledat, zda se daný znak vyskytuje ve zkoumaném textu.

Pravda, jeden znak je trochu málo. Naštěstí můžete regulární výrazy řetězit za sebe a Perl se pak snaží najít odpovídající zřetězení. A hned tu máme první aplikaci – regulární výraz v podmínce. Píše se ve tvaru:

```
»zkoumaný text« =~ /»regulární výraz«/
```

Říká „Porovnej zkoumaný text se vzorem daným regulárním výrazem. Pokud se dá najít část textu, která vyhovuje regulárnímu výrazu, je podmínka splněna. Jinak ne.“ Například následující podmíněný příkaz vypíše oslavné hlášení, pokud řetězec v proměnné *$radek* obsahuje „Pepa“:

```
if ( $radek =~ /Pepa/ ) {
   print "Je tam Pepa!\n";
}
```

Dvojznakem =~ se zapisuje tak zvaný *operátor srovnání*[1]. Má dva operandy. Levým je zpracovávaný či zkoumaný řetězec znaků. Pravý určuje operaci, která se s ním má provést (hledání už znáte, ale také tu může být nahrazování či transformace). Výsledná hodnota signalizuje úspěšnost

1: Nerad to dělám, ale zde se odchyluji od „oficiální“ terminologie českého překladu knihy [1]. Ta pro =~ používá termín vazebný operátor. Nemohu si pomoci, ale žádnou vazbu v jeho funkci nevidím. Snad jedině tu, která měla být uvalena na překladatele za nekvalitní práci.

či neúspěšnost srovnání. Existuje ještě doplňkový operátor !~, který se chová úplně stejně, ale jeho výsledná hodnota je negována. Přiznám se, že jej prakticky nepoužívám.

Vraťme se však od obalových technologií k vlastním regulárním výrazům. Zatím mnoho síly nepředvedly – jen prostý řetězec znaků. Při skutečném hledání je zpravidla potřeba projevit jistou benevolenci, ale na určitých věcech nekompromisně trvat. Kouzlo regulárních výrazů spočívá především v košatých možnostech pro vyjadřování, na čem nám záleží a na čem ne.

Řekněme, že budu hledat Pepu ve svých textech. Jedním z mých notorických neduhů je nedomačkávání klávesy **Shift**.

Cvičení 6.1: Najděte v knize alespoň jednu větu, která díky nedomačknutému **Shift** začíná malým písmenem. ■

Jelikož se znám, určitě bych na to při hledání pamatoval a chtěl bych Pepu hledat s malým i velkým počátečním písmenem. V regulárních výrazech k tomu slouží konstrukce:

[»*znaky*«]

Říká „Na tomto místě může stát libovolný z uvedených »*znaků*«.“ Čili regulárnímu výrazu [Pp]epa vyhoví „Pepa“ i „pepa“, ale nic jiného.

Pokud je znaků hodně a navíc spolu sousedí, můžete je nahradit intervalem. Například libovolnou číslici můžete zapsat jako [0123456789], ale také snadněji [0–9].

Cvičení 6.2: Napište regulární výraz, kterému vyhoví rodné číslo. ■

Nabídku znaků můžete dokonce negovat. Jestliže je prvním znakem v hranatých závorkách „^“, znamená to „zde může být jakýkoli znak kromě těch, které jsou uvedeny v těchto hranatých závorkách“. Například libovolný znak odlišný od číslice lze vyjádřit pomocí [^0–9].

Na negace si dávejte pozor. Perl je založen na ryze pozitivním myšlení (ostatně pochází z USA) a při srovnávání se pouze snaží vyhledat, zda zkoumaný řetězec obsahuje něco, co vyhovuje vámi zadanému vzoru. Ostatního si nevšímá. Chcete-li najít řetězce, které *neobsahují* Pepu, nebude fungovat třeba:

if (*$radek* =~ /[^Pp]epa/)

Ve skutečnosti říká: hledám podřetězec, jehož prvním znakem není „P“ ani „p“ a zbývající tři jsou „epa“. Takže pokud *$radek* bude obsahovat „Pepa vyšel z depa“, perfektně vyhoví díky slovu „depa“.

Chcete-li říci, že hledáte řetězec který neobsahuje určitý vzor, musíte negovat celé srovnání, nikoli jeho prvky. V našem případě by řešením bylo:

```
if ( $radek !~ /[Pp]epa/ )          nebo
if ( not $radek =~ /[Pp]epa/ )
```

Abychom měli práci ještě snazší, obsahuje Perl předvařené konstrukce pro nejběžnější kategorie znaků. Shrnuje je tabulka 6.1. Význam \w a \W se může mírně měnit. Použijete-li locales (viz strana 82), budou za alfanumerické považovány i znaky národní abecedy. V našem případě čárkovaná, háčkovaná, pletená a další písmena.

zápis	význam	odpovídá
\d	číslice	[0–9]
\D	nečíslice	[^0–9]
\w	alfanumerický znak	[a–zA–Z0–9_]
\W	nealfanumerický znak	[^a–zA–Z0–9_]
\s	prázdný znak	[␣\t\n\r\f]
\S	neprázdný znak	[^␣\t\n\r\f]

Tabulka 6.1: Kategorie znaků

Občas je vám úplně jedno, jaký znak se na dané pozici nachází. V takovém případě napište do regulárního výrazu tečku. Například vzoru „.ostel“ vyhoví „Postel“, „kostel“, „dostel“ a řada dalších. Jediný znak, který nevyhoví tečce, je konec řádku.

Ale co když potřebujete hledat právě některý ze speciálních znaků, jako je tečka či hranatá závorka? Prostě mu předřaďte zpětné lomítko. Například tři tečky byste hledali pomocí vzoru „\.\.\.“.

V Perlu obecně platí tato pravidla: Má-li znak speciální význam (např. [), vyřadíte jej předřazením zpětného lomítka (\[). U některých znaků se předřazením zpětného lomítka speciální význam naopak zapíná (např. \d). Pokud kombinace \»*znak*« nemá speciální význam, chová se stejně jako samotný »*znak*«. Příjemnou vlastností je, že zpětné lomítko následované nealfanumerickým znakem v Perlu nikdy nemá speciální význam.

Mezi speciální znaky se uvnitř vzoru řadí i lomítko, které zde slouží k zahájení a ukončení celého vzoru. Hledáte-li například HTML značku , použijte vzor /<\/div>/.

6.2 Opakování matka hledání

Zatím dovedete vytvářet jen vyhledávací vzory s pevně danou délkou. Často však nelze předem stanovit délku hledaného řetězce. Nevadí. Máme tu opakovací konstrukce. Říkají, že předchozí regulární výraz se může opakovat. Jejich souhrn uvádí tabulka 6.2.

zápis	význam
*	libovolný počet výskytů
+	alespoň jeden výskyt
?	nanejvýš jeden výskyt
{3,5}	alespoň 3 a nanejvýš 5 výskytů
{,10}	nanejvýš 10 výskytů
{2,}	alespoň 2 výskyty
{7}	přesně 7 výskytů

Tabulka 6.2: Opakování v regulárních výrazech

Například klasickou konstrukcí je „.*“. Označuje libovolný řetězec. Podrobněji rozepsáno: libovolný počet libovolných znaků. Alespoň jedno písmeno anglické abecedy bych napsal jako „[a–zA–Z]+“.

Příklad: Podívejme se na něco malinko složitějšího: zápis celého čísla. To začíná nepovinným znaménkem, za kterým následuje alespoň jedna číslice. Odpovídající regulární výraz zní:

```
[–+]?\d+
```

Všimněte si, že uvnitř hranatých závorek potřebuji uvést mínus čili pomlčku. Ta zde má speciální význam – vytváří intervaly. Proto je vhodné zvyknout si ji psát jako první. Perl tak pochopí, že se nejedná o interval, ale že jedním z povolených znaků je pomlčka/mínus. ■

Řekněme, že jsem si stanovil jako vyhledávací vzor libovolný řetězec začínající písmenem „o“ a končící „e“, čili „o.*e“. Když takový vzor vypustím na řetězec „Popokatepetl“, která jeho část mu vlastně vyhoví? Bude to P opokate petl nebo P opokatepe tl či Pop okatepe tl nebo jen Pop okate petl? Dokud jde o pouhé hledání, na odpovědi příliš nezáleží. Důležité je pouze to, zda existuje vyhovující část či nikoli. Jakmile však začnete regulární výrazy používat pro nahrazování, rázem bude velice důležité vědět *přesně*, která část odpovídá regulárnímu výrazu, a tudíž bude nahrazena. Odpověď je prostá:

Regulární výrazy jsou hladové.

To znamená, že se snaží pozřít co nejvíce ze zkoumaného řetězce. Hledání začíná vždy od jeho začátku. V našem případě by tudíž regulárnímu výrazu vyhovělo P opokatepe tl.

Podívejme se na mechaniku hledání podrobněji, protože je poměrně důležitá. Perl používá techniku zvanou backtracking. Regulární výraz zkoumá postupně po jeho jednotlivých prvcích. Každému se pokusí přidělit co největší kus zkoumaného řetězce a pak pokračuje k dalšímu prvku. Jestliže se podaří dospět až na konec regulárního výrazu, zaznamená vítězství. Pokud však některý z následujících prvků selže, vrátí se k předchozímu a zkusí jej zkrátit a zase vyrazí vpřed. Když se takto vrátí až na samý začátek regulárního výrazu a není už kde co ubírat, zkusí se přesunout na následující znak zkoumaného řetězce. U něj zahájí hledání znovu od začátku. Takto pokračuje, dokud buď neuspěje nebo neprojde celý zkoumaný řetězec.

V našem konkrétním případě má regulární výraz tři prvky: písmeno „o“, libovolný řetězec a písmeno „e“. Zahájí zkoumání na začátku řetězce „Popokatepetl“ a zjistí, že písmeno „P“ neodpovídá „o“. Jelikož nevyhověl hned první prvek regulárního výrazu, posune se ve zkoumaném řetězci o jeden znak doprava. Nyní tedy zkoumá písmeno „o“, které vyhovuje. Pokročí ke druhému prvku, kterým je libovolný řetězec. Zkusí mu přiřadit maximum – „pokatepetl“. Pokračuje třetím prvkem, písmenem „e“. Na ně však ve zkoumaném řetězci nic nezbylo. Proto se vrátí ke druhému a zkusí vyhovující řetězec o znak zkrátit (pokatepet). Tentokrát porovnává písmeno „e“ se znakem „l“. To zase nesedí, takže se znovu vrátí ke druhému prvku a zkrátí jej o další znak (pokatepe). Následuje neúspěšné srovnání „e“ a „t“, které vynutí další návrat a zkrácení libovolného řetězce na „pokatep“. Tentokrát již následující písmeno vyhovuje třetímu prvku regulárního výrazu. Jelikož se jednalo o prvek poslední, zaznamená celkový úspěch a jako vyhovující vezme řetězec „opokatepe“.

Pochopili? Tak schválně:

Cvičení 6.3: Na řetězec „Popokatepetl“ uplatním vzor „.*.*“. Jaká část řetězce bude odpovídat prvnímu „.*“ a jaká druhému? ■

Cvičení 6.4: Napište regulární výraz, kterému vyhoví řetězec znaků uzavřený do uvozovek. Na první pohled by se mohlo zdát, že řešením je:

```
".*"
```

Existují však případy, kdy tento regulární výraz funguje špatně. Přijdete na to, které to jsou případy a jak jej vylepšit, aby se choval korektně i za těchto okolností? ■

Skutečnost, že se prvky z tabulky 6.2 chovají hladově, se někdy hodí a jindy ne. Abyste se nemuseli trápit, nabízí Perl i jejich sytou variantu. Vyrobíte ji prostě – za kvantifikátor připíšete otazník. Například „.*?“ znamená libovolný řetězec, který se ale snaží spolykat co nejméně znaků. Při srov-

návání mu Perl vždy přiřadí nejprve prázdný řetězec a pokud se nepodaří vzor dokončit, postupně jej prodlužuje.

Cvičení 6.5: Vraťte se k předchozím dvěma cvičením a zkuste je přepracovat pro syté kvantifikátory. Jak by dopadlo srovnání řetězce „Popokatepetl“ se vzorem „.*?.*?“ a jak by se dala tato konstrukce využít pro hledání řetězce v uvozovkách? ■

6.3 Regulární Kámasútra čili polohy

Zatím jsme se zabývali pouze tím, *co* hledáme. Občas je však potřeba určit, *kde* se má dotyčný řetězec nacházet. K tomu slouží speciální znaky uvedené v tabulce 6.3.

zápis	význam
^	začátek řetězce
$	konec řetězce
\b	hranice slova (mezi \w a \W)
\B	mimo hranici slova

Tabulka 6.3: Určení polohy v regulárním výrazu

Všimněte si významného rozdílu. Zatímco dosavadní regulární výrazy odpovídaly určitým znakům či podřetězcům zkoumaného textu, konstrukce z tabulky 6.3 označují pozice v textu. Lze to chápat tak, že jim vyhovující místa leží *mezi* znaky zkoumaného řetězce.

Začátek a konec je, myslím, jasný. Hranice slova (\b) leží všude tam, kde se z jedné strany nachází alfanumerický znak a z druhé strany nealfanumerický či začátek/konec řádku. Její pomocí můžete například vyjádřit, že vás určitý řetězec zajímá, jen pokud tvoří samostatné slovo: „\b»*slovo*«\b“.

Příklad: Existují programy, které vynechávají nadbytečné prázdné řádky. Jako nadbytečné jsou přitom chápány tyto prázdné řádky:

1. na začátku souboru,
2. pokud následují za prázdným řádkem.

Druhé pravidlo vlastně říká, že libovolně dlouhá skupina po sobě jdoucích prázdných řádků bude zredukována na jeden. Napišme si takový program:

prazdne.pl

```
my $prazdny = 1;
while ( my $radek = <> ) {
    chomp($radek);
    if ( $radek =~ /^\s*$/ ) {
        if ( not $prazdny ) {
            print "$radek\n";
            $prazdny = 1;
        }
    } else {
        print "$radek\n";
        $prazdny = 0;
    }
}
```

Program funguje tak, že si v proměnné *$prazdny* pamatuje, zda předchozí řádek byl prázdný. Narazí-li na prázdný řádek a tato proměnná je pravdivá, nevypíše jej. V opačném případě řádek opíše do svého výstupu a nastaví hodnotu *$prazdny* podle skutečnosti.

Všimněte si regulárního výrazu pro posuzování prázdného řádku. Ve skutečném nefalšovaném prázdném řádku následuje konec bezprostředně za začátkem (protože v něm nic není). Vzor by vlastně měl vypadat „^$“. Ovšem v praxi těžko poznáte skutečně prázdný řádek od řádku obsahujícího pět mezer. Proto jsem do regulárního výrazu vložil „\s*“ a připustil tak řádek obsahující nanejvýš nějaké prázdné znaky. ■

Cvičení 6.6: Vezmu standardní program pro zpracování souboru po řádcích. **while** cyklus vždy načte jeden řádek, následně se pomocí **chomp** odstřihne znak konce řádku a přijde ke slovu podmíněný příkaz, který řádek porovná se vzorem. Rozhodněte, pro které řádky bude úspěšné srovnání s následujícími vzory:

1. /\d/
2. /^\d/
3. /^\s*\d/
4. /\d*/
5. /^\d+$/

■

6.4 Vyhledej a nahraď!

Posouzení, zda určitý řetězec vyhovuje danému vzoru či nikoli, je pouze jednou ze dvojice základních aplikací pro regulární výrazy. Tou druhou je nahrazování: regulárním výrazem stanovíte co hledáte a připojíte řetězec, kterým se ten nalezený má nahradit. Zapisujte ve tvaru:

$proměnná =~ **s**/*»vzor«*/*»nahrazující řetězec«*/;

Funkce se jmenuje **s** jako „substituce“. *»vzor«* je klasický regulární výraz, *»nahrazující řetězec«* se chová stejně jako řetězec ve dvojitých uvozovkách (včetně dosazování proměnných a interpretace speciálních znaků). Příkaz funguje tak, že obsah *$proměnné* srovná se *»vzorem«*. Je-li srovnání úspěšné, odstraní z proměnné řetězec vyhovující *»vzoru«* a na jeho místo vloží *»nahrazující řetězec«*. Výstupní hodnotou celé konstrukce je pravdivostní údaj signalizující úspěch. Pokud byl vzor nalezen a tudíž došlo k nahrazení, je výsledkem „pravda“, v opačném případě „nepravda“.

Nahrazuje se pouze první výskyt vzoru! Chcete-li nahradit všechny, použijte volbu /g (viz dále).

Příklad: Nahrazování lze použít třeba v programu, který spočítá slova ve vstupujícím souboru. Slovo lze snadno definovat jako neprázdnou posloupnost alfanumerických znaků (řečí regulárních výrazů \w+). Zpracování bude jako obvykle postupovat po řádcích. Dokud řádek není prázdný, opakují se tyto dva kroky:

1. Z jeho začátku se odstraní nealfanumerické znaky (jsou-li jaké).
2. Začíná-li alfanumerickým znakem, vymaže se slovo a započítá.

Nalezené řetězce zde likviduji, takže nahrazující řetězec je v obou případech prázdný. Výsledek vypadá takto:

pocetslv.pl

```
use locale;

my $pocetslov = 0;
while ( my $radek = <> ) {
    chomp($radek);
    while ( $radek ne "" ) {
        $radek =~ s/^\W*//;
        if ( $radek =~ s/^\w+// ) {
            $pocetslov++;
        }
    }
}
print "Počet slov je $pocetslov\n";
```

Použití locales (viz strana 82) je zde důležité. Jinak by program znaky s háčky, čárkami a dalšími přízdobami považoval za nealfanumerické a třeba „mašinku“ počítal jako dvě slova. ■

Také funkce **s** standardně používá k vymezení hledaného vzoru a nahrazujícího řetězce znak lomítko. Chcete-li je použít uvnitř, musíte mu předsadit zpětné lomítko a potlačit tak jeho roli oddělovače. A nebo prostě použít jiný znak. Obecný tvar nahrazující funkce **s** totiž zní:

s *»znak«* *»vzor«* *»znak«* *»náhrada«* *»znak«* *»volby«*

Čili bezprostředně za **s** můžete uvést libovolný znak a ten bude sloužit k vymezení jednotlivých částí. Stejný trik můžete použít u samotného hledání. Před regulární výraz pak musíte přidat **m**. Například místo:

if (*$soubor* =~ /^\/home\/satrapa/)

lze zcela rovnocenně psát:

if (*$soubor* =~ **m**#^/home/satrapa#)

Oblíbenými oddělovači jsou například znaky # či ;, které se ve vzorech vyskytují dost vzácně.

Už jsem naznačil, že za nahrazující řetězec můžete přihodit ještě volby. Jimi lze ovlivnit chování funkce **s**. Obdobné volby lze připojit i za hledání – ať už používáte funkci **m** nebo jen /…/. Nabídku těch nejčastějších najdete v tabulce 6.4.

i	nerozlišovat malé/velké znaky
g	vyhledat/nahradit všechny výskyty
x	uvolněná syntaxe

Tabulka 6.4: Volby pro nahrazování a vyhledávání

Volby /i a /g počítám nepotřebují vysvětlení, ale co je to ta uvolněná syntaxe? Jejím cílem je umožnit přehlednější zápis regulárních výrazů. Pokud ji použijete, můžete regulární výraz psát podobně jako vlastní program: ignorují se prázdné znaky (pokud před nimi není zpětné lomítko), znak „#“ zahajuje komentář a celý regulární výraz můžete uzavřít do složených závorek. Například místo:

$radek =~ **s**/raz/dva/g;

můžete psát:

```
$radek =~ s{
                raz #tohle hledám!
          }
          {dva}gx;
```

Pozor! Uvolněná syntaxe se týká pouze regulárního výrazu. V nahrazujícím řetězci se nic nemění a stále záleží na každém znaku. U takto jednoduchého příkladu vypadá uvolněná syntaxe jako naprostý nesmysl. Při složitějších kouscích ale pravděpodobně změníte názor.

Cvičení 6.7: Svůj věčný zápas s klávesou **Shift** jsem už popsal. Řekněme, že mi (jako všem ješitům) bude záležet na tom, abych své jméno psal vždy správně s velkým počátečním písmenem[2]. Napište příkaz či příkazy, které v obsahu proměnné *$radek* opraví případné chyby ve velikosti písmen u mého (nebo vašeho, ať si také něco užijete) jména a příjmení. ■

6.5 Perl pamětníkem

Tahle kapitola by se také mohla jmenovat „Dejte mi závorku a pohnu zeměkoulí". Bude totiž pojednávat o nebetyčné úloze závorek v Perlu. A jak už to u velkých věcí bývá, začne zcela nenápadně.

Závorky lze v regulárních výrazech využít podobně jako v matematických vzorcích: k seskupování. Hodí se, pokud chcete povolit opakování nějakého většího celku. Například regulární výraz:

```
m{^\s*
   (
      [−+]?\d+    # celé číslo
      \s*         # případná mezera
   )*             # kolikrát chcete
$}x
```

říká „řetězec smí být tvořen jen celými čísly". Jak vidíte, povoluje opakování celé složité konstrukce – celého čísla a případných mezer za ním. Mimochodem, právě jste viděli přínos uvolněné syntaxe. Srovnejte si ji s klasickým tvarem:

```
/^\s*([−+]?\d+\s*)*$/
```

2: Ale možná je to chyba. V poslední době se stále častěji píše Internet (což je také vlastní jméno) s malým „i", protože prý se jedná o zcela běžnou věc. Mé jméno je pro mne pochopitelně naprosto všední záležitostí a podle tohoto pravidla bych se zřejmě měl psát pavel satrapa. Ovšem pro vás tak běžné není, takže jste povinni to číst jako Pavel Satrapa. Tenhle nový pravopis nebude nic pro prosťáčky…

Závorky lze také využít k ovlivnění priority. Pravda, v regulárních výrazech není mnoho prvků s různou prioritou, ale jeden přece: znak „|“, jehož významem je „nebo“. Má nižší prioritu než zřetězení, aby se dalo přirozeně psát:

```
ahoj|nazdar
```

a znamenalo to „hledám řetězec ahoj nebo řetězec nazdar“. Pomocí závorek snadno vymezíte odkud kam ono „nebo“ vlastně sahá. Například vzoru:

```
(1|jedna) lentilka
```

vyhoví řetězec „1 lentilka“ nebo „jedna lentilka“. Kdybych nepoužil závorky, vyhovoval by buď řetězec „1“ nebo „jedna lentilka“.

Hlavní přínos závorek však spočívá v tom, že obohacují regulární výrazy o paměť. Perl si zapamatuje řetězec, který vyhověl regulárnímu výrazu uzavřenému do závorek. Pro identifikaci těchto řetězců slouží pořadová čísla – prvnímu páru závorek odpovídá jednička, druhému dvojka apod. Vnořené závorky nijak nevadí. Rozhoduje pořadí levé závorky z páru. Chcete-li se na zapamatovaný řetězec odkázat, použijte \»*číslo*«.

Příklad: Klasické použití tohoto mechanismu. Podmínka:

```
$slovo =~ /^(.)(.).\2\1$/
```

bude splněna jen v případě, že proměnná *$slovo* obsahuje pětiznakový palindrom. To je slovo, které čteno normálně i pozpátku je stejné – například radar či rotor.

Podívejme se na způsob jejího vyhodnocení, pokud *$slovo* obsahuje „radar“. Regulární výraz začíná začátkem řádku, kterému vyhoví jen začátek řetězce. Následuje jeden libovolný znak. Odpovídá mu „r“ a jelikož je uzavřen do závorek, uloží se pod číslem 1. Další znak je opět libovolný a vyhovující „a“ se uloží pod číslem 2. Třetímu libovolnému znaku vyhoví „d“. Následující znak musí být stejný jako druhý uložený, čili „a“. To je v pořádku a odpovídá i další znak, který se musí shodovat s prvním uloženým „r“. Následně regulární výraz požaduje konec řetězce, na kterém se skutečně nacházíme. Takže velký úspěch, může se jít slavit. ■

Použití zapamatovaných prvků rovnou v regulárním výrazu je poměrně vzácné. Daleko častěji se uplatní v nahrazujícím řetězci funkce **s**. Každou chvilku se totiž ocitnete v situaci, kdy máte k dispozici potřebné informace, ovšem ve špatné podobě. Regulární výrazy lze využít k jejich přeměně.

Mimo regulární výraz se na zapamatované řetězce můžete odkazovat také jako na proměnné *$»číslo«*. Tento způsob je v páté verzi Perlu dokonce doporučen a jste nabádáni, abyste své \1 přepsa-

li na *$1*. Nedělejte to uvnitř vzoru! Proměnné jsou nadefinovány teprve po jeho *úspěšném* srovnání s řetězcem.

Příklad: Následující příklad je z praxe. Nasadili jsme nový DHCP server, který požadoval konfigurační soubor ve zcela jiném tvaru, než ten starý. Z původního:

```
cosi.kdesi.cz:  ha=004038967B6A:ip=147.230.11.8:
```

bylo potřeba udělat:

```
host cosi.kdesi.cz {
        hardware ethernet 00:40:38:96:7B:6A;
        fixed-address 147.230.11.8;
}
```

S regulárními výrazy nic obtížného. Poněkud nepříjemná je jen nutnost rozdělit Ethernet adresu na dvojice šestnáctkových číslic. Díky ní jsem musel psát závorek poněkud víc, než by se mi líbilo:

ethernet.pl

```
if ( $radek =~ m/(.*):\s* #jméno počítače
           ha=(..)(..)(..)(..)(..)(..):\s* #Ethernet
           ip=([^:]*):/x ) {
     print "host $1 {\n";
     print "  hardware ethernet $2:$3:$4:$5:$6:$7;\n";
     print "  fixed-address $8;\n";
     print "}\n";
}
```

■

A ještě třetí alternativa: pokud výsledek srovnání přiřadíte do proměnné typu pole, budou prvky tohoto pole naplněny hezky popořadě jednotlivými zapamatovanými řetězci. Například příkaz:

```
($prvni,$druhe) = $radek =~ m/\s*(\d+)\s+(\d+)\s*/;
```

najde v řetězci v proměnné *$radek* první dvě celá čísla (pro jednoduchost je beru bez znaménka) a přiřadí je do proměnných *$prvni* a *$druhe*.

Pokud je srovnání neúspěšné (v našem případě pokud řetězec neobsahuje dvě mezerami oddělená čísla), budou hodnoty přiřazované v poli nedefinovány. Výsledek proto vždy testujte.

Cvičení 6.8: Napište program, který přeorganizuje jmenný seznam. Jeho vstupem bude text, kde každý řádek obsahuje křestní jméno, alespoň jednu mezeru a příjmení. Například:

```
Jan Smutný
Alois Čvančara
Leoš Lesklý
```

Vyměňte pořadí jména a příjmení a vypište seznam uspořádaný abecedně podle příjmení jednotlivých osob. ■

⚡ Existují ještě čtyři speciální proměnné s paměťovým efektem. Do *$+* Perl automaticky uloží poslední řetězec vyhovující vzoru v závorkách, do *$&* řetězec vyhovující celému naposledy použitému regulárnímu výrazu, do *$`* část zkoumaného řetězce před ním a do *$'* část řetězce za ním. *Snažte se jim za každou cenu vyhýbat!* Jakmile některou z těchto proměnných použijete, výrazně se zpomalí zpracování *všech* regulárních výrazů v celém programu. Je to nepochopitelné, ale je to tak.

Na rozdíl od lidí můžete v Perlu jeho sklerózu ovládat. Pokud vám překáží, že si závorky zapamatovávají svůj obsah (chcete je použít čistě k seskupení či vyjádření priority), lze tomu snadno zabránit. Použijte místo obyčejných závorek konstrukci (:?*»regulární výraz«*) a Perl si vyhovující řetězec nezapamatuje.

6.6 Hromadná výroba

Někdy budete pracovat s daty, která jsou oddělována pevně danými oddělovači. Typickým příkladem je soubor */etc/passwd*, jehož položky jsou oddělovány dvojtečkami. V podobném formátu lze také exportovat data z tabulkových kalkulátorů či databází. Psát regulární výrazy typu:

```
([^:]*):([^:]*):…
```

je mimořádně otravné. Naštěstí je Larry Wall také líný a nabídl elegantnější řešení – funkci **split**. Její nejčastější použití je:

```
»pole« = split( /»vzor«/, »řetězec« )
```

»vzor« definuje oddělovač, kterým jsou v *»řetězci«* rozděleny jednotlivé hodnoty. Funkce *»řetězec«* rozkrájí v místech výskytu oddělovačů a jako svůj výsledek vydá seznam hodnot ležících mezi nimi. Častým oddělovačem je prázdné místo. Například po provedení:

```
@pole = split( /\s/, "raz dva tři čtyři" )
```

bude *$pole*[0] obsahovat „raz“, *$pole*[1] „dva“ a tak dále. Specialitou je prázdný vzor (//). Když jej použijete, bude řetězec rozdělen po jednotlivých znacích. Zadáte-li vzor, který se v řetězci vůbec nevyskytuje, vznikne jako výsledek jediná hodnota – celý řetězec.

Funkci můžete předat ještě třetí parametr. Udává maximální počet kousků, na které chcete řetězec rozdělit. Poslední část pak bude obsahovat celý zbytek řetězce, i když v něm lze najít další výskyty oddělovače.

Příklad: Příkaz:

```
( $a, $b ) = split( /\s/, "raz dva tři čtyři" )
```

uloží do proměnné *$a* hodnotu „raz“ a do *$b* „dva“. Naproti tomu:

```
( $a, $b ) = split( /\s/, "raz dva tři čtyři", 2 )
```

uloží do *$a* „raz“ a do *$b* „dva tři čtyři“. ■

Cvičení 6.9: Soubor */etc/passwd* obsahuje informace o uživatelích operačních systémů typu Unix. Každému je věnován jeden řádek ve tvaru:

uživatel:*heslo*:*UID*:*GID*:*jméno*:*domácí adresář*:*interpret*

Sestrojte program, jehož vstupem bude obsah souboru */etc/passwd* a výstupem bude seznam uživatelů ve tvaru:

jméno příjmení (*uživatelské jméno*)

uspořádaný abecedně podle příjmení uživatelů. Uživatelské jméno je první položkou na řádku v */etc/passwd*, jméno a příjmení tvoří pátou položku. Ignorujte systémové uživatele (jako je třeba *root*). Poznáte je podle toho, že v páté položce nemají buď nic nebo jen jedno slovo. ■

K funkci **split** existuje i funkce inverzní. Jmenuje se **join** a volá se takto:

join(»*výraz*«, »*seznam*«)

Jejím výsledkem je řetězec znaků, který vznikne tak, že se jednotlivé hodnoty »*seznamu*« spojí a mezi každé dvě se vždy vloží jeden »*výraz*«. Takže po provedení:

```
@pole = split( /:/, $radek );
$radek = join( ":", @pole );
```

bude mít proměnná *$radek* stejný obsah, jako před touto dvojicí příkazů.

Speciálním případem hromadného nahrazování znaků je operátor **tr**. V této kapitole vlastně nemá co dělat, protože nepoužívá regulární výrazy, ale jednoduché skupiny znaků. Píše se ve tvaru:

$proměnná =~ **tr**/»*nahrazované znaky*«/»*nahrazující znaky*«/

Prohledá řetězec uložený v »*proměnné*« a kdykoli v něm najde znak uvedený mezi nahrazovanými, vymění jej za znak, který je na stejné pozici v nahrazujících. Například **tr**/a–z/A–Z/ převede malá písmena anglické abecedy na velká.

Tak to bychom měli! Pokud propadáte pocitu, že se vám každou chvíli rozskočí hlava a potřebovali byste našroubovat novou, nepropadejte panice. Já jsem nesliboval, že regulární výrazy jsou lehké. Sliboval jsem, že jsou neuvěřitelně silné. Pokud se vám podaří s nimi skamarádit, dokážete obrovské věci. Chlapce od chlapů totiž nerozlišuje vojna, jak se snaží tvrdit zelení mužíčci, ale pochopení regulárních výrazů.

Jestli ještě nemáte dost, zkuste `man perlfaq6` nebo knihu [6].

7 Asociativní pole, česky hashe

Reklama:
Lojza a Pepa jsou kamarádi. Zatímco Lojza používá běžný programovací jazyk, Pepa dává přednost Perlu. Oba dostali stejnou úlohu: spočítat, kolik peněz se za co doma utratí. Lojza si navrhl datové struktury, ale to již Pepa psal základní cyklus a analýzu vstupu. Opravdu hodně času Lojza věnoval algoritmům pro vyhledávání řetězců znaků a jejich vkládání do pole. Během té doby Pepa pohodlně napsal přičítání dílčích hodnot i výstup výsledků. A ještě si stačil vyprat ponožky v pracím prášku… ale to už je jiná reklama.

A proč že byl Pepa tak úspěšný? Použil asociativní pole.

7.1 To chci také

Asociativní pole, pro které se i v české literatuře zhusta používá anglický termín „hash“, je speciální typ pole, v němž jako index slouží řetězec znaků. V praxi to znamená, že máte přímo mezi základními konstrukcemi jazyka šikovný a silný vyhledávací mechanismus. Místo názvu index se zpravidla používá pojem „klíč“.

Aby se asociativní pole odlišila od obyčejných, předsazuje se jejich proměnným znak „%“ místo „@“. Opět se jedná o zcela samostatný prostor proměnných, takže proměnná *$a*, pole *@a* a asociativní pole *%a* jsou tři zcela nezávislé objekty. Druhou syntaktickou odlišností je, že index (klíč) se zapisuje do složených závorek. Například:

```
$nakupy{”drogerie”} = 150;
```

uloží do asociativního pole *%nakupy* na pozici danou klíčem „drogerie“ číslo 150. Potřebujete-li inicializovat asociativní pole celou skupinou prvků, můžete na pravé straně přiřazovacího příkazu použít seznam a na levé pak cílové pole. Liché prvky seznamu budou interpretovány jako klíče a sudé jako hodnoty, které mají být přiřazeny jednotlivým klíčům. Nejprve se však všechny stávající prvky přiřazovaného pole vymažou. Díky tomu je ekvivalentní, zda napíšete:

```
%nakupy = ( ”potraviny”, 86, ”oděvy”, 1230 );
```

nebo:

```
%nakupy = ();       # vymazání všeho
$nakupy{”potraviny”} = 86;
$nakupy{”oděvy”} = 1230;
```

Aby byl seznam srozumitelnější, můžete mezi klíč a jemu příslušející hodnotu psát „=>“. Je to vlastně synonymum pro čárku, ale zápis je tak přehlednější:

```
%nakupy = ( ”potraviny” => 86, ”oděvy” => 1230 );
```

Je-li klíč tvořen jen písmeny, číslicemi a podtržítky a nezačíná číslicí, nemusíte kolem něj psát uvozovky. *$nakupy*{”elektro”} a *$nakupy*{elektro} je totéž, Perl is uvozovky doplní automaticky a interpetuje obsah složených závorek jako řetězec znaků. Nicméně je bezpečnější uvozovky psát, aby se zdůraznilo, že se jedná o řetězec.

Příklad: Jednou ze situací, kdy asociativní pole přijde k duhu, je kontrola klíčových slov. Vstupem programu bude textový datový soubor, kde jednotlivé řádky mají tvar:

```
»název položky«:  »hodnota«
```

Názvy dostupných položek jsou přitom pevně dány a chcete, aby program případnou odchylku od povolených názvů ohlásil jako chybu. Například se může jednat o vaši databázi kontaktů, kde jedinými povolenými položkami jsou „jméno“, „telefon“ a „e-mail“:

klice.pl

```
my %polozka_ok = ( ”jméno” => 1, ”telefon” => 1, ”e−mail” => 1 );

while ( my $radek = <> ) {
    chomp($radek);
# vynechám prázdné řádky
    if ( $radek =~ /^\s*$/ ) { next; }
# získám název položky a zkontroluji
    my ($polozka, $hodnota) = $radek =~ m/^([^:]*):\s*(.*)$/;
    if ( not $polozka_ok{$polozka} ) {
        print STDERR ”Chybný řádek: \”$radek\”\n”;
        next;
    }
# následuje zpracování řádku
# ...
}
```

Posouzení správnosti zajišťuje asociativní pole *%polozka_ok*. Na začátku do něj pro všechny povolené hodnoty uložím jedničku. Když potom použiji *$polozka_ok*{*řetězec*}, výsledkem pro některý z povolených řetězců bude 1 čili pravda. Je-li řetězec jiný, nebyla mu přiřazena žádná hodnota a výsledkem bude **undef**. To je podle pravidel Perlu nepravda, tudíž se vypíše chybové hlášení a program pokračuje zpracováním dalšího řádku.

Veškeré testování pro vás obstará jediný přístup do asociativního pole. Jak elegantní! Jak snadné! ■

7.2 Operace a funkce

Těžko vás překvapím sdělením, že asociativní pole lze navzájem přiřazovat. Naproti tomu jejich použití ve skalárním kontextu nevydá počet prvků, jak byste možná očekávali. Výsledkem je jakási interní hodnota, která je v praxi použitelná jen k testování, zda je asociativní pole neprázdné.

Základní nabídka funkcí, které pracují s asociativním polem, je obsažena v tabulce 7.1. K práci s individuálními prvky slouží **exists** a **delete**. První zjistí, zda je dotyčný klíč obsažen v poli. V minulém příkladu jsem použil jednodušší test – prosté použití *$pole{klíč}*. Pokud však klíči byla přiřazena hodnota, kterou Perl interpretuje jako nepravdivou, tento jednoduchý test selže.

exists(*$pole{klíč}*)	obsahuje pole daný klíč?
delete(*$pole{klíč}*)	vymaže klíč z pole, vydá jeho hodnotu
keys(*%pole*)	seznam všech klíčů v poli
values(*%pole*)	seznam všech hodnot v poli
each(*%pole*)	další dvojice (klíč,hodnota) z pole

Tabulka 7.1: Funkce pro asociativní pole

Prostřednictvím **delete** můžete z asociativního pole vymazat klíč a jemu příslušející hodnotu. Ta tvoří výstup funkce **delete**.

Zbývající trojice funkcí slouží nejčastěji k procházení všech hodnot uložených v poli. Osobně mám v největší oblibě funkci **keys**, která vydá seznam všech klíčů. Jejich pořadí je dáno vnitřním uspořádáním pole, takže je de facto náhodné. Chcete-li je upravit do rozumnější podoby, prožeňte výsledek **keys** funkcí **sort**.

Funkce **values** vydá seznam všech hodnot, které byly do pole uloženy. Jejich pořadí je stejné, jako u **keys**. Nejpodivnější chování vykazuje funkce **each**. Jejím výsledkem je seznam se dvěma prvky, který obsahuje první klíč z asociativního pole a jemu příslušející hodnotu. Funkce si pamatuje, kde minule skončila, a při následujícím volání vydá další prvek z pole. Pořadí procházení je opět stejné jako u **keys**. Když dorazí na konec, vydá jako výsledek prázdný seznam. Při příštím opakování pak začne zase od začátku.

Pomocí funkce **each** lze vypsat obsah asociativního pole *%pokus* třeba takto:

```
while ( ($klic,$hodnota) = each(%pokus) ) {
    print "$klic => $hodnota\n";
}
```

Příklad: Vrátím se k reklamnímu šotu ze začátku kapitoly a zkusím vyřešit Pepovu úlohu. Vstupem programu budou údaje o nákupech ve tvaru:

```
potraviny 153
oděvy 492
potraviny 49
```

Cílem je spočítat celkovou útratu za jednotlivé druhy zboží (zde potraviny 202, oděvy 492) a vypsat ji v přehledné formě.

financ.pl

```
my %celkem = ();
while (my $radek = <> ) {
    chomp($radek);
    if ( my ($zbozi,$cena) = $radek =~ m/^(\S+)\s+(\d+(\.\d*)?)$/ ) {
        $celkem{$zbozi} += $cena;
    } else {
        print "Chybný řádek '$radek'\n";
    }
}

foreach my $zbozi ( sort keys(%celkem) ) {
    print "$zbozi ... $celkem{$zbozi}\n";
}
```

Datovým centrem programu je asociativní pole *%celkem*, do kterého sčítám celkové ceny za jednotlivé druhy nákupů. V obvyklém cyklu přes jednotlivé řádky vždy rozložím řádek na název zboží a jeho cenu, název použiji jako klíč a přičtu k němu v poli *%celkem* aktuální cenu. Všimněte si, že regulární výraz pro číslo připouští i desetinnou část (desetinná tečka následovaná libovolným počtem číslic, to celé nepovinné).

Závěrečný cyklus je klasickou ukázkou výpisu hodnot z asociativního pole. Prostřednictvím **keys** získám seznam klíčů, abecedně jej uspořádám a probírám cyklem **foreach**. Jelikož mám k dispozici klíč (zde *$zbozi*), snadno se dostanu k jemu odpovídající hodnotě (*$celkem*{*$zbozi*}). ■

Cvičení 7.1: Napište program, který spočítá výskyty jednotlivých slov textu. Vstupem bude analyzovaný text. Výstup má tvořit seznam slov, která se v textu vyskytují. U každého bude uvedeno, kolikrát je v textu obsaženo. ■

8 Podprogramy

Vraťme se do mlhou zahalených prvopočátků výpočetní techniky, kdy stateční mužové, ošlehaní výbuchy elektronek, kladivem a pájkou sestavovali první programy. Již tehdy si všimli, že některé části programu se používají opakovaně a bylo by záhodno, aby se nemusely pokaždé znovu psát. Zrodila se myšlenka podprogramu.

Jedná se o logicky ucelenou část programu, která řeší určitou dílčí část celého problému. Je zapsána na jednom místě a kdykoli ji potřebujete použít, pouze ji zavoláte. Typickou ukázkou jsou standardní funkce Perlu, jako je například **sqrt** pro výpočet druhé odmocniny. Kdesi v útrobách překladače leží předpis, který říká, jak se vlastně ta druhá odmocnina počítá. Kdykoli ve svém programu zavoláte **sqrt**, bude tento předpis proveden a jako data dostane vámi zadané údaje.

Z dosavadního dramatického líčení jasně vysvítají dvě důležité přednosti používání podprogramů: ušetří práci (dotyčnou část kódu píšete jen jednou, bez ohledu na počet jejích použití) a zajišťují konzistenci. Nestane se vám, že by se jeden exemplář stejné funkce choval jinak než jiný, protože jste se v tom druhém překlepli. Tyto dvě výhody stály u kolébky podprogramů.

Z pohádek jistě víte, že sudičky bývají tři. Ta třetí se přidala později, ale o to větší význam jí v současnosti přikládáme. Je jí vyšší srozumitelnost programu. Podprogramy do něj zavádějí abstrakci, navíc abstrakci s libovolným počtem úrovní.

Voláte-li funkci **sqrt**, vlastně říkáte „Tady si spočítám druhou odmocninu z …“ S využitím vlastních podprogramů se můžete vyjadřovat ve stylu „teď vyhledám v databázi záznam XY“, „teď přesunu figurku na šachovnici“, „teď vyberu další tah“ a tak dále a tak podobně. Čtenář programu se pokaždé může rozhodnout: buď mu vyjádření typu „teď přesunu figurku na šachovnici“ stačí a nezajímají jej technické detaily (například proto, že dotyčný podprogram již zná) nebo může odbočit a prostudovat si, co vlastně daný podprogram dělá. Každý si tak může volit, do jaké úrovně detailů hodlá program zkoumat.

Jelikož žijeme v době monumentálních programů a mnohohlavých programátorských týmů, je právě na snadnou srozumitelnost a dobré strukturování programu kladen velký důraz. Podívejme se, jak se dělají.

8.1 Podprogramy v Perlu

Podprogram, stejně jako jiné prvky programu, musí mít své jméno. To je součástí jeho definice, která má tvar:

```
sub »jméno podprogramu« {
    »tělo podprogramu«
}
```

V podstatě se dá říci, že definice podprogramu je blok, kterému jste přidělili jméno. Prostřednictvím tohoto jména lze později podprogram zavolat.

Klasický způsob volání se zapisuje tak, že jménu podprogramu předřadíte znak „&“. Čili obvyklá zdejší konvence, kdy speciální počáteční znak určuje druh identifikátoru, který stojí za ním.

Později přibyla druhá alternativa, která se obejde bez úvodního „&“. Zato zde povinně musíte přidat seznam parametrů v kulatých závorkách (bude o nich řeč v části 8.3), i kdyby byl prázdný. Tento zápis odpovídá konvencím drtivé většiny ostatních programovacích jazyků. Zřejmě i proto se prosadil, je v současné době obvyklejší a budu se jej držet i já. Zvykněte si nedělat mezeru mezi jménem podprogramu a seznamem jeho parametrů, jinak se v některých případech může interpretovat špatně.

Příklad: Předvedu velmi primitivní nástroj pro pseudografické zobrazení hodnot. V poli *@vyroba* je 12 položek, obsahujících objem výroby v jednotlivých měsících. Mám je zobrazit ve formě primitivního textového grafu, kdy každému měsíci bude odpovídat jeden řádek a na něm bude tolik hvězdiček, kolik činila výše výroby (předpokládám, že je převedena na vhodné jednotky – např. tisíce kusů – aby se výsledek vešel na řádek). Ukázku výstupu vidíte na obrázku 8.1.

```
 1 | **************
 2 | ************
 3 | ***************
 4 | ******************
 5 | **************
 6 | ****************
 7 | *********
 8 | **********
 9 | ************
10 | ******************
11 | ****************
12 | **************
```

Obrázek 8.1: Ukázka „grafu“ výroby

Základem řešení bude podprogram *hvezdicky*, který zobrazí jeden řádek grafu – vypíše číslo měsíce a příslušný počet hvězdiček. V hlavním programu pouze zajistím, aby došlo k jeho vyvolání pro každý měsíc.

```
1   for ( $mesic=1; $mesic<=12; $mesic++ ) {                    graf.pl
2       hvezdicky();
3   }
4
5   sub hvezdicky {
6       if ( $mesic < 10 ) { print " "; }
7       print "$mesic | ";
8       for ( $i=int( $vyroba[$mesic]+0.5 ); $i>0; $i-- ) {
9           print "*";
10      }
11      print "\n";
12  }
```

Podle tradiční konvence by volání podprogramu na druhém řádku vypadalo:

```
2       &hvezdicky;
```

Poznamenejme, že výstup řady hvězdiček by se dal efektivněji realizovat pomocí operátoru „x“ pro opakování předchozího řetězce. Místo řádků 8 až 11 by podprogram *hvezdicky* obsahoval:

```
8       print "*" x int( $vyroba[$mesic]+0.5 ), "\n";
```

■

Ideovým otcem podprogramu je nepochybně Jára Cimrman, protože základní myšlenka odpovídá jeho slavnému kroku stranou. Procesor prostě pozastaví vykonávání hlavního programu a na chvilku si odskočí stranou, aby provedl podprogram. Jakmile s ním skončí, vrátí se zase zpět a pokračuje v provádění programu bezprostředně za místem, ze kterého si odskočil.

Například v předchozím příkladu vždy, když na řádku 2 dojde k vyvolání podprogramu *hvezdicky*, je přerušeno provádění hlavního programu a pokračuje se instrukcemi dotyčného podprogramu (řádky 5 až 12). Po jeho dokončení se pokračuje na řádku 2 za voláním podprogramu. V praxi to znamená, že bude provedena další obrátka hlavního cyklu programu.

Toto je jen nejjednodušší případ. V praxi často jeden podprogram volá druhý, který si zavolá třetí a tak dále, a tak podobně. Nicméně mechanismus stále zůstává stejný: odložit stávající práci, provést podprogram a po návratu pokračovat.

8.2 Lokální proměnné

V příkladu, který jsem právě předvedl, používá podprogram *hvezdicky* k řízení svého **for** cyklu globální proměnnou *$i*. Ostatně proměnné, o kterých jsem se dosud zmiňoval, byly z valné většiny globální. To znamená, že proměnná existuje na úrovni hlavního programu a je společná pro všechny jeho části. Kdykoli kdekoli napíšete *$i*, jedná se stále o jednu a tu samou proměnnou.

Nechci vás strašit, ale za tímhle rohem číhá ošklivý chlap s klackem. Jakmile začnete ve větší míře používat podprogramy, je jen otázkou času, kdy v globálních proměnných ztratíte přehled. Pak jeden podprogram nečekaně přepíše proměnnou jinému a výsledkem bude Velký Průšvih.

Proto je velmi záhodno, aby si podprogram pro své interní záležitosti používal své interní proměnné. Už dříve jsem naznačil, že k tomu slouží klíčové slovo **my**. Je na čase podívat se mu pořádně na zoubek. Obecně vypadá deklarace proměnných takto:

my (»*seznam proměnných*«)

Jakmile takto definujete proměnnou, stává se z ní proměnná lokální. To znamená, že existuje pouze uvnitř bloku (resp. souboru, pokud není uzavřena v žádném bloku), v němž byla definována. Z ostatních částí programu je nedostupná.

Je přípustné, aby lokální proměnná měla stejné jméno jako jiná proměnná globální. V takovém případě se jedná o dva zcela nezávislé objekty (v reálném životě o takové situaci říkáme „je to pouhá shoda jmen“). Od okamžiku své definice příkazem **my** však lokální proměnná zastiňuje globální. Až do konce bloku (souboru) bude globální proměnná nedostupná a kdykoli použijete tento identifikátor, bude interpretován jako jméno lokální proměnné.

Jak už víte, v rámci deklarace můžete proměnným rovnou přiřadit hodnoty. Například:

my (*$jmeno*, *$vek*) = ("Pepa", 33);

zavede lokální proměnné *$jmeno* a *$vek* a přiřadí první z nich hodnotu „Pepa“ a druhé 33.

Příklad: Sestrojím program, který sice nedělá mnoho užitečného, ale hezky ilustruje vlastnosti lokálních proměnných:

```
1   my $i = 1;
2   raz();
3   dva();
4   $i++;
5   print "$i\n";
6
```

```
7   sub raz {
8       my ( $i ) = 8;
9       print "$i\n";
10  }
11
12  sub dva {
13      $i++;
14      print "$i\n";
15  }
```

Předmětem vašeho zájmu budiž proměnná *$i*, přesněji řečeno proměnné *$i* a *$i*. Přiřazovací příkaz na prvním řádku zavedl globální proměnnou *$i* a přiřadil jí hodnotu 1. Následuje volání podprogramu *raz*.

Jeho tělo začíná deklarací lokální proměnné, která se shodou okolností také jmenuje *$i* (řádek 8). Nemá s globálním *$i* vůbec nic společného. Ovšem vzhledem k tomu, že se stejně jmenují, bude globální proměnná *$i* až do konce těla podprogramu *raz*, tedy do řádku 10 včetně, nedostupná. Když se na řádku 9 vypisuje hodnota *$i*, jedná se proto o lokální proměnnou a výsledkem bude číslo 8.

```
$i = 1;
raz();
dva();
$i++;
print "$i\n";

sub raz {
    my ( $i ) = 8;
    print "$i\n";
}

sub dva {
    $i++;
    print "$i\n";
}
```

Obrázek 8.2: Rozsahy platnosti globálního *$i* a lokálního *$i*

Tím je podprogram *raz* dokončen a program pokračuje za jeho voláním, čili na konci řádku 2. Následuje volání podprogramu *dva*, který zvětšuje hodnotu *$i* a vzápětí ji vypisuje. Tentokrát se nachází již mimo dosah lokální deklarace *$i* z osmého řádku (ta platila jen uvnitř podprogramu

raz). Proto identifikátor *$i* označuje staré dobré globální *$i* z prvního řádku. Jeho hodnota se tedy zvětší na 2 a vypíše.

Mimochodem – kdyby i podprogram *dva* začínal příkazem **my** (*$i*), jednalo by se o další lokální proměnnou, která je zcela nezávislá na předchozích dvou *$i*. Tentokrát by její existence a dostupnost byla omezena na tělo podprogramu *dva*.

Po dokončení *dva* následuje návrat do hlavního programu. Zde se na řádcích 4 a 5 opět pracuje s proměnnou *$i*. Jsme mimo dosah jakýchkoli lokálních deklarací, takže se jedná o globální *$i*, jehož hodnotu podprogram *dva* zvýšil na dvojku. Nejprve se zvětší a vzápětí vypíše, takže posledním výstupem programu bude číslo 3. Obrázek 8.2 názorně ilustruje, kde platí které *$i*. ■

Obecné doporučení zní: proměnné by měly být co nejlokálnější. Jakmile potřebujete uvnitř podprogramu pracovní proměnnou, která bude sloužit pouze zde, a nechcete jejím prostřednictvím předávat výsledky někam ven, vždy by měla být deklarována jako lokální. Díky tomu se značně zjednoduší vzájemné vztahy mezi podprogramy a ubyde vám jeden míček, se kterým musíte žonglovat. Vaše spotřeba prášků proti bolení hlavy zaručeně poklesne.

Obecný tvar **my** jsem popsal výše, nicméně při deklaraci jediné proměnné se obvykle používá zápis bez závorek:

```
my $i = 8;
```

Pokud deklarace zahrnuje také inicializaci proměnné, má přítomnost závorek důležitý dopad na pravou stranu přiřazení. Bez závorek je vyhodnocena ve skalárním kontextu, zatímco s nimi v kontextu seznamovém. Po provedení:

```
my @pole = ( 10, 20, 30 );
my $delka = @pole;
my ( $prvni ) = @pole;
my $posledni = ( 10, 20, 30 );
```

bude *$delka* obsahovat 3, protože *@pole* se vyhodnotí ve skalárním kontextu, výsledkem tudíž bude počet jeho položek. *$prvni* získá hodnotu 10, protože je v závorkách, tudíž se pravá strana vyhodnotí v seznamovém kontextu a následně se do proměnných ze seznamu na levé straně dosadí hodnoty na odpovídajících pozicích vpravo. Jelikož levý seznam obsahuje jedinou proměnnou, bude do ní dosazen první prvek pravého seznamu a jeho zbytek se ignoruje. Na posledním řádku se vyhodnocuje seznam ve skalárním kontextu a vydá svou poslední hodnotu. Proměnná *$posledni* tedy zahájí svou existenci s hodnotou 30.

Deklarujete-li několik lokálních proměnných jedním **my**, *vždy jejich seznam zavírejte do závorek*. **my** má vyšší prioritu než čárka, takže:

my *$raz*, *$dva* = (10, 20);

znamená ve skutečnosti:

my *$raz*;
$dva = (10, 20);

Čili vznikne lokální proměnná *$raz* s hodnotou **undef** a *globální* proměnná *$dva* s hodnotou 20 (seznam je vyhodnocován ve skalárním kontextu).

Existuje ještě jeden mechanismus, který se podobá lokálním proměnným. Dokonce oplývá svůdným jménem **local**. Ve skutečnosti ale pracuje poněkud jinak. Když deklarujete proměnnou:

local (*»seznam proměnných«*)

(opět lze deklarovat několik proměnných najednou a opět jim lze přiřadit hodnoty), vezme se její aktuální hodnota a odloží kamsi do bezpečí. Dotyčná proměnná se nadále chová jako globální, mohou k ní přistupovat všechny části programu (například volané podprogramy) a její případná změna bude mít vliv na činnost všech zúčastněných. Jakmile však skončí blok, ve kterém byla proměnná deklarována jako **local**, vrátí se jí původní bezpečně odložená hodnota. Po dokončení podprogramu tudíž bude mít proměnná stejnou hodnotu jako před jeho začátkem.

Cvičení 8.1: Jaký výstup vytvoří následující program? Změnil by se, kdybych na šestém řádku nahradil klíčové slovo **local** slovem **my**?

local.pl

```
1   my $a = 10;
2   raz();
3   print "a = $a\n";
4
5   sub raz {
6       local ( $a ) = 20;
7       dva();
8   }
9
10  sub dva {
11      print "a = $a\n";
12  }
```

■

Mechanismus lokálních proměnných a klíčové slovo **my** byly zavedeny až pátou verzí Perlu. Některé starší struktury odkojené Perlem 4 (mne nevyjímaje) měly potíže odtrhnout se od zažitého **local**. Nicméně **my** je opravdu mnohem lepší a je s námi dnes už dostatečně dlouho na to, abychom **local** poslali do zaslouženého důchodu, protože pouze omezuje vzájemné interference mezi podprogramy zatímco **my** je vylučuje.

8.3 Parametry a výstupní hodnoty

Někdy je naopak žádoucí, místy až potřebné, aby se podprogramy navzájem ovlivňovaly. Aby jeden předal druhému hodnoty, které má zpracovat. Například podprogram *hvezdicky* v příkladu ze strany 111 potřebuje znát číslo měsíce a počet hvězdiček, které má vypsat. Tam jsem situaci vyřešil pomocí globální proměnné, což ale není nejvhodnější.

Proč? Představte si, že by tímto způsobem byla řešena standardní funkce **sqrt**. Bylo by řečeno, že existuje proměnná s pevně definovaným jménem *$tohle_odmocni*. Kdykoli chcete spočítat druhou odmocninu, musíte do ní přiřadit danou hodnotu a pak zavolat **sqrt**. Takže místo:

```
$y = sqrt($x);
```

byste museli psát:

```
$tohle_odmocni = $x;
$y = sqrt();
```

Líbí? Těžko. Je to neohrabané, nepřehledné a náchylné k chybám. Proto panuje snaha, aby co nejvíce vstupních informací přicházelo do podprogramů prostřednictvím jejich parametrů – stejně jako u standardních funkcí.

Podle metody cukru a biče teď musí přijít studená sprcha. V předchozích kapitolách jsem vás hojně pocukroval regulárními výrazy a asociativními poli a nastal čas si zahrát na dominu. Parametry podprogramů se v Perlu definují dost nechutným způsobem. V podstatě je definovat nemusíte. Perl automaticky vytvoří pole s obskurním názvem @_ (ano, podtržítko), do kterého přiřadí všechny parametry, které byly podprogramu předány.

Aby se vyhovělo alespoň elementárním základům dobrého vkusu, panuje mezi perlícími programátory konvence, že vytvoří skupinu lokálních proměnných, které převezmou hodnoty z tohoto pole. Jednotlivým parametrům tak přidělí jména, jak bývá ve slušné společnosti zvykem. Řekněme, že bych definoval funkci *max3*, která hledá největší z trojice čísel. Tato tři čísla dostane jako své parametry a uvnitř funkce je budu nazývat *$a*, *$b* a *$c*. Deklarace funkce by vypadala takto:

```
sub max3 {
    my ( $a, $b, $c ) = @_;
    »tělo funkce«
}
```

Její volání odpovídá standardním funkcím – předáte jí příslušný počet hodnot, které budou přiřazeny jednotlivým parametrům podle pořadí:

```
$nejvetsi = max3( 27, 38, 19 );
```

- Přiřazujete-li pole @_, vždy při deklaraci parametru pište za **my** závorky, i kdyby byl parametr jen jeden. Příkaz:

```
my ( $x ) = @_;
```

správně do parametru *$x* přiřadí první prvek pole @_, zatímco:

```
my $x = @_;
```

vyhodnotí pravou stranu ve skalárním kontextu a do *$x* přiřadí počet prvků pole @_.

Někdy programátoři parametry přesouvají pomocí standardní funkce **shift** z pole @_ do příslušných proměnných. Při použití tohoto přístupu by funkce *max3* vypadala následovně:

```
sub max3 {
    my $a = shift;
    my $b = shift;
    my $c = shift;
    »tělo funkce«
}
```

Volání **shift** bez parametrů způsobí, že funkce bude použita na implicitní pole @_. Rozdíl obou variant je v tom, že při použití **shift** jsou parametry z pole @_ odstraňovány, zatímco při prostém přiřazení zůstává pole beze změny a kdykoli později se k němu můžete v těle podprogramu vrátit.

Jiným způsobem, jak parametrům přidělit jména, je použít asociativní pole. V definici podprogramu jednoduše přiřadíte pole @_ do asociativního pole a následně z něj vyzvedáváte hodnoty pomocí klíčů:

```
sub max3 {
    my %param = @_;
    my $max = $param{a};
    if ( $max < $param{b} ) { $max = $param{b} };
    …
}
```

V takovém případě ovšem musíte jména uvádět i při volání funkce, protože hodnoty na lichých pozicích budou interpretovány jako klíče (čili jména parametrů) a jejich následnice jako hodnoty klíčům přiřazené:

```
$nejvetsi = max3( a => 27, c => 19, b => 38 );
```

Výhodou je, že lze parametry uvádět v libovolném pořadí. Záleží jen na jejich jménech. Tento přístup se hodí, pokud mají parametry jasně definované významy. Například když budete volat podprogram pro založení zaměstnance:

```
zaloz_zamestnance(
    jmeno => "Pavel Satrapa",
    oddeleni => "NTI",
    osobniCislo => 12345
);
```

Z volání je na první pohled zřejmé, k čemu je která hodnota určena. Naopak funkce *max3* se pro tento způsob práce s parametry vysloveně nehodí. Zde je význam všech parametrů stejný a jména jim dáváme jen proto, abychom s nimi mohli pracovat. Nutit uživatele, aby je při volání funkce pojmenovával, jen zbytečně přidělává práci.

Přiřazování hodnot z pole @_ do proměnných či asociativního pole nemá jen estetický význam. Pole @_ je zvláštní a funguje jako skupina přezdívek pro skutečné parametry, které jste podprogramu předali při jeho volání. Když do některé z položek @_ přiřadíte hodnotu, přiřadí se do proměnné, která parametru odpovídá. A byla-li daným parametrem konstanta, program skončí běhovou chybou.

Pokud ovšem do celého pole @_ přiřadíte seznam, s parametry se nestane nic, pole @_ získá nové hodnoty a vazby na parametry se z něj nenávratně ztratí.

Příklad: Kuriózní chování pole @_ předvedu na dvou ukázkových podprogramech. *meni* do svého prvního parametru přiřazuje číslo 10, proto změní hodnotu proměnné, se kterou podprogram zavoláte. Naproti tomu *nemeni* přiřazuje celé nové pole, takže neudělá v podstatě nic.

zmeny-param.pl

```
1   sub meni {
2       @_[0] = 10;
3   }
4
5   sub nemeni {
6       @_ = (99);
7   }
8
9   my $x = 1;
10  print $x, "\n";
11  meni($x);
12  print $x, "\n";
13  nemeni($x);
14  print $x, "\n";
15  meni(5);
16  print $x, "\n";
```

Výstupem budou čísla 1, 10, 10 a chybové hlášení, protože funkce *meni* volaná na řádku 11 změní hodnotu proměnné *$x*, zatímco funkce *nemeni* s ní na řádku 13 neudělá nic. Při volání na řádku 15 se funkce *meni* pokusí změnit konstantu a program skončí chybou. ■

Chování pole @_ je nezvyklé. Abyste si nenatloukli ústa, doporučuji držet se následujících pravidel:

1. Nikdy nepřiřazujte do pole @_ ani jeho položek.
2. Pravidlo 1 porušte jen tehdy, pokud je změna parametrů hlavním účelem funkce. Nezapomeňte na to upozornit v dokumentaci nebo komentářích.

Až v roce 2014 přišla verze 5.20 s možností zadávat parametry podprogramů v podobě připomínající zbytek světa. Zatím Perl spíš jen namáčí palec do rybníka, protože se jedná o experimentální vlastnost, kterou musíte zapnout pomocí **use** *experimental* 'signatures' a která se v příštích verzích může změnit nebo zmizet. Když ji zapnete, můžete podprogram definovat v podobě:

```
sub »jméno« ( »parametry« ) {
    »tělo«
}
```

»*Parametry*« jsou jména parametrů podprogramu, pod nimiž se v »*těle*« používají. Není třeba volat **shift** ani přiřazovat @_, máte je rovnou k dispozici. Zároveň je tím určen jejich počet, který Perl bude hlídat. Má-li být počet parametrů proměnlivý, uveďte na posledním místě jejich seznamu pole a všechny zbývající hodnoty při volání budou uloženy do tohoto pole.

Parametru lze v hlavičce podprogramu „přiřadit“ hodnotu a tím jej učinit nepovinným. Když bude při volání chybět, přiřadí se mu tato hodnota.

Příklad: Podívejme se na jednoduchý příklad – definuji funkci s jedním povinným a jedním nepovinným parametrem:

```
use experimental 'signatures';

sub ahoj ( $jmeno, $pozdrav = "Ahoj" ) {
    print "$pozdrav $jmeno.\n";
}

ahoj( "Pepo" );
ahoj( "Pepo", "Dobrý den" );
ahoj();
```

Jejím výstupem bude:

```
Ahoj Pepo.
Dobrý den Pepo.
Too few arguments for subroutine 'main::ahoj' at pozdrav.pl line 9.
```

■

Příklad: Proměnlivý počet parametrů s použitím experimentální syntaxe by se dal využít k vytvoření funkce *pasmo*, která dostane minimum, maximum a libovolný počet hodnot. Ty vypíše, ovšem omezené daným rozpětím – hodnoty překračující meze budou nahrazeny minimem či maximem.

pasmo.pl

```
use experimental 'signatures';

sub pasmo ( $min, $max, @hodnoty ) {
    foreach my $x ( @hodnoty ) {
        if ( $x < $min ) {
            print $min, "\n";
        } elsif ( $x > $max ) {
            print $max, "\n";
        } else {
            print $x, "\n";
        }
    }
}
```

pasmo(–10, 10, 52, 8, –3, –20, 7, 15);

Při volání na posledním řádku začínají zpracovávané hodnoty číslem 52 a budou omezeny na interval od –10 do 10. Program tedy vypíše čísla 10, 8, –3, –10, 7 a 10.

■

8.4 Výstupní hodnoty

Veškeré podprogramy v Perlu jsou vlastně funkce. To znamená, že jejich provedením vznikne určitá výstupní hodnota, se kterou lze dále pracovat. Můžete ji také ignorovat, což je u některých více než žádoucí. Například podprogram *hvezdicky* slouží k tomu, aby zobrazil nějaký výstup, nikoli aby vypočítal hodnotu. V jeho případě nemají úvahy o výstupní hodnotě žádný smysl – úkol podprogramu spočívá v něčem jiném.

Nicméně kdybych se zbláznil[1] mohl bych i s ním pracovat jako s funkcí a psát věci typu:

$pishweytz = 2 * *hvezdicky*() + 19;

Neřeknete-li jinak, je jako výstupní hodnota funkce brána výstupní hodnota posledního příkazu, který je v ní proveden. Takže každý podprogram má definovánu svou výstupní hodnotu, byť někdy neúmyslně.

Pokud má podprogram sloužit skutečně jako funkce, doporučuji výstupní hodnotu explicitně stanovit příkazem:

return *»výraz«*

Jeho účinek je dvojí: okamžitě ukončí provádění daného podprogramu a použije výsledek vyhodnocení *»výrazu«* jako jeho výstupní hodnotu.

Příklad: Dokončím již nakousnutý příklad funkce *max3*. Dostane jako parametry tři hodnoty a jako svůj výsledek vydá největší z nich.

max3.pl

```
sub max3 {
    my ( $a, $b, $c ) = @_;
    if ( $a < $b ) { $a = $b; }
```

1: Nevylučuji, že k tomu do konce téhle knížky nedojde, hé, hé…

```
    if ( $a < $c ) { $a = $c; }
    return $a;
}
```

Můžete ji volat například způsobem:

```
$maxplat = max3( $plat1, $plat2, 15000 );
```

Všimněte si, jak uvnitř *$max3* s chladným srdcem devastuji hodnoty lokálních proměnných (konkrétně *$a*). Právě k tomu jsou určeny – po skončení podprogramu přestanou existovat, takže je uvnitř mohu používat, jak se mi hodí a nemusím se ohlížet na to, abych některé z okolních částí programu nešlápl do úsměvu. Tato nezávislost je hlavní předností lokálních proměnných. ■

Příklad: Jelikož se parametry předávají v podobě pole a pole se nemusí v Perlu nijak předem deklarovat, klidně můžete používat podprogramy s proměnlivým počtem parametrů. Snadno například zhotovím obecnou funkci *max*, která dostane libovolný počet parametrů a jako výstupní hodnotu vydá největší z nich:

max.pl

```
sub max {
    my @hodnoty = @_;
    my $max = $hodnoty[0];
    foreach my $hodnota ( @hodnoty ) {
        if ( $hodnota > $max ) { $max = $hodnota; }
    }
    return $max;
}
```

Upřímně řečeno, v praxi bych se u takto jednoduchého podprogramu pravděpodobně nenamáhal s kopírováním pole parametrů @_ do *@hodnoty*. Ušetřil bych tak špetku času. U složitějších podprogramů však symbolické pojmenování může významně zvýšit srozumitelnost. Konstrukce typu @_[0] přece jen vypadají dosti obskurně. ■

A co když potřebujete, aby výsledkem funkce bylo více hodnot? Není nic snazšího. Prostě příkazu **return** předáte seznam hodnot a výsledkem bude tento seznam. Musíte tomu pochopitelně přizpůsobit i zpracování výsledků v místě volání funkce.

Příklad: Poněkud rozvinu funkci *max* z předchozího příkladu. Na jejím základě vybuduji funkci *minimax*, která dostane jako parametr libovolně dlouhý seznam hodnot. Jako svůj výsledek vydá seznam dvou hodnot, z nichž první bude nejmenší z parametrů a druhá největší. Rovnou kolem ní postavím celý program, který si vyžádá od uživatele sérii čísel a na závěr vypíše největší a nejmenší z nich.

minimax.pl

```
my @hodnoty;
print "Určení extrémů\n";
print "Zadávejte čísla, prázdný řetězec ukončí vstup.\n";
while ( (my $hodnota = vstup_hodnoty()) ne "" ) {
    push(@hodnoty, $hodnota);
}
my ($min, $max) = minimax(@hodnoty);
print "Nejmenší je $min\n";
print "Největší je $max\n";

sub vstup_hodnoty {
# funkce, která zajistí zadání jedné hodnoty uživatelem
# vstup: žádný
# výstup: zadaná hodnota
    print "hodnota: ";
    my $hodnota = <STDIN>;
    chomp($hodnota);
    return $hodnota;
}

sub minimax {
# funkce pro určení nejmenší a největší hodnoty
# vstup: seznam hodnot
# výstup: dvojice ( nejmenší, největší )
    my @hodnoty = @_;
    my ( $max, $min ) = ( $hodnoty[0], $hodnoty[0] );
    foreach my $hodnota ( @hodnoty ) {
        if ( $hodnota > $max ) { $max = $hodnota; }
        if ( $hodnota < $min ) { $min = $hodnota; }
    }
    return ( $min, $max );
}
```

Všimněte si, že v příkladu jsou podprogramy použity výlučně ke zvýšení srozumitelnosti. Každý se používá jen na jediném místě, takže délku programu rozhodně nezmenší. Ba právě naopak. Rozdělí ho ale na několik nepříliš velkých a logicky ucelených částí, což usnadňuje jeho pochopení.

Je velmi vhodné doprovodit každý podprogram stručným komentářem, ze kterého bude patrné:

- co dělá,
- jaké parametry se mu mají předat,
- co vydá jako výsledek.

Upozorňuji, že konstrukce předvedeného programu má k ideálu daleko. Slouží jako ukázka použití daného podprogramu *minimax*. Extrémy ve vstupujících hodnotách nevyžadují jejich načtení do pole. Stačí jednoduchá skalární proměnná, do které bych načetl vždy jednu hodnotu, porovnal se stávajícími extrémy a případně uložil jako nový extrém. A tak dále pořád dokola, dokud neskončí vstup. ■

Pud sebezáchovy říká: Chcete-li přežít své podprogramy, snažte se, aby se veškerá jejich komunikace s okolím odehrávala prostřednictvím parametrů a výstupních hodnot. Všechny ostatní proměnné, se kterými podprogram pracuje, by měly být lokální.

Uvedené pravidlo směřuje k jasnému cíli: aby vazby podprogramu se zbytkem světa byly co nejjednodušší. Existují pochopitelně výjimky. Například pokud máte ve svém programu jakousi centrální datovou strukturu (např. databázi zaměstnanců), mohou k ní jednotlivé podprogramy přistupovat přímo. Avšak takovéto „postranní vchody" by měly být skutečně výjimečné a dobře odůvodněné.

Cvičení 8.2: V posledním příkladu se k uživatelovu zadání stavím zcela bezelstně a nechám si od něj napsat, co bude chtít. Pokud zadá něco jiného než číslo, povede to k chybovým hlášením. Upravte funkci *vstup_hodnoty* tak, aby svou výzvu opakovala tak dlouho, dokud uživatel nezadá prázdný řetězec nebo celé číslo. Doporučení: zaveďte si další podprogram *platny_vstup*, který dostane jako parametr uživatelem zadanou hodnotu a vydá hodnotu pravda/nepravda podle toho, zda vstup byl korektní. ■

8.5 Dekompozice

Nastal čas zmínit se o návrhu programů metodou shora dolů. Název „shora dolů" neznamená, že to s vašimi programátorskými dovednostmi půjde od desíti k pěti nebo že snad zoufalstvím vyskočíte z patnáctého patra. Ba právě naopak. Cílem této metody je, abyste se dokázali popasovat i se značně složitými problémy a nevykloubili si při tom hlavu.

Jako přípravu ke studiu bych si dovolil vřele doporučit povídku *Kusejr* z pera Ivana Vyskočila. V ní je tato metoda popsána velmi výstižně. Kusejr dokázal hovořit o problémech. Během řeči rozložil původní problém na několik dílčích problémů, každý z nich pak na pár problémků, ze kterých vznikly problémečky a tak dále. Nakonec původní vážný problém rozdělil na hromadu pidiproblémků tak snadných, že to vlastně ani žádné problémy nebyly[2].

Návrh programu metodou shora dolů postupuje úplně stejně. Máte napsat program, který cosi dělá, řeší jakousi úlohu. Vaše úvahy by měly začít návrhem datových struktur – rozmyslet si, jak

2: Pointu povídky vám neprozradím, jen doporučím, že opravdu stojí za přečtení – ostatně jako všechno, co jsem kdy od Ivana Vyskočila četl.

budete mít uloženy informace, které potřebujete. Když je máte, začněte přemýšlet o výkonné části programu.

Nejprve si vymezte základní činnosti, které budete k vyřešení úlohy potřebovat, a jejich vzájemné vztahy. Na jejich základě můžete napsat hlavní program, který bude řídit celý výpočet. Tyto činnosti v něm budou vystupovat v podobě volání jednotlivých podprogramů. Řekněme, že hodláte napsat program, který bude hrát šachy. To je jistě dost složitá úloha, nicméně hlavní program přesto může být víceméně triviální:

```
inicializace();       # rozestaví figurky a podobně
while ( not konec_hry() ) {
      udelej_tah();
}
vyhodnot_viteze();
```

Nyní máte základní konstrukci programu hotovou a zbývá „jen“ napsat dotyčné podprogramy. V našem případě budou jistě dost komplikované, ale můžete na ně uplatnit stejný postup. Takovýmto způsobem problém postupně ožižláváte, až složitost jednotlivých dílčích podprogramů klesne natolik, že je budete schopni zapsat už přímo základními příkazy Perlu.

Cílem je, abyste při řešení nedělali příliš velké logické kroky. Existuje zlaté pravidlo: délka hlavního programu či libovolného z podprogramů by neměla přesáhnout obrazovku. Proč? Je to prosté. Děláte-li mnoho věcí najednou, s vysokou pravděpodobností něco neuhlídáte a vše se pokazí. Jestliže při řešení postupujete po malých krocích a každý váš podprogram dělá vždy jen poměrně malou a jasně definovanou část práce, máte dobrou šanci, že si udržíte přehled a svůj program pod kontrolou.

Toto rozkládání problémů na dílčí podproblémy se nazývá dekompozice. Jeho zvládnutí – tedy dovednost určit vhodné dílčí části – patří ke klíčovým vlastnostem dobrého programátora.

Příklad: Předvedu ukázku postupného řešení netriviální úlohy. Je jí zpracování jednoho tahu jisté loterie. Účastníci hádají pět čísel z padesáti. Pokud se někomu podaří uhodnout všech pět, vyhrává první cenu. Za čtyři uhodnutá čísla je druhá cena a za tři cena třetí.

Vstupem programu jsou informace o jednotlivých losech. Každý los je na jednom řádku. Začíná svým identifikačním číslem, za nímž následuje pětice tipovaných čísel. Vše je oddělováno mezerami – například:

```
4682387456   8   19   23   38   46
```

Program má náhodně vylosovat pět čísel a porovnat s tipovanými. Výstupem bude informace o tažených číslech a seznam identifikačních čísel losů, které vyhrály první, druhou a třetí cenu.

Jako základní datová struktura se nabízí asociativní pole (nazvu je *%tazeno*), kam pro vylosovaná čísla uložím jedničky. Dotaz, zda určité číslo bylo vylosováno, pak vyřeším snadným *$tazeno*{*$cislo*}. Dále si zavedu tři obyčejná pole *@prvni*, *@druzi* a *@treti*, do kterých budu ukládat čísla losů vyhrávajících odpovídající pořadí.

Hlavní program lze formulovat snadno: nejprve proběhne inicializace (to je standardní začátek prakticky každého programu), pak se vylosují čísla. Následuje cyklus zpracovávající jednotlivé losy. Určí se pořadí losu a pokud byl úspěšný, zařadí se do odpovídajícího pole. Na závěr vypíši vítěze:

losy.pl

```
# globální proměnné
my $pocetcisel = 5; # počet losovaných čísel
my $maxcislo = 50; # největší losované číslo
my %tazeno; # tažená čísla
my (@prvni, @druzi, @treti); # výherci

inicializace();
vylosuj_cisla();
while ( my @los = dalsi_los() ) {
    my $poradi = urci_poradi(@los);
    if ( $poradi <= 3 ) {
        zarad_viteze($los[0], $poradi);
    }
}
vypis_viteze();
```

Inicializace je v tomto případě triviální – pouze inicializuje generátor náhodných čísel (standardní podprogram **srand**). Zajímavější je podprogram *vylosuj_cisla*, který naplní pole *%tazeno*. Generátor náhodných čísel nezaručuje, že se hodnoty nebudou opakovat. Proto je při přidávání třeba kontrolovat, zda dané číslo již nebylo vylosováno. Na závěr se vypíše seznam tažených čísel.

```
sub inicializace {
    srand;
}

sub vylosuj_cisla {
    my ( $cislo, $vylosovano ) = ( 0, 0 );
    while ( $vylosovano < $pocetcisel ) {
        $cislo = int( rand($maxcislo) ) + 1;
```

```
            if ( not $tazeno{$cislo} ) {
                $tazeno{$cislo} = 1;
                $vylosovano++;
            }
        }
        print "Tažena byla čísla: ";
        foreach $cislo ( sort {$a <=> $b} keys %tazeno ) {
            print "$cislo ";
        }
        print "\n";
    }
```

Funkce *dalsi_los* načte informace o dalším losu a vydá je v podobě pole. Jeho první položkou je číslo losu, dalšími pak tipovaná čísla.

```
    sub dalsi_los {
        my $radek = <>;
        chomp($radek);
        return split( /\s+/, $radek );
    }
```

Funkce *urci_poradi* nejprve spočítá, kolik tažených čísel daný los uhodl. Svůj výsledek vydá v podobě čísla od 1 do 6, kde 1 znamená „uhodl vše – první cena“ a 6 „neuhodl nic“.

```
    sub urci_poradi {
        my @los = @_;
        my $uhodl = 0;
        shift(@los); # zbavím se čísla losu
        foreach my $cislo ( @los ) {
            if ( $tazeno{$cislo} ) { $uhodl++; }
        }
        return $pocetcisel + 1 – $uhodl;
    }
```

Podprogram *zarad_viteze* dostane dva parametry: číslo losu a cenu, kterou získal. Podle toho jej zařadí do odpovídajícího pole vítězů.

```
    sub zarad_viteze {
        my ( $los, $poradi ) = @_;
        if ( $poradi == 1 ) { push(@prvni, $los); }
        elsif ( $poradi == 2 ) { push(@druzi, $los); }
```

```
        elsif ( $poradi == 3 ) { push(@treti, $los); }
    }
```

A konečně *vypis_viteze* je oslavou zvítězivších. Postupně vypíše seznamy losů vyhrávajících první, druhou a třetí cenu. Jelikož je výpis obsahu pole opět logicky uceleným podproblémem, zavedl jsem si pro něj další podprogram.

```
    sub vypis_viteze {
        print "Výherci první ceny:\n";
        vypis_seznam(@prvni);
        print "\nVýherci druhé ceny:\n";
        vypis_seznam(@druzi);
        print "\nVýherci třetí ceny:\n";
        vypis_seznam(@treti);
    }

    sub vypis_seznam {
        foreach my $los ( sort {$a <=> $b} @_ ) {
            print "$los\n";
        }
    }
```

■

Snadnější vytvoření není jedinou předností dobré dekompozice programu. Ta přináší celou řadu dalších výhod.

V první řadě *větší srozumitelnost*. Když neděláte velké logické veletoče a případná triková místa opatříte vysvětlujícím komentářem, vaše programy bude radost číst. Případné pozdější úpravy či rozšíření by neměly pro nikoho (včetně vás samotných!) představovat problém.

Snadnější lokalizace chyb. Má-li každá část programu přesně vymezenou funkci, dokážete poměrně přesně určit, která z nich by mohla mít na svědomí vzniklé problémy. Ostatně díky větší přehlednosti se výrazně snižuje riziko, že vůbec nějakou chybu uděláte.

Možnost *odděleného ladění*. Pokud ladíte určitou část programu, můžete ty ostatní vyměnit za jednoduché náhražky, které nic nepokazí a pomohou vám v ladění. Například když jsem ladil loterijní program z předchozího příkladu, potřeboval jsem ověřit, zda funguje dobře vyhodnocování a vypisování vítězů. Proto jsem dočasně nahradil podprogram *vylosuj_cisla* šidítkem, které nic nelosovalo, ale naplnilo pole *%tazeno* pětici mnou zvolených čísel. Díky tomu jsem mohl snadno ověřovat, zda se vyhodnocování chová, jak má.

Dobrá přizpůsobitelnost. Jednotlivé podprogramy nemusíte nahrazovat jen šidítky, která simulují jejich funkci a jsou plně pod vaší kontrolou. Můžete je také upravovat podle měnících se okolních podmínek.

Řekněme, že by se změnila pravidla a místo pěti čísel z padesáti by se tahalo šest čísel ze čtyřiceti. Jedinou změnou, kterou bych v programu musel provést, by byl zásah do hodnot globálních proměnných na začátku. Pokud by byla změna radikální (např. by se směly opakovat hodnoty), musel bych přizpůsobit i podprogramy *vylosuj_cisla* a *urci_poradi*. Vše ostatní by zůstalo identické.

Nebo by údaje o losech vstupovaly jiným způsobem – například z databáze nebo jakýmsi on-line spojením se vzdáleným serverem. Opět by stačilo zasáhnout na jediné místo – do podprogramu *dalsi_los*. V dobře strukturovaném programu zpravidla změníte jen velmi malou část a navíc je okamžitě patrné, která část by to měla být.

Cvičení 8.3: Pěkně jsem si svůj loterijní program vychválil. Ale přece jen musím přiznat, že má jisté mušky. Spoléhá totiž na to, že vstupní data jsou korektní. Pokuste se do něj doplnit kontrolu, která se vypořádá s následujícími potenciálními nedostatky:

- Duplicitní čísla losů – vyskytne-li se vícekrát stejné číslo losu, bude se posuzovat jen první z těchto losů, ostatní budou ignorovány.
- Opakování čísel – je-li na jednom losu několikrát tipováno stejné číslo, ponechá se jen jediný jeho výskyt.
- Příliš mnoho tipů – pokud los obsahuje více než pět (použijte hodnotu *$pocetcisel*, ať se nepřipravíte o obecnost) různých hodnot, los bude ignorován.

Pokud si netroufnete rovnou modifikovat program, zkuste napsat samostatný program, který realizuje tyto kontroly. Ten by se použil na vstupní data jako preprocesor (předžvýkávač) a do svého výstupu by pustil jen losy vyhovující výše uvedeným podmínkám.

Ve druhé fázi si pak zkuste rozmyslet, jak by se tyto kontroly daly zabudovat přímo do hlavního programu. ■

9 Vstupy a výstupy

Nejzákladnější sadu cviků souvisejících se vstupem a výstupem dat již znáte. V této kapitole se ji pokusím poněkud rozšířit a prohloubit. Podíváme se nejprve na možnosti formátovaného výstupu a poté odtajním práci se soubory.

9.1 Jednoduchý formátovaný výstup

Základní výstupní funkcí je **print**, to už dávno víte. Jistou nevýhodou je, že si do své práce nenechá příliš mluvit – například si nemůžete poručit, s jakou přesností mají být vypisovány výsledné hodnoty. Tyto vaše ambice naplní až spřátelená funkce **printf** dobře známá programátorům v C:

printf *»vzor«*, *»seznam hodnot«*

»vzor« je řetězec znaků a představuje základní kostru výstupu. V podstatě lze říci, že funkce **printf** vypíše do standardního výstupu *»vzor«*, do kterého ale na patřičná místa dosadí hodnoty ze *»seznamu«*. Ona patřičná místa se ve *»vzoru«* vyznačují speciálními kombinacemi znaků. Zároveň tak určujete, jakým způsobem má být dotyčná hodnota formátována.

Tyto speciální konstrukce začínají vždy znakem „%“, za kterým následuje písmeno určující způsob formátování dané hodnoty. Seznam nejběžnějších písmen najdete v tabulce 9.1. Mezi tyto dva prvky můžete ještě vložit číslo udávající požadovanou šířku. Přesněji řečeno udává minimální počet znaků, které má výstup zabrat. Pokud je šířka vystupující hodnoty menší, bude zleva doplněna patřičným počtem mezer. Zadáte-li požadovanou šířku záporným číslem, budou mezery doplněny zprava. Jestliže číslo zahájíte nulou, doplní se na požadovanou šířku zleva nulami místo mezer.

%	znak „%“		
c	jeden znak	**s**	řetězec znaků
d	desítkové číslo		
x	šestnáctkové číslo	**e**	exponenciální číslo
o	osmičkové číslo	**f**	destinné číslo
b	binární číslo	**g**	krátký tvar reálného čísla

Tabulka 9.1: Formátování výstupů funkce **printf**

Požadovanou šířku lze dokonce uvést prostřednictvím desetinného čísla. Celý zápis pak má tvar:

%*m.n»znak«*

Jak již bylo řečeno, *m* udává minimální šířku. Naproti tomu význam *n* se mění v závislosti na typu hodnoty. Pro znak či řetězec znaků určuje maximální šířku výstupu. Je-li řetězcová hodnota delší, vystoupí jen jejích prvních *n* znaků. U celých čísel je *n* ignorováno – zkracovat čísla opravdu není dobrý nápad. Tady je lepší, když se poruší výstupní formát, než když se 1024 promění na 102. Pro reálná čísla *n* stanoví počet míst za desetinnou tečkou.

Příklad: Předvedu, jak se tatáž hodnota proměňuje, když ji vypíšete v ruzných formátech. Proměnná *$a* bude obsahovat číslo 109. Použiji příkaz:

printf *formát*, *$a*;

Následující tabulka uvádí, jak bude vypadat výstup pro různé formáty:

formát	*výstup*
”%d”	109
”%x”	6d
”%o”	155
”%b”	1101101
”%5d”	␣␣109
”%05d”	00109
”%–5d”	109␣␣
”%12.3e”	␣␣␣1.090e+02
”%12.3f”	␣␣␣␣␣109.000
”%1.2s”	10
”%c”	m

Možná vám bude připadat podivný poslední řádek. Pokud totiž necháte něco vypsat jako jediný znak, předpokládá se, že dotyčná proměnná obsahuje ASCII kód. Vlastně se zavolá funkce **chr** a vystoupí její výsledek. ASCII kód 109 má písmeno „m“ a to také vystoupí. ■

Příklad: A ještě jednu ryze praktickou ukázku. Jsme v závěru programu, který do proměnné *$cena* spočítal celkovou cenu zboží. Je třeba ji vypsat, a to bez DPH i s ním. Problém vyřeší příkaz:

printf ”Celkem %0.2f Kč (%0.2f včetně DPH)\n”, *$cena*, *$cena**1.21;

který vypíše například:

Celkem 685.40 Kč (829.33 včetně DPH)

Použití **printf** je zde de facto nutností. Sice se můžete pokusit vyjít s obvyklou funkcí **print** a zápis bude dokonce o chloupek kratší:

```
print "Celkem $cena Kč (", $cena*1.21, " včetně DPH)\n";
```

ovšem výsledek vypadá prapodivně:

```
Celkem 685.4 Kč (829.334 včetně DPH)
```

Při srovnání obou výstupů vidíte, že Perl zaokrouhluje přesně tak, jak je v kraji zvykem. ■

Jak dokazuje příklad, **printf** poslouží všude tam, kde potřebujete zaokrouhlovat na pevný počet desetinných míst (např. při práci s cenami). Hodí se také v situacích, kdy sestavujete řádek z několika vypočítávaných hodnot. Tehdy bývá úspornější.

Tento způsob formátování můžete používat také interně uvnitř programu. Existuje totiž příbuzná funkce **sprintf**. Chová se úplně stejně jako **printf**, až na to, že výsledný řetězec znaků nepošle do standardního výstupu, ale vydá jako svůj výsledek. Můžete jej uložit do proměnné a dále zpracovávat.

Funkci **sprintf** lze použít třeba k zaokrouhlování zpracovávaných čísel. Následující příkaz zaokrouhlí hodnotu v proměnné *$cena* na dvě desetinná místa:

```
$cena = sprintf( "%0.2f", $cena );
```

Cvičení 9.1: Napište program, který vypíše formátovaný ceník. Jeho vstupem budou informace o cenách jednotlivých výrobků. Každý výrobek je na jednom řádku ve tvaru:

»název«#»cena«

Výstupem vašeho programu by měla být přehledná tabulka obsahující tři sloupce: názvy výrobků, cenu bez DPH a cenu s DPH. Můžete předpokládat, že názvy výrobků nebudou delší než 40 znaků a ceny že nedosáhnou 1 miliónu. Sloupce s cenami by měly být zarovnány tak, aby desetinné tečky byly umístěny pod sebou. ■

9.2 Výstup podle šablony

Občas potřebujete vytvořit program, jehož výstupy mají neměnný a neustále se opakující tvar. Například na základě databáze zaměstnanců vytisknete karty s údaji o nich, které si zručný byrokrat zařadí do šanónů a teprve potom bude moci pořádně úřadovat.

Takový výstup pochopitelně lze realizovat hejnem **print** a **printf**, ovšem Perl nabízí něco lepšího. Jedná se o tak zvané formáty. Dá se říci, že formáty vycházejí z podobné myšlenky jako **printf**,

avšak dotažené ještě o krok dál. Formát totiž definuje několikařádkový vzor, do něhož se vkládají jednotlivé hodnoty. Zapisuje se ve tvaru:

format *»jméno«* =
»řádky se vzory a hodnotami«
.

Ta tečka na konci není mušinec, ale samotná tečka na řádku, která ukončuje definici formátu. Jako *»jméno«* se zpravidla používá jméno totožné s názvem ovladače souboru, pro který chcete formát používat. O ovladačích souborů se dočtete zanedlouho. Dobrou zprávou je, že pro standardní výstup ovladač psát nemusíte – pokud jednoduše vynecháte *»jméno«*, definuje se formát pro standardní výstup.

A opět stejná písnička: ve vzorových řádcích vyznačují speciální řetězce znaků, kam se mají umístit jednotlivé hodnoty. Tyto řetězce začínají vždy znakem „@“ a jejich celková délka určuje, kolik znaků má hodnota zabrat. Význam jednotlivých speciálních řetězců uvádí tabulka 9.2.

`@<<<<<<<`	hodnota zarovnaná doleva
`@>>>>>>>`	hodnota zarovnaná doprava
`@|||||||`	centrovaná hodnota
`@#####.##`	desetinné číslo v daném tvaru
`@*`	libovolně dlouhá hodnota

Tabulka 9.2: Vyznačení hodnot ve formátu

Poněkud neobvyklý je řetězec „@*“, který se musí vyskytovat samotný na řádku. Vyznačuje položku neomezené délky, která může zabrat libovolný počet řádků. Používá se především pro informace, jako je slovní popis předmětu, komentář a podobně.

Formát může mít libovolný počet řádků. Za každým řádkem s místy pro vkládání hodnot musí následovat řádek s hodnotami. Může se jednat o proměnné i výrazy, které se vyhodnotí a výsledek se vloží na příslušné místo do předchozího vzorového řádku. Párují se pozičně – první hodnota patří na první místo, druhá na druhé atd.

Chcete-li vyvolat výstup formátovaných dat, použijte příkaz **write**. Nedostane žádné parametry. Před jeho voláním musíte zajistit, že proměnným použitým na řádcích hodnot v definici formátu byly přiřazeny odpovídající hodnoty.

Příklad: Ve cvičení v předchozí části jsem po vás chtěl sestavit program pro formátování ceníku. Při využití formátu by mohl vypadat následovně (počet znaků „<“ jsem zmenšil, aby se mi definice formátu vešla na řádek, správně by jich mělo být 39):

format.pl

```
my ($nazev,$cena);
while ( my $radek = <> ) {
    chomp($radek);
    ($nazev,$cena) = split( /#/, $radek );
    write;
}

format =
@<<<<<<<<<<<<<<<<<<<<<<<<<<<< @#####.## Kč @#####.## Kč
$nazev, $cena, $cena*1.21
.
```

■

Použitý příklad je dost jednoduchý, protože šablona ve formátu má jen jediný vzorový řádek. Nejcennější služby formáty odvedou pro víceřádkové vzory, kdy jediným příkazem **write** vytvoříte celý (mnohdy dost složitý) blok vystupujícího textu.

Faktem je, že formáty představují velmi specifický nástroj s dost úzce vymezeným polem působnosti. Proto jsem jejich použití spíše naznačil a zamlčel celou řadu jejich dovedností. Pokud jsou pevně formátované výstupy vaším denním chlebem, určitě si o nich přečtěte více – například v knize [1] nebo na stránce *perlform* v dokumentaci Perlu.

9.3 Práce se soubory

Veškeré dosavadní vstupní a výstupní operace jsem prováděl pouze se standardními soubory. Program se pak chová jako filtr: data do něj vstupují ze standardního vstupu (případně ze souborů, které mu zadáte jako parametry na příkazovém řádku) a výsledky posílá do svého standardního výstupu. Každý, kdo si někdy v Unixu montoval rourami svůj vlastní mnohaprogramový slepenec, ví, že takové chování je pravé dobrodiní. Ovšem čas od času je potřeba použít i jiný soubor – například si načíst konfigurační soubor, čerpat informace o uživatelích z */etc/passwd* či přidat řádek do protokolu o činnosti.

Perl samozřejmě nabízí pohodlný způsob, jak pracovat se soubory podle svého gusta. Budete postupovat obvyklým způsobem: nejprve soubor musíte otevřít, pak jej nějakou dobu používáte a když s ním skončíte, zase jej uzavřete. Souhrn funkcí, které pro tyto činnosti Perl nabízí, najdete v tabulce 9.3.

otevření	**open**(*»ovladač«*,*»název«*)
použití	<*»ovladač«*>
	getc(*»ovladač«*)
	print *»ovladač« »hodnoty«*
	printf *»ovladač« »hodnoty«*
	write *»ovladač«*
uzavření	**close**(*»ovladač«*)

Tabulka 9.3: Práce se soubory

Klíčovým pojmem je tak zvaný *ovladač souboru*. Jedná se o interní proměnnou, která představuje otevřený soubor a jejímž prostřednictvím k němu přistupujete. Ovladače jsou další druh perlivých komponent a opět mají svůj vlastní prostor oddělený od proměnných, polí a dalších kategorií. Důrazně se doporučuje psát jejich názvy velkými písmeny, aby se nepletly s ostatními prvky programu. Nepíší se před nimi totiž žádné paznaky.

Implicitně vám Perl dá do vínku trojici ovladačů, které zpřístupňují standardní soubory každého procesu v Unixu: standardní vstup, výstup a chybový výstup (viz tabulka 9.4). Tyto soubory není třeba otevírat ani zavírat. Vše se děje automaticky, stačí je jen používat (což jsem dosud hojně dělal). Kdybyste je náhodou chtěli otevírat sami, najdete v pravém sloupci tabulky 9.4 „názvy“, které to zajistí.

STDIN	standardní vstup	”–”
STDOUT	standardní výstup	”>–”
STDERR	standardní chybový výstup	

Tabulka 9.4: Standardní ovladače souborů

Ostatní soubory musíte nejprve otevřít. To zajistí funkce **open**, která jako parametry dostane ovladač, jehož prostřednictvím bude se souborem dále zacházeno, a název diskového souboru, který má otevřít. Například:

```
open( KONFIG, ”program.cfg” )
```

Otevře soubor *program.cfg* z aktuálního adresáře a přiřadí jej ovladači KONFIG. Pokud tomuto ovladači již byl přiřazen nějaký soubor, automaticky se uzavře a jeho spojení s ovladačem se zruší.

Veškeré vstupní a výstupní funkce jsou inteligentní. Pokud jim jako první parametr předáte ovladač, budou pracovat s „jeho“ souborem.

Funkce **open** vydá jako svůj výsledek pravdivostní údaj o tom, zda se otevření podařilo. Nabízí se použít ji jako podmínku a podle výsledku pak přizpůsobit další běh programu. Nemožnost otevřít soubor bývá často fatální a život programu bez něj ztrácí smysl. Standardní konstrukcí v takových případech bývá:

```
open( KONFIG, ”program.cfg” )
    or die ”Nemohu otevřít soubor program.cfg!\n”;
```

Funkce **die**[1] způsobí okamžité ukončení programu a do jeho standardního chybového výstupu vypíše hodnoty, které obdrží jako parametry. Já osobně bych raději psal něco jako:

```
if ( not open(KONFIG, ”program.cfg”) ) {
    die ”Nemohu otevřít soubor program.cfg!\n”;
}
```

což je významově totéž, ale konstrukce **open or die** je v perlovské komunitě hluboce zakořeněná a používá se vždy a všude.

Opakem funkce **open** je **close**, která dotyčný soubor uzavře. Jako jediný parametr dostane ovladač souboru, který tím okamžikem de facto zanikne (nebude k němu již připojen žádný soubor, takže případný pokus o jeho použití povede k chybovému hlášení).

Příklad: Udělám si malou statistiku interpretů příkazů, které používají uživatelé mého Unixu. Informace o nich jsou uloženy v souboru */etc/passwd*, jehož strukturu jsem popsal ve cvičení na straně 102. Cesta k interpretu příkazů je poslední položkou na řádku.

shells.pl

```
my %interprety = ();
open(DATA, ”/etc/passwd”)
    or die ”Nemohu otevřít /etc/passwd !\n”;
while ( my $radek = <DATA> ) {
    chomp($radek);
    # interpret je za poslední dvojtečkou
    my ($shell) = $radek =~ m/:([^:]*)$/;
    if ( $shell ne ”” ) {
        $interprety{$shell}++;
    }
}
```

1: Pro angličtiny neznalé: „die“ znamená „zhyň!“

```
close(DATA);

# výstup výsledků
foreach my $shell ( sort keys %interprety ) {
    print "$shell ... $interprety{$shell}\n";
}
```

■

Popsaný způsob otevření souboru umožňuje pouze čtení jeho obsahu. Funkce **open** však dovede víc. Záleží jen na tom, jaký bude její druhý parametr. Přesněji řečeno, jak začíná. Konvence jsou převzaty z přesměrovávání standardních souborů na příkazovém řádku, takže by vám neměly dělat potíže.

Jestliže jako druhý parametr uvedete ">*soubor*", bude dotyčný soubor otevřen pro zápis. Pokud existoval, veškerý jeho dosavadní obsah se vymaže. Příslušný ovladač pak můžete používat ve funkcích **print**, **printf** či **write**. Píše se jako první parametr a od ostatních parametrů se odděluje mezerou, nikoli čárkou.

Existuje i druhý způsob otevření pro zápis, při kterém se stávající obsah souboru zachová a výstupy programu budou přidávány na jeho konec. V takovém případě použijte ve funkci **open** řetězec ">>*soubor*".

Příklad: Myslím, že nastal čas pro krapet složitější a ryze praktický příklad: databázi žáků ve třídě. Pro každého žáka se eviduje jméno a průměrný prospěch. Program umožňuje přidat žáka, vymazat jej, vypsat seznam uspořádaný podle abecedy a podle prospěchu a konečně uložit či načíst třídu.

Základní datovou strukturou je pole *@zaci*. Každému žákovi je přidělena jedna jeho položka, obsahující jméno a průměrný prospěch. Tyto dvě informace jsou navzájem odděleny znakem dvojtečka (který se nevyskytuje ani ve jménech, ani v prospěchu, takže nehrozí nějaká nejednoznačnost).

Svá data program ukládá (je-li požádán) do souboru *trida.dat*. Zároveň vede v souboru *trida.log* protokol o své činnosti, do kterého zaznamenává všechny důležité změny v datech.

trida.pl

```
1  use locale;
2  my $jmenodat = "trida.dat";
3  my $jmenologu = "trida.log";
4  my @zaci = ();
5
6  open(PROTO, ">>$jmenologu");
```

```
7   while ( my $pokyn = dej_znak() ) {
8       if ( $pokyn =~ /n/i ) { pridej_zaka(); }
9       elsif ( $pokyn =~ /d/i ) { smaz_zaka(); }
10      elsif ( $pokyn =~ /j/i ) { seznam_abecedne(); }
11      elsif ( $pokyn =~ /p/i ) { seznam_prospech(); }
12      elsif ( $pokyn =~ /r/i ) { nacti_tridu(); }
13      elsif ( $pokyn =~ /s/i ) { uloz_tridu(); }
14      elsif ( $pokyn =~ /q/i ) { last; }
15      else { print "Neplatný příkaz!"; }
16  }
17  close(PROTO);
18
19  sub dej_znak {
20      print "Zadejte příkaz\n";
21      print " n ... přidat žáka\n";
22      print " d ... vymazat žáka\n";
23      print " j ... seznam žáků abecedně\n";
24      print " p ... seznam žáků podle prospěchu\n";
25      print " r ... načíst z disku\n";
26      print " s ... uložit na disk\n";
27      print " q ... konec\n";
28      print "> ";
29      my $vstup = <STDIN>;
30      return substr( $vstup, 0, 1 );
31  }
32
33  sub do_logu {
34      print PROTO @_;
35  }
36
37  sub pridej_zaka {
38      print "Příjmení a jméno: ";
39      my $jmeno = <STDIN>; chomp($jmeno);
40      print "Průměr: ";
41      my $prumer = <STDIN>; chomp($prumer);
42      push( @zaci, "$jmeno:$prumer" );
43      do_logu( "Přidán žák $jmeno.\n" );
44  }
45
46  sub smaz_zaka {
47      print "Příjmení a jméno: ";
```

```
48          my $jmeno = <STDIN>; chomp($jmeno);
49          for ( my $i=0; $i<@zaci; $i++ ) {
50              if ( $zaci[$i] =~ /^$jmeno:/ ) {
51                  # našel jsem – vymažu a ukončím podprogram
52                  splice( @zaci, $i, 1 );
53                  print "Žák $jmeno vymazán.\n";
54                  do_logu( "Vymazán žák $jmeno.\n" );
55                  return;
56              }
57          }
58          print "Žák $jmeno nenalezen.\n";
59      }
60
61      sub pis_zaka {
62          my ( $zak ) = @_;
63          my ($jmeno,$prumer) = $zak =~ m/(.*):(.*)/;
64          printf "%-30s %0.2f\n", $jmeno, $prumer;
65      }
66
67      sub seznam_abecedne {
68          foreach my $zak ( sort @zaci ) {
69              pis_zaka($zak);
70          }
71      }
72
73      sub seznam_prospech {
74          foreach my $zak ( sort srovnej_prospech @zaci ) {
75              pis_zaka($zak);
76          }
77      }
78
79      sub srovnej_prospech {
80          my ( $prvy, $druhy ) = ( $a, $b );
81          $prvy =~ s/.*://; # odstraním jména a porovnám
82          $druhy =~ s/.*://;
83          return ( $prvy <=> $druhy );
84      }
85
86      sub nacti_tridu {
87          if ( not open(TRIDA, $jmenodat) ) {
88              print STDERR "Nelze načíst soubor $jmenodat !\n";
```

```
89              return 0;
90          }
91          @zaci = ();
92          while ( my $zak = <TRIDA> ) {
93              chomp($zak);
94              push(@zaci, $zak);
95          }
96          close(TRIDA);
97          print "Třída načtena.\n";
98          do_logu( "Načteno ze souboru.\n" );
99      }
100
101     sub uloz_tridu {
102         if ( not open(TRIDA, ">".$jmenodat) ) {
103             print STDERR "Nelze psát do souboru $jmenodat !\n";
104             return 0;
105         }
106         foreach my $zak ( @zaci ) { print TRIDA "$zak\n"; }
107         close(TRIDA);
108         print "Třída uložena.\n";
109         do_logu( "Uloženo do souboru.\n" );
110     }
```

Program používá celkem pět ovladačů souborů. Ze STDIN načítá uživatelské vstupy. STDOUT používá pro výzvy a oznamování výsledků. Funkce **print**, která se vyskytuje u práce se soubory, svůj výstup pošle do STDERR (a budeme všichni rádi, když k tomu pokud možno nedojde). Dále zde existuje ovladač TRIDA, který je otevírán vždy jen dočasně během načítání či ukládání třídy. A konečně PROTO, který zastupuje protokol o činnosti programu. Je otevřen trvale a zapisuje do něj podprogram *do_logu*. ■

Někdy si chcete odložit mezivýsledky do pracovního souboru. Ovšem může se stát (například když program dáte k dispozici všem uživatelům či jej startuje WWW server), že bude spuštěno několik exemplářů zároveň. Bylo by velmi záhodno, aby měl každý svůj vlastní pracovní soubor. Jinak hrozí nepříjemné kolize.

Ideálním nástrojem pro řešení tohoto problému je proměnná *$$*. Obsahuje identifikační číslo (PID) běžícího programu. Tato čísla přiděluje operační systém a zároveň ručí za jejich jednoznačnost – nemůže se stát, že by současně běžely dva programy se stejným PID. Takže problém jednoznačnosti snadno vyřeší například:

```
open( TMP, ">/tmp/$$.data" ) or die "…";
```

Nezapomeňte před skončením programu vymazat pracovní soubor. Zanedlouho se dozvíte jak.

Důležitou skutečností je, že pojmenované ovladače souborů jsou globální. Kdykoli kdekoli v programu použiji například ovladač *TRIDA*, bude se vždy pracovat se souborem, který na něj navázalo poslední otevření. Proto například následující program je zcela korektní a zapíše do souboru *yyy.txt*:

```
sub otevri {
    open( SOUBOR, ">>yyy.txt" )
        or die "Nelze zapisovat do souboru yyy.txt\n";
}

otevri();
print SOUBOR "Tohle sem zapsal proces $$\n";
```

Soubor se otevírá v podprogramu, nicméně ovladače jsou globální, takže podprogram sice skončí, ale *SOUBOR* žije dál a dále je navázán na soubor *yyy.txt* otevřený pro zápis. Díky tomu **print** volaný po skončení podprogramu do něj bez problémů zapíše.

Pokud by vám to vadilo a chtěli jste se pojistit proti tomu, aby podprogramu do jeho otevřeného souboru zasahoval někdo zvenčí, ovladač nelze omezit. Pokus o něco jako:

```
my SOUBOR;
```

skončí hlášením o syntaktické chybě. Nicméně dá se to obejít použitím anonymního ovladače. Jestliže při otevírání jako první argument použijete proměnnou, vytvoří se pro otevřený soubor anonymní ovladač a uloží se do ní. Daná proměnná pak představuje jedinou cestu, žádný jiný způsob, jak se dostat k ovladači, neexistuje. A proměnnou už omezit jde.

Příklad: Vytvořím podprogram *pripis*, který dostane jméno souboru a nějaký text, který má připsat na jeho konec. Používá k tomu popsanou metodu, kdy si pro práci se souborem vytváří anonymní ovladač a ukládá jej do lokální proměnné *$OVLADAC*. Díky tomu má jistotu, že svou činností nenaruší práci se soubory v jiné části programu a pokud by se třeba později rozkošatil a sám volal jiné podprogramy, ty zase nemohou zasahovat do jeho souboru.

lokovladac.pl

```
1  sub pripis {
2      my( $soubor, $obsah ) = @_;
3      open( my $OVLADAC, ">>$soubor" )
4          or die "Nelze přidávat do souboru $soubor\n";
5      print $OVLADAC $obsah;
6      close( $OVLADAC );
7  }
```

```
8
9     pripis( "yyy.txt", "A teď jsem přidal ještě něco!!!\n" );
```

Všimněte si, že při zápisu do souboru na řádku 5 není mezi proměnnými čárka, stejně jako není mezi pojmenovaným ovladačem a zapisovaným obsahem v příkazech **print** příkladu se třídou. Proměnná zde prostě představuje ovladač se vším všudy. ■

Perl také nabízí sadu operátorů, kterými si můžete o souborech zjistit řadu informací – zda existují, jsou zapisovatelné, jedná se o soubor nebo adresář a podobně. Za inspiraci si vzal interpret příkazů, který v tomto směru dovede ledacos. A když inspirace, tak totální. Zlí jazykové by řekli, že Perl to prostě opsal. My hodní jazykové víme, že se pouze snaží být konzistentní. Nejdůležitější testy obsahuje tabulka 9.5.

–r	soubor lze číst	**–f**	je to soubor
–w	do souboru lze zapisovat	**–d**	je to adresář
–x	soubor je spustitelný	**–l**	je to symbolický odkaz
–e	soubor existuje	**–z**	soubor má nulovou délku
		–s	soubor má nenulovou délku

Tabulka 9.5: Testování souborů

Všechny tyto testovací operátory jsou unární a jako jediný argument dostanou jméno souboru. Používají se v podmínkách. Například:

```
if ( –r "/etc/resolv.conf" ) { … }
```

ověří, zda existuje soubor */etc/resolv.conf* a zda program má právo číst jeho obsah. Místo jména můžete jako operand předhodit i ovladač souboru, ovšem to už musí být otevřen a tehdy bývá na testování dost pozdě.

Příklad: Hezkým příkladem využití této vlastnosti je sestavování seznamu domácích stránek uživatelů pro WWW server. V Unixu si každý uživatel může ve svém domácím adresáři vytvořit podadresář (v našem případě se jménem *html*) a do něj uložit své osobní stránky. Jejich vstupním souborem musí být *index.html*. Z hlediska WWW se k těmto stránkám přistupuje prostřednictvím lokátoru */~uživatel/*.

Bývá zvykem zpřístupnit na WWW serveru seznam uživatelů, kteří zde mají své osobní stránky. Je tedy třeba sestavit seznam těch uživatelů, kteří ve svém domácím adresáři mají soubor *html/in-*

dex.html. Následující program to udělá. Doufám, že se alespoň trochu vyznáte v HTML. Pokud ne, zkuste si přečíst knihu [8]. Určitě vás to chytne.

htmluz.pl

```
my $htmlindex = "/html/index.html";
my $vystup = "seznamuziv.html";

open(DATA, "/etc/passwd")
    or die "Nemohu otevřít /etc/passwd !\n";
open(VYSTUP, ">$vystup")
    or die "Nemohu zapisovat do $vystup !\n";
print VYSTUP "<ul>\n";
while ( my $radek = <DATA> ) {
    my @uziv = split( /:/, $radek );
    my ($login,$jmeno,$home) = ($uziv[0],$uziv[4],$uziv[5]);
    if ( -r "$home$htmlindex" ) {
        if ( $jmeno ) {
            pis_uzivatele($jmeno, "/~$login/");
        } else {
            pis_uzivatele($login, "/~$login/");
        }
    }
}
print VYSTUP "</ul>\n";
close(VYSTUP);
close(DATA);

sub pis_uzivatele {
    my ( $jmeno, $url ) = @_;
    print VYSTUP "<li><a href='$url'>$jmeno</a></li>\n";
}
```

Program vytváří jen holý základ stránky – vlastní seznam uživatelů. K tomu, aby se z něj stala skutečná stránka, je třeba jej obalit patřičnými hlavičkami, patičkami a podobným materiálem. Na jeho konstrukci však z hlediska Perlu není nic zajímavého. ■

9.4 Zpátky na stromy (adresářové)

V souvislosti se soubory nelze neuvažovat o souborech a adresářích operačního systému. Například si vypsat obsah určitého adresáře či smazat existující soubor. Základní sadu funkcí zajišťujících práci se soubory operačního systému obsahuje tabulka 9.6. Znalci Unixu snadno poznají své staré známé.

unlink(*»seznam«*)	vymaže soubory
rename(*»staré«*, *»nové«*)	přejmenuje/přesune soubor
chmod(*»práva«*, *»seznam«*)	nastaví přístupová práva
chown(*»uid«*, *»gid«*, *»seznam«*)	změní vlastníka
link(*»existující«*, *»odkaz«*)	vytvoří pevný odkaz
symlink(*»existující«*, *»odkaz«*)	vytvoří symbolický odkaz
chdir(*»jméno«*)	změní aktuální adresář
mkdir(*»jméno«*, *»práva«*)	vytvoří adresář
rmdir(*»jméno«*)	odstraní adresář

Tabulka 9.6: Souborové funkce

Pro práci s adresáři máte k dispozici hned několik cest. Tou nejserióznější je použití trojice funkcí **opendir**, **readdir** a **closedir**, které jsou přímou analogií trojice **open**, <> a **close** pro soubory. Adresář nejprve otevřete, pak z něj čtete položky a nakonec zase zavřete.

Funkce **readdir** se podobá diamantovému operátoru i svou závislostí na kontextu. Pokud ji zavoláte ve skalárním kontextu, vydá jméno jednoho souboru (a při následujícím volání dalšího…). Při volání v seznamovém kontextu vydá seznam všech souborů, které obsahuje.

Příklad: Udělám primitivní napodobeninu programu *tree*. Vypíše strom souborů začínající v aktuálním adresáři. Vzájemné vnoření souborů a adresářů vyznačí délkou odsazení od levého okraje.

tree.pl

```
1   zpracuj_adr( ".", "" );
2
3   sub zpracuj_adr {
4       my ( $adresar, $mezery ) = @_;
5
6       if ( not opendir(ADR, $adresar) ) {
7           print STDERR "Nelze otevřít $adresar \n";
8           return 0;
9       }
10      my @soubory = readdir(ADR);
11      closedir (ADR);
12
```

```
13          foreach my $jmeno ( sort @soubory ) {
14              if ( $jmeno =~ /^\./ ) { next; }
15              my $cesta = $adresar . "/" . $jmeno;
16              if ( -d $cesta ) {
17                  print "$mezery$jmeno/\n";
18                  zpracuj_adr( $cesta, $mezery." " );
19              } else {
20                  print "$mezery$jmeno\n";
21              }
22          }
23      }
```

■

Funkce **readdir** poctivě vrací všechny položky v adresáři včetně skrytých, a to v jejich syrovém pořadí. Tedy bez jakéhokoli řazení. Proto jsem v příkladu musel použít **sort** a ručně vynechávat soubory, jejichž jméno začíná tečkou.

Cvičení 9.2: Všimněte si, jak v předchozím příkladu střídám pouhé jméno souboru při výstupech (řádek 17) s cestou k němu při otevírání a testování adresářů (řádky 15, 16, 18). Pokuste se tuto nejednotnost odstranit za pomoci přepínání aktuálního adresáře. ■

Pokud se vám zastesklo po žolíkových znacích, klidně zamáčkněte slzu. Perl totiž nabízí alternativní způsob zacházení s adresáři – funkci **glob**. Příkaz:

```
@soubory = glob( "*.c" );
```

přiřadí do pole *@soubory* abecedně uspořádaný seznam všech souborů z aktuálního adresáře, jejichž jména končí příponou *.c*. Lze použít i zápis:

```
@soubory = <*.c>;
```

Z toho vidíte, že diamantový operátor je pěkný chameleón. Pokud mu předložíte ovladač souboru nebo proměnnou, která obsahuje jméno ovladače, bude číst obsah dotyčného souboru. Dostane-li jiný parametr, vydá seznam souborů, jejichž jména mu vyhovují. Právě kvůli této nejednoznačnosti vám doporučuji dávat přednost funkci **glob**.

Díky ní bych mohl příklad dále zestručnit: řádky 6 až 11 by se srazily do jediného:

```
@soubory = glob( "*" );
```

Na řádku 13 bych mohl vynechat **sort** a řádek 14 vypustit docela.

Zavoláte-li **glob** ve skalárním kontextu, chová se, jak je v kraji zvykem. Při každém volání poskytne jméno jednoho vyhovujícího souboru. Až všechny vyčerpá, vydá **undef**. Jedinou nevýhodou je jistá neefektivita. K realizaci funkce **glob** se totiž volá interpret příkazů. Pokud je rychlost vaším cílem, dejte přednost **readdir**.

Doposud jsem psal výlučně o jménech souborů a vůbec mne nezajímaly další informace o nich: jak jsou velké, komu patří, kdy byly naposledy změněny a podobně. Nastal čas podívat se jim trochu na zoubek.

V Unixu jsou veškeré informace o souboru (s výjimkou jména) uloženy v datové struktuře zvané I-uzel. Perl ji zpřístupňuje prostřednictvím funkce **stat**. Jako parametr dostane soubor (jméno nebo ovladač) a odmění se polem s třinácti atraktivními prvky – viz tabulka 9.7.

index	zkratka	význam
0	dev	číslo zařízení systému souborů
1	ino	číslo I-uzlu
2	mode	typ souboru a přístupová práva
3	nlink	počet pevných odkazů na soubor
4	uid	vlastník souboru
5	gid	skupina vlastnící soubor
6	rdev	identifikátor zařízení (jen u speciálních souborů)
7	size	velikost (v bajtech)
8	atime	čas posledního přístupu
9	mtime	čas poslední změny souboru
10	ctime	čas poslední změny I-uzlu
11	blksize	doporučená velikost bloku
12	blocks	počet alokovaných bloků

Tabulka 9.7: Výsledky funkce **stat**

Příklad: Tento velmi jednoduchý program vypíše seznam souborů v aktuálním adresáři uspořádaný vzestupně podle jejich velikosti.

lssize.pl

```
my %soubory = ();
foreach my $soubor ( glob("*") ) {
    my @iuzel = stat($soubor);
    $soubory{$soubor} = $iuzel[7];
}

foreach my $soubor (
  sort {$soubory{$a}<=>$soubory{$b}} keys %soubory ) {
    printf "%12d %s\n", $soubory{$soubor}, $soubor;
}
```

Používá lehce kuriózní, leč funkční způsob uložení dat: informace o souborech si ukládá do asociativního pole, kde jméno souboru je klíčem a jeho velikost hodnotou. Podle toho pak vypadá porovnání hodnot ve funkci **sort**. ■

9.5 Zamykání souborů

Dokud si programujete sami pro sebe, zpravidla se dokážete celkem dobře uhlídat a nehrozí vám, že byste si současně spustili několik programů, které budou jeden druhému pod rukama měnit datové soubory. Ovšem jakmile své programy dáte veřejně k dispozici nebo budete programovat pro WWW, vyvstane problém sdílení.

Například se uživatel prostřednictvím WWW formuláře přihlásil na seminář a obslužný program jej má přidat do seznamu účastníků. Nikdo vám však nezaručí, že se nepřihlásí několik uživatelů současně, což způsobí, že spustí paralelně několik exemplářů obslužného programu. Ty všechny budou zapisovat do stejného seznamu účastníků.

Nebude-li váš program proti podobným situacím zabezpečen, může docházet k chybám. Některé záznamy se ztratí nebo se dokonce může stát, že dojde k nenapravitelnému poškození celého souboru. Lék na tuto bolest se nazývá zamykání souborů a provádí se standardní funkcí:

flock(*»soubor«*, *»režim«*)

Prvním parametrem je ovladač zamykaného souboru, druhým přístupový režim (typ zámku), který mu chcete nastavit. Zámek by se zhruba dal přirovnat k dopravnímu semaforu. Signalizuje, zda je dotyčný soubor volný nebo nikoli. A stejně jako semafor funguje jen tehdy, pokud na něj ostatní berou ohled. **flock** tedy dokáže zablokovat jen volání **flock** v dalších programech, nikoli vlastní operace se souborem.

Pokud budete psát sadu programů využívajících zamykání, musíte zamykat všude. Když u některého zapomenete použít **flock**, ztratí celé vaše snažení smysl.

»režim« zámku je určen několika bity. Jejich hodnoty, symbolické názvy a významy shrnuje tabulka 9.8. Jednotlivé hodnoty sice můžete vzájemně kombinovat (bitové „nebo“ či prostý součet), jejich význam to však zpravidla vylučuje. Jedinou výjimkou je neblokující zamčení, které můžete připojit k jednomu z prvních dvou zámků.

1	LOCK_SH	sdílený zámek (pro čtení)
2	LOCK_EX	výlučný zámek (pro zápis)
4	LOCK_NB	neblokující zamčení
8	LOCK_UN	uvolnění zámku

Tabulka 9.8: Režimy zamykání souboru

⚡ Číselné hodnoty zamykacích režimů se mohou lišit v závislosti na operačním systému.

Sdílený zámek umožňuje, aby dotyčný soubor otevřelo několik programů současně. Používá se, pokud chcete pouze číst. Chrání před tím, aby někdo během vašeho čtení změnil obsah souboru. K zápisu by si totiž měl vyžádat výlučné zamčení. Pokud je získá, znamená to, že soubor má momentálně uzamčen pouze a jedině on a že změny, které s ním provede, nezmatou žádný jiný program.

Zamykání je blokující operací. To znamená, že pokud program nemůže soubor zamknout v požadovaném režimu, bude operačním systémem pozastaven a vzbudí se až po úspěšném dokončení operace.

Příklad: Sestavím dva jednoduché programy, které nedělají nic užitečného. Zato intenzivně zamykají a odemykají soubor. První z nich čte jeho obsah. Číst data může spousta programů zároveň aniž by se navzájem rušily. Proto zde použiji sdílený zámek (hodnota 1).

ctenar.pl

```
1  open(DATA, ”program.log”)
2       or die ”Nelze otevřít soubor program.log\n”;
3  flock(DATA, 1) or die ”Nelze zamknout soubor\n”;
4  while ( my $radek = <DATA> ) {
5       print $radek;
6  }
7  flock(DATA, 8);
8  close(DATA);
```

Jak vidíte, soubor se nejprve otevře a zamyká se až před vlastním přístupem k němu (na řádku 3). Jakmile je vstupně/výstupní operace dokončena, můžete zase odemknout (řádek 7). Mimochodem – pokud hodláte soubor uzavřít, je odemykání zbytečné. **close** totiž automaticky uvolní ja-

kýkoli zámek, který váš program na daném souboru nastavil. Této vlastnosti využívá následující program, který do souboru zapisuje:

pisec.pl

```
open( DATA, ">>program.log" )
    or die "Nelze otevřít soubor program.log\n";
flock( DATA, 2 ) or die "Nelze zamknout soubor\n";
print DATA "Tohle sem zapsal proces $$\n";
close( DATA );
```

Tentokrát se používá výlučný zámek. V jednom okamžiku může mít soubor *program.log* otevřeno buď libovolné množství programů *ctenar* nebo jediný *pisec*. ■

Zamykat je třeba s rozumem. Především se musíte vyhnout tomu, aby některý z programů zamkl datový soubor dlouhodobě. Pokud například sestavíte program, ve kterém bude uživatel pracovat s daty v paměti a jen tu a tam je načte z disku či uloží, mělo by se zamykat jen v těchto načítacích/ukládacích podprogramech. V žádném případě není vhodné, aby si interaktivní program zamkl data při svém startu a odemkl je až když končí.

Zamykáte-li více souborů, tiše se vám pod židlí stulí strašidlo uváznutí. Nejjednodušší uváznutí vypadá takhle: Program *A* si výlučně zamkne soubor *x* a program *B* si výlučně zamkne soubor *y*. Nyní program *A* požádá o uzamčení souboru *y*. To momentálně není možné, takže musí čekat, až jej program *B* uvolní. Ten však při své další práci chce zamknout soubor *x* a musí čekat, až jej *A* odemkne. Jestli neumřeli, čekají dodnes.

Před uváznutím se dá chránit. Například tak, že soubory budete zamykat vždy ve stejném pořadí a odemykat v obráceném. Nebo budete přístup k celé skupině souborů vázat jediným zámkem. Když chce některý program přistupovat k libovolnému souboru, musí zamknout tento jediný zámek. Existují i chytřejší a efektivnější metody. Dočtete se o nich v každé slušné knize věnované teorii operačních systémů či paralelnímu programování.

Až se v příští kapitole naučíte používat moduly, můžete si nahradit číselné konstanty textovými. Modul *Fcntl* totiž nabízí symbolická jména vyjadřující jednotlivé režimy zamykání (jsou uvedena v prostředním soupci tabulky 9.8. Stačí použít:

```
use Fcntl qw(:DEFAULT :flock);
```

a můžete psát věci jako:

```
flock( DATA, LOCK_EX );
```

Část III

Na hranicích Perlu

Třetí část pojednává o pokročilejších programátorských technikách a také o styku Perlu s jeho okolím.

Velmi důležitá je hned první kapitola o modulech, jejich vytváření a používání. Prostřednictvím modulů totiž můžete snadno a rychle rozšiřovat schopnosti jazyka v těch směrech, ve kterých to potřebujete. Perliví programátoři zastávají názor, že byste pokud možno měli psát moduly do svých programů tak, aby byly později znovu použitelné při řešení dalších problémů. Hovoří v této souvislosti o ekologickém programování.

Následují odkazy, jejichž prostřednictvím můžete vytvářet libovolně složité datové struktury. Kniha pokračuje kapitolou o komunikaci vašeho programu s uživatelem a s prostředím, které jej obklopuje.

Objektově orientované programování je další z velkých témat současnosti. Dozvíte se zde, jak tyto techniky používat v Perlu a co od nich můžete očekávat. V kapitole o funkcionálním programování si pak trochu zažonglujeme s funkcemi.

Závěrečná dvojice kapitol je věnována práci s databázemi a vytváření programů, které se používají ke zpracování informací pro web. Právě na tomto poli si Perl vydobyl velmi silnou pozici.

10 Moduly

Nenamoduluje-li mne Julie, namoduluji ji já. Protože moduly jsou moderní. Moduly jsou módní.

Moduly umožňují rozdělit zdrojový text vašeho programu do několika značně nezávislých částí. To se uplatní v řadě případů: U velkých projektů, aby velikost souborů nepřesáhla rozumnou mez. U týmových projektů, kdy různí programátoři pracují na různých částech programu. A především při využívání celé řady již existujících modulů.

Znalci tvrdí, že samotný Perl je jen polovina zábavy. Zbytek tvoří využívání početného množství modulů, které lze získat v archivu CPAN (viz příloha 18 na straně 275), případně i jinde.

V této kapitole vás seznámím se základními mechanismy, které budete při využívání a vytváření modulů potřebovat.

10.1 Balíky

Jakmile na programu pracuje několik lidí nebo chcete použít nějakou existující knihovnu, objeví se problém s identifikátory. Jak se vyhnout jejich kolizím? Aby programátor Vomáčka nepoužil pro svůj pomocný podprogram vyzvedávající data z databáze název *vyzvedni*, protože ho již použila programátorka Svíčková pro svůj podprogram realizující vyzvednutí zboží ze skladu.

Pokud si vzpomínáte, tenhle problém jste už jednou viděli. Tehdy se jednalo o názvy proměnných v podprogramech a jeho řešením bylo zavedení lokálních identifikátorů. Podobné řešení existuje i zde. Nazývá se balík (package).

Pomocí balíků lze identifikátory rozdělit do izolovaných skupin, oficiálně se jim říká *jmenné prostory*. Nerad to přiznávám, ale až dosud jsem vám vlastně lhal. Ve skutečnosti se *každý* identifikátor skládá ze dvou částí: jména balíku a vlastního jména. Navzájem se oddělují dvojicí dvojteček. Například název proměnné má plný tvar:

$»balík«::»proměnná«

Každý balík je světem sám pro sebe a identifikátory v něm nemají nic společného s identifikátory ostatních balíků. Takže pokud bude programátor Vomáčka realizovat balík *databaze* a programátorka Svíčková balík *sklad*, nic jim nebrání, aby si zavedli podprogramy *databaze::vyzvedni* a *sklad::vyzvedni*. Nebudou si nijak překážet.

Abyste se neupsali, zavádí Perl příkaz:

package *»jméno«*;

Od okamžiku jeho použití se *»jméno«* stává implicitním jménem balíku. Pokud u identifikátoru neuvedete balík, automaticky se mu předřadí „*»jméno«::*“. Příkaz **package** platí tak dlouho, dokud nenastane jedna z následujících situací:

- použijete další **package**, kterým nastavíte jiný implicitní balík,
- příkaz **package** byl použit uvnitř bloku a tento blok skončil nebo
- skončil soubor, ve kterém byl **package** použit.

Implicitním balíkem je *main*. Takže všechny identifikátory, o kterých jsem se dosud zmiňoval, měly plný název *main::cosi*. Viděli jste to třeba v ukázce chybového hlášení na straně 34.

Příklad: Jednoduchý program:

```
1  package Data;
2  $cena = 23.50;
3
4  package main;
5  $pocet = 10;
6  $cena = $Data::cena * $pocet;
7
8  print "Celková cena: $cena (s DPH ";
9  print $cena*1.22, ")\n";
```

zavádí dva balíky (*Data* a *main*) a celkem tři proměnné (*$Data::cena*, *$main::pocet* a *$main:cena*). Skutečnost, že proměnná *$cena* existuje v obou balících není nijak na závadu. Jen když chci použít identifikátor z jiného balíku, musím to explicitně vyjádřit, jako na řádku 6. ■

Vřele vám *ne*doporučuji střídat několik balíků v rámci jednoho souboru. Je dobrým zvykem použít **package** jen jednou, a to na začátku souboru. Pokud si zafixujete tuto konvenci, výrazně si zjednodušíte posuzování, do kterého balíku patří identifikátor, který se vyskytuje kdesi daleko ve zdrojovém textu.

10.2 Moduly

Balíky jsou jen prvním krokem na cestě k vícehlavým programátorským týmům. Umožňují, aby se programátoři nepoprali o identifikátory. To pravé ořechové však přinášejí až moduly.

Modul je logicky ucelená skupina proměnných a podprogramů, která řeší určitou část problému. Například budete mít jeden modul definující globální datové struktury, další modul pro uživatelské rozhraní, další pro práci s databází a tak dále a tak podobně.

Neexistuje jednoznačný a snadno popsatelný recept, co všechno zahrnout do jednoho modulu. Pouze doporučení, že by měl být logicky ucelený a do značné míry samostatný. Často to znamená, že se v jednom modulu nacházejí určité datové struktury společně s jejich obslužnými podprogramy.

Modul je uložen v jednom souboru. Jeho jméno je totožné se jménem modulu a povinně má příponu *.pm* (Perl Module). Bývá zvykem zahajovat jména modulů velkými písmeny. Například modul *Databaze* by byl uložen v souboru *Databaze.pm*. Chcete-li ve svém programu použít existující modul, zajistíte to příkazem:

use *»modul«*;

Tento nejzákladnější mechanismus pro práci s moduly sám o sobě nezajišťuje skoro nic, jak předvede následující příklad.

Příklad: Mám soubor *Data.pm*, který obsahuje:

```
$a = 3;
$b = 2;
1;
```

Všimněte si posledního řádku, který je charakteristickým znakem valné většiny modulů. Součástí modulu může být jeho inicializace – nastavení proměnných a podobné kroky. Hodnota posledního provedeného příkazu je pak interpretována jako návratový kód vložení modulu, kterým lze signalizovat, zda vše dobře dopadlo.

Jelikož většina modulů ani nemůže nedopadnout, obsahují na svém konci tento „příkaz“, ohlašující úspěšné vložení. Zvykněte si jej psát zcela automaticky. Vyhnete se tak nepříjemným chybovým hlášením, ke kterým by jinak mohlo docházet.

Následující program vloží soubor *Data.pm* a použije v něm definované proměnné:

```
use Data;

print "$a x $b = ", $a*$b, "\n";
```

Výsledek bude strhující: 3 × 2 = 6. ■

Jak je vidět, základní mechanismy samy o sobě neposkytují oddělování prostorů pro proměnné ani žádné další pokročilejší služby. V podstatě jen zajistí vložení souboru[1]. Proto nedílnou částí práce s moduly je jistá sebekázeň a dodržování obecných konvencí. Moduly sice lze psát i bez nich (bohužel), ale pak budete mít problémy při spolupráci s ostatními programátory. Všichni totiž očekávají určité obvyklosti. Je záhodno se jich držet.

Slušný modul (a žádné jiné pokud možno nepište) obsahuje následující prvky:

- Jako implicitní nastaví balík stejného jména jako modul (jména se skutečně *musí* shodovat, jinak obdržíte zdánlivě nepochopitelné chyby).
- Prostřednictvím standardního modulu *Exporter* nadefinuje své rozhraní – které prvky jsou určeny pro veřejnost.
- Dále následuje vlastní tělo modulu; v něm jsou realizovány výše nabídnuté prvky.
- Skončí signalizací úspěšného vložení – zpravidla příkazem „1;“.

Základní myšlenka stojící v pozadí modulů by se dala nazvat „jsou věci veřejné a jsou věci soukromé“. Modul svému okolí nabízí určité služby – proměnné či podprogramy. Tento souhrn nabízených služeb se nazývá rozhraní modulu.

Mělo by být popsáno v dokumentaci. Z ní se potenciální uživatel například dozví, že váš modul *Sklad* nabízí funkci *vyzvedni*, jejímiž parametry jsou identifikace zboží a počet kusů, které chcete vyzvednou. Jejím výsledkem je počet skutečně vyzvednutých kusů. Analogická funkce *uloz* bude zboží do skladu naopak přidávat. Znalost těchto „veřejných služeb“ je vlastně vše, co potřebujete k používání modulu.

Druhou věcí je jejich implementace. Jak je uložen stav skladu – zda se jedná o asociativní pole, o databázi, či o snímání výstupů jakýchsi čidel, která jsou umístěna ve skutečném fyzickém skladu a ve spojení s umělou inteligencí se určují identifikace a počty zboží… Tohle jsou interní záležitosti modulu *Sklad* a ostatním částem programu po nich vůbec nic není.

Tento přístup, kdy okolí zajímá jen rozhraní a implementace je interní záležitostí, má obrovskou výhodu: pokud se rozhodnete změnit implementaci (například přejít od primitivního asociativního pole ukládaného do běžného textového souboru k SQL databázi), ale zachováte stávající rozhraní, pro okolí modulu se nic nezmění. Používá jeho služby stejně, jako je používalo dříve, a vůbec netuší, že volání funkce *Sklad::uloz* vyvolá úplně jinou sekvenci operací než před týdnem.

Dobrá, dobrá. Dost už teorie a konečně prozradím, jak se to v praxi dělá. Pro definici rozhraní se používá standardní modul *Exporter* a jeho funkce *import*, která se v Perlu automaticky volá,

1: Později uvidíte, že přeci jen dělají trochu víc.

když modul použijete. Nejvýznamnější informací je pole *@EXPORT*, do kterého přiřadíte jména proměnných a podprogramů, které daný modul nabízí svému okolí.

Příklad: Zmínil jsem se o modulu *Sklad*. Soubor *Sklad.pm*, který jej obsahuje, by vypadal zhruba takto:

Sklad.pm

```
package Sklad;

# definice rozhraní pomocí Exporter
use Exporter 'import';
our @EXPORT = qw(uloz vyzvedni);

# implementace
sub uloz { ... }
sub vyzvedni { ... }

1;
```

Klíčové slovo **our** zpřístupní pole *@EXPORT* ostatním částem programu. Zatím je berte jako dogma, dostaneme se k němu později. Hodnotou pole *@EXPORT* je seznam seznam všech nabízených identifikátorů (prostřednictvím standardní funkce **qw**). ■

A proč vlastně to všechno? Výsledkem je, že uživatel modulu získá přímý přístup k prvkům rozhraní, aniž by musel používat identifikátor balíku. Pokud tedy ve svém programu použiji:

```
use Sklad;
```

mohu psát rovnou *uloz* a nemusím se obtěžovat se *Sklad::uloz*. Zde se totiž projeví druhá funkce příkazu **use**. Kromě toho, že vloží modul do programu, zároveň vytvoří „zkratky" ke všem identifikátorům uvedeným v poli *@EXPORT*. Toto je jádro činnosti funkce *import* z modulu *Exporter*.

Možná si teď významně poklepáváte na čelo a brumláte cosi o vývoji v kruhu. Zavedli jsme balíky, abychom oddělili adresní prostory identifikátorů, a teď se namáháme s jakýmsi *Exporterem*, abychom je zase slili dohromady. Ovšem my, kteří jsme byli oblaženi marxisticko-leninským vzděláním, víme, že vývoj vždy probíhá ve spirále. Pouze zarytí revizionisté, trockisté, bernštajnovci či jiní odpadlíci od společnosti by, soudruzi, mohli tvrdit, že jsme tam, kde na začátku! (potlesk)

Kouzlo spočívá v tom, že modul *Exporter* zpřístupní jen vybrané prvky z modulu. Typický modul bývá často poměrně rozsáhlý s řadou pomocných proměnných a podprogramů. Použitím *Exporteru* z něj zveřejníte jen několik málo identifikátorů, které jsou určeny k veřejnému použití.

Udržet i v rámci rozsáhlého projektu jednoznačnost v názvech takto exportovaných identifikátorů by nemělo být problémem. Pokud přece jen dojde k tomu, že dva různé moduly nabídnou stejnojmenný identifikátor (to se může stát například při použití modulů získaných v Internetu a vytvářených nezávislými autory), lze stále použít konvenci *Modul::jméno* k jejich rozlišení.

Aby byla věc ještě trochu veselejší, mohou se v názvech modulů vyskytovat dvojice dvojteček. Například *$Karty::Marias::trumfy* označuje proměnnou *$trumfy* z modulu *Karty::Marias*. Při převodu názvu modulu na jméno souboru, který jej obsahuje, bude každá dvojice dvojteček převedena na znak sloužící v daném systému k oddělování adresářů (pro Unix `/`, pro Windows \). Takže zmiňovaný modul se bude hledat v souboru *Karty/Marias.pm*.

A kde že se vlastně moduly vyhledávají? Rozhoduje o tom obsah standardního pole *@INC*. Obsahuje seznam adresářů. Kdykoli použijete příkaz **use**, Perl postupně vyhledává soubor *»modul«.pm* ve všech adresářích obsažených v *@INC*. První, který najde, použije. Pokud se nepodaří soubor najít, obdržíte chybové hlášení:

```
Can't locate neni.pm in @INC
```

Jeho součástí je i stávající obsah tohoto pole. V mém případě se moduly hledají v následujících adresářích:

/etc/perl
/usr/local/lib/x86_64-linux-gnu/perl/5.26.1
/usr/local/share/perl/5.26.1
/usr/lib/x86_64-linux-gnu/perl5/5.26
/usr/share/perl5
/usr/lib/x86_64-linux-gnu/perl/5.26
/usr/share/perl/5.26
/usr/local/lib/site_perl
/usr/lib/x86_64-linux-gnu/perl-base

Jak je vidět, jedná se o jakési globální knihovní adresáře. Dříve seznam obsahoval i aktuální adresář, verze 5.26 této praxi učinila z bezpečnostních důvodů přítrž[2]. Pokud chcete do *@INC* přidat další prvek, použijte příkaz:

use *lib »adresář«*;

2: Kdyby vám to příliš komplikovalo život, lze jej vrátit zpět nastavením proměnné prostředí `PERL_USE_UNSAFE_INC` na hodnotu 1.

Například zmíněný aktuální adresář přidáte pomocí:

```
use lib '.';
```

Perl vloží uvedený »*adresář*« na začátek pole *@INC*. Navíc zkontroluje, zda v něm neexistuje podadresář, jehož jméno odpovídá názvu vaší platformy (v mém případě *x86_64-linux-gnu*). Pokud ano, vloží tento podadresář do *@INC* také, opět na začátek. Předpokládá se totiž, že v tomto adresáři budou moduly závislé na dané platformě, které by měly mít přednost před moduly obecnými.

Po použití **use** *lib* budou všechny následující příkazy **use** hledat nejprve ve vámi uvedeném adresáři.

Příklad: Předložit v knížce smysluplný příklad programu rozděleného na moduly je vždycky oříšek. Mají totiž rozumný smysl především u rozsáhlých problémů. Na ty však v publikaci tohoto charakteru není místo (nemohu si dovolit příklad na dvacet stran, který bych navíc musel na dalších deseti vysvětlovat). U relativně jednoduchých příkladů pak moduly nutně budí dojem zbytečné komplikace. Chápejte proto prosím následující příklad skutečně jako ilustraci modulů, nikoli jako návod typu „kdykoli máte dva tematicky související podprogramy, měli byste je zařadit do modulu".

Nejrozsáhlejším dosavadním příkladem byla třídní agenda. Proto se k ní vrátím a zkusím ji rozdělit do modulů. Když se zamyslím nad operacemi, které jsem při jejím řešení prováděl, celkem přirozeně se vyloupnou dva tematicky uzavřené okruhy problémů: komunikace s diskovými soubory (ukládání a načítání dat) a uživatelské rozhraní. Toto jsou ideální kandidáti na moduly.

Třetí je poněkud méně nápadný – vedení protokolu o činnosti programu. Všimněte si, že zápisy do protokolu jsou roztroušeny po celém zdrojovém textu. Proto se bude hodit, když budu mít modul, který jednoduše použiji a dá mi k dispozici podprogram *do_logu*.

Při konstrukci modulů se zároveň budu snažit o vyšší míru jejich soběstačnosti a použitelnosti i v jiných programech. V případě modulu *Log* to znamená, že do něj zařadím i podprogramy pro otevření a uzavření protokolového souboru:

Log.pm

```
package Log;

use Exporter 'import';
our @EXPORT = qw(otevri_log zavri_log do_logu);

my $log_otevren = 0;
my $PROTO;

sub otevri_log {
    my ( $jmeno ) = @_;
```

```
    if ( open($PROTO, ">>$jmeno") ) {
        $log_otevren = 1;
        return 1;
    } else {
        print STDERR "Log::Nemohu otevřít soubor $jmeno !\n";
        return 0;
    }
}

sub zavri_log {
    if ( $log_otevren ) { close($PROTO); }
    $log_otevren = 0;
}

sub do_logu {
    if ( $log_otevren ) { print $PROTO @_; }
}

1;
```

Dalším modulem bude *Trida::Soubory*, který dostane na starost práci s diskovými soubory (*nacti_tridu* a *uloz_tridu*). Abych dosáhl univerzálnější použitelnosti těchto podprogramů, zruším jejich přímou práci s globálním polem *@zaci*. Tentokrát budou konstruovány tak, že *nacti_tridu* vydá načtené pole jako výsledek svého volání (při neúspěchu bude prázdné) a podprogramu *uloz_tridu* zase předám ukládané pole jako parametry.

Trida/Soubory.pm

```
package Trida::Soubory;

use Exporter 'import';
our @EXPORT = qw(uloz_tridu nacti_tridu);

use Log;

sub nacti_tridu {
    my @zaci;
    my $TRIDA;
    print "Jméno souboru: ";
    my $jmenodat = <STDIN>; chomp( $jmenodat );
    if ( not open($TRIDA,$jmenodat) ) {
        print STDERR "Nelze načíst soubor $jmenodat !\n";
        return ();
    }
```

```
    while ( my $zak = <$TRIDA> ) {
        chomp( $zak );
        push( @zaci, $zak );
    }
    close($TRIDA);
    do_logu( "Načteno ze souboru $jmenodat.\n" );
    return @zaci;
}

sub uloz_tridu {
    my @zaci = @_;
    my $TRIDA;
    print "Jméno souboru: ";
    my $jmenodat = <STDIN>; chomp( $jmenodat );
    if ( not open($TRIDA,">$jmenodat") ) {
        print STDERR "Nelze zapisovat do souboru $jmenodat !\n";
        return 0;
    }
    foreach my $zak ( @zaci ) { print $TRIDA "$zak\n"; }
    close($TRIDA);
    do_logu( "Uloženo do souboru $jmenodat.\n" );
    return 1;
}

1;
```

Nejobjemnější bude modul pro komunikaci s uživatelem *Trida::Rozhrani*. Zajistí komunikaci s uživatelem – načítání dat a výpisy seznamu žáků. Všechny podprogramy, které ke své činnosti potřebují seznam žáků, jej opět obdrží jako parametr.

Určitý oříšek představují *smaz_zaka* a *pridej_zaka*, které mají měnit složení třídy. Původně pracovaly s globální proměnnou, tomu bych se ale rád vyhnul. Rozhodně není vhodné, aby podprogramy jednoho balíku měnily datové struktury jiného balíku. Při takovém způsobu programování je jen otázkou času, kdy si pořádně natlučete. Raději jim opět předám stávající pole žáků jako parametr a ve výsledku vrátí jeho upravenou verzi.

Jako bonus modul nabízí i funkci *najdi_zaka*, která se zeptá na jméno žáka a vrátí jeho index v poli (nebo –1). Sice se využívá jen uvnitř funkce *smaz_zaka*, nicméně taková funkce by se potenciálně mohla hodit, proto jsem ji přidal mezi exportované.

A teď už slíbený modul:

Trida/Rozhrani.pm

```
package Trida::Rozhrani;

use Exporter 'import';
our @EXPORT = qw(dej_znak pridej_zaka smaz_zaka najdi_zaka
                 seznam_abecedne seznam_prospech);

use locale;
use Log;

sub dej_znak {
    print "Zadejte příkaz\n";
    print " n ... přidat žáka\n";
    print " d ... vymazat žáka\n";
    print " j ... seznam žáků abecedně\n";
    print " p ... seznam žáků podle prospěchu\n";
    print " r ... načíst z disku\n";
    print " s ... uložit na disk\n";
    print " q ... konec\n";
    print "> ";
    my $vstup = <STDIN>;
    return substr( $vstup, 0, 1 );
}

sub pridej_zaka {
    print "Příjmení a jméno: ";
    my $jmeno = <STDIN>; chomp( $jmeno );
    print "Průměr: ";
    my $prumer = <STDIN>; chomp( $prumer );
    push( @_, "$jmeno:$prumer" );
    do_logu( "Přidán žák $jmeno.\n" );
    return @_;
}

sub najdi_zaka {
    my @zaci = @_;
    print "Příjmení a jméno: ";
    my $jmeno = <STDIN>; chomp( $jmeno );
    for ( my $i=0; $i<@zaci; $i++ ) {
        if ( $zaci[$i] =~ /^$jmeno:/ ) {
            return $i;
        }
```

```
    }
    return −1;
}

sub smaz_zaka {
    my $index = najdi_zaka(@_);
    if ( $index > −1 ) {
        my ( $jmeno ) = $zaci[$index] =~ /^(.*):/;
        splice( @_, $index, 1 );
        print "Žák $jmeno vymazán.\n";
        do_logu( "Vymazán žák $jmeno.\n" );
    } else {
        print "Žák nenalezen.\n";
    }
    return @_;
}

sub pis_zaka {
    my ( $zak ) = @_;
    my ($jmeno,$prumer) = $zak =~ m/(.*):(.*)/;
    printf "%−30s %0.2f\n", $jmeno, $prumer;
}

sub seznam_abecedne {
    my ( @zaci ) = @_;
    foreach my $zak ( sort @zaci ) {
        pis_zaka( $zak );
    }
}

sub seznam_prospech {
    my ( @zaci ) = @_;
    foreach my $zak ( sort srovnej_prospech @zaci ) {
        pis_zaka( $zak );
    }
}

sub srovnej_prospech {
    my ( $prvy, $druhy ) = ( $a, $b );
    $prvy =~ s/.*://; # odstraním jména a porovnám
    $druhy =~ s/.*://;
```

```
    return ( $prvy <=> $druhy );
}
```

Zároveň se jedná o první příklad modulu, který nezveřejňuje všechno. Některé podprogramy (*pis_zaka*, *srovnej_prospech*) jsou určeny pro jeho interní potřebu a nejsou zařazeny do pole *@EXPORT*.

A co zbylo na hlavní program? Není toho mnoho:

trida3.pl

```
use locale;
use lib ".";
use Log;
use Trida::Soubory;
use Trida::Rozhrani;

my @zaci;

otevri_log("trida.log");
while ( my $pokyn = dej_znak() ) {
    if ( $pokyn =~ /n/i ) { @zaci = pridej_zaka(@zaci); }
    elsif ( $pokyn =~ /d/i ) { @zaci = smaz_zaka(@zaci); }
    elsif ( $pokyn =~ /j/i ) { seznam_abecedne(@zaci); }
    elsif ( $pokyn =~ /p/i ) { seznam_prospech(@zaci); }
    elsif ( $pokyn =~ /r/i ) { @zaci = nacti_tridu(); }
    elsif ( $pokyn =~ /s/i ) { uloz_tridu(@zaci); }
    elsif ( $pokyn =~ /q/i ) { last; }
    else { print "Neplatný příkaz!"; }
}
zavri_log();
```

Všechny podprogramy získá skupinou příkazů **use** na začátku. Sám pak definuje jen hlavní cyklus, obhospodařující jejich volání. Všimněte si přidávání a odebírání žáků, které předává pole *@zaci* v parametru a nahrazuje je jeho změněnou verzí. Kopírování pole tam a zpět sice není ten nejefektivnější způsob, reálně je ale jeho režie zanedbatelná. ■

Uf! Já to říkal, že moduly mají smysl jen u rozsáhlejších programů. Doufám, že jste se tímhle dlouhým příkladem nenechali udolat. Já už to víckrát neudělám…

Rozhraní obvykle dává ostatním k dispozici podprogramy. Pokud chcete zpřístupnit proměnnou, musíte ji deklarovat pomocí **our**. Tím určíte, že proměnná sice existuje v daném modulu, nicméně

pracovat s ní mohou i ostatní části programu. Ve výčtu exportovaných identifikátorů před její název nezapomeňte přidat $, @ nebo %.

Typickým příkladem datových struktur, které chcete zpřístupnit okolí, jsou servisní proměnné modulu, jako je třeba pole *@EXPORT*. Podrobněji se k nim dostanu zanedlouho. Nicméně můžete samozřejmě dát k dispozici i vlastní proměnné. Pokud bych ve skladovém modulu chtěl zveřejnit i vlastní datovou strukturu skladu, což bude pole *@sklad*, vypadalo by jeho zahájení asi takto:

package *SkladRisk*; *SkladRisk.pm*

```
# definice rozhraní pomocí Exporter
use Exporter 'import';
our @EXPORT = qw(@sklad uloz vyzvedni);

our @sklad = ();

# implementace
sub uloz { ... }
sub vyzvedni { ... }

1;
```

Název modulu zároveň naznačuje, že takový přístup je nebezpečný. Zejména složitější datové struktury mívají různé vnitřní vazby, kdy je třeba vzájemně koordinovat hodnoty několika proměnných. Pokud zpřístupníte vlastní data, mohou být zvenku změněna nekoncepčním způsobem, což může rozložit činnost celého modulu.

Bývá vhodnější přistupovat k proměnným modulu jen lokálně a okolnímu světu nabídnout jen podprogramy. Ty zajistí potřebné funkce (v našem případě změny ve skladu, zjišťování jeho aktuálního stavu a podobně) a zároveň zajistí, že změny budou prováděny korektně a zachovají všechny potřebné vnitřní vazby v datových strukturách.

Ochranu proměnných před zlým vnějším světem zajistí jejich standardní deklarace pomocí **my**. Takové proměnné jsou dosažitelné jen zevnitř modulu.

Příklad: Vytvořím jednoduchý ukázkový modul s chráněnou proměnnou. Svému okolí nabízí jen funkci, která vypíše její aktuální hodnotu:

package *Ohradka*; *Ohradka.pm*

```
use Exporter 'import';
our @EXPORT = qw(stav);
```

```
my $hodnota = 10;

sub stav { print "Hodnota zevnitř: $hodnota\n" }
1;
```

V následujícím programu modul použiji a pokusím se proměnnou číst a měnit:

kolem-ohradky.pl

```
use lib ".";
use Ohradka;

stav();
print "Hodnota zvenčí: ", $Ohradka::hodnota, "\n";
$Ohradka::hodnota = 333;
print "Hodnota zvenčí: ", $Ohradka::hodnota, "\n";
stav();
```

Výsledkem bude:

```
Hodnota zevnitř: 10
Hodnota zvenčí:
Hodnota zvenčí: 333
Hodnota zevnitř: 10
```

Jak vidíte, Perl vytvořil novou proměnnou, se kterou mohu pracovat pod názvem *$Ohradka::hodnota*, ale závěrečný *stav()* dokazuje, že se skutečnou proměnnou *$hodnota* modulu *Ohradka* nemá nic společného. ■

Sečteno a podtrženo: Je-li proměnná určena pouze pro daný modul, deklarujte ji **my**. Pokud k ní má být přístup i z ostatních částí programu, použijte **our**.

Problémy s dostupností globálních proměnných, na které jsem narazil v příkladu se třídou, jsou celkem běžným jevem. Existují na ně dva recepty. Jeden jsem předvedl v příkladu – je jím snaha o maximální nezávislost modulů, kdy veškerá potřebná data jejich podprogramům předáváte ve formě parametrů. V kapitole 11 na straně 173 se dozvíte, jak se dá předat parametr tak, aby bylo možné jej i měnit. Tento přístup je záhodno nasadit především tehdy, když se snažíte o univerzálně použitelné moduly, které pak budou k dispozici i pro jiné programy či programátory.

V některých případech však takové ambice nemáte a chcete moduly především jako prostředek logického členění programu. Pak lze uvažovat o druhém způsobu řešení problému globálních da-

tových struktur – prostě pro ně vytvořit samostatný modul obsahující jen globální data. V našem případě bych vytvořil třeba modul *Trida::Data*:

Trida/Data.pm

```
package Trida::Data;

use Exporter 'import';
our @EXPORT = qw(@zaci);

our @zaci = ();

1;
```

Kdekoli bych pak potřeboval přístup ke globálním datovým strukturám, stačí uvést:

```
use Trida::Data;
```

a mohu s nimi vesele pracovat. Stejný příkaz bych pochopitelně musel přidat i do hlavního programu, abych si nedopatřením nezavedl vedle pole *@Trida::Data::zaci* také *@main::zaci*.

10.3 Definice a použití rozhraní

V předchozí části jsem definici rozhraní (použití modulu *Exporter*) poněkud zjednodušil. Ve skutečnosti vám dává větší možnosti, které ale v řadě případů nebudete vůbec potřebovat. Proto jsem se je rozhodl odložit na později. Konkrétně na teď.

Pole *@EXPORT* není jedinou proměnnou, která řídí chování *Exporter*u. Jak již bylo řečeno, obsahuje jména těch identifikátorů, které mají být automaticky importovány do každého programu či modulu, který použije ten váš. Jaké jsou další řídicí proměnné?

$VERSION je celkem obyčejná proměnná, do které přiřadíte reálné číslo identifikující verzi vašeho modulu. Například:

```
our $VERSION = 2.30;
```

V příkazu **use** totiž lze uvést, že se požaduje určitá minimální verze. Pokud pro svůj program potřebujete verzi alespoň 2.0, napište:

```
use »modul« 2.0;
```

Jestliže Perl najde jen starší verzi, skončí s chybovým hlášením typu:

```
Data version 2 required--this is only version 1.5
```

Podle poslední perlivé módy by měla být verze součástí příkazu **package**, začínat písmenem „v“ a obsahovat tři tečkami oddělená čísla, asi takto:

package *Trida::Data* v1.0.0;

Nicméně řada existujících balíků se drží tradičních hodnot v podobě proměnné *$VERSION*.

Pole *@EXPORT_OK* obsahuje názvy těch identifikátorů, které sice lze importovat do dalších programů a modulů, avšak nedělá se to automaticky. Jejich import musí programátor explicitně nařídit v příkazu **use**. Například:

use *Trida::Rozhrani* **qw**(*seznam_abecedne seznam_prospech*);

importuje z modulu *Trida::Rozhrani* jen dva identifikátory – *seznam_abecedne* a *seznam_prospech*. (Všechny identifikátory zařazené do *@EXPORT* jsou automaticky také *@EXPORT_OK*.) Jakmile v příkazu **use** uvedete seznam identifikátorů, budou importovány skutečně jen ony a obsah pole *@EXPORT* se ignoruje. Chcete-li importovat všechny implicitní a k nim navíc některý z pole *@EXPORT_OK*, použijte:

use »*modul*« **qw**(:DEFAULT »*další_identifikátory*«);

Asociativní pole *%EXPORT_TAGS* využijete především u rozlehlých modulů, které nabízejí k importu velké množství identifikátorů. Díky *%EXPORT_TAGS* můžete tyto identifikátory shlukovat do tematických skupin a prostřednictvím názvu skupiny pak snadno vložit několik najednou.

Klíčem v poli je název skupiny, hodnotou seznam prvků, které do této skupiny patří. Všechny identifikátory ve skupině musí být uvedeny v poli *@EXPORT* či *@EXPORT_OK*. Například identifikátory z modulu *Trida::Rozhrani* bych mohl rozdělit do několika tematických skupin: vstup od uživatele, výstup, změny a hledání.

```
our %EXPORT_TAGS = (
    Vstup => [ qw(dej_znak) ],
    Vystup => [ qw(seznam_abecedne seznam_prospech) ],
    Zmeny => [ qw(pridej_zaka smaz_zaka) ],
    Hledani => [ qw(najdi_zaka) ]
);
```

Zápis zatím berte jako dogma. Až si přečtete příští kapitolu, pochopíte. Kdybych pak chtěl ve svém programu použít jen výstupní podprogramy a *pridej_zaka*, zapsal bych **use** ve tvaru:

use = *Trida::Rozhrani* **qw**(:Vystup *pridej_zaka*);

- Rozhraní, export a import identifikátorů představují pouze zkratky k jednotlivým identifikátorům. Nejedná se o bezpečnostní mechanismy, jak bývá zvykem u jiných jazyků. Pokud některý identifikátor není označen jako importovatelný, přesto se k němu cizí modul může dostat – prostě tím, že napíše jeho plné jméno (například si mohu zavolat *Trida::Rozhrani::pis_zaka*).

Pokud chcete soukromí, lze je dosáhnout jen u proměnných prostřednictvím deklarace **my**. Takto zavedená proměnná bude lokální v daném souboru (jestliže ji uvedete uvnitř bloku, bude samozřejmě lokální jen v rámci svého bloku) a zvenčí se k ní nelze nijak dostat. Nemůžete ji ani exportovat. Podprogramy však znepřístupnit nelze.

Perlovská komunita se k této otázce staví celkem zajímavým způsobem. Prosazuje názor, že programování v Perlu je prostě založeno raději na zásadách slušnosti než na technických bariérách. Do cizích modulů se nešťoural proto, že si to jejich autor prostě nepřeje (neuvedl dotyčné prvky v seznamu exportovaných). Slušnému člověku prostě stačí nápis „Nevstupujte“ a nemusíte ani zamykat.

Jako filozofie je to opravdu hezké. Z vlastní praxe bohužel vím, že řada lidí ty dveře prostě vyzkouší. A pokud bude zamčeno, půjdou za vaším nadřízeným, aby vám přikázal dát jim klíče…

10.4 Když se řekne pragma

V Perlu existuje ještě jeden, dost speciální druh modulů. Zatímco běžný modul zpravidla přidává nové konstrukce, modul z této kategorie spíše mění chování těch stávajících. Pro moduly tohoto typu se používá název *pragma* a jsou zařazeny do standardní distribuce Perlu. Existuje jich kolem čtyřicet, vybrané najdete v tabulce 10.1.

Konvence ukládá zahajovat jejich jména malým písmenem, aby se na první pohled odlišila od běžných modulů. Některá už jste viděli v akci – konkrétně *locale* a *lib*. U pár dalších se teď zastavím.

Pragma *constant* je jedinou cestou, jak v Perlu vyrobit konstantu. Tedy symbolické jméno s pevně přiřazenou hodnotou. Zápis vypadá takto:

```
use constant PI => 3.1415;
use constant ADRESA => 'kdosi@kdesi.cz';
```

Od tohoto okamžiku můžete používat symbolická jména PI a ADRESA a Perl si za ně dosadí hodnoty, které jste jim přidělili. Pochopitelně je nelze měnit. Je zvykem psát názvy konstant velkými písmeny.

autouse	odkládá vložení modulu na dobu běhu
bignum	aritmetika s velkými čísly
constant	definice konstant
diagnostics	podrobnější informace o chybách
integer	celočíselná aritmetika
lib	mění obsah *@INC*
locale	národní prostředí
overload	přetěžování operací
sigtrap	zpracování signálů
strict	silnější kontroly
subs	předběžná deklarace podprogramů
vars	předběžná deklarace proměnných

Tabulka 10.1: Vybrané pragma moduly

Příklad: Použití konstanty předvedu na jednoduchém programu pro výpočet obvodu a obsahu kruhu. Když už se kolem toho π pořád ochomýtám, ať se také jednou uplatní:

kruh.pl

```
use constant PI => 3.1415926;
print "Obvod a obsah kruhu\n";
print "zadejte poloměr: ";
my $r = <STDIN>; chomp($r);
print "Obvod: ", 2*PI*$r, "\n";
print "Obsah: ", PI*$r*$r, "\n";
```

■

Pokud vám chybová hlášení a varování Perlu připadají příliš strohá, zkuste:

```
use diagnostics;
```

Seriózní překladač tím proměníte v pavlačovou babu. Něco tak užvaněného jste v životě neviděli.

Veškeré výpočty se odehrávají s reálnými čísly. Už jsem se zmiňoval o možnosti zaokrouhlování prostřednictvím funkce **sprintf**. Můžete si také přepnout na celočíselnou aritmetiku. Obstará to příkaz:

```
use integer;
```

který zajistí, že od tohoto okamžiku až do konce bloku (nebo souboru) budou veškeré výpočty prováděny s celými čísly. Takže:

```
{
    use integer;
    print 10/3;
}
```

vypíše 3 místo obvyklých 3.33333333333333.

Pro výpočty s vyššími nároky na přesnost se hodí pragma *bignum*, které veškerou aritmetiku přepne na velká čísla. Celá čísla se stanou téměř nekonečnými[3], u reálných se více než zdvojnásobí počet desetinných míst. Platí se pamětí a rychlostí – výpočty jsou o něco pomalejší a hodnoty spotřebují více paměti.

Pragma *strict* má na starosti jisté zmírnění benevolence, kterou je Perl proslulý. Jeho použití má následující efekty:

- zakáže symbolické odkazy (v příští kapitole se dozvíte, co to je),
- zakáže použití názvů podprogramů bez úvodního & i závorek,
- povolí jen proměnné deklarované jako **my**, importované z modulu, deklarované pomocí **use** *vars* nebo plně kvalifikované, tedy včetně jména balíku.

Ruku v ruce s ním zpravidla kráčejí *subs* a *vars*, díky nimž můžete deklarovat povolené podprogramy a proměnné – například:

```
use strict;
use subs qw(nacti_tridu uloz_tridu);
use vars qw(@trida);
```

deklaruje jako povolené podprogramy *nacti_tridu* a *uloz_tridu* a pole *@trida*. Všechny ostatní identifikátory podléhají výše uvedeným omezením.

3: Výsledná hodnota jednoho mého výpočtu na obrazovce zabrala přes 150 řádků.

11 Odkazy, datové struktury a propletence

Jednou z nezpochybnitelných nevýhod Perlu je jeho velmi omezený sortiment základních typů: skalár, jednorozměrné pole a asociativní pole netvoří nijak impozantní výčet. Už jen realizace dvojrozměrného pole (např. šachovnice) vyžaduje jisté intelektuální vypětí. Složitější datové struktury pak zavánějí neřešitelnem nebo přinejmenším nechutnou manuální dřinou.

Odkazy vaše možnosti podstatným způsobem rozšíří. Jejich prostřednictvím lze popsat i velmi komplikované vztahy mezi daty. Ovšem nebude to zadarmo. Odkazy nedoporučují čtyři z pěti psychiatrů. Jedná se o poměrně náročný koncept a jednoznačně patří mezi programátorskou vyšší dívčí. S trochou nadsázky se dá říci, že dobré zvládnutí odkazů mění zelenáče a příležitostné softwarové kutily na zkušené programátory.

Proto počítejte s tím, že nadcházející kapitola bude tuhá. Doporučuji vám nechat svou hlavu zrelaxovat, než se pustíte do čtení.

11.1 Co je odkaz

Jednoduše řečeno: odkaz je skalární proměnná, která jako hodnotu obsahuje adresu jiné proměnné (čili odkaz na ni). Proměnná má totiž v programovacím jazyce dva významy: jednak je to určité místo v paměti počítače, kam ukládáte data, jednak je to symbolické jméno, kterým ono místo označujete. Když v programu napíšete *$a*, interpret si místo tohoto jména dosadí adresu v paměti, na které sídlí hodnota proměnné *$a*. Odkazy vám umožňují pracovat přímo s těmito adresami.

Nejjednodušší variantou odkazu je odkaz prostřednictvím jména, tak zvaný *symbolický odkaz*. Myšlenka prostá a v jiných jazycích celkem nevídaná. Řekněme, že v proměnné *$prom* mám uložen řetězec „jmeno“. Co udělá, když před *$prom* předřadím ještě jeden *$*? Perl se zachová celkem logicky: za *$prom* si dosadí hodnotu dotyčné proměnné a před ní ponechá onen druhý dolar, čímž vznikne proměnná *$jmeno*.

Analogicky fungují i ostatní datové typy, takže třeba *@$pole* představuje pole, jehož jméno je uloženo v proměnné *$pole*.

Příklad: Za příklad tohoto typu odkazů může posloužit následující podprogram *pis_pole*, který vypíše obsah pole. Jako parametr však nedostane přímo ono pole (přesněji řečeno seznam jeho hodnot), jak jste dosud byli zvyklí, ale jen jméno:

pispole.pl

```
sub pis_pole {
    my ( $jmeno_pole ) = @_;
    foreach my $prvek ( @$jmeno_pole ) {
```

```
            print "$prvek\n";
        }
    }
```

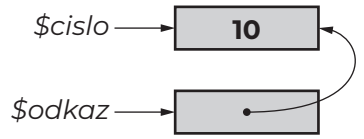
Pro výpis obsahu pole *@vyroba* je třeba volat *pis_pole*("vyroba"). ■

Ovšem odkazy prostřednictvím jména nejsou ty jedině pravé odkazy. Tady jsme se nevymanili ze závislosti na jménu, jen to jméno používáme nepřímo. Upřímně řečeno, podle současných pravidel slušného perlivého programování jsou považovány za poněkud nečistou techniku a doporučuje se jim vyhýbat.

Skutečně pravý odkaz neobsahuje jméno, ale pravověrnou adresu určité části paměti. Ta vznikne, když před název některé z proměnných vepíšete zpětné lomítko. Takže například příkazy:

```
$cislo = 10;
$odkaz = \$cislo;
```

přiřadíte do proměnné *$cislo* hodnotu 10 a do proměnné *$odkaz* odkaz na tuto proměnnou. Čili adresu, na které sídlí. Vznikne situace, kterou znázorňuje obrázek 11.1.



Obrázek 11.1: Proměnná a odkaz na ni

Když si necháte vypsat hodnotu proměnné *$odkaz*, dostanete cosi jako:

```
SCALAR(0x80e2500)
```

Ono příšerně vyhlížející číslo v šestnáctkové soustavě je adresa paměti, kam ukazuje dotyčný ukazatel[1]. Slůvko SCALAR signalizuje, že je na ní uložena skalární proměnná. V tomto případě si Perl hlídá typy dat a pokud byste se například s tímto odkazem pokusili zacházet jako s odkazem na pole, skončíte s chybovým hlášením.

Zpřístupnění hodnoty (označované též jako následování odkazu či dereference) se zapisuje stejně jako u symbolických odkazů: *$$odkaz* (výsledkem je hodnota 10).

1: Teď trochu lžu. Odkazy v Perlu neobsahují skutečné adresy paměti, ale jakési abstraktní identifikátory, které si Perl až při běhu programu převádí na opravdové paměťové adresy. Ten rozdíl ale není nijak podstatný a pro zjednodušení se dál budu tvářit, že odkazy jsou adresy.

Je třeba si uvědomit, že po provedení výše zmíněných příkazů ukazují *$cislo* a *$$odkaz* na stejný úsek paměti. To znamená, že když provedete:

```
$$odkaz = 20;
print $cislo;
```

vypíše se hodnota 20. Tímto způsobem můžete jednomu kousku paměti dát jmen, co hrdlo ráčí.

Analogicky lze zacházet i s oběma druhy polí. Ovšem pro zpřístupnění jejich položek zavedl Perl zajímavou alternativu, která přispívá k vyšší srozumitelnosti zdrojového textu. Místo *$$pole*[1] můžete psát *$pole*–>[1]. Operátor –> se vkládá mezi název proměnné obsahující odkaz na pole a index či klíč. Graficky připomíná odkazující šipku a velmi pěkně tak symbolizuje, co vlastně děláte: následujete odkaz uvedený v proměnné vlevo a v cílovém poli pak vyberete index, resp. klíč.

Cvičení 11.1: Zavedu pole, asociativní pole a odkazy na ně:

```
@hodnoty = ( "raz", "dva", "tři", "čtyři" );
%asocp = @hodnoty;
$opole = \@hodnoty;
$ohash = \%asocp;
```

Co vypíše následující čtveřice příkazů?

```
print "$opole–>[2]\n";
print "$ohash–>[2]\n";
print "$ohash–>{'raz'}\n";
print "$ohash–>{'dva'}\n";
```

■

Zejména u komplikovanějších složenin mohou být výsledné tvary dost krkolomné. Doporučuje se, abyste se pokud možno snažili vyhýbat po sobě jdoucím sekvencím znaků určujících typ proměnné (jako například *$$opole*[2]). Tento zápis totiž může být pochopen dvojím způsobem:

- Proměnná *$opole* obsahuje odkaz na pole a programátor žádá obsah jeho položky s indexem 2 (tento výklad je správný).
- V poli *@opole* je s indexem 2 uložen odkaz na skalární proměnnou a programátor chce získat její obsah (tak to není).

Aby k podobným pochybnostem nedocházelo, používejte buď operátor –> nebo význam explicitně vyznačte prostřednictvím složených závorek. Pokud je vaším cílem první význam, pište *${$opole}*[2] (běžný přístup k položce pole, až na to, že jeho jméno je nahrazeno proměnnou obsahující odkaz

na něj). Chcete-li druhý význam, použijte *${$opole*[2]} (vezme se položka s indexem 2 a použije jako odkaz).

Jedno z míst, kde odkazy hrají dost významnou roli, jsou parametry podprogramů. Jejich hlavním přínosem je, že umožňují zachovat identitu polí.

Jistě si vzpomínáte, že seznam má tu vlastnost, že do sebe pohlcuje vnořené seznamy včetně polí. Vše rozvine a vytvoří jeden úhledný seznam hodnot. Jenže co když máte vytvořit funkci, která sčítá dva vektory? Potřebuje dostat *dvě* pole čísel a sečíst v nich první prvek s prvním, druhý s druhým atd. Ovšem podprogramy v Perlu dostávají *jeden* seznam parametrů. Pokud jím budou dvě pole, slijí se do jednoho a vy nepoznáte, kde končí první a začíná druhé. Odkazy vám naštěstí nabídnou řešení.

Příklad: Kýžená funkce, která sečte dva vektory a jako výsledek vydá třetí (jejich součet), prostě dostane jen dva parametry – odkaz na první a druhý vektor (pole):

soucetvek.pl

```
sub secti_vektory {
    my ( $a, $b ) = @_;
    my ( @c, $delka );
    if ( @$a > @$b ) {
        $delka = @$a;
    } else {
        $delka = @$b;
    }
    for ( my $i=0; $i<$delka; $i++ ) {
        push( @c, $a->[$i] + $b->[$i] );
    }
    return @c;
}
```

Použití je prosté: *@c = secti_vektory*(\@x, \@y). ■

Cvičení 11.2: Napište podprogram *obrat*, který dostane jako parametr odkaz na pole a který toto pole převrátí – vymění první prvek s posledním, druhý s předposledním a podobně. ■

11.2 Anonymní data a práce s pamětí

Doposud se zdálo, že odkazy figurují jen jako alternativní jména pro existující proměnné. Pravda, v parametrech podprogramů ukázaly jistou sílu, ale stále to nebyl žádný velký zázrak.

Skutečné kouzlo odkazů spočívá ve vytváření potenciálně velmi rozlehlých datových struktur, které jsou dostupné jen prostřednictvím odkazů. Taková struktura, která může čítat tisíce položek, bývá „zavěšena" na jediné vstupní proměnné. Veškerý další pohyb v ní se odehrává výlučně prostřednictvím jejích interních odkazů.

Abychom takové konstrukce mohli přenést ze světa snů do bitů našeho počítače, potřebujeme osamostatnit odkazy od přízemní existence pojmenovaných proměnných. Jinými slovy vyřešit otázku jak vytvořit proměnnou, která není dostupná pod žádným jménem, ale jen prostřednictvím odkazu.

První (komplikovanější) cesta využívá lokality proměnných. Založím pojmenovanou lokální proměnnou, odkaz na ni přiřadím do proměnné globální a ukončím blok, takže ono lokální pojmenování zanikne.

Příklad: Tento postup demonstruje následující úsek programu:

```
my $odkaz = nove_cislo(15);
print "$$odkaz\n";

sub nove_cislo {
    my ( $x ) = @_;
    return \$x;
}
```

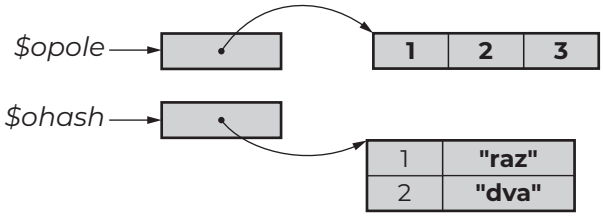
Co se stalo? Voláním funkce *nove_cislo* vznikla její lokální proměnná *$x*, které interpret přidělil určitou část paměti. Vzápětí se adresa této přidělené paměti oznámí jako výsledek funkce. Tím končí její blok a také platnost identifikátoru *$x*. Ovšem přidělená část paměti (proměnná) žije dál, protože její adresa byla uložena do proměnné *$odkaz*. Ta nyní obsahuje odkaz na proměnnou, ke které již neexistuje žádná jiná cesta. ■

Jednodušší cestu představuje použití tak zvaných anonymních dat. Jedná se o konstrukce, které rovnou přidělí úsek paměti a vydají odkaz na něj, aniž by dotyčné paměti přidělovaly nějaké jméno. Například pro obyčejnou skalární hodnotu použijte:

```
$$odkaz = 20;
```

V případě polí a asociativních polí máte na výběr dva alternativní způsoby zápisu. Jeden je klasický a vychází ze standardního zápisu:

```
@$opole = ( 1, 2, 3 );
%$ohash = ( 1 => "raz", 2 => "dva" );
```



Obrázek 11.2: Dvojice anonymních polí

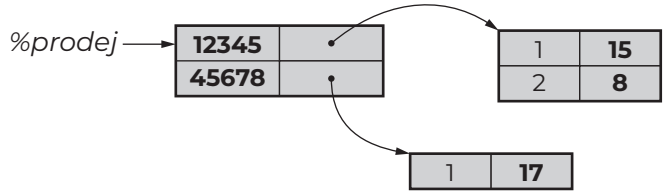
Výsledkem této dvojice příkazů je situace, kterou znázorňuje obrázek 11.2. Druhý způsob zápisu spotřebuje menší počet speciálních znaků. Stejný efekt jako výše uvedená dvojice vyloudí i tyto dva příkazy:

```
$opole = [ 1, 2, 3 ];
$ohash = { 1 => "raz", 2 => "dva" };
```

Typ zakládané proměnné se zde nevyjadřuje speciálním znakem před odkazující proměnnou, ale druhem závorek kolem přiřazovaného seznamu.

Prostřednictvím těchto nástrojů můžete budovat datové struktury se složitostí, jakou si zamanete. Umožňují obejít podstatné omezení Perlu, že do polí (obyčejných i asociativních) lze ukládat jen skalární hodnoty. Potřebujete-li do pole vložit jiné pole, vložíte odkaz na ně. Například šachovnici lze reprezentovat prostřednictvím pole s osmi odkazy na řádky, což budou opět pole, každé s osmi údaji pro jednotlivé sloupce.

Příklad: Vyrobím jednoduchý program pro vyhodnocení prodeje zboží v síti prodejen. Vstupní data jsou tvořena sadou údajů o jednotlivých realizovaných prodejích. Každý řádek obsahuje vždy kód zboží, kód prodejny a počet prodaných kusů. Cílem programu je určit, kolik kusů jednotlivých druhů zboží se celkem prodalo ve které prodejně.



Obrázek 11.3: Datová struktura pro vyhodnocení prodeje

Datovou strukturou, ve které budu sbírat údaje, bude asociativní pole asociativních polí. V hlavním asociativním poli budou jako klíče sloužit kódy druhů zboží. Pro každý druh bude obsahovat odkaz na asociativní pole indexované kódy prodejen. Do jeho položek pak budu sčítat celkový prodej daného zboží v dané prodejně. Například vstupní data:

```
12345 1 15
45678 1 7
12345 2 8
45678 1 10
```

vytvoří datovou strukturu na obrázku 11.3. Realizující program by vypadal třeba takto:

zbozi.pl

```
1   my %prodej = ();
2
3   # načítání dat
4   while ( my $radek = <> ) {
5       chomp($radek);
6       my ( $zbozi, $prodejna, $pocet ) = split( /\s+/, $radek );
7       zapocitej_zbozi( $zbozi, $prodejna, $pocet );
8   }
9   # výstup výsledků
10  foreach my $zbozi ( sort keys %prodej ) {
11      print "\nZboží: $zbozi\n";
12      foreach my $prodejna ( sort keys %{$prodej{$zbozi}} ) {
13          print " prodejna $prodejna ... ";
14          print $prodej{$zbozi}->{$prodejna}, "\n";
15      }
16  }
17
18  sub zapocitej_zbozi {
19      my ( $zbozi, $prodejna, $pocet ) = @_;
20      $prodej{$zbozi}->{$prodejna} += $pocet
21  }
```

Všimněte si, že ve druhém příkazu **foreach** (řádek 12) jsem musel použít složené závorky, abych vyznačil, že *$prodej*{*$zbozi*} je odkaz na asociativní pole, jehož klíče chci procházet. ■

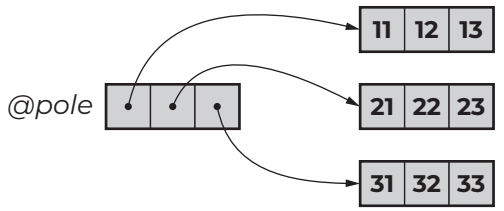
Perl připouští i kompaktní syntaxi a při vnořování polí nemusíte mezi jednotlivými indexy a klíči psát odkazové šipky. Následující zápisy jsou korektní:

```
$pole[0][1]          odpovídá  $pole[0]->[1]
$pole[0]{"raz"}      odpovídá  $pole[0]->{"raz"}
$hash{"raz"}[0]      odpovídá  $hash{"raz"}->[0]
$hash{"raz"}{"dva"}  odpovídá  $hash{"raz"}->{"dva"}
```

Je otázkou, zda těch pár ušetřených znaků stojí za potenciální zmatení, protože kompaktní zápis skrývá přítomnost odkazů. Ty tam však stále jsou. Například provedením příkazů:

```
my @pole = ();
foreach my $i (0..2) {
    foreach my $j (0..2) {
        $pole[$i][$j] = ($i+1)*10 + $j + 1;
    }
}
```

vznikne dvojrozměrné pole, jehož vnitřní strukturu vidíte na obrázku 11.4.



Obrázek 11.4: Dvojrozměrné pole

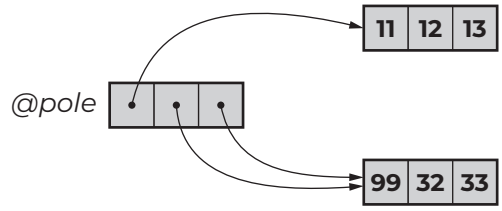
Kdybyste zapomněli na existenci odkazů v něm, mohlo by vás překvapit, že po provedení dvojice příkazů:

```
$pole[1] = $pole[2];
$pole[1][0] = 99;
```

bude mít *$pole*[2][0] hodnotu 99. Je to dáno tím, že první příkaz nepřiřazuje hodnoty uložené v řádku pole, ale odkaz na příslušný řádek. Po jeho provedení vedou odkazy na druhý a třetí řádek na totéž anonymní pole v paměti. Vznikne datová struktura z obrázku 11.5.

Pokud bych chtěl zachovat samostatné řádky a jen zkopírovat hodnoty ze třetího do druhého, musel bych to buď provést po jedné v cyklu:

```
foreach my $j (0..2) {
    $pole[1][$j] = $pole[2][$j];
}
```



Obrázek 11.5: Dvojrozměrné pole po změně

nebo pro druhý řádek vytvořit nové anonymní pole a do něj vložit seznam hodnot ze třetího:

```
$pole[1] = [ @{$pole[2]} ];
```

Kdyby ovšem datová struktura byla složitější a řádek obsahoval další odkazy, opakoval by se podobný problém. Nejbezpečnější (a nejpohodlnější) je řádek klonovat funkcí *dclone* z modulu *Storable*, ke kterému se dostanu na stránce 184:

```
$pole[1] = dclone( $pole[2] );
```

Cvičení 11.3: Šlápněte Karpovovi na paty! Sestavte základ každého šachového programu – podprogram, který spočítá figurky na šachovnici. Ta bude realizována výše popsanou datovou strukturou: polem odkazů na osm polí, z nichž každé obsahuje osm položek. Pokud je dané políčko šachovnice prázdné, obsahuje příslušná položka prázdný řetězec. V opačném případě je v ní uložen popis figurky (barva a hodnota).

Sestrojte funkci, která dostane *odkaz* na takovouto datovou strukturu a jako výsledek vydá celkový počet figurek na šachovnici (tedy počet neprázdných položek ve všech 64 polích. ■

Doposud jsem lehce přecházel mechanismy, které Perl musí při práci s vytvářením a likvidací proměnných používat. Prostě jsem předpokládal, že proměnná automaticky vznikne, když je potřeba, a stejně automaticky i zaniká. Možná vás to překvapí, ale ono to doopravdy tak funguje.

S tímto prohlášením bych se mohl spokojit, ale já vás nechám nahlédnout Perlu do kuchyně. Přidělování paměti není problém. Jednoduše si udržuje seznam volné paměti a kdykoli se objeví nová proměnná, vezme z něj potřebný kus a přidělí.

Daleko zajímavější je uvolňování paměti, která přestala být potřeba. Perl k tomu používá podobný mechanismus, jako Unix pro mazání souborů: počítá si odkazy na každou proměnnou. Identifikátory proměnných při tom bere stejně jako odkazy. Takže dvojice příkazů:

```
$a = 8;
$b = \$a;
```

vede ke vzniku dvou skalárních proměnných. Na první z nich vedou dva odkazy (identifikátor *$a* a odkaz uložený v proměnné *$b*), na druhou jen jeden (identifikátor *$b*).

Když některý identifikátor zanikne (v případě našeho *$a* když do proměnné *$b* přiřadíte jinou hodnotu nebo když skončí blok, ve kterém byl zaveden lokální identifikátor *$a*), odečte jedničku od počtu odkazů na příslušnou proměnnou. Pokud tím došlo k jeho vynulování, zruší alokaci a paměť označí jako volnou. Je to logické, protože k uložené hodnotě nevede už žádná cesta a pro váš program neexistuje způsob, jak se k ní dostat.

Tento algoritmus má svá omezení (lze sestrojit datovou strukturu, která bude mít u svých proměnných nenulový počet odkazů, a přesto do ní nevede žádný odkaz, takže Perl nepřijde na to, že by ji měl vymazat), ovšem ve většině běžných případů bude fungovat k vaší naprosté spokojenosti.

11.3 Záznamy

Jedním z běžných datových typů, které v Perlu chybí, je záznam (v Pascalu **record**, v C **struct**). Mívá dvě charakteristické vlastnosti:

- jeho položky mohou být různých datových typů a
- jsou identifikovány názvy, nikoli indexy.

Tento způsob ukládání dat je velmi dobře srozumitelný, protože umožňuje pojmenovávat datové položky mnemotechnickými názvy. Například když uchováváte informace o automobilu, budou údaje o jeho SPZ a spotřebě dostupné jako *auto.spz* a *auto.spotreba*.

Jak nahradit záznam v Perlu? Při ležérním přístupu jazyka k datovým typům nebude s první vlastností žádný problém. Identifikace pomocí názvů pak jasně napovídá, že vhodným kandidátem na ztělesnění záznamů bude asociativní pole. Roli identifikátoru položky sehraje klíč. Zápis se proti tvaru obvyklému z jiných programovacích jazyků změní na *$auto*{"spz"} a *$auto*{"spotreba"}, což není žádná tragédie.

Ovšem záznamy bývají často zařazovány do složitějších struktur. Například základní evidenční mantrou je pole záznamů. Třeba údaje o jednom žákovi budou podchyceny záznamem obsahujícím jméno, bydliště, datum narození, známky z jednotlivých předmětů atd. Celá třída pak bude realizována polem těchto záznamů.

Jelikož asociativní pole nemůže být zařazeno jako položka pole, zažil se jako nejobvyklejší způsob reprezentace záznamu v Perlu *odkaz na asociativní pole*. Například:

```
$zak = { ”jmeno” => ”Horáček Jiljí”,
         ”narozen” => [ 11, 12, 1985 ],
         ”adresa” => ”Jilmová 13, Praha 1” };
```

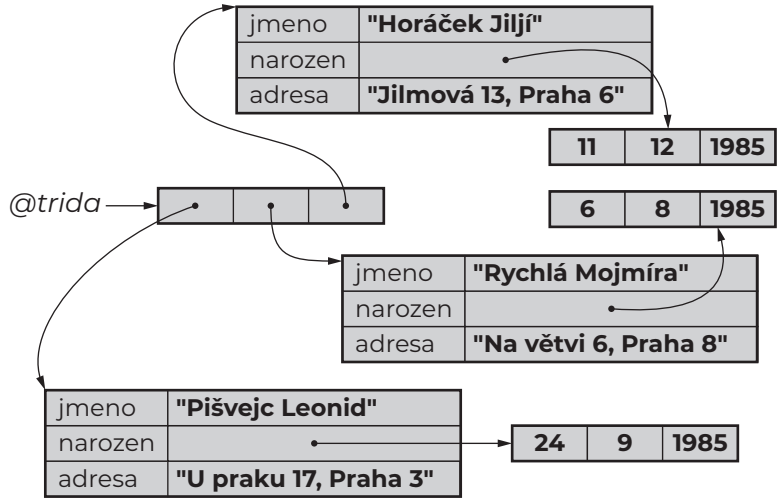
vytvoří „záznam“ pro žáka a odkaz na něj uloží do proměnné *$zak*. Všimněte si, že datum narození je řešeno odkazem na pole, jehož jednotlivými prvky jsou den, měsíc a rok. Takže například rok narození získáte pomocí:

```
$zak–>{”narozen”}–>[2]
```

Zařazení tohoto žáka mezi ostatní (do pole *@trida*) by pak obstaral příkaz:

```
push ( @trida, $zak );
```

Obrázek 11.6 znázorňuje příklad takto reprezentované třídy.



Obrázek 11.6: Struktura dat pro třídu

Popsaný způsob realizace záznamů je v jádru funkční, nicméně trpí stejnými nedostatky, jako celá práce s proměnnými v Perlu. Pokud se překlepnete v názvu některé položky, prostě se vytvoří nová nebo vydá prázdný řetězec (podle toho, zda hodnotu ukládáte či vybíráte). V řadě případů dokonce

nedostanete vůbec žádné varování a na chybu si musíte přijít sami. Zkrátka bezpečnost práce si na perlivém žebříčku hodnot nestojí nijak vysoko.

11.4 Datové struktury a práce s nimi

Psát v Perlu o datových strukturách je dost ošidná práce. Pod tento pojem obecně spadá uspořádání dat v programu. V praxi se tak nejčastěji označují konstrukce složené z elementárních datových buněk propojené navzájem ukazateli. Existují dvě nejzákladnější datové struktury, které jsou popsány ve valné většině učebnic programování: lineární seznam a binární vyhledávací strom.

Lineární seznam je odkazy propojený řetízek proměnných, kde každá nese vždy určitou užitečnou informaci a odkaz na svého následníka. Hlavním smyslem jejich existence je obejít nutnost stanovit předem a pevně počet prvků v poli, kterou má řada programovacích jazyků. Jenže v Perlu si s polem můžete dělat, co vás napadne, a lineární seznam vám de facto nemá co nabídnout.

Binární vyhledávací strom obsahuje v každém svém prvku (uzlu) užitečnou hodnotu a dva odkazy – na levý a pravý podstrom. Vše je uspořádáno tak, aby hodnoty v levém podstromu byly menší než hodnota uzlu a v pravém podstromu naopak větší. Takže když něco hledáte, porovnáním velikostí hned víte, do kterého podstromu máte jít.

Jak název napovídá, hlavním smyslem binárního vyhledávacího stromu je rychlé hledání. Ovšem na to má Perl asociativní pole. Nechci vás podceňovat, ale s binárním stromem proti němu nemáte šanci ani v rychlosti ani ve snadnosti použití.

Sečteno a podtrženo: v datových strukturách je Perl hodně netradiční. Jeho základní konstrukce se výrazně liší od toho, co je v kraji zvykem. Takže některé prvky, které jinde patří mezi samozřejmosti (vícerozměrná pole, záznamy), jsou zde relativně pracné. A naopak levou zadní zvládá konstrukce, které by vám jinde daly spoustu práce. Perl je prostě svůj.

Proto se spokojím s prohlášením, že datové struktury v Perlu je třeba vytvářet vždy „na tělo“ konkrétnímu řešenému problému (jako třeba data o třídě před chvilkou).

Zde se zmíním jen o prostředcích pro práci se složitějšími datovými strukturami. Konkrétně o jejich kopírování a ukládání do souborů. Tyto operace si jistě dokážete zajistit sami, ale proč vynalézat kolo? V instantním balení je totiž nabízí modul *Storable*, který sice nepatří mezi standardní výbavu Perlu, ale lze jej získat na serverech CPAN.

Modul nabízí několik základních podprogramů, ty nejzajímavější jsou uvedeny v tabulce 11.1. Funkce *dclone* dostane jako parametr odkaz na datovou strukturu. Jejím výsledkem je přesná kopie

dclone(»*odkaz*«)	vydá kopii datové struktury
store(»*odkaz*«,»*soubor*«)	uloží do souboru
nstore(»*odkaz*«,»*soubor*«)	uloží v přenositelném formátu
retrieve(»*soubor*«)	načte ze souboru
freeze(»*odkaz*«)	zakóduje do řetězce
nfreeze(»*odkaz*«)	zakóduje v přenositelném formátu
thaw(»*řetězec*«)	rekonstruuje z řetězce

Tabulka 11.1: Podprogramy z modulu *Storable*

dotyčné struktury (bez ohledu na její složitost). Pokud například proměnná *$opole* obsahuje odkaz na dvourozměrné pole (tedy pole odkazů na další pole), pak příkaz:

```
$odruhepole = dclone( $opole );
```

vytvoří druhé dvourozměrné pole se stejným obsahem a odkaz na ně přiřadí do proměnné *$odruhepole*. Naproti tomu prosté:

```
$otretipole = $opole;
```

vloží do proměnné *$otretipole* stejnou hodnotu, jako má *$opole*. Jinými slovy po provedení těchto dvou příkazů budou proměnné *$opole* a *$otretipole* ukazovat na tutéž datovou strukturu, zatímco *$odruhepole* bude obsahovat odkaz na její přesnou kopii, která je ale od svého vytvoření zcela samostatná a případné změny originálu na ni už nebudou mít žádný vliv.

K vytvoření kopie je nutné, aby funkce *dclone* postupně procházela celou strukturou, na niž dostala odkaz. Obdobně se chová podprogram *store*, který však nalezené hodnoty a jejich vzájemné vazby ukládá do souboru. Určitým problémem je, že tyto podprogramy musí jako parametr dostat adresu datové struktury. Když tedy chcete uložit například třídu, která je polem odkazů, musíte uměle vytvořit odkaz na ni:

```
store( \@trida, "trida.dat" );
```

Druhým parametrem podprogramu je název souboru, do něhož se data mají uložit. O vyzvednutí se postará funkce *retrieve*, která dostane jako parametr název souboru. Načte jeho obsah, podle něj postaví datovou strukturu shodnou s uloženou a jako výsledek vydá odkaz na ni. Pokud jej potřebujete uložit do proměnné neodkazovitého charakteru, musíte si jej „rozbalit“:

```
@trida = @{ retrieve( "trida.dat" ) };
```

Modul nabízí ještě podprogram *nstore*, který je sice pomalejší než *store*, ale ukládá data v přenositelném formátu. Ten pak můžete přenést na jinou platformu a načíst obvyklým *retrieve*.

Chcete-li si ukládání raději zajistit sami, nabízí modul ještě funkce *freeze* a *thaw*. První dostane odkaz na datovou strukturu, kterou projde a zakóduje do řetězce znaků, který vydá jako svůj výsledek. Můžete jej uložit do proměnné nebo třeba do databáze. Když později tento řetězec předhodíte funkci *thaw*, rekonstruuje z něj původní datovou strukturu a vrátí odkaz na ni. Funkce *store* interně volá *freeze* a její výsledek uloží do souboru. Analogicky *retrieve* získá ze souboru hodnotu a použije *thaw*, aby z ní postavila datovou strukturu. Pro „zmrazení" v přenositelném formátu slouží funkce *nfreeze*.

Příklad: S využitím modulu *Storable* bych mohl výrazně zjednodušit modul pro ukládání a načítání třídy z příkladu ze strany 160:

Trida/SouborySt.pm

```
package Trida::SouborySt;

use Exporter 'import';
our @EXPORT = qw(uloz_tridu nacti_tridu);

use Log;
use Storable;

sub nacti_tridu {
    print "Jméno souboru: ";
    my $jmenodat = <STDIN>; chomp( $jmenodat );
    do_logu( "Načteno ze souboru $jmenodat.\n" );
    return @{retrieve($jmenodat)};
}

sub uloz_tridu {
    print "Jméno souboru: ";
    my $jmenodat = <STDIN>; chomp( $jmenodat );
    store(\@_, $jmenodat);
    do_logu( "Uloženo do souboru $jmenodat.\n" );
    return 1;
}

1;
```

Zároveň je to pěkná ukázka charakteristické vlastnosti modulů – přestože jsem zcela přepracoval implementaci a třídy se ukládají úplně jinak, rozhraní zůstává stejné (podprogramy *uloz_tridu* a *nacti_tridu* s nezměněnými parametry a výslednými hodnotami) a pro zbytek programu se vůbec nic nemění. ■

Při práci se složitějšími strukturami a především při ladění takových programů občas potřebujete zobrazit, co vlastně daná struktura momentálně obsahuje. Docela rozumně použitelné prostředky vám dává k dispozici modul *Data::Dumper*. Nabízí funkci *Dumper*, která dostane odkaz a vydá jej v podobě formátovaného textu.

Příklad: Následující program:

```
use Data::Dumper;

my $pole = [ 15, "Adam", 33.2, "Božena" ];
$pole->[1] = { "raz"=>1, "dva"=>2 };
print Dumper($pole);
```

vytvoří tento výstup:

```
$VAR1 = [
          15,
          {
            'dva' => 2,
            'raz' => 1
          },
          '33.2',
          'Božena'
        ];
```

■

Existují i složitější způsoby použití modulu (včetně objektově orientovaného), ale tento základní ve většině případů postačí.

12 Styk s okolím

V této kapitole popíši cosi, co by se dalo nazvat ministerstvem zahraničí Perlu. Čili mechanismy a postupy, které mu umožňují komunikovat s okolním světem, ostatními programy a operačním systémem.

12.1 Příkazový řádek

Jedním ze základních způsobů, jak předávat informace programu v prostředí řádkově orientovaného operačního systému, je příkazový řádek. Jeho prostřednictvím můžete zadat informace ke zpracování či pomocí voleb ovlivnit chování programu. Pokud to umí. Nyní se podíváme, jak zařídit, aby perlivý program uměl.

Elementární spolupráci s příkazovým řádkem zajistí diamantový operátor <>. Díky němu lze programu předložit soubor či soubory, které má zpracovat. Ovšem <> se zabývá pouze jejich obsahem. Neříká například nic o tom, jak se který soubor jmenuje a dokonce ani nepoznáte, kdy jeden skončil a začal další. Kdybyste třeba chtěli sestrojit program, který dostane několik jmen souborů a vypíše pro každý z nich počet řádků, <> vám nestačí.

Zajímáte-li se o parametry a volby, které váš program dostal do vínku, poslouží vám pole *@ARGV*. Obsahuje skutečně jedině parametry, nikoli jméno vašeho programu, jak bývá zvykem u stejnojmenného pole v jiných jazycích. Jméno programu Perl ukládá do speciální proměnné *$0*. Takže pokud zadáte:

```
delky pokus.c dopis.txt db.h
```

budou hodnoty jednotlivých proměnných:

$0 = ”delky”
$ARGV[0] = ”pokus.c”
$ARGV[1] = ”dopis.txt”
$ARGV[2] = ”db.h”

Jistě si vzpomínáte na proměnné *$1* až *$99*, které obsahují řetězce zapamatované z regulárního výrazu. Zdejší *$0* nemá s regulárními výrazy vůbec nic společného, což je lehce matoucí.

Příklad: Zmiňovaný program, který spočítá délky souborů zadaných jako parametry, může vypadat třeba takto:

```
foreach my $jmeno ( @ARGV ) {                                    delkysou.pl
    open(DATA, $jmeno) or next;
    my $radku = 0;
    while ( <DATA> ) { $radku++; }
    printf "%10d %s\n", $radku, $jmeno;
}
```

Nemožnost otevřít soubor by neměla být fatální – uživatel se prostě přepsal ve jméně souboru. Proto jej jednoduše přeskočím. ■

Obvyklým způsobem ovlivňování činnosti států a programů jsou volby. Na rozdíl od států je v softwarovém světě situace zcela jednoduchá: napíšete na příkazový řádek cosi jako `-n` a program změní své chování.

Jednoduchý způsob, jak zpracovávat volby z příkazového řádku, představuje standardní modul *Getopt::Std*. Jeho základní službou je funkce:

getopts(»*charakteristika_voleb*«)

kde »*charakteristikou_voleb*« je obyčejný řetězec znaků. Obsahuje písmeno pro každou volbu, kterou váš program přijímá. Jedná-li se o volbu typu ano/ne, stačí uvést jen písmeno. Pokud volba očekává hodnotu, přidejte za ni dvojtečku.

Funkce *getopts* vytáhne z příkazového řádku všechny řetězce, které vypadají jako volby: leží mezi jménem programu a jeho parametry a začínají pomlčkou. V jejich zápisu je uživateli ponechána obvyklá volnost: „`-ab`“ je totéž co „`-a -b`“; hodnotu lze psát hned za volbu nebo ji oddělit mezerou. Pro každou volbu, kterou funkce najde, zavede globální proměnnou *$opt_x* (kde *x* je písmeno dané volby). U pravdivostních voleb do ní přiřadí 1, u voleb s hodnotou převezme hodnotu z příkazového řádku. Pro volby, které se na příkazovém řádku nevyskytují, zůstane *$opt_X* nedefinováno. V poli *@ARGV* ponechá jen zbývající řetězce (tedy skutečné parametry programu).

Příklad: Chci mít v programu pravdivostní volby `-h` a `-R` a volbu `-f`, která očekává jméno souboru. Začátek programu bych konstruoval takto:

```
use Getopt::Std;
getopts( "hRf:" );
```

Pokud by uživatel dotyčný program zavolal příkazem:

```
program -h -bf pokus.txt head.c tail.txt
```

obdržel by chybové hlášení:

```
Unknown option: b
```

a jednotlivé proměnné by získaly následující hodnoty:

$opt_h = 1
$opt_f = ”pokus.txt”
$ARGV[0] = ”head.c”
$ARGV[1] = ”tail.txt”

Proměnná *$opt_R* by zůstala nedefinována, protože příkazový řádek neobsahuje volbu `-R`. ■

Funkci *getopts* můžete také jako druhý parametr předat odkaz na asociativní pole – například *getopts*(”hRf:”,*%volby*). V tom případě se hodnoty voleb uloží do tohoto pole. Tedy místo proměnné *$opt_h* bude nastaveno *$volby*{”h”}.

Omezením modulu *Getopt::Std* je, že podporuje pouze klasické jednopísmenné volby, jak je známe z ortodoxních dob Unixu. Pokud chcete dlouhé názvy ve stylu „`--help`“, poslouží vám modul *Getopt::Long*. Jeho ovládání je o něco složitější, takže je přenechám vašemu sebevzdělávání.

Cvičení 12.1: Napište analogii příkazu *cat*. Dostane jako parametr seznam souborů, jejichž obsah opíše do standardního výstupu. Program by měl přijímat volbu „`-r`“, která způsobí, že řádky vystoupí v obráceném pořadí: jako první poslední řádek posledního souboru, pak předposlední řádek posledního souboru atd. až nakonec první řádek prvního souboru. ■

12.2 Proměnné prostředí

Tak tohle bude opravdu krátké. Proměnné prostředí Perl zpřístupňuje prostřednictvím asociativního pole *%ENV*. Jako klíče k němu slouží názvy proměnných, hodnotou je hodnota dotyčné proměnné. Takže například obsah proměnné prostředí *PATH* získáte v Perlu pomocí konstrukce *$ENV*{”PATH”}.

Příklad: Jednou z užitečných konvencí Unixu jsou standardní názvy pro individuální konfigurační soubory programů. Bývají uloženy v uživatelově domácím adresáři a mají jméno *.»program«rc* (například *.vimrc* pro konfiguraci editoru *vim*). Podprogram, který zajistí načtení takového konfiguračního souboru, může vypadat takto:

konfig.pl

```
sub cti_konfiguraci {
    my $jmenokonf = $ENV{"HOME"}."/.$0rc";
    if ( open(KONFIG,$jmenokonf) ) {
        while ( my $radek = <KONFIG> ) {
            # zpracování konfiguračních příkazů...
        }
        close(KONFIG);
    }
}
```

Všimněte si, že díky použití proměnné *$0* obsahující název běžícího programu je dotyčný podprogram zcela obecný a můžete jej do svého programu včlenit tak, jak je. Stačí jen dodefinovat zpracování obsahu jednotlivých konfiguračních řádků. ■

Zdálo by se, že přístup k proměnným prostředí prostřednictvím asociativního pole *%ENV* je tak jednoduchý, že to snadnější být ani nemůže. Může.

Mezi standardními moduly Perlu najdete i modul *Env*, který pro každou existující proměnnou prostředí zavede stejnojmennou globální proměnnou. Takže s proměnnou prostředí *HOME* můžete ve svém programu zacházet stejně jako v interpretu příkazů – prostřednictvím *$HOME*.

A jak se modul používá? Je to triviální. Stačí napsat pouhé:

```
use Env;
```

a proměnné jsou vaše.

12.3 Spouštění externích programů

Časem možná zlenivíte a dospějete k závěru, že nemá cenu psát všechno sami. Že se zkrátka čas od času hodí sáhnout po některém již existujícím programu a pouze využít jeho výsledky. Máte k dispozici hned několik alternativ.

Nejvyšší míru autonomie přidělíte externímu programu prostřednictvím standardní funkce **system**. Jako parametr dostane řetězec znaků tvořící příkazový řádek. Funkce **system** zavolá interpret příkazů a nechá jej vykonat dotyčný příkaz. Přitom jeho standardní vstup a výstup napojí na své, takže uživatel může s programem normálně komunikovat.

Perlovský program je pozastaven a čeká na ukončení volaného programu. Teprve pak pokračuje ve své činnosti. Výsledkem funkce **system** je návratový kód, kterým skončil spuštěný program. Tento způsob spouštění se používá, pokud externí program souvisí s vaším kódem jen volně.

Příklad: Řekněme, že budete psát v Perlu jednoduchý nástěnkový systém, který umožní uživatelům zveřejňovat různá oznámení. Chtěli byste, aby texty byly vytvářeny interaktivně, ale z pochopitelných důvodů se vám nechce psát textový editor. Takže si zavoláte externí a pokud vše dopadne dobře, jen vystavíte vytvořený soubor do veřejné části nástěnek:

```
$status = system( "vim $jmenosouboru" );
if ( $status == 0 and –s $jmenosouboru ) {
      vystav_soubor( $jmenosouboru );
} else {
      print "Příspěvek se nepodařilo vytvořit.\n";
}
```

■

Existují však i těsnější modely spolupráce, kdy externí program živíte svými daty nebo naopak přebíráte a dále zpracováváte jeho výstup. Zde poslouží stará dobrá funkce **open**, která místo jména souboru dostane jako druhý parametr příkazový řádek začínající nebo končící znakem „|“. Stejně jako v interpretu příkazů i zde ztělesňuje rouru.

Uvedete-li jej na začátku řetězce, představuje rouru vedoucí do spouštěného programu. Cokoli zapíšete do příslušného ovladače souboru, bude předáno na standardní vstup programu. Pokud řetězec znakem „|“ zakončíte, povede roura z programu a data z něj můžete přebírat obvyklým načítáním.

Příklad: Vyrobím program *on-line*, který dostane jako parametr uživatelské jméno a ohlásí, zda dotyčný uživatel je či není momentálně přihlášen. Ke své funkci využívá standardní Unixovský program *who*, který vypíše seznam právě přihlášených uživatelů a pár informací o nich. Jméno uživatele je uvedeno vždy na začátku řádku.

on-line.pl

```
if ( not $ARGV[0] ) {
      print "Použití: on–line uživatelské_jméno\n";
      exit;
}

my $jetu = 0;
open(UZIV, "who |")
      or die "Nelze spustit příkaz who\n";
```

```
while ( my $radek = <UZIV> ) {
    if ( $radek =~ /^$ARGV[0]\s/ ) {
        $jetu = 1;
        last;
    }
}
close(UZIV);

if ( $jetu ) {
    print "$ARGV[0] je přihlášen(a)\n";
} else {
    print "$ARGV[0] není přihlášen(a)\n";
}
```

Jak vidíte, není na spolupráci s externím programem nic magického. Zpracovává se úplně stejně, jako běžný soubor. ■

Bohužel nelze umístit roury na oba konce. Z externího programu můžete buď číst data nebo mu je předávat, nikoli však oboje.

Cvičení 12.2: Pokuste se napsat podprogramy *uloz_pole* a *nacti_pole*, které uloží resp. načtou obsah globální proměnné *@pole* do/ze souboru, jehož jméno dostanou jako parametr. Soubor by měl být uložen v komprimovaném tvaru prostřednictvím programu *gzip*.

Lehce napovím: *gzip* bez parametrů komprimuje svůj standardní vstup a posílá jej do standardního výstupu. *gunzip* s volbou `-c` posílá rozbalenou verzi souboru na svůj standardní výstup. ■

12.4 Interaktivní programy

Často se na to zapomíná, ale součástí okolí programu je také jeho uživatel. Přestože Perl původně vznikl spíše za účelem výroby programů pracujících neinteraktivně, má i v této části co nabídnout. Kvalita komunikace s uživatelem ještě výrazně vzroste, když použijete některé z rozšiřujících modulů.

Bohužel musím ponechat stranou grafické uživatelské rozhraní. Existují moduly, které umožňují psát programy využívající nejrozšířenější grafické knihovny Unixu: klasické Tcl/Tk i moderní Qt a Gtk. Ovšem jejich používání je pro tuhle knížku příliš velké sousto. Třeba někdy příště…

Vrátím se proto zpět do textového režimu a prozradím pár jednoduchých modulů, které vylepšují styk s uživatelem. Co nám chybí? Přímý vstup kláves (aby uživatel nemusel při volbě z menu mačkat **Enter**) a větší možnosti při práci s obrazovkou.

Přímé načítání kláves zajišťuje modul *Term::ReadKey*, který získáte v archivu CPAN. Nabízí dvě základní funkce: *ReadMode* pro změnu režimu práce se vstupem a *ReadKey* pro vlastní načtení klávesy. K přímému načítání musíte nejprve přepnout vstup do režimu „cbreak“:

```
ReadMode( ”cbreak” );
```

Když nyní použijete:

```
$klavesa = ReadKey(0);
```

funkce *ReadKey* počká, dokud uživatel nestiskne klávesu. Jakmile to udělá, vydá příslušný znak jako svůj výsledek. Parametr funkce určuje způsob čtení, nulu berte prosím jako dogma. Je záhodno po načtení znaku vrátit vstup do režimu „normal“.

Příklad: Jednoduchá ukázka použití modulu:

anone.pl

```
use Term::ReadKey;

print ”Souhlasíte [a/n]? ”;
if ( ano_ne() ) {
    print ”Jsme dohodnuti.\n”;
} else {
    print ”No tak ne.\n”;
}

sub ano_ne {
    ReadMode(”cbreak”);
    my $znak = ReadKey(0);
    ReadMode(”normal”);
    if ( $znak =~ /a/i ) {
        return 1;
    } else {
        return 0;
    }
}
```

Funkce *ano_ne* načte od uživatele jeden znak a pokud se jednalo o „a“ či „A“, vydá pravdivou hodnotu. ■

Existují i další moduly pro vylepšování vstupů. Zajímavý je například *Term::ReadLine*, který implementuje příkazový řádek s historií a umožňuje uživateli editovat jeho obsah. Je však založen na objektech, takže s jeho zkoumáním ještě chvíli počkejte.

Rozhraní programu lze citelně vylepšit použitím barev. Za tímto účelem si v CPAN můžete obstarat modul *Term::ANSIColor*. Jeho nosným prvkem je funkce *color*, která jako parametr dostane řetězec určující barvu a odmění se kódem pro ANSI terminál, který ji zapne. Typicky se používá uvnitř **print** a zpravidla se kolem jejího parametru nepíší závorky, protože to vypadá dobře:

```
print color "blue on_white";
```

Slov určujících požadovaný vzhled písma je kolem dvaceti. Jejich přehled vám poskytne *man Term::ANSIColor*.

Uvedený způsob použití funguje jako přepínač. Dokud písmo nezměníte, budou všechny výstupy modré na bílém pozadí. O návrat k implicitnímu vzhledu se postará řetězec „reset“.

Příklad: Jeden z prvků, které je záhodno barevně odlišit, jsou chybová hlášení. Následující zdrojový kód definuje podprogram *ohlas_chybu*, který dostane jako parametr text chybového hlášení. Vypíše jej tučným bílým písmem na červeném pozadí.

barvy.pl

```
use Term::ANSIColor;

print "Běžný nápis.\n";
ohlas_chybu("Chybové hlášení!\n");
print "Další běžný nápis.\n";

sub ohlas_chybu {
    my ( $napis ) = @_;
    print color("bold white on_red");
    print $napis;
    print color "reset";
}
```

■

⚡ Barvy jsou dvojsečná zbraň a je potřeba je používat s rozmyslem. Snažte se být raději střídmí než hýřiví. Definujte si jasné schéma, jak budou vypadat jednotlivé typy informací. A nemusí mít každý svou vlastní barvu odlišnou od ostatních! Méně je skoro vždy více.

Barva by měla rozhraní vašeho programu zpřehledňovat, ne naopak. Pokud *totiž* zvýrazníte každé **druhé** slovo, vznikne NESROZUMITELNÝ a **totálně** *nepřehledný* s̬ɐ̗lnɓ.

Hodláte-li podnikat větší výboje ve směru interaktivních programů, budete potřebovat nástroje pro práci s obrazovkou. To je bohužel opět celkem rozsáhlá tématika, takže jen poradím, že vám poslouží modul *Curses*, který je k dostání v archivu CPAN.

12.5 Čas

Důležitou složkou okolního prostředí je čas. Prostředky, které vám Perl dává pro práci s ním, jsou dost minimalistické, ale buďme rádi alespoň za ně. Základní způsob ukládání časových údajů převzal z operačního systému Unix. Je jím prosté celé číslo udávající počet sekund, které uplynuly od 1. ledna 1970.

Chcete-li získat aktuální čas v tomto tvaru, použijte funkci **time**. Nemá žádné parametry a jako výstupní hodnotu vydá zmíněný počet sekund.

Použitá reprezentace času je sice úsporná a používá ji řada funkcí, ovšem pro komunikaci s člověkem je třeba nabídnout něco srozumitelnějšího. O polidštění časových údajů se postará funkce **localtime**. Jejím parametrem je časový údaj v interním tvaru. Jako výstup vydá seznam devíti čísel, jehož jednotlivé položky uvádí tabulka 12.1.

index	*význam*
0	sekundy
1	minuty
2	hodiny
3	den v měsíci
4	měsíc (z intervalu 0…11)
5	rok (zmenšený o 1900)
6	den v týdnu (z intervalu 0…6)
7	den v roce (z intervalu 0…365)
8	příznak letního času

Tabulka 12.1: Výstupní hodnoty **localtime**

Počínaje měsícem jsou hodnoty různě upravovány, takže je třeba dávat si na ně pozor. Intervaly většinou začínají od nuly, aby byly hodnoty snadno použitelné jako index v poli.

Příklad: Za použití funkcí **time** a **localtime** si snadno postavíte program, který ohlásí aktuální čas.

```
my @mesice = qw(leden únor březen duben květen červen                    cas.pl
                červenec srpen září říjen listopad prosinec);
my @denvtydnu = qw(neděle pondělí úterý středa čtvrtek pátek sobota);
my @cas = localtime(time);

print "Přesný čas: $cas[3]. $mesice[$cas[4]] ";
print $cas[5] + 1900, " ($denvtydnu[$cas[6]]), ";
printf "%d:%02d:%02d", $cas[2], $cas[1], $cas[0];
if ( $cas[8] > 0 ) { print " (letní čas)"; }
print "\n";
```

Vytvoří výstup ve tvaru:

```
Přesný čas: 18. srpen 2018 (sobota), 13:00:50 (letní čas)
```

■

Vedle **localtime** nabízí Perl ještě podobnou funkci **gmtime**. Fungují úplně stejně, rozdíl je jen v časové zóně, kterou používají. **localtime** vydá výsledný čas upravený podle časové zóny, kterou máte nastavenu ve svém počítači. Měl by odpovídat tomu, co máte na hodinkách. Naproti tomu **gmtime** přepočítává na greenwichský čas.

Obrácenou konverzi (ze sekund, minut, hodin atd. na interní tvar) byste ve standardních funkcích hledali marně. Nabízí ji modul *Time::Local* v podobě funkcí:

timelocal(*sek*, *min*, *hod*, *den*, *měs*, *rok*)
timegm(*sek*, *min*, *hod*, *den*, *měs*, *rok*)

Každá z nich vydá jako výstupní hodnotu uvedený čas přepočítaný na počet sekund od 1. ledna 1970. Opět se liší pouze časovou zónou.

13 Objektivně vzato

Když ve skupině programátorů pronesete slova „objektově orientované programování“, dostaví se podobný efekt, jako když mezi -náctiletými dívkami řeknete „Mandrage“. Většina upadne do nekontrolovatelného třeštění, bude poskakovat, výskat a vydávat jiné neartikulované, leč zjevně pochvalné zvuky. Pár s vámi okamžitě přestane mluvit a stanou se vašimi zarytými nepřáteli do konce života (přinejmenším do zítřka). A konečně zbývající zanedbatelný hlouček příliš neví, o čem je řeč, a usilovně přemýšlí, ke které z předchozích dvou skupin se přidat.

Zkrátka objektově orientované programování je módní pojem. Pravda, ve srovnání s Mandrage je populární už poměrně dlouho (někdy od konce osmdesátých let) a zatím se nezdá, že by na programátorském nebi vycházela nějaká nová hvězda, která by hrozila tuto zastínit[1].

13.1 Základní principy

Ve vývoji programovacích jazyků a technik je patrná jasná snaha o co největší přiblížení reprezentace dat a práce s nimi skutečnému světu kolem nás. Stále se hledají takové abstrakce, které by věrně, logicky a jednoduše odrážely objekty a vztahy z našeho okolí. Objektově orientované programování (OOP) je jedním z nejrozšířenějších kroků na této cestě.

Nosná myšlenka je poměrně prostá: pro reprezentaci prvků reálného světa je výhodné spojit data s podprogramy, které s nimi pracují. Například když budete psát grafický program, reprezentujete každý grafický prvek (bod, úsečku, kružnici) prostřednictvím objektu. Ten bude mít některé pasivní datové složky (souřadnice na obrazovce, barvu a podobě) a některé aktivní podprogramy (např. se umí nakreslit či přesunout na jinou pozici). Těmto podprogramům, které jsou svázány s objekty, se v terminologii OOP říká *metody*.

Základní myšlenka bývá rozvinuta do trojice principů:

Zapouzdření je právě ono spojení dat s podprogramy do jednoho společného obalu. U ortodoxních objektově orientovaných jazyků je dokonce zakázáno manipulovat přímo s datovými položkami objektů. Jediné, co máte k dispozici, jsou metody. Perl tomuto trendu přitakává. Samozřejmě ne tak, že by vám něco zakazoval[2]. Ale prohlásil přímý přístup k proměnným objektu za nežádoucí. Perlivá programátorská etika velí používat jen metody.

1: Trochu mu začíná foukat do polévky funkcionální programování. To je koncept ještě starší, nicméně poslední dobou zájem o ně zřetelně roste.

2: Pokud jste se knihou pročetli až sem, jistě něco takového neočekáváte.

Dědičnost vám umožňuje budovat mezi objekty hierarchické vztahy. Například můžete říct, že objekt typu „obdélník" je pouhým zpřesněním – potomkem – obecnějšího objektu „grafický prvek". Obdélník tak zdědí veškeré součásti typu grafický prvek (např. souřadnice či viditelnost) a může si je rozšířit nebo pozměnit (v případě obdélníku jistě bude třeba přidat délky stran, zatímco pro kružnici vás bude zajímat poloměr).

Polymorfismus zajišťuje, že chování metod se může měnit v závislosti na tom, jaký konkrétní objekt je použit. Například obrázek by byl reprezentován skupinou objektů typu „grafický prvek". Tento typ nabízí metodu „nakresli se", která způsobí vykreslení dotyčného prvku. Pokud potřebujete nakreslit obrázek, projdete všechny členy skupiny a každému přikážete „nakresli se". Ovšem skutečnými členy nejsou obecné grafické prvky, ale jejich konkrétnější potomci: obdélníky, elipsy, úsečky… Polymorfismus zajišťuje, že se u každého z nich zavolá jeho konkrétní metoda „nakresli se", která se pochopitelně bude u různých grafických objektů lišit (jinak se nakreslí bod, jinak obdélník, jinak elipsa). Objektový model Perlu je veskrze dynamický a kontroly mizivé, takže polymorfismus je u něj samozřejmostí.

V souvislosti s OOP se často vyskytuje pojem *třída*[3]. Je to vlastně typ objektů. Jakýsi obecný prototyp, který říká, jaké datové položky a metody nabízí objekty, které do ní patří. Jako *objekt* je pak nazýván určitý konkrétní zástupce této třídy s konkrétními hodnotami datových položek. Někdy se pro něj používá pojem instance.

Udělám-li analogii s houbařením, jsou „houba" či „muchomůrka červená" třídy. Popisují obecné vlastnosti houby resp. muchomůrky červené. Houbařský atlas je pak vlastně katalogem tříd. Když na pasece najdete tři muchomůrky červené, jedná se o trojici objektů (konkrétních představitelů, instancí) třídy muchomůrka červená, která je podtřídou (potomkem) obecnější třídy houba.

13.2 Objekty a třídy v Perlu

Perl není od kolébky objektově orientovaný jazyk. Koncepce objektů a mechanismy pro práci s nimi byly zařazeny až do jeho páté verze. Snad už jste si zvykli na perlovský minimalismus, takže by vás nemělo překvapit, že objekty jsou realizovány prostřednictvím stávajících konstrukcí jazyka a celé objektově orientované programování je v podstatě soubor zásad a doporučení pro programátory.

Aby to bylo ještě trochu složitější, základní konstrukce Perlu byly programátorům málo pohodlné. Postupem času se proto objevilo několik modulů, které se snažily objektově orientované programování usnadnit a zároveň přiblížit zvyklostem z jiných programovacích jazyků. Pozici de facto standardu si vydobyl modul *Moose*. Budu se jej držet i já.

3: Navzdory nejmenovanému vousáči to v programování k beztřídní společnosti asi nikdy nedotáhneme.

Třída je reprezentována balíkem. Z obyčejného balíku uděláte třídu tak, že v ní použijete *Moose* a pomocí *has* deklarujete jeho datové položky. Podprogramy definované v balíku se stávají metodami této třídy.

Vytváření datových položek má na starosti *has*, jemuž předáte jméno položky a atributy určující její vlastnosti. Syntaxe je proměnlivá, držme se tvaru z dokumentace *Moose*:

has »název« => (*»atr1«* => *»hod1«*, *»atr2«* => *»hod2«*, …)

Různých atributů je k dispozici kolem dvaceti, ty nejvýznamnější shrnuje tabulka 13.1. Musíte především stanovit atributem *is*, jak se s položkou bude pracovat. Jestli bude její hodnota přiřazena při vytvoření objektu a dále už se bude jen číst, nebo může být kdykoli změněna.

název	význam
is	lze jej měnit ('rw') nebo ne ('ro')?
isa	datový typ hodnoty
default	implicitní hodnota

Tabulka 13.1: Nejběžnější atributy datových položek pro *has*

Pomocí *isa* lze oznámit, jakého typu bude její hodnota. Perl si na typy obecně moc nehraje, takže dopady nebudou zásadní. Nicméně *Moose* vám zajistí alespoň základní kontroly při práci s příslušnou položkou. Několik nejběžněji používaných typů najdete v tabulce 13.2. Výchozí hodnoty datových položek se obvykle zadávají při vytvoření objektu. Pokud ji nechcete zadávat, můžete atributem *default* předepsat její výchozí hodnotu.

název	význam
Any	libovolný, výchozí hodnota
Str	řetězec znaků
Num	jakékoli číslo
Int	celé číslo
ArrayRef	odkaz na pole
Object	odkaz na objekt

Tabulka 13.2: Nejběžnější typy datových položek pro *isa*

Jak jsem již prozradil, metody třídy mají podobu běžných podprogramů, zařazených do daného balíku. Jednu specialitu ale přece jen mají: Když je zavoláte, Perl automaticky přidá první parametr, kterým je odkaz na objekt, pro který byla metoda vyvolána, resp. název třídy (pokud byla metoda vyvolána pro třídu). V těle metody s tím musíte počítat. Zavedenou konvencí je používat pro něj jméno *$self* a doporučuji se jí držet. Těla metod typicky začínají příkazem:

```
my $self = shift;
```

kterým se přidaný parametr odstraní z pole @_ a přesune do lokální proměnné *$self*.

Ale dost už povídání, pojďme do první třídy.

Příklad: Vytvořím základ potenciálního programu pro řešení automobilistických úloh. Nejprve předvedu definici třídy *Auto*. Jelikož se jedná o modul, bude uložena v souboru *Auto.pm*.

Objekty této třídy budou mít tři datové položky: SPZ (řetězec znaků), průměrnou spotřebu (číslo) a stav nádrže (číslo). První dvě se nemění[4], takže je zavedu jako „jen pro čtení", zatímco stav nádrže bude pracovat. U něj také přidám nulovou výchozí hodnotu. Definice třídy *Auto* by vypadala nějak takto:

Auto.pm

```
package Auto;
use Moose;
has spz => ( is => 'ro', isa => 'Str' );
has spotreba => ( is => 'ro', isa => 'Num' );
has nadrz => ( is => 'rw', isa => 'Num', default => 0 );

sub tankuj {
    my $self = shift;
    my $kolik = shift;
    $self->nadrz( $self->nadrz + $kolik );
}

sub jed {
    my $self = shift;
    my $km = shift;
    my $spaleno = ( $km/100 ) * $self->spotreba;
```

4: Teď nepochybně vyskočili metr vysoko všichni notoričtí automobilisté, protože spotřeba přece závisí na tlaku, teplotě, rosném bodu, opotřebení motoru, stylu jízdy a řadě dalších faktorů, zatímco byrokrati si klotovým rukávem otřeli studený pot z čela, protože každý přece ví, že vyplněním formuláře 239c s přílohami Be, Fe, Le a Me se dá změnit SPZ vozidla. Prostě zjednodušuji, to už se tak při programování dělává.

```
        if ( $spaleno <= $self->nadrz ) {
            $self->nadrz( $self->nadrz - $spaleno );
            return 1;
        } else {
            print "Autu ", $self->spz, " došel benzín.\n";
            $self->nadrz(0);
            return 0;
        }
    }

    sub stav {
        my $self = shift;
        print "Auto ", $self->spz, "\n";
        print "Nádrž: ", $self->nadrz, "\n";
    }

    return 1;
```

Přidal jsem zároveň tři metody. *tankuj* dostane počet litrů a přidá je do nádrže. O něco složitější je metoda *jed*, která s autem „popojede". V mé jednoduché reprezentaci se jízda projeví jen úbytkem benzínu v nádrži. Metoda *jed* dostane vzdálenost, kterou má automobil ujet. Vypočítá spotřebovaný benzín, odebere jej z nádrže a jako výsledek svého volání vydá, zda auto dojelo do cíle nebo mu cestou došel benzín, což zároveň vypíše. Metoda *stav* vypíše aktuální informace o vozidle.

V tělech metod se používají zatím poněkud tajemné konstrukce, které hnedle objasním.

■

Objekt je ukazatelem na asociativní pole, který byl přidělen dotyčné třídě. O jeho vytvoření se postará konstruktor – speciální metoda, která má na starosti inicializaci objektu a přiřazení výchozích hodnot datovým položkám. Pokud jste položku omezili pouze na čtení, je toto jediné místo, kde jí lze přiřadit hodnotu.

Modul *Moose* pro každou třídu automaticky vytvoří konstruktor pojmenovaný *new*. Hodnoty datových položek mu předáte v podobě stejnojmenných pojmenovaných parametrů. Vynecháte-li některý, přiřadí mu implicitní hodnotu nebo (není-li ve třídě určena) nedefinovanou hodnotu. Například:

```
my $muj_vuz = Auto->new( spz=>"1L2 3456", spotreba=>5.9 );
```

uloží do proměnné *$muj_vuz* odkaz na objekt třídy *Auto*, jehož datové položky *spz* a *spotreba* získají uvedené hodnoty a do *nadrz* se přiřadí implicitní nula, protože mezi parametry *new* chybí. Konstruktor hlídá datové typy. Pokud bych se pokusil přiřadit něco nepřípadného, třeba:

```
my $muj_vuz = Auto->new( spz=>"1L2 3456", spotreba=>"ok" );
```

dostanu chybové hlášení, že „ok“ není číslo.

Pro volání metod se opět používá šipková notace. Příklad už jsem vlastně předvedl na konstruktoru, častěji se ovšem volají metody konkrétních objektů, nikoli tříd. Volání má tvar:

```
»objekt«->»metoda«(»parametry«);
```

Svůj vůz bych tedy natankoval pomocí:

```
$muj_vuz->tankuj( 35 );
```

Vraťme se k metodám třídy *Auto* z příkladu. V jejich tělech se na několika místech používají volání typu:

```
$self->nadrz
$self->nadrz(0)
```

$self je lokální proměnná, do níž jsem na začátku metody uložil její první parametr, čili odkazuje na objekt, pro který byla volána. Jedná se tedy o standardní volání metody objektu až na to, že metoda *nadrz* není nikde definována. Jedná se o další konstrukci, kterou automaticky vytváří modul *Moose*. Pro každou datovou položku zavedenou pomocí *has* vytvoří stejnojmennou přístupovou metodu. Zavoláte-li ji bez parametru, vrátí aktuální hodnotu položky. Pokud dostane parametr, uloží jeho hodnotu do položky, ovšem jen když byla v *has* definována jako zapisovatelná.

Příklad: Primitivní program používající třídu *Auto* by vypadal třeba takto:

jezdec.pl

```
use Auto;

my $muj_vuz = Auto->new( spz => '1L2 3456', spotreba => 7.3 );
$muj_vuz->tankuj( 10 );
if ( $muj_vuz->jed(110) ) {
    print "Sláva, jsem v Praze.\n";
} else {
    print "Budu muset dotankovat.\n";
}
$muj_vuz->stav();
```

Jeho výstupem bude:

```
Sláva, jsem v Praze.
Auto 1L2 3456
Nádrž: 1.97
```

■

Existuje i metoda určená pro likvidaci objektu. Nazývá se destruktor a její jméno je povinné: *DEMOLISH*. Pokud má objekt definovánu metodu *DEMOLISH*, bude automaticky zavolána v okamžiku, kdy se správa paměti rozhodne jej odstranit, protože přestal být dostupný.

Destruktor by měl zajistit úklidové práce, aby po objektu nezůstaly nějaké nedodělky. Například pokud objekt zprostředkovává přístup k souboru, měl by jej destruktor zavřít. V jiných jazycích bývá nejčastější úlohou destruktorů uvolnění dynamicky přidělené paměti. To však Perl obvykle zvládne automaticky, takže si troufnu tvrdit, že destruktory se zde vyskytují jen velmi vzácně.

Cvičení 13.1: Zpátky do škamen. Opět se vrátím k příkladu s evidencí třídy a tentokrát budu chtít, abyste napsali třídu *Zak*, jejímiž objekty budete reprezentovat jednotlivé žáky. Třída *Zak* by měla obsahovat:

- datovou položku *jmeno* se jménem žáka (řetězec, neměnný),
- datovou položku *prospech* s průměrem jeho známek (číslo, měnitelné),
- metodu *vypis*, která vypíše informace o žákovi (náhrada *pis_zaka*).

Školní třída bude nyní realizována polem odkazů na takové objekty. Nechci po vás, abyste psali celý program znovu, ale zkuste si napsat alespoň některý podprogram (třeba *pridej_zaka*). ■

13.3 Dědičnost

Možnost dědit datové složky a metody patří mezi nejsilnější zbraně objektově orientovaného programování. Podstatným způsobem totiž zvyšuje možnosti opětovného využití již existujícího kódu. Mnohdy existuje určitá knihovna, která se skoro přesně hodí pro vaše záměry. K odstranění onoho „skoro“ však potřebujete některé její prvky pozměnit.

Je-li dotyčná knihovna objektově orientovaná, neměli byste mít vážnější problémy. Prostě si vytvoříte svou třídu, o které prohlásíte, že je potomkem příslušné třídy z knihovny. Tento potomek zdědí všechny komponenty své mateřské třídy a může k nim přidat své vlastní nebo některé z nich změnit.

Mechanismus dědičnosti se v modulu *Moose* definuje na úrovni třídy. Slouží k tomu funkce *extends*, kterou použijete v podtřídě a uvedete v ní jméno rodičovské třídy. Název funkce naznačuje, že podtřída rozšiřuje svého rodiče. Například třída *OsobniAuto*, která by byla potomkem třídy *Auto* by začínala:

```
package OsobniAuto;
use Moose;
extends 'Auto';
```

⚡ Název rodičovské třídy musíte zadávat jako řetězec. Pokud zapomenete na uvozovky, *Moose* vás odměňuje různými chybovými zprávami a není úplně snadné dešifrovat, co je jejich příčinou.

Ve valné většině případů má *extends* jediný prvek, protože vaše třída je potomkem jedné třídy. V takovém případě se mluví o jednoduché dědičnosti. Rodičovská třída samozřejmě může mít své *extends* a být potomkem další třídy atd. atd. Můžete definovat libovolně hluboké dědičné stromy a vytvořit mezi třídami složitou hierarchii.

Pokud má funkce *extends* více argumentů, znamená to, že třída je potomkem hned několika tříd zároveň. Říká se tomu vícenásobná dědičnost a mezi objektově orientovanými programátory se vedou zuřivé spory, zda přináší nějaké praktické přednosti nebo ne. Já osobně jsem odkojen jazyky s jednoduchou dědičností a zatím jsem nenarazil na případ, kdy bych použil vícenásobnou. Nicméně ta možnost v Perlu je.

Jak se dědičnost projeví v praxi? Řekněme, že máte proměnnou *$obj*, která je ukazatelem na objekt třídy *X*, která je podtřídou třídy *Y* a ta je podtřídou třídy *Z*. Zavoláte metodu *$obj–>delej*(). Perl se nejprve podívá, zda je metoda *delej* definována pro třídu *X*, ke které objekt *$obj* náleží. Pokud ji nenajde, zkoumá, zda není něčím potomkem. Najde *Y* a proto následně zkoumá, zda třída *Y* obsahuje metodu *delej*. Není-li úspěšný ani zde, pokračuje v dědičnosti u třídy *Y* – prohledá její *extends*, najde *Z* a pokračuje v hledání tam.

Takto Perl postupuje, dokud nenarazí na třídu, ve které je definována příslušná metoda. Potom takto nalezenou metodu použije. Druhý konec nastane, pokud se metodu stále nedaří nalézt a Perl dojde ke třídě bez rodičů. V takovém případě bylo hledání neúspěšné a pokus o vyvolání metody skončí běhovou chybou[5].

⚡ Jak je vidět, hledání metod probíhá až za běhu programu. Pokud se spletete v názvu metody, Perl to zjistí až při pokusu o její provedení, nikoli hned při překladu. Toto je poměrně velké riziko. Snažte se své objektově orientované programy důkladně odladit a přesvědčit se, že program během ladění

5: Přesněji řečeno může ještě pokračovat hledáním mezi metodami vkládanými za běhu. Slouží k tomu moduly *DynaLoader*, *AutoLoader* a *SelfLoader*.

prošel opravdu všemi možnými cestičkami. Jinak hrozí, že v některé zřídka navštěvované větvi zůstane chybné jméno a uživatelé na ně podle zákona schválnosti brzy narazí.

Dobrá. Díky *extends* vaše třída zdědí všechny metody po svém rodiči. Pokud použijete některou, kterou jste ve své třídě nedefinovali, výše popsaným mechanismem se najde. Ovšem co když jste metodu pro svůj objekt definovali a přesto potřebujete stejnojmennou metodu svého rodiče.

Tato situace není nijak výjimečná. Jestliže podtřída přizpůsobuje nebo rozšiřuje svého rodiče, nabízí se, aby metoda zavolala tu rodičovskou a následně k ní něco přidala nebo pozměnila. Nejvhodnějším způsobem, jak zavolat rodičovu metodu, je použít předponu *SUPER::*. Jedná se vlastně o speciální jméno třídy, které vždy označuje třídu, která je rodičem té aktuální.

Kdybychom například ve třídě *OsobniAuto* chtěli rozšířit metodu *jed* tak, aby kromě změny stavu nádrže zobrazila ujetou vzdálenost, mohla by její definice vypadat takto:

```
sub jed {
    my $self = shift;
    my $km = shift;
    SUPER::jed( $km );    # změní stav nádrže
    print "Auto ", $self->spz, " ujelo ", $km, "km\n";
}
```

Existuje i speciální syntaxe, kterou explicitně zdůrazníte, že přepisujete rodičovskou metodu. V našem případě by zápis vypadal následovně:

```
override 'jed' => sub {
    …
}
```

Rozdíly proti standardnímu zápisu podprogramu jsou minimální (zavádí funkci *super*(), kterou lze jednoduše volat rodičovu stejnojmennou metodu) a připadá mi, že nestojí za to rozmělňovat zdrojový text další nezvyklou formou zápisu.

Příklad: Na ukázku rozvinu automobilistický příklad z předchozí části. Definuji novou třídu *Auto::Nakladni* jako potomka třídy *Auto*. Je dobrým zvykem vyjadřovat přímo v názvu třídy její zařazení do dědičné hierarchie. Odrazí se pak v adresářové struktuře souborů s objekty. Mějte však na paměti, že jméno *Auto::Nakladni* samo o sobě nemá nic společného s dědičností. Tu musíte explicitně vyjádřit pomocí *extends*. A teď už definice třídy:

Auto/Nakladni.pm

```
package Auto::Nakladni;
# definuje třídu Auto::Nakladni jako potomka třídy Auto
# přidává metody:
# naloz(kolik) přidá náklad; vrátí, zda se vešlo
# vyloz() vyprázdní náklad
# mění metody:
# stav() zobrazení aktuálního stavu

use Moose;
extends "Auto";

has nosnost => ( is => 'ro', isa => 'Num' );
has naklad => ( is => 'rw', isa => 'Num', default => 0 );

sub naloz {
    my $self = shift;
    my $kolik = shift;
    if ( $kolik + $self->naklad > $self->nosnost ) {
        print "Překročena nosnost! Naloženo ",
            $self->nosnost - $self->naklad, " t.\n";
        $self->naklad( $self->nosnost );
    } else {
        $self->naklad( $self->naklad + $kolik );
    }
    return $self->{"naklad"};
}

sub vyloz {
    my $self = shift;
    my $vylozeno = $self->naklad;
    $self->naklad( 0 );
    return $vylozeno;
}

sub stav {
    my $self = shift;
    $self->SUPER::stav();
    print "Náklad: ", $self->naklad, "\n";
}

1;
```

Přidal jsem datové položky pro nosnost nákladního vozu a jeho aktuální náklad. S nimi jsou spojeny metody *naloz* (naloží danou hmotnost, zároveň kontroluje dodržení nosnosti) a *vyloz* (vyloží celý náklad). Metodu *stav* jsem rozšířil tak, že přidá do výpisu informaci o aktuálním nákladu.

Konstruktor nadále řešit nemusíte, *Moose* jej opět vytvoří automaticky a postará se, aby zavolal konstruktor rodičovské třídy a předal mu příslušné parametry. Při jeho volání máte k dispozici pojmenované parametry pro všechny třídy v hierarchii, v našem případě *spz*, *spotreba* a *nadrz* z třídy *Auto* a *nosnost* a *naklad* z třídy *Auto::Nakladni*. A ještě předvedu jednoduchý program, který třídu používá:

nakladak.pl

```
use lib ".";
use Auto::Nakladni;

my $muj_vuz = Auto::Nakladni->new(
    spz => '2L3 4567',
    spotreba => 12.7,
    nosnost => 10.0
);
$muj_vuz->tankuj( 50 );
$muj_vuz->naloz(3.5);
if ( $muj_vuz->jed(110) ) {
    print "Sláva, jsem v Praze.\n";
} else {
    print "Budu muset dotankovat.\n";
}
$muj_vuz->stav();
$muj_vuz->naloz(8);
$muj_vuz->stav();
```

Jak je vidět, směle používám metody obou tříd (*tankuj* a *jed* patří do třídy *Auto*, ostatní metody do *Auto::Nakladni*) a vše funguje, jak má. Výstupem bude:

```
Sláva, jsem v Praze.
Auto 2L3 4567
Nádrž: 36.03
Náklad: 3.5
Překročena nosnost! Naloženo 6.5 t.
Auto 2L3 4567
Nádrž: 36.03
Náklad: 10
```

■

13.4 Ochrana a tak dál

Perl nemá mechanismy, kterými byste mohli některé metody třídy nebo datové položky prohlásit za neveřejné a znepřístupnit je jejímu okolí. Nemá je ani *Moose*. Vše je opět postaveno na principu slušného vychování:

Autor třídy zdokumentuje její veřejně poskytované služby (metody). Všechny ostatní metody jsou považovány za neveřejné. Kdo používá neveřejné metody, je 

Modul *Moose* lehce nakročil směrem k ochraně datových položek, když umožňuje přejmenování metod pro práci s nimi. Standardně se používá stejnojmenná metoda, která umožní komukoli číst a zapisovat (je-li zapisovatelná) hodnotu položky. Atributy *reader* a *writer* můžete předepsat, jak se tyto metody budou jmenovat. Přitom platí konvence, že metody s názvy začínajícími podtržítkem se zvenčí nevolají[6].

Pokud bych například chtěl, aby stav nádrže auta měnily jen metody *tankuj* a *jed* a nesahal by na ni nikdo zvenčí, dala by se položka *nadrz* ve třídě *Auto* deklarovat následovně:

```
has nadrz => (
    is => 'rw',
    isa => 'Num',
    default => 0,
    writer => _set_nadrz
);
```

Neměním *reader*, čili nadále je volně k dispozici zjištění stavu nádrže pomocí *$vuz->nadrz*. Ke změně hodnoty ale budou muset ostatní metody použít *$self->_set_nadrz(»hodnota«)* a počáteční podtržítko naznačuje, že by to neměl dělat nikdo jiný, než metody třídy *Auto* či jejích potomků.

K dispozici jsou ještě silnější nástroje, ale to už musíte sáhnout po rozšíření *Moose* nazvaném *MooseX::Privacy*. Zde už jsou k dispozici nástroje pro regulaci přístupu obvyklé v ostatních objektově orientovaných jazycích. Nádrž bych prohlásil za soukromou datovou položku atributem *traits*:

```
has nadrz => (
    is => 'rw',
    isa => 'Num',
    default => 0,
    traits => [qw/Private/]
);
```

6: Ano, stále jsme na poli konvencí, ale tentokrát ještě důrazněji říkáte „nesahat“.

Můžete mít i soukromé metody:

private_method »jméno_metody« => **sub** { …};

ale to už jsme hodně daleko ze zelenáčských prérií. Berte to jako námět na samostudium.

Mějte nicméně na paměti, že ochrana poskytovaná modulem *MooseX::Privacy* platí jen do té doby, dokud se s objekty a třídami pracuje slušně a využívají se zde popsané postupy. Jakmile si dá někdo práci a prostuduje vnitřní mechanismy modulu, dokáže je obejít. Stoprocentní ochrana v Perlu zkrátka neexistuje.

13.5 Jak je to doopravdy

Zatím jsem popsal objektově orientované programování realizované prostřednictvím modulu *Moose*. Mohlo by vás nicméně zajímat, jak to funguje uvnitř. Například proto, že byste na „syrový" přístup mohli narazit u některého existujícího programu, který potřebujete upravit.

Pro vytváření objektů je klíčový standardní podprogram **bless**, který dostane proměnnou a balík (třídu) a deklaruje proměnnou jako objekt dané třídy. V Perlu může být objektem prakticky cokoli, nicméně nejčastěji se používají odkazy na asociativní pole. Klasické vytvoření objektu vypadá nějak takto:

```
$muj_vuz = { };
bless( $muj_vuz, Auto );
$muj_vuz->new( "XYZ–12–34", 7.23 );
```

kde *new* je konstruktor definovaný v balíku *Auto*. Ve skutečnosti první dva příkazy bývají obsaženy v konstruktoru a vytvoření objektu se díky nim zkrátí na:

```
$muj_vuz = Auto->new( "XYZ–12–34", 7.23 );
```

S metodami pak pracujete stejně jako výše, navíc můžete pracovat přímo i s datovými položkami jako s každým jiným odkazem na asociativní pole:

```
$muj_vuz->{"spotreba"} = 7.15;
```

Tentokrát se nic nevytváří automaticky, konstruktor i metody pro přístup k datovým položkám si musíte napsat sami. Jméno konstruktoru je díky tomu zcela libovolné, třída jich může mít i několik. Záleží jen na tom, aby konstruktor volal **bless** a vytvořil tak objekt. Destruktor, pokud by byl potřeba, má ve standardním Perlu povinné jméno *DESTROY*.

Dědičnost definuje pole *@ISA*, do nějž vložíte jména tříd, jejichž potomkem je daná třída. Ano, Perl nabízí vícenásobnou dědičnost. Ne, neznamená to že byste ji měli používat. Třída *Auto::Nakladni*, která je potomkem třídy *Auto* by při použití interních mechanismů Perlu začínala takto:

```
package Auto::Nakladni;
use Auto;
@ISA = qw(Auto);
```

K volání rodičových funkcí slouží prefix *SUPER::*. Například konstruktor nákladního auta by typicky zakládal automobil voláním rodičova konstruktoru:

```
my auto = $self->SUPER::new( $spz, $spotreba );
```

Datové položky se nedědí. Potomek s rodičovými daty pracuje stejně jako všichni ostatní – prostřednictvím přístupových metod.

14 Aby to bylo funkční

Poslední dobou se do popředí zájmu tlačí funkcionální programování. Nejedná se o žádnou žhavou novinku, jeho kořeny sahají k jazyku LISP do 50. let 20. století. Dlouhá léta však funkcionální programátoři platili za skupinku neškodných podivínů, mumlajících si své „Hare funkce, hare, hare“ kdesi na okraji zájmu.

Jenže funkcionálně psané programy se dobře paralelizují a paralelismus je dnes v kurzu. Proto můžeme sledovat renesanci zájmu o tohle paradigma a celkem běžně se setkávat s články, které se věnují funkcionálnímu programování v tom či onom programovacím jazyce. Podívejme se, jak je na tom Perl.

14.1 Funkcionální programování

Základní myšlenkou funkcionálního programování je, že program je něco jako matematická funkce. Ta dostane v parametrech nějaké hodnoty a vrátí z nich vypočtený výsledek. Program se jí podobá – dáte mu vstupní hodnoty a dostanete podle nich určený výsledek. A podobným způsobem by měl fungovat i uvnitř.

Tady se projevuje stáří paradigmatu. V padesátých letech se na počítačích opravdu počítalo a funkcionální koncept programu odpovídal realitě. Celá řada současných programů, jako jsou třeba Skype nebo World of Tanks, se tomuto popisu vymyká. Nejde v nich o výsledek, ale zejména o interakce během jeho činnosti. Ovšem pokud budete těžit kryptoměny nebo počítat obtékání tělesa, klíčové pasáže programu mají charakter funkcí. Dívat se na program jako funkci tedy není všeobjímající koncept, nicméně v řadě případů uplatnění najde.

Tento styl programování staví program jako skupinu spolupracujících a navzájem se volajících funkcí. Situaci výrazně usnadní, pokud veškerá jejich vzájemná komunikace probíhá prostřednictvím parametrů a výsledných hodnot. Funkcionální programování se obecně snaží minimalizovat ukládání informací o stavu řešení do proměnných[1]. Za největší prohřešek je považováno, pokud má funkce vedlejší efekt. Tedy pokud kromě toho, že vydá svůj výsledek, například změní hodnotu globální proměnné nebo provede jinou akci, po níž zůstanou trvalé změny.

Problém vedlejších efektů spočívá v tom, že se na ně snadno zapomene. Programátor pak zavolá funkci, neuvědomí si, že kromě své hlavní činnosti ještě někde něco změní, a pak se nestačí divit. Proto je záhodno se vedlejším efektům vyhýbat.

1: Nejkrutopřísnější funkcionální jazyky proměnné vůbec nemají.

Úplně se jich zbavit nebývá vždy reálné. Píšete-li například aplikaci pro telefonní seznam, musíte mít globální datovou strukturu s telefonním seznamem a některé části ji musí měnit – přidávat, odebírat a měnit kontakty. Nicméně snažte se, aby takových funkcí bylo co nejméně a bylo jasně patrné, co dělají. Vůbec nejhorší je funkce, která něco mění pokoutně jako přívažek ke své hlavní činnosti. Naopak funkci, pro kterou je vedlejší efekt ve skutečnosti hlavní a jedinou náplní práce, lze považovat za přijatelnou.

Příklad: Pěkným příkladem záludné funkce je funkce *cislo_uz_bylo* z mého řešení kontroly opakování losů (program *losyb.pl* na straně 262). Název sugeruje, že funkce testuje, zda se číslo losu už vyskytlo. To skutečně ve své hlavní náplni dělá, nicméně jako vedlejší efekt zapíše číslo ověřovaného losu do pole již zpracovaných.

Při návrhu jsem se spolehl na to, že úloha je celkem jednoduchá a snadno udržím přehled. Nicméně pokud by se postupem času program zkomplikoval[2] nebo jej měl upravovat jiný programátor, mohlo by se šeredně vymstít, že podprogram nejen testuje, ale také mění.

Čistější by bylo vytvořit dvě funkce: jednu, která opravdu jen testuje, zda číslo losu už bylo zpracováno, a druhou na uložení čísla losu ke zpracovaným. Tedy něco jako:

```
sub cislo_uz_bylo {
    my ( $cislo ) = @_;
    return $zpracovano{$cislo};
}

sub zapamatuj_cislo {
    my ( $cislo ) = @_;
    $zpracovano{$cislo} = 1;
}
```

Funkce *kontola_dat* bude teď skutečně jen kontrolovat a zapamatování zpracovaného losu by bylo součástí hlavního zpracovávajícího cyklu:

```
while ( my @los = dalsi_los() ) {
    zapamatuj_cislo($los[0]);
    my $poradi = urci_poradi(@los);
    if ( $poradi <= 3 ) {
        zarad_viteze($los[0], $poradi);
    }
}
```

■

2: Což se programům často stává…

Matematické funkce jdou ještě o něco dál. Jejich výsledek závisí jen na hodnotě parametrů. Odmocnina z devíti jsou tři bez ohledu na to, jestli je pondělí nebo pátek, jaký máte průměrný plat nebo kdy jste byli naposledy u zubaře. Ideálem funkcionálního programování je, aby se stejně chovaly i funkce v programu. Říká se jim *čisté funkce* a mají dvě charakteristické vlastnosti:

- nemají žádné vedlejší efekty a
- při stejných parametrech vrátí vždy stejný výsledek.

Právě čisté funkce je radost paralelizovat. Nijak se vzájemně neovlivňují, můžete jich spustit hromadu najednou, každá se stará jen o své parametry a jejich zpracování. Obecně je žádoucí, aby funkce ve vašem programu byly co nejčistší. Usnadníte tím jejich vzájemnou spolupráci a ušetříte za Ibalgin.

Funkcionální programátoři často a rádi *skládají funkce*. Tedy připraví parametry pro jednu funkci voláním funkce jiné. Přirozeným důsledkem je, že funkcionálně psaný program inklinuje ke skupině jednoduchých funkcí, které se vzájemně kombinují k dosažení kýženého výsledku. Podobá se to stavebnici Lego nebo operačnímu systému Unix. Zároveň to programátora vede k oddělování jednotlivých činností a jejich vyjadřování samostatnými funkcemi.

Příklad: Řekněme, že mám za úkol napsat funkci, která dostane seznam libovolných čísel a má spočítat průměr všech kladných čísel z něj. Dalo by se to napsat přímočaře, nějak takto:

prumerkladnych.pl

```
sub prumer_kladnych {
    my $pocet = 0;
    my $soucet = 0;
    foreach my $cislo ( @_ ) {
        if ( $cislo > 0 ) {
            $pocet++;
            $soucet += $cislo;
        }
    }
    if ( $pocet > 0 ) {
        return $soucet / $pocet;
    } else {
        return undef;
    }
}
```

Výsledkem je zcela jednoúčelová funkce, která sice funguje, ale nedává možnost žádného jiného využití. Funkcionální programátor by v zadání okamžitě rozpoznal dva podúkoly, které spolu nemají mnoho společného a potenciálně by mohly být k užitku i samostatně. Jedním je výpo-

čet průměru ze seznamu čísel, druhým výběr kladných čísel. Aby umožnil jejich využití v dalších částech programu, napsal by pro ně dvě samostatné funkce a cílovou pak vytvořil jejich složením:

prumerkladnych2.pl

```
sub prumer {
    my $pocet = 0;
    my $soucet = 0;
    foreach my $cislo ( @_ ) {
        $pocet++;
        $soucet += $cislo;
    }
    if ( $pocet > 0 ) {
        return $soucet / $pocet;
    } else {
        return undef;
    }
}

sub vyber_kladna {
    my @vysledek = ();
    foreach my $cislo ( @_ ) {
        if ( $cislo > 0 ) {
            push( @vysledek, $cislo );
        }
    }
    return @vysledek;
}

sub prumer_kladnych {
    prumer( vyber_kladna( @_ ) );
}
```

Ve funkci *prumer_kladnych* na seznam jejích parametrů nejprve vypustím funkci *vyber_kladna*, která z něj vybere kladná čísla a jejich seznam vydá jako svůj výsledek. Ten se následně předá jako parametr funkci *prumer*, aby vypočetla průměr všech kladných čísel z původního seznamu. ■

Jak vidíte, základ funkcionálního programování tvoří především metodika. Nevyžaduje žádné speciální konstrukce programovacího jazyka, řeší především uspořádání programu a způsob jeho tvorby. Pojďme se ale podívat na několik pokročilejších konstrukcí, které už od jazyka něco potřebovat budou.

14.2 Funkce jako parametr

Zkusím trochu rozvinout výběr kladných čísel z posledního příkladu. Vytvořil jsem pro něj funkci *vyber_kladna*, která si založí pole pro výsledek, projde všechny hodnoty z parametrů a kdykoli je hodnota kladná, přidá ji do pole. Po zpracování všech parametrů vydá pole jako výsledek:

```
1   sub vyber_kladna {
2       my @vysledek = ();
3       foreach my $cislo ( @_ ) {
4           if ( $cislo > 0 ) {
5               push( @vysledek, $cislo );
6           }
7       }
8       return @vysledek;
9   }
```

Nicméně požadavek „vybrat ze skupiny hodnot ty, které splňují určitou podmínku" je dost obecný a nepochybně se bude často opakovat. Může se stát, že budu vybírat čísla v rozsahu od 0 do 255, čísla dělitelná sedmi, nebo třeba neprázdné řetězce ze seznamu řetězců.

Je zřejmé, že kód všech takových funkcí bude prakticky stejný a lišit se bude jen v podmínce na řádku 4, která rozhoduje, zda hodnotu přidat do výsledku nebo ne. Bylo by dobré toto uchopit nějak koncepčně, protože:

1. Opakování je nuda.
2. Když se kopíruje a upravuje kód, snadno se může přihodit, že při úpravách uděláte drobnou chybu. Ta způsobí, že upravená verze se bude chovat jinak. Chyby tohoto typu se velmi špatně hledají, protože programátor přece *ví*, že to je stejná funkce, takže určitě dělá totéž a chyba musí být někde jinde…
3. Pokud přijdete na geniální myšlenku, jak podprogram vylepšit, budete muset upravovat všechny jeho varianty.

Naštěstí se to koncepčně uchopit dá, protože Perl umožňuje, abyste podprogramům v parametrech předávali nejen hodná pasivní data, ale také jiné podprogramy, které ovlivní jejich činnost. Přesněji řečeno, umožňuje předat odkaz na podprogram. V anglickojazyčné literatuře se pro funkcionální parametry často používá termín *zpětné volání (callback)*, protože podprogram při své činnosti volá zpět části kódu, které dostal. Můžete se též setkat s pojmem *háček (hook)*, protože umožňuje „navěsit" kód podle potřeby.

V našem konkrétním případě bych funkci přidal jeden parametr navíc, jímž by bylo kritérium pro výběr položek do výsledku. Bude se jednat o funkci s jedním parametrem, kterým je posuzovaná

hodnota, a výsledkem pravdivostního typu. Kritérium bude prvním parametrem funkce, za ním následují vlastní hodnoty. Funkce by vypadala následovně:

filtr.pl

```
1   sub filtr {
2       my $kriterium = shift;
3       my @vysledek = ();
4       foreach my $hodnota ( @_ ) {
5           if ( $kriterium->($hodnota) ) {
6               push( @vysledek, $hodnota );
7           }
8       }
9       return @vysledek;
10  }
```

Na druhém řádku si první parametr přesunu do lokální proměnné *$kriterium*, která bude obsahovat odkaz na funkci rozhodující o zařazení hodnoty do výsledného pole. Používám **shift**, abych funkci odstranil z pole parametrů, kde mi zbudou jen zpracovávané hodnoty a mohu je pak v klidu projít cyklem **foreach**.

Volání funkce z parametru vidíte na řádku 5. Jedná se o odkaz, čili používám „šipkový" zápis, jaký jste viděli při práci s odkazy na pole v předchozích kapitolách. Za šipkou následuje obvyklý tvar parametrů v kulatých závorkách. Alternativně by se dal použít i zápis ve tvaru:

```
&{$kriterium}($hodnota)
```

Ten je ovšem méně elegantní. A jak se předává funkce při volání filtru? Historie se opakuje, pro vytvoření odkazu slouží zpětné lomítko před názvem příslušné funkce. Tentokrát ale název musí zahrnovat i úvodní znak &. Pro výběr kladných čísel si vytvořím funkci *kladne*, která porovná svůj parametr s nulou, a předám odkaz na ni jako první parametr filtru:

```
sub kladne {
    return @_[0] > 0;
}

my @x = ( 10, -5, 15, 8, -3, -22 );
my @kladne_x = filtr( \&kladne, @x );
```

A výběr neprázdných řetězců? Žádný problém, pomocí regulárních výrazů vytvořím kritérium, jemuž vyhoví jen řetězec obsahující alespoň jeden viditelný znak, a vypustím s ním filtr na seznam řetězců:

```
sub neprazdny_ret {
    return @_[0] !~ /^\s*$/;
}

my @y = ("", "raz", " ", "dva", "", " ", "tri");
my @neprazdne_y = filtr( \&neprazdny_ret, @y );
```

Tady se velmi hodí benevolentní přístup Perlu k datovým typům. Stejnou filtrovací funkci mohu použít pro hodnoty nejrůznějších typů, jen je třeba, aby je kritérium dokázalo posoudit. Kdyby například kritérium zkoumalo dělitelnost sedmi a dostalo by k posouzení hodnotu „ahoj", výsledek by nedával valný smysl.

Pro funkce, které jako svůj parametr dostávají další funkce, se používá pojem *funkce vyššího řádu*. Zatímco *vyber_kladna*, kterou jsem tuto část začínal, je jen obyčejnou funkcí, *filtr*, ke kterému jsem se dopracoval, je funkcí vyššího řádu.

Tyto funkce umožňují popsat obecné principy a oddělit je od detailů specifických pro konkrétní případ. *filtr* definuje obecné filtrování skupiny hodnot. Konkrétní kritérium, podle nějž se bude v určité specifické situaci filtrovat, se nachází mimo něj a bude mu předáno jako parametr.

Tento přístup se hodí zejména pro knihovní funkce. Ty mají být použitelné v různých případech, proto by měl mít jejich uživatel možnost přizpůsobit si jejich chování svým potřebám. Funkcionální parametry se k tomu skvěle hodí.

Cvičení 14.1: Vedle filtrování je další typickou úlohou ověření, zda všechny hodnoty v seznamu mají určitou vlastnost. Vytvořte funkci vyššího řádu *vsechny*, která dostane kritérium (funkci s jedním parametrem vracející pravdivostní hodnotu) a seznam hodnot. Výsledkem funkce bude „pravda", pokud všechny hodnoty v seznamu splňují kritérium. V opačném případě bude výsledkem „nepravda". Například když jako kritérium použijete výše uvedenou funkci *kladne*, ověří se, zda jsou všechna čísla v seznamu kladná. ■

Cvičení 14.2: Numerický výpočet derivace funkce jedné proměnné f v bodě x se provádí tak, že se zvolí nějaké velmi malé číslo e a spočítá se:

$$\frac{f(x+e) - f(x-e)}{2e}$$

Vytvořte funkci *derivace*, která dostane funkci f a hodnotu x a spočítá podle výše uvedeného vzorce derivaci f v bodě x. Za e zvolte např. 0.001. ■

14.3 Funkce jako hodnota

Při volání funkcí vyššího řádu není nijak neobvyklé, že podprogramy předávané jako jejich parametry jsou docela jednoduché. Třeba nedávno použité porovnání s nulou nebo s regulárním výrazem není žádný výkvět kreativity. Standardní postup, kdy se vytvoří plnohodnotná, ovšem jednoduchoučká funkce (třeba *kladne*) a následně se předává odkaz na ni vypadá jako výtvor Antonia Salieriho. Hodilo by se něco elegantnějšího, abychom tu jednoduchou funkci napsali rovnou do parametru.

To něco skutečně existuje a jmenuje se *anonymní funkce*. Syntakticky se velmi podobá definici podprogramu, chybí jen jeho jméno:

```
sub {
    »tělo«
}
```

Formálně se jedná o výraz, který můžete zapsat kamkoli, kde má být odkaz na funkci. Jeho vyhodnocením vznikne kód implementující *»tělo«* funkce a jako výsledek vydá odkaz na něj.

V příkladu filtrování kladných čísel ze strany 218 bych díky tomu nemusel definovat funkci *kladne*, ale mohl bych *filtr* stručněji zavolat s anonymní funkcí, která zajistí porovnání s nulou:

```
my @kladne_x = filtr( sub { @_[0] > 0 }, @x );
```

Připomínám, že příkaz **return** není povinný a v takto jednoduchých případech se běžně vynechává. Výsledkem funkce se pak stane výsledek jejího posledního (a často jediného) příkazu.

Jestliže lze odkazem na pojmenovanou funkci nebo použitím anonymní funkce vytvořit „funkční hodnotu“ a přiřadit ji do parametru, můžete ji samozřejmě přiřadit i do běžné proměnné. Používat se pak bude stejným způsobem jako funkční parametr – pomocí šipkové konvence:

```
my $kladne = sub {
    return @_[0] > 0;
};

print $kladne->(10), "\n";
print $kladne->(-5), "\n";
```

Nezapomeňte na závěrečný středník. Jedná se o deklaraci proměnné a ta musí končit středníkem, jinak k sobě přibere i kus následujícího kódu a budou se dít věci. Pochopitelně takovou proměnnou můžete také předat jako funkční parametr, třeba zmíněnému filtru:

```
my @kladna_cisla = filtr( $kladne, @cisla );
```

Filtr očekává odkaz na funkci a dostal odkaz na funkci, uložený v proměnné. A když už zacházíme s funkcemi jako s hodnotami, nemohly by být třeba i výsledkem funkce? „Ale jistě, jak si přejete." ozvalo se ze sluchátka:

fce-vysledek.pl

```
1  sub vytvor_pozdrav {
2      return sub {
3          my $jmeno = shift;
4          print "Nazdar $jmeno!\n";
5      }
6  }
7
8  my $pozdrav = vytvor_pozdrav();
9  $pozdrav->("lidi");
```

Funkce *vytvor_pozdrav* jako svůj výsledek vrátí odkaz na anonymní funkci s jedním parametrem, která vypíše pozdrav obsahující hodnotu tohoto parametru. Na řádku 8 jsem ji zavolal a odkaz na funkci, kterou jsem dostal, jsem uložil do proměnné *$pozdrav*. Díky tomu mohu funkci na následujícím řádku zavolat a do výstupu se vypíše „Nazdar lidi!"

Možná vám touto dobou vrtá hlavou, k čemu je taková prapodivná akrobacie dobrá. Zatím to vypadá, že se drbu levou rukou za pravým uchem a abych si to ještě ztížil, nasadil jsem si baseballovou rukavici.

Vracet ve výsledku stále stejnou funkci nedává valný smysl, to ji můžete definovat rovnou a nekomplikovat si život. Věc začne být zajímavá, když výslednou funkci budete upravovat podle předaných parametrů. V takovém případě mluvíme o *generátoru funkcí*. Ten dostane nějaké parametry a jako výsledek vrátí funkci, jejíž tělo přizpůsobil podle jejich hodnot. Tím vznikne funkce ušitá na míru konkrétní situaci.

Zapněte si bezpečnostní pásy, tohle je docela turbulentní.

Příklad: Vraťme se k filtrování hodnot. Řekněme, že v programu budete často potřebovat procházet seznamy čísel a vybírat z nich hodnoty ležící v různých rozmezích. Jistě by se dal napsat specializovaný filtr, který by kromě seznamu filtrovaných čísel dostal i nejmenší a největší hodnotu. Dal by se také využít náš generický filtr, kterému by se předalo příslušné kritérium.

Ale zkusme to ještě jinak – budu generovat filtry tohoto typu pro pásma podle potřeby. Generátor dostane mezní hodnoty a vytvoří filtrovací funkci, která při svém volání dostane seznam čísel a vrátí seznam všech hodnot z něj, které leží v uvedeném rozmezí:

```
1   sub generuj_filtr {                                          generatorfiltru.pl
2       my( $minimum, $maximum ) = @_;
3       return sub {
4           my @vysledek = ();
5           foreach my $cislo ( @_ ) {
6               if ( $cislo >= $minimum and $cislo <= $maximum ) {
7                   push(@vysledek, $cislo);
8               }
9           }
10          return @vysledek;
11      }
12  }
```

Je důležité si uvědomit, že v kódu generátoru se pracuje s parametry dvou různých funkcí. Jednak to jsou parametry generátoru *generuj_filtr*, zde *$minimum* a *$maximum*. Jejich hodnoty zadáte při volání generátoru. Používají se v těle generované funkce a díky tomu ovlivní její činnost. Jakmile jednou funkci vytvoříte, hodnoty *$minimum* a *$maximum* jsou v ní pevně dány a už se nezmění.

Na nich zcela nezávislé jsou parametry generované funkce. Jim jsem nepřidělil jméno, pracuje se s nimi na řádku 5 v podobě pole @_. Tyto hodnoty předáte vygenerované funkci, až ji budete volat. Ona z nich vybere ty, které leží v rozmezí od *$minimum* po *$maximum* a jejich seznam vrátí jako svůj výsledek.

Mám generátor a nebojím se ho použít. Řekněme, že mne často budou zajímat jednobajtové hodnoty. Vytvořím si pro ně dva specializované filtry: *$bez_znamenka* bude vybírat jednobajtové hodnoty bez znaménka, tedy od 0 do 255, *$se_znamenkem* bude vybírat čísla od –128 do 127. Jejich volání vidíte na řádcích 18 a 20:

```
14  my $bez_znamenka = generuj_filtr(0, 255);
15  my $se_znamenkem = generuj_filtr(–128, 127);
16  my @hodnoty = ( 100, –50, 80, 320, –400, 99, 220 );
17  print "Jednobajtové bez znaménka:\n";
18  print join( "\n", $bez_znamenka–>(@hodnoty) ), "\n\n";
19  print "Jednobajtové se znaménkem:\n";
20  print join( "\n", $se_znamenkem–>(@hodnoty) ), "\n\n";
```

■

Generátor filtrů zároveň ilustruje jeden podstatný princip: Dokud existuje odkaz na funkci, Perl zachová existenci proměnných, s nimiž funkce pracuje, i když jejich vlastní blok již zanikl. Říká se tomu *uzávěr (closure)* a v našem příkladu se týká proměnných *$minimum* a *$maximum*.

Jedná se o lokální proměnné (ve skutečnosti parametry) funkce *generuj_filtr*. Funkce je volána na řádku 14, parametrům se přidělí hodnoty, provede se tělo a funkce skončí. Tím zanikne blok, ve kterém byly proměnné *$minimum* a *$maximum* deklarovány jako lokální, takže by normálně byly odstraněny. Výsledkem *generuj_filtr* je ovšem odkaz na vytvořený filtr, který se uloží do proměnné *$bez_znamenka*. Tento filtr s proměnnými pracuje, proto budou zachovány v jeho uzávěru. Dokud je funkce dostupná (nezanikne proměnná *$bez_znamenka* nebo se nezmění její hodnota), Perl udržuje v paměti její uzávěr.

Hned na dalším řádku je funkce *generuj_filtr* volána znovu s jinými hodnotami, vytvoří další filtr a uloží odkaz na něj tentokrát do proměnné *$se_znamenkem*. Tato funkce má svůj uzávěr a v něm své hodnoty *$minimum* a *$maximum*. Uzávěr opět bude existovat tak dlouho, dokud je daná funkce dosažitelná.

Pro zobecňování základních postupů a jejich oddělování od specialit konkrétních případů jsou tedy k dispozici dva nástroje. Buď lze konkrétní části kódu doplňovat pomocí funkcionálních parametrů, nebo lze specifické verze funkcí vytvářet pomocí obecných generátorů. Který je lepší? To nelze obecně říci, každý máme jiné preference. Silné argumenty ve prospěch jedné z těchto variant neexistují.

Obvyklejší a používanější jsou určitě funkcionální parametry, které jsou přece jen bližší způsobu uvažování většiny programátorů. Svou roli jistě hraje i to, že podobné nástroje jsou k dispozici i v řadě konvenčních programovacích jazyků. Naproti tomu funkce jako výsledek je přece jen koncept méně vídaný.

Jak jste viděli, funkce[3] je v Perlu hodnota jako každá jiná. Můžete ji předávat v parametrech, vracet jako výsledek, přiřazovat do proměnných… Pokud programovací jazyk toto umožňuje, říká se, že má *funkce první třídy*. Má-li být jazyk považován za vhodný pro „opravdové“ funkcionální programování, měl by tuto vlastnost mít.

Cvičení 14.3: Generátor samozřejmě nemusí dostávat v parametrech jen data jako v příkladu s filtrováním čísel, ale i funkce, které bude jím vytvořená funkce volat.

Ve cvičení 14.2 na straně 219 jsem po vás chtěl numerický výpočet derivace v konkrétním bodě. Zkuste vytvořit generátor derivací *derivuj*, který dostane funkci jedné proměnné a jako výsledek vydá funkci, která numericky počítá derivaci této funkce v bodě, který dostane jako parametr. Tedy aby po provedení příkazů:

```
sub kvadrat { my $x = shift; return $x*$x; }
my $kvadrat_der = derivuj( \&kvadrat );
```

3: Přesněji řečeno odkaz na ni.

proměnná *$kvadrat_der* obsahovala odkaz na derivaci kvadratické funkce a mohl jsem si vypočíst její hodnotu v libovolném bodě:

```
print $kvadrat_der->(3), "\n";
```

■

14.4 Rekurze

Když je řeč o funkcionálním programování, dříve či později odněkud vykoukne rekurze. Ne, že by s ním byla nerozlučně spojena. Rekurze je obecný princip a rekurzivní podprogramy lze psát skoro v každém jazyce. Nicméně ti dva jsou velcí kamarádi už jen proto, že v některých funkcionálních jazycích chybí cykly a rekurze se v nich používá, kdykoli je třeba zpracovat seznam hodnot.

V Perlu tuhle potřebu nemáme, nicméně zařadit rekurzi mezi dostupné přístupy k řešení problémů považuji za velmi prospěšné. Základní princip rekurze spočívá v tom, že funkce řešený problém trochu zjednoduší, zavolá sama sebe k vyřešení této jednodušší verze a výsledek pak využije k vytvoření vlastního řešení. Dva důležité principy, které je třeba dodržovat při vytváření rekurzivních funkcí, jsou:

1. Pokud je problém dostatečně jednoduchý, vyřešit jej rovnou.
2. Jinak zavolat sama sebe na zjednodušený problém a využít výsledek pro celkové řešení.

Zjednodušení ve druhém bodě je klíčové. Jen když při každém dalším volání problém zjednodušíte, po nějakém počtu kroků se dostanete do situace, kdy je natolik jednoduchý, že jej lze vyřešit podle bodu 1.

Notorickým příkladem rekurze je výpočet faktoriálu. Budu se jej držet i já, protože se na něm dobře ilustrují klíčové vlastnosti. Faktoriál lze už z matematického pohledu definovat dvěma způsoby. Lze to udělat iterativně a prohlásit, že *n*! je součin všech čísel od 1 do *n*.

Tomu odpovídá i programové řešení pomocí cyklu. Vytvořím si proměnnou, ve které budu konstruovat výsledek, v cyklu projdu hodnoty až po *n* a každou z nich proměnnou vynásobím:

faktorial1.pl

```
sub faktorial {
    my( $n ) = @_;
    my $vysledek = 1;
    foreach my $i (2..$n) {
        $vysledek *= $i;
    }
    return $vysledek;
}
```

Ovšem součin čísel od 1 do $n-1$ není nic jiného než faktoriál čísla $n-1$. Mohu tedy už na úrovni matematiky faktoriál definovat rekurzivně a prohlásit, že $1! = 1$ a pro $n > 1$ je $n! = n \cdot (n-1)!$. Přímočarým převodem této definice do Perlu vznikne rekurzivní funkce:

faktorial2.pl

```
sub faktorial {
    my( $n ) = @_;
    if ( $n <= 1 ) {
        return 1;
    } else {
        return $n * faktorial($n-1);
    }
}
```

Když se vrátím k obecným principům tvorby rekurzivních funkcí, složitost problému je zde reprezentována velikostí čísla *n*. Je-li *n* dostatečně malé (konkrétně 1), vrátí funkce výsledek rovnou. Jinak zavolá sama sebe na jednodušší verzi problému – výpočet faktoriálu čísla o jedničku menšího – a až dostane výsledek, vynásobí jej *n* a má hotovo.

Nabízí se porovnání obou verzí. Ta rekurzivní je elegantní a velmi přesně odpovídá matematické definici. Naproti tomu řešení cyklem je poněkud upocené. Musel jsem vytvořit dvě pomocné proměnné a od „vynásobit čísla od 1 do *n*“ k výslednému kódu vede delší cesta.

Na druhé straně ovšem iterativní verze bude rychlejší. Rekurzivní řešení problému zahrnuje velký počet volání podprogramů (zde *n*krát), což je časově náročná operace. Musí se vždy odložit aktuální stav na zásobník, vytvořit nové parametry a lokální proměnné a po skončení zase vše uklidit a vrátit se k původnímu podprogramu.

Tyto vlastnosti jsou docela obecné. Pro řadu problémů existuje iterativní i rekurzivní řešení, přičemž rekurzivní obvykle bývá elegantnější, zatímco iterativní rychlejší. Pro ilustraci naznačím rekurzivní řešení některých problémů, které mívají programátoři ve zvyku řešit iterativně:

- Součet seznamu hodnot: Je-li seznam prázdný, je součet 0. Jinak jej vypočtu jako první hodnotu plus součet všech zbývajících.
- Hledání hodnoty v seznamu: Je-li seznam prázdný, je výsledkem neúspěch. Je-li v něm hledaná hodnota na začátku, je výsledkem úspěch. Jinak zahodím první prvek a pokračuji v hledání ve zbytku seznamu.
- Spojení dvou vzestupně seřazených seznamů do jednoho, který bude opět vzestupně seřazen: Je-li některý ze seznamů prázdný, výsledkem je druhý seznam. Jinak přesunu menší z obou počátečních hodnot na začátek výsledku a připojím za ni výsledek spojení zbývajících hodnot.

Jak vidíte, konstrukce rekurzivního řešení znamená aplikaci dvou výše zmíněných principů na konkrétní problém. Programátor hledá odpovědi na dvě otázky: Kdy je daný problém tak jednoduchý, že jej lze vyřešit rovnou? Jakým způsobem jej zjednodušit a jak využít řešení této zjednodušené verze?

Některé problémy jsou ovšem rekurzivní ze své podstaty. Typickým příkladem je třeba průchod stromem, kde samotná datová struktura má rekurzivní charakter – strom je buď prázdný, nebo je tvořen kořenem, jehož potomky jsou podstromy. Proto jste první příklad rekurzivního podprogramu viděli už při průchodu adresářovou strukturou na straně 145, kde podprogram *zpracuj_adr* volá sám sebe na své podadresáře. I taková věc se sice dá řešit iterativně, ale dá to spoustu práce a tentokrát to ani nebude rychlejší.

Jak velký či malý je rozdíl ve výkonu iterativní a rekurzivní verze záleží na konkrétním problému a konstrukci příslušných podprogramů. Obecně lze říci, že volání podprogramu je sice náročnou operací, ale pokud se rekurze neutrhne ze řetězu a hloubka vnoření je lineárně úměrná velikosti řešeného problému, nebývá rozdíl příliš patrný.

Existují ovšem i případy, kdy se rekurze ze řetězu utrhne. Jejich typickým představitelem je Fibonacciho posloupnost, jejíž první dva členy jsou 0 a 1 a každý další se spočítá jako součet dvou předchozích. Pokud výpočet implementujete přímočarým přepisem do podoby rekurzivní funkce, dopadne to katastrofálně:

fibo1.pl

```
1   sub fibo {
2       my( $n ) = @_;
3       if ( $n < 2 ) {
4           return $n;
5       } else {
6           return fibo($n-2) + fibo($n-1);
7       }
8   }
```

Jádro problému se skrývá na řádku 6, kde se funkce *fibo* volá dvakrát, aby vypočetla dvojici předchozích členů posloupnosti. Výpočet *fibo*(*$n*–1) ovšem zopakuje celý výpočet *fibo*(*$n*–2) a ještě k němu přidá *fibo*(*$n*–3). Toto se děje pro každý člen posloupnosti, což vede k masivnímu opakování stále stejných výpočtů.

Zvětšíte-li *$n* o jedničku, složitost výpočtu se bezmála zdvojnásobí. Důsledkem je, že celkový počet prováděných operací rychle naroste do nepřijatelných mezí. Na mém počítači výpočet *fibo*(35) trvá něco přes 6 sekund, *fibo*(36) přes 10 s, na výsledek *fibo*(50) už nejsem ochoten čekat dvě hodiny a *fibo*(100) by trvalo nějaké 4 miliony let. Přitom iterativní řešení hodnotu spočítá hned.

Pokud je nízký výkon způsoben masivním opakováním stále stejných výpočtů, dá se situace řešit pomocí *zapamatování* (anglicky *memoization*). Jednou spočítané výsledky si budu ukládat do vyrovnávací paměti (cache) tvořené nejčastěji asociativním polem a pokud dojde k volání funkce pro stejné parametry, jednoduše vyzvednu výsledek z ní. Upravená funkce by mohla vypadat nějak takhle:

fibo2.pl

```
1   my %fibocache;
2   sub fibo {
3       my( $n ) = @_;
4       if ( exists($fibocache{$n}) ) {
5           return $fibocache{$n};
6       } elsif ( $n < 2 ) {
7           return $fibocache{$n} = $n;
8       } else {
9           return $fibocache{$n} = fibo($n-2) + fibo($n-1);
10      }
11  }
```

Tělo se změnilo jen velmi málo. Začíná teď testem, zda nemám uložené řešení (řádek 4). Pokud ne, hodnotu spočítám a ještě před odevzdáním volajícímu ji uložím do vyrovnávací paměti pro příští volání (řádky 7 a 9). Takto upravená funkce počítá *fibo*(1000) čtyři tisíciny sekundy…

Ještě rychleji by to šlo cyklem, nicméně mechanismus zapamatování je obecný a lze jej použít na řadu funkcí. Pokud má funkce více parametrů, je třeba je spojit (obvykle se používá **join** s vhodným oddělovačem, ve složitějších případech lze sáhnout po modulu *Storable*, o kterém jsem psal na straně 184) a vytvořit tak klíč do asociativního pole. Aby ale mělo zapamatování smysl, musí být splněno několik podmínek:

- Funkce musí být čistá. Zapamatování výsledků funkce *time* nebo funkce závislé na aktuální hodnotě globální proměnné by vedlo k chybným výsledkům.
- Funkce musí být výpočetně náročná. Práce s vyrovnávací pamětí má svou režii (sestavujete klíč, prohledáváte a měníte asociativní pole) a pokud se výsledek spočítá rychlostí blesku, nemůže se vyplatit. Například převod mezi stupni Celsia a Fahrenheita, který zahrnuje jedno sčítání/odečítání a jedno dělení, zapamatováním zpomalíte.
- Musí docházet k opakovaným výpočtům se stejnými parametry. Jen pokud si často ušetříte náročný výpočet a získáte řešení rovnou z vyrovnávací paměti, zrychlíte.

Platí se samozřejmě pamětí, ukládání výsledků k daným parametrům něco spotřebuje. Ale při současných velikostech pamětí se to zpravidla bohatě vyplatí. Pro ty líné z nás je navíc k dispozici modul *Memoize*. Opatřit funkci pamětí pro výsledky dá ve většině případů přesně tolikhle práce:

```
use Memoize;
memoize("fibo");
```

Doplněním těchto dvou řádků do programu s první verzí funkce *fibo* ji zrychlím na úroveň verze druhé. Nic jiného měnit netřeba.

15 Perl a databáze

K práci s daty neodmyslitelně patří potřeba jejich trvalého ukládání. Nejjednodušší a nejuniverzálnější formou uložení dat je textový soubor. Na této prosté skutečnosti, regulárních výrazech a vzájemném propojování vstupů a výstupů je postaven základ úspěchu operačního systému Unix.

Nelze však přehlížet, že textové soubory mají vážná omezení. Jsou zcela nevhodné pro náhodné přístupy, kdy potřebujete sáhnout tu sem a vypustit z evidence dvě židle, za chvilku zase támhle a změnit zůstatkovou cenu popelnice. Zkrátka textový soubor je velmi vhodný pro výměnu informací mezi programy, ale jako pracovní médium selhává. Nemluvě o tom, že při velkých objemech informací končí s dechem.

V takových situacích nastupují databáze. Jejich způsob uložení dat a algoritmy, které jej obalují, jsou navrženy právě pro realizaci výše popsaných cílů: uložení velkých objemů dat, rychlé vyhledávání v nich, možnost aktualizovat jednotlivé záznamy v libovolném pořadí.

15.1 Co je k dispozici

Perl vám nabízí v tomto oboru dvě alternativy: spolupráci s DBM databázemi a s SQL databázemi.

Pod názvem DBM se skrývají jednoduché databázové nástroje, které jsou dostupné ve valné většině Unixů. Díky volně šiřitelné implementaci Berkeley DB si je můžete doplnit do těch zbývajících. Implementace Berkeley DB je rychlá a velmi kvalitní, takže se těší nemalé popularitě.

Základní výhodou použití DBM je, že si vystačíte s pouhým Perlem. Nemusíte se učit zacházení s databázovým strojem, zkoumat jeho mechanismus přístupových práv a dělat další ne zrovna příjemné věci.

Nevýhoda DBM spočívá v tom, že tyto databáze jsou postaveny na jednoduchém principu klíč–hodnota. Z toho plynou dvě omezení: vyhledávat lze jen podle jednoho klíče a můžete mu přiřadit jen jednoduchou hodnotu. Tou naštěstí může být řetězec znaků a existují metody, jak do něj zabalit i poměrně složité struktury. Díky tomu druhá nevýhoda tolik nebolí. A – ruku na srdce – vícenásobné klíče jsou také dost vzácné.

Na poli seriózních databází se celkem nekompromisně prosadily relační databáze používající dotazovací jazyk SQL. Jsou postaveny na principu klient–server. Perl zde bude hrát klientskou úlohu, ale potřebujete k němu databázový server.

Musíte si tudíž nainstalovat některou SQL databázi. Nabídka je velmi široká. Začíná volně šiřitelnými produkty[1] a končí vlajkovými loděmi zvučných jmen, jejichž ceny dosahují řádu milionů.

Řešení na bázi SQL zpravidla bývá robustnější a nabízí košatější možnosti. Ty pochopitelně závisí na schopnostech konkrétního databázového serveru, nicméně existuje společný velmi solidní základ, který máte k dispozici vždy. Díky tomu jsou řešení založená na SQL vysoce přenositelná. Platíte nutností mít databázový server, naučit se s ním pracovat a musíte také umět dotazovací jazyk SQL.

Z uvedeného popisu je myslím patrné, že pro menší úlohy si vystačíte s DBM, zatímco pro složitější či přístup k již existujícím agendám je zpravidla výhodnější použít SQL.

15.2 Spolupráce s DBM

Pro DBM je k dispozici geniálně jednoduchý model: databázi napojíte na asociativní pole. Dále s ní pak zacházíte jako s obyčejným asociativním polem a všechny vaše operace se ve skutečnosti provádějí s databázovým souborem.

K použití tohoto triku potřebujete standardní modul *DB_File* (nebo příbuzný – viz tabulka 15.1). O vytvoření vazby mezi asociativním polem a databází se pak postará funkce:

tie(*»asoc_pole«*, ”DB_File”, *»jméno_souboru«*);

Vydá pravdivostní hodnotu, zda se otevření databáze podařilo (podobně jako **open**), takže typicky následuje **or die** …

Od tohoto okamžiku se *»asociativní pole«* dá používat jako kterékoli jiné. Když do něj něco zapíšete, uloží se příslušná dvojice klíč+hodnota do databázového souboru. Jestliže naopak hledáte určitý klíč, hledá se v databázi. Fungují dokonce i takové funkce, jako **keys** či **each**.

O uzavření databázového souboru a uložení všech provedených změn se postará:

untie(*»asoc_pole«*);

Příklad: Jako příklad této techniky použiji jednoduchý telefonní seznam. Informace ukládá do databázového souboru *telefony.db*. Program pracuje interaktivně. Zadáte-li mu řetězec znaků, hledá záznam s tímto klíčem. Obecnější hledání vyvoláte pomocí *»vzor«. Jako odpověď obdržíte

1: Nepodceňovat! Zpracování kompletní statistiky toků dat v síti CESNET2, což představuje řádově desítky milionů údajů za hodinu, je postaveno na bázi Perlu a volně šiřitelné databáze MySQL.

všechny záznamy, jejichž klíč vyhovuje danému *»vzoru«* (samotná hvězdička pak vypíše všechny). Záznam přidáte pomocí *+»jméno«:»číslo«* a odeberete *-»jméno«*. Prázdným vstupem pak ukončíte činnost programu.

telefony.pl

```
use locale;
use DB_File;

tie( my %seznam, "DB_File", "telefony.db" );
while ( my $vstup = zadej_vstup() ) {
    if ( $vstup =~ s/^\+// ) {
    # přidám záznam
        my ($jmeno,$cislo) = split( /:/, $vstup );
        $seznam{$jmeno} = $cislo;
    } elsif ( $vstup =~ s/^–// ) {
    # odstraním záznam
        delete( $seznam{$vstup} )
    } elsif ( $vstup =~ s/^\*// ) {
    # hledání vzoru
        foreach my $jmeno ( sort keys %seznam ) {
            if ( $jmeno =~ /$vstup/ ) {
                pis_zaznam( $jmeno, $seznam{$jmeno} );
            }
        }
    } else {
    # hledání přesného jména
        if ( exists($seznam{$vstup}) ) {
            pis_zaznam( $vstup, $seznam{$vstup} );
        } else {
            print "Není v databázi\n";
        }
    }
}
untie( %seznam );

sub zadej_vstup {
    print "dotaz> ";
    my $vstup = <STDIN>;
    chomp($vstup);
    return $vstup;
}
```

```
sub pis_zaznam {
    my ( $jmeno, $cislo ) = @_;
    printf "%-30s %8s\n", $jmeno, $cislo;
}
```

Telefonní databázi ztělesňuje asociativní pole *%seznam*. Jak vidíte, až na volání funkcí **tie** a **untie** tu není vůbec nic neobvyklého. Jak je ten svět jednoduchý! ■

Do standardní výbavy Perlu patří hned několik modulů pro různé implementace DBM. V dokumentaci systému si najděte tu pravou a pokud má Berkeley DB nebo si ji můžete nainstalovat, dejte jí přednost. Přehled modulů uvádí tabulka 15.1. Funkci **tie** musíte jako druhý parametr dát vždy řetězec shodný se jménem modulu.

modul	implementace
DB_File	Berkeley DB
GDBM_File	GDBM (GNU implementace)
NDBM_File	NDBM (často v BSD)
SDBM_File	SDBM (součást X11)

Tabulka 15.1: Moduly pro různé verze DBM

Ovšem pozor! Moduly **DBM_File* vyžadují kromě jména povinně i režim zamykání databázového souboru a jeho přístupová práva. Takže v jejich případě bude otevírání poněkud složitější:

```
use SDBM_File;
use Fcntl;

tie(%db, 'SDBM_File', "test.db", O_CREAT|O_RDWR, 0644)
  or die "Nelze otevřít databázi!\n";
```

tie je ve skutečnosti ještě obecnější, než jsem naznačil. Díky této funkci můžete na obyčejné proměnné navázat objekty, které mohou dělat nejrůznější věci. Například se dá vytvořit asociativní pole, které nerozlišuje malá písmena od velkých a spousta dalších triků. To už se ale ocitáme za hranicemi této knížky. Zajímavé kousky najdete například v [3].

15.3 DBM a datové struktury

Nastal čas vypořádat se s nepříjemným omezením DBM, kterým je možnost ukládat jen jednoduché datové typy. Naštěstí si můžete v CPAN obstarat modul *MLDBM*, který tento problém vyřeší za vás.

Postupuje jednoduchým a pochopitelným způsobem: datovou strukturu si interně převede na řetězec znaků a ten pak uloží. Při načítání pracuje opačně: vyzvedne z databáze řetězec a z něj rekonstruuje původní strukturu. Pro tyto převody používá standardní modul *Data::Dumper*, ovšem umí využívat i jiné. Zajímavý je především modul *Storable*, o kterém jsem se zmiňoval již v kapitole 11. Je totiž o poznání rychlejší.

Pokud potřebujete uložit do databáze pole či asociativní pole, můžete směle pracovat. Má-li vaše datová struktura více vrstev (pole obsahuje další odkazy na něco), nebude přístup k ní zcela přímočarý. Nemůžete si s ní pohrávat přímo v databázi, ale musíte ji vždy nejprve vytáhnout do pomocné proměnné, tam změnit a zase uložit:

```
$odkaz = $hash{»klíč«};
…změnit strukturu $odkaz…
$hash{»klíč«} = $odkaz;
```

Příklad: Následující program demonstruje, jak se zachází s *MLDBM*:

mldbm.pl

```
use MLDBM qw(DB_File Storable);

tie( my %db, 'MLDBM', "test.db")
     or die "Nelze otevřít test.db!\n";

# jednoúrovňové struktury lze přímo
$db{"alfa"} = [ 0.15, 1.23, 2.81 ];

# tohle nefunguje!
$db{"beta"}->[1]->[3] = 2.7182;

# víceúrovňové jen přes pomocné proměnné
my $pole = $db{"alfa"};
$pole->[1] = [ "aaa", "bbb", "ccc" ];
$db{"alfa"} = $pole;

# test, zda se podařilo...
$pole = $db{"alfa"};
```

```
print $pole->[1]->[1], "\n";
untie( %db );
```

Všimněte si příkazu **use**, který zajistí, že *MLDBM* bude používat Berkeley DB (*DB_File*) a modul *Storable*. Pokud byste neuvedli **qw**(…), použije se implicitní nastavení, které znamená SDBM a *Data::Dumper*. To jsou hodnoty sice všeobecně kompatibilní, ale dost pomalé. ■

15.4 Špetka SQL

V názvu části lehce lžu. Hodlám totiž psát o věci zvané *DBI* čili *DataBase Interface*. Jedná se o univerzální databázové rozhraní Perlu, které můžete používat pro nejrůznější typy databází (včetně textových souborů). Ovšem ve valné většině bývá nasazeno právě pro komunikaci s databázovými servery na bázi SQL. Proč? Pro jiné případy je zbytečně složité.

DBI má dva základní cíle: zprostředkovat Perlu styk s databází a udělat to jednotně. Je to modul, který nabízí určitou standardní paletu podprogramů nezávislých na použitém databázovém stroji. Díky tomu se případná změna databázového serveru projeví změnou jediného řádku. Když se totiž připojujete k serveru, musíte uvést jeho typ. Dále už je vše nezávislé.

Vlastní komunikaci s databázovým serverem zajišťuje speciální ovladač zvaný *DBD* (*DataBase Driver*). Toto je jediná součást *DBI*, která je závislá na druhu databáze. V době psaní této knihy existovaly ovladače pro desítky různých databází včetně nejslavnějších komerčních i volně šiřitelných titulů.

Modul *DBI* pracuje na bázi objektů. Jeho použití typicky vypadá tak, že zavoláte určitou metodu, která vytvoří nový objekt a vydá odkaz na něj (ve zdejší terminologii se mu říká „ovladač“). Prostřednictvím metod tohoto objektu pak provádíte další operace. Všechny používané objekty nabízejí metodu *errstr*, která vydá chybové hlášení vyvolané poslední operací. Používá se při zpracování chyb.

Otevření databáze má na starosti metoda *connect*. Jako parametr dostane řetězec ve tvaru:

”DBI:»*druh DBD*«:»*jméno databáze*«:»*počítač*«:»*port*«”

»*Druh DBD*« identifikuje ovladač, který se má použít. Je určen databázovým strojem, na kterém se dotyčná databáze nachází. Její identifikace tvoří zbytek řetězce, který bude předán ovladači. Může se jednat o samotné její jméno, pokud databázový server sídlí na tomtéž stroji, na kterém běží váš program. V opačném případě musíte přidat ještě doménové jméno či IP adresu serveru a port, na kterém server čeká. I databázové stroje mívají svá přístupová práva, takže metodě *connect* často musíte sdělit ještě dva parametry s přístupovým jménem a heslem k databázi. Celé otevření pak vypadá takto:

```
use DBI;
my $databaze =
    DBI->connect( "DBI:mysql:zamestnanci:db.firma.cz:3306",
                  "novak", "priSne+ajnE" )
    or die "Nelze otevřít databázi: ", DBI->errstr, "\n";
```

Proměnná *$databaze* nyní obsahuje odkaz na objekt, jehož prostřednictvím budete s otevřenou databází komunikovat. Až nastane její čas, uzavřete ji prostřednictvím:

```
$databaze->disconnect;
```

Předtím by však bylo záhodno ji alespoň trochu používat. Nejjednodušší metodou, která umožňuje provedení SQL dotazu, je *do*. Má jediný parametr – řetězec znaků obsahující dotaz. Například vytvoření tabulky *osoby*, která bude mít sloupce *jmeno*, *adresa*, *telefon* a *deti* obstará tento příkaz:

```
$databaze->do( "CREATE TABLE osoby (
                  jmeno CHAR(30),
                  adresa CHAR(50),
                  telefon INT,
                  deti INT
                )" );
```

Metodu *do* můžete použít všude tam, kde nepotřebujete z databáze získávat konkrétní data. Jejím výsledkem totiž je úspěch/neúspěch, případně počet řádků, kterých se operace dotkla. *do* poslouží pro vytváření a likvidaci tabulek a řádků v nich či pro změny údajů.

Jakmile však chcete položit dotaz SELECT, kterým se dotazujete na uložené hodnoty, bude situace komplikovanější. Musíte vlastně v malém opakovat známý postup: připravíte si dotaz (čímž vznikne objekt, jehož metody budete dále používat), následně jej provedete (i opakovaně s různými parametry) a po každém provedení zpracujete výsledky.

Dotaz vznikne voláním metody *prepare*. Vytvoří objekt reprezentující váš dotaz a jako výsledek vydá odkaz na něj. Další operace s dotazem budete provádět prostřednictvím metod tohoto objektu. Parametrem metody *prepare* je opět řetězec znaků tvořící SQL dotaz. Například dotaz, který později zjistí seznam jmen všech zaměstnanců, by vznikl pomocí:

```
$dotaz_vsichni = $databaze->prepare(
            "SELECT jmeno FROM osoby" );
```

Ovšem dotaz nemusí být v okamžiku přípravy ještě zcela konkrétní. Snadno se může stát, že se budete opakovaně dotazovat na adresu různých zaměstnanců. K tomu si připravte dotaz s parametrem, který se do SQL kódu zapisuje v podobě otazníku. V našem případě:

```
$dotaz_adresa = $databaze->prepare(
                "SELECT jmeno, adresa
                 FROM osoby WHERE jmeno = '?'"
                );
```

Proměnná *$dotaz_vsichni* resp. *$dotaz_adresa* nyní obsahuje odkaz na nově vzniklý objekt. Jelikož je příprava poměrně náročnou záležitostí, snažte se vyhýbat opakovaným přípravám stejných dotazů. Častým jevem je cyklus, v němž se provádí stále tentýž dotaz, ovšem pro různé hodnoty. V takovém případě je záhodno si jej připravit jen jednou – před zahájením cyklu – a uvnitř jej pouze opakovaně provádět.

Provedení zajistí metoda *execute*, které můžete v parametrech předat hodnoty. Ty se doplní na místa jednotlivých otazníků. Dotaz na adresu zaměstnance jménem Novák Josef bych provedl pomocí:

```
$dotaz_adresa->execute("Novák Josef");
```

Výsledkem metody je pouhá signalizace úspěch/neúspěch. Získaná data si objekt uloží v sobě v podobě výsledkové tabulky. Počet jejích řádků (čili počet řádků, které vyhověly dotazu) získáte metodou *rows*.

Samotné řádky pak vydává po jednom metoda *fetchrow_array*. Chová se podobně jako většina ostatních: každé zavolání vydá další řádek. Až dojdou, vrátí prázdný seznam. Řádek obdržíte v podobě seznamu hodnot, kdy každému sloupci odpovídá jedna jeho položka.

Příklad: Dosavadní konstrukce shrnu do jednoho konkrétního příkladu. Nebude to nic velikého, jen prostý nástroj na vyhledávání informací o zaměstnancích z výše uvedené databáze:

dbi.pl

```
use DBI;

my $databaze =
    DBI->connect( "DBI:mysql:zamestnanci:db.firma.cz:3306",
                  "novak", "priSne+ajnE" )
    or die "Nelze otevřít databázi: ", DBI->errstr;
my $dotaz = $databaze->prepare(
              "SELECT * FROM osoby WHERE jmeno=?"
          );
```

```
while ( my $jmeno = nacti_jmeno() ) {
    $dotaz->execute($jmeno);
    if ( $dotaz->rows == 0 ) {
        print "Není v databázi.\n";
    } else {
        while ( my @odpoved = $dotaz->fetchrow_array ) {
            print "Jméno: $odpoved[0]\n";
            print "Adresa: $odpoved[1]\n";
            print "Telefon: $odpoved[2]\n\n";
        }
    }
}
$databaze->disconnect;

sub nacti_jmeno {
    print "hledané jméno: ";
    my $vstup = <STDIN>; chomp($vstup);
    return $vstup;
}
```

■

Ještě lepší variantou přípravné metody je *prepare_cached*, která si pamatuje poslední připravený dotaz. Pokud se zopakuje, nenamáhá se a sáhne rovnou po zapamatovaném.

DBI nabízí i podporu transakcí. Databázový objekt oplývá metodami *commit*, která transakci potvrdí, a *rollback*, která ji odvolá a vrátí databázi do stavu před jejím zahájením. Typické použití: sbíráte návratové kódy prováděných dotazů, které jsou součástí transakce. Pokud všechny dopadly dobře, potvrdíte ji. V opačném případě ji odvoláte. Například:

```
my $ok = $dotaz1->execute();
$ok &&= $dotaz2->execute($jmeno);
if ( $ok ) {
    $databaze->commit;
} else {
    $databaze->rollback;
}
```

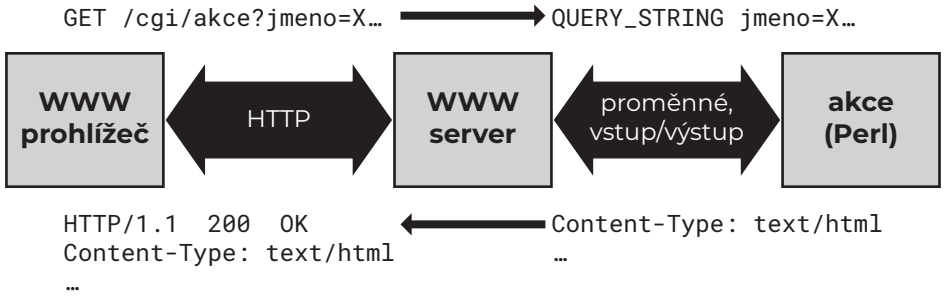
16 Perl motorem Webu

Rozhodující část komunikace mezi webovým prohlížečem a serverem spočívá ve výměně textových informací. Posílají se samozřejmě i binární data (třeba obrázky), ale HTML je text a řídicí informace také. Jelikož je Perl na práci s texty dobrý, nemělo by nikoho překvapit, že si zde vydobyl silnou pozici, svého času bezmála monopol.

Místo na slunci si nachází tam, kde je třeba stránky generovat dynamicky podle aktuální situace – obsahu datových struktur, hledaných řetězců a podobně. Tradičně pro tento účel sloužilo rozhraní zvané CGI, kdy se generovaly celé stránky, v současnosti se používá spíše AJAX, který poskytuje jen jejich části. Podívejme se, co Perl může nabídnout.

16.1 CGI

Zkratka CGI znamená *Common Gateway Interface* a označuje jednoduché rozhraní, kterým lze napojit téměř libovolný externí program na WWW server. Nejčastější uspořádání vypadá tak, že uživatel na WWW stránce vyplní formulář a odešle data serveru. Ten spustí obslužný program a prostřednictvím rozhraní CGI mu data předá ke zpracování. Odpověď, kterou od obslužného programu dostane, server předá prohlížeči jako výsledek odeslání formuláře.



Obrázek 16.1: Rozhraní CGI

Vstup dat do programu se odehrává prostřednictvím několika proměnných prostředí. Nejdůležitější je *REQUEST_METHOD*, podle které poznáte, zda data přicházejí v proměnné prostředí *QUERY_STRING* (hodnota `GET`) nebo standardním vstupem (hodnota `POST`). Jména proměnných jsou standardizována a měl by je dodržovat každý WWW server. Výsledky musí CGI program odevzdat ve standardním výstupu. Jeho povinným začátkem jsou hlavičky identifikující typ odesílaných dat.

Na podrobnější popis tu bohužel není místo (a upřímně řečeno ho ani nepotřebujete, jak uvidíte za chvíli). Dovolím si vás odkázat na stránky [9], kde je srozumitelně popsáno CGI i vytváření formulářů (kapitoly 6 až 11). Dále si dovolím předpokládat, že jste seznámeni s HTML a CGI.

Nic vám nebrání zpracovávat data hezky ručně. Nota bene je to jen práce s texty, v kterézto oblasti je Perl skutečně dobrý. Ovšem proč pracovat zbytečně, když máte k dispozici instantní řešení. Existují hotové nástroje, s jejichž použitím je výroba CGI programů potěchou pro oko i ducha.

K dispozici jich máte celou řadu, ale v širším měřítku se prosadily jen dva: knihovna *cgi-lib* a modul *CGI*. *cgi-lib* je starší, kratší a jednodušší. Její kód je opradu historický a bylo v něm objeveno několik bezpečnostních problémů. Proto je oficiálně doporučovaným prostředkem modul *CGI*. Zmíním se alespoň ve stručnosti o obou variantách.

16.2 Knihovna cgi-lib

Tato knihovna vznikla v době, kdy ještě neexistoval modul *CGI*. Je jednoduchá, ale bohatě dostačuje běžným potřebám. Oficiální distribuční stránku najdete na adrese:

http://cgi-lib.berkeley.edu/

Jelikož se nejedná o modul, vkládá se do programu příkazem:

```
require cgi-lib.pl;
```

require je jednodušší alternativou příkazu **use**. Vloží a provede uvedený soubor, ale nemá žádnou obalující inteligenci. Používá se pro vkládání souborů, které nejsou koncipovány jako moduly.

Základní funkcí *cgi-lib* je *ReadParse*, kterou musíte zavolat kdesi na začátku svého programu. Postará se o převzetí dat z CGI rozhraní, vše adekvátním způsobem dekóduje a hodnoty připraví do asociativního pole *%in*. Jako klíče slouží názvy položek z formuláře, jehož zpracování má váš program na starosti.

Knihovna poskytuje i pár funkcí usnadňujících vytváření HTML stránky, která bude odpovědí na odeslání formuláře. Není to žádný velký zázrak, ale potěší – viz tabulka 16.1.

Příklad: Jedním z nejčastějších CGI programů je odeslání údajů z formuláře elektronickou poštou. Uživatel vyplní jakýsi dotazník (např. objednávku) a zadané informace se pošlou elektronickou poštou osobě, která je má zpracovat.

PrintHeader	generuje povinné zahájení
HtmlTop(»*nadpis*«)	generuje úvod stránky
HtmlBot	generuje konec stránky

Tabulka 16.1: Generování HTML v *cgi-lib*

Řekněme, že existuje jednoduchoučký formulář pro zasílání připomínek k čemusi. Má jen tři položky: jméno uživatele (položka *jmeno*), adresu pro zaslání odpovědi (*email*) a vlastní připomínku (*pripominka*). Obslužný CGI program, který zadané informace pošle na adresu *kdosi@kdesi.cz* by vypadal takto:

wwwmail.pl

```
#!/usr/bin/perl
require "./cgi–lib.pl";

my $adresat = "kdosi\@kdesi.cz";
my $mailprog = "/usr/sbin/sendmail";

ReadParse();
if ( kontrola() ) {
      odesli_dopis();
      print PrintHeader();
      print HtmlTop("Dopis odeslán");
      print "Děkujeme za Vaše připomínky.\n";
      print HtmlBot();
} else {
      print PrintHeader();
      print HtmlTop("Neúplné údaje");
      print "Neuvedl jste žádnou připomínku.\n";
      print HtmlBot();
}

sub kontrola {
      if ( $in{"pripominka"} ) {
            return 1;
      } else {
            return 0;
      }
}
```

```
sub odesli_dopis {
    open(MAIL, "|$mailprog -t")
        or die "Nelze odeslat dopis.\n";
    print MAIL "To: $adresat\n";
    print MAIL "From: spravce\@www.kdesi.cz\n";
    print MAIL "Subject: Pripominka k www.kdesi.cz\n\n";

    print MAIL "Pripominka k www.kdesi.cz\n";
    print MAIL "(generovano z formulare)\n\n";
    print MAIL "Odesilatel: $in{'jmeno'}\n";
    print MAIL "E-mail: $in{'email'}\n";
    print MAIL "Pripominka:\n";
    print MAIL $in{'pripominka'}, "\n\n";
    close(MAIL);
}
```

Program předvádí typický postup: načtení zadaných dat (*ReadParse*), jejich kontrola (zde velmi primitivní) a podle výsledku buď zpracování nebo chybové hlášení. Vlastní odeslání dopisu zajišťuje podprogram *odesli_dopis*. Jak vidíte, komunikuje přímo s poštovním démonem *sendmail*. Pokud vám to připadá příliš syrové či málo objektově orientované, můžete si v CPAN obstarat modul *Mail::Mailer* a dopisy odesílat jeho prostřednictvím. Mně osobně to připadá zbytečné ale proti gustu… ■

16.3 Modul CGI

Jak již bylo řečeno, tento modul je doporučovanou cestou k vytváření CGI programů. Ve srovnání s *cgi-lib* nabízí podstatně širší paletu funkcí, především v oblasti generování HTML. Ostatně přístup ke vstupujícím datům nic moc nového ani nabídnout nemůže. Za poskytnutý komfort však zaplatíte velikostí. Soubor *CGI.pm* má v současné době zhruba 125 KB, zatímco *cgi-lib.pl* jen 15.

Zvláštností modulu je, že poskytuje dvě alternativní rozhraní: procedurální a objektově orientované. Procedurální je jednodušší a efektivnější. Jeho omezení jsou minimální a v běžných CGI programech vás jen těžko budou utlačovat. Proto jsem je použil v následujících příkladech. Naproti tomu v dokumentaci modulu (*man CGI*) jsou zpravidla objektově orientované příklady.

Procedurální rozhraní se neimportuje automaticky. Musíte použít příkaz:

```
use CGI qw(:standard);
```

CGI nemá žádnou funkci pro načítání dat z CGI rozhraní. To se provede automaticky při vložení modulu. Takže se nemusíte o nic starat a jen si užíváte. Přístup ke vstoupivším datům zajišťuje funkce *param*. Nejčastěji ji voláte s parametrem, kterým je název dotyčné položky formuláře. Například:

param('jmeno')

vydá hodnotu, kterou uživatel zadal do položky *jmeno*. Pokud je dotyčná položka vícehodnotová (např. sada vypínačů), bude výsledkem *param* seznam hodnot. Při volání bez parametru vydá seznam všech platných položek formuláře, které pak můžete používat jako parametry k dotazování na jednotlivé hodnoty.

Pokud byste chtěli hodnoty jednotlivých datových položek měnit, poslouží vám opět funkce *param*. Když jí zadáte více parametrů, vezme první jako jméno položky a následující jako seznam hodnot, které jí má přiřadit.

header	povinné záhlaví CGI výstupu
start_html	zahájení HTML stránky (hlavička)
end_html	zakončení HTML stránky

Tabulka 16.2: Generování HTML v modulu *CGI*

Mimořádně košatá je nabídka funkcí generujících HTML kód. Několik základních je uvedeno v tabulce 16.2. Kromě nich však máte k dispozici hromadu funkcí, jejichž jména se shodují s názvy HTML značek. Pokud je použijete bez parametrů, vytvoří dotyčnou značku. Když jim jako parametr předáte řetězec, obalí jej značkami. Takže například:

h1('Nadpis stránky');

vytvoří řetězec:

<h1>Nadpis stránky</h1>

A ještě do třetí nohy: v modulu *CGI* je přehršel specializovaných funkcí pro generování součástí formulářů (sady vypínačů, menu a podobné věci). Takže například přepínač, kterým uživatel sdělí jakého je pohlaví, můžete vyrobit následovně:

```
print "Jste ";
print radio_group(
        -name=>'pohlavi',
        -values=>['muž','žena','nevím'],
        -default=>'nevím' );
```

Z příkladu zároveň vidíte, že modul využívá pojmenované parametry[1]. Ovšem valná většina volání vystačí s jediným. V takovém případě si můžete název odpustit a zadat jen hodnotu, jak jste nedávno viděli u funkce *h1*.

U funkcí generujících HTML značky mají parametry stejná jména jako atributy dotyčných značek. Takže:

```
a(–href=>'http://www.kdesi.cz/', 'náš server');
```

vyrobí:

```
<a href="http://www.kdesi.cz/">náš server</a>
```

Příklad: Tentokrát jako příklad poslouží druhá z běžně se vyskytujících úloh: registrace účastníků nějaké akce. Uchazeč vyplní formulář a údaje o něm se uloží do souboru obsahujícího seznam zaregistrovaných. Formulář může obsahovat řadu položek, povinně však musí uvést své jméno a adresu. Přítomnost těchto položek je třeba prověřit.

registrace.pl

```
#!/usr/bin/perl
use CGI qw(:standard);
use Fcntl qw(:flock);
use constant JMENOSOUB => "/www/data/ucastnici.dat";

if ( kontrola() ) {
    zaregistruj();
    print header(),
        start_html("Registrace provedena"),
        h1("Registrace provedena"),
        "Byl(a) jste zaregistrován(a).\n",
        end_html();
} else {
    print header(),
        start_html("Neplatné údaje"),
        h1("Neplatné údaje"),
        "Prosím opravte zadání.\n",
        end_html();
}
```

1: Popisuji je na straně 117.

```
sub kontrola {
    # odstraním případné dvojtečky
    foreach my $jmeno ( param() ) {
        my $hodnota = param($jmeno);
        if ( $hodnota =~ s/:/;/g ) {
            param($jmeno,$hodnota);
        }
    }
    # a prověřím, zda je zadáno vše potřebné
    if ( param('jmeno') and param('adresa') ) {
        return 1;
    } else {
        return 0;
    }
}

sub zaregistruj {
    open(DATA, ">>".JMENOSOUB)
        or die "Nelze otevřít soubor\n";
    flock(DATA, LOCK_EX)
        or die "Nelze zamknout soubor\n";
    print DATA, join( ":", param() ), "\n";
    close(DATA);
}
```

Jelikož při ukládání do souboru odděluji jednotlivé položky prostřednictvím dvojteček, je třeba zajistit, aby se nevyskytovaly ve vlastních datech. Proto je kontrolní funkce nahradí vzhledově podobnými středníky. ■

Dlužno podotknout, že naznačené kontroly jsou velice primitivní. Nijak se nesnaží zjistit, jestli jméno a adresa obsahují alespoň trochu smysluplné údaje. Skutečný kontrolní mechanismus by se neměl spokojit s prostou existencí určité položky, ale zajímat se o její obsah (např. zda jsou ve jméně alespoň dvě slova, zda E-mail vypadá realisticky a podobně). Také chybová hlášení v případě neúspěchu by měla konkrétněji sdělovat příčinu problému (které konkrétní položky chybí a podobně).

Nezapomeňte na zamykání při přístupech k souborům. WWW je vysoce paralelní prostředí a snadno se může stát, že tentýž CGI program spustí několik uživatelů současně.

16.4 AJAX

Tradiční přístup ke generování dynamických stránek ve stylu „kliknu na odkaz a dostanu celou novou stránku" je pro moderní webové aplikace příliš neohrabaný. Typický příklad: prohlížíte si na webu mapu a změníte její měřítko. Vlastní obrázek mapy je třeba překreslit (a obstarat pro něj ze serveru příslušná data), ale jeho okolí zůstává beze změny. Přenášet znovu celou stránku je neefektivní, navíc se bude znovu celá vykreslovat a uživatel nebude nadšený.

Proto vznikl *AJAX (Asynchronous JavaScript and XML)*. Jeho základní myšlenkou je, že v prohlížeči běží jako součást stránky program v JavaSciptu, který hlídá důležité události. Dojde-li k nějaké (např. zmíněná změna měřítka mapy), pošle dotaz serveru a na základě dat z jeho odpovědi změní příslušnou část stránky. Její zbytek zůstane beze změny.

Původně se pro výměnu dat mezi prohlížečem a serverem používal formát XML, který figuruje i v názvu technologie. Postupně byl ale vytlačen formátem *JSON (JavaScript Object Notation)*, který je úspornější a navíc pro JavaScript nativní. Vlastně by se AJAX měl přejmenovat na AJAJ, ale to by neznělo tak dobře.

Co to znamená na straně serveru, kde může sloužit Perl? Upřímně řečeno se zase tak mnoho nezmění. Opět dorazí dotaz protokolem HTTP, server zavolá příslušný program, předá mu dotaz ve standardním vstupu a očekává odpověď k odeslání v jeho standardním výstupu. V programu se opět bude analyzovat vstup modulem *CGI*, který zpřístupní informace z něj. Změny se projeví zejména v odpovědi. Ty nejpodstatnější jsou:

- Odpovědí není celá stránka, ale jen relevantní data.
- Jako typ odpovědi je třeba poslat `application/json` místo původního `text/html`.
- Odpověď je třeba zakódovat do formátu JSON.

Co jsou ona relevantní data a zda se skutečně používá formát JSON nebo třeba XML vždy záleží na konkrétní webové aplikaci. Klíčový je soulad mezi klientem a serverem – program běžící v prohlížeči musí rozumět odpovědím, které od serveru dostává. Popsal jsem typický případ.

Čili v podstatě jedinou novinkou je kódování do formátu JSON. Ten je textový, takže by dal generovat ručně, nicméně proč vynalézat kolo, když existuje modul *JSON*. Opět nabízí jednoduché procedurální i složitější objektové rozhraní, které oceníte, když potřebujete něco speciálního.

V běžných případech si vystačíte s jednoduchým voláním funkce *encode_json*, které předáte proměnnou a ona její hodnotu zakóduje do JSON. Proměnná musí být skalární. Místo pole nebo asociativního pole předejte odkaz na ně. Výstupem je řetězec, nezapomeňte jej přiřadit do proměnné nebo rovnou vytisknout do standardního výstupu, kde webový server očekává výstup z aplikace. Pokud by se předávala data i v opačném směru, máte k dispozici i *decode_json*.

Příklad: Řekněme, že jako součást webové stránky bude někde zobrazen přehled procesů běžících na serveru. Uživatel si zadáním řetězce znaků může omezit, že jej zajímají jen některé. Přehled se bude aktualizovat přes AJAX, řetězec omezující jména bude prohlížeč serveru předávat v položce `nazev`.

Vytvořím aplikaci, která na dotaz odpoví – spustí *ps –x* a zašle zpět pole řádků z jeho výstupu, které obsahují zadaný řetězec:

procesy.pl

```
1   use JSON;
2   use CGI;
3   my $cgi = CGI->new;
4
5   my $nazev = $cgi->param("nazev");
6   my @nalezeno = ();
7
8   open( PROCESY, "ps -x |" )
9       or die "Nelze spustit ps\n";
10  while ( my $radek = <PROCESY> ) {
11      chomp($radek);
12      if ( $radek =~ /$nazev/ ) {
13          push( @nalezeno, $radek);
14      }
15  }
16  close(PROCESY);
17
18  print $cgi->header("application/json;charset=UTF-8");
19  print encode_json(\@nalezeno);
```

Zároveň využívám tuto příležitost, abych ilustroval objektové rozhraní modulu *CGI*. Jak vidíte, není to nic speciálního. Je třeba vytvořit příslušný objekt (řádek 3) a jeho služby se pak využívají v podobě metod či datových položek, jejichž jména odpovídají procedurálnímu rozhraní (řádky 5 a 18).

Upozorňuji, že informace o běžících procesech a jiném dění uvnitř serveru jsou zneužitelné a jejich zveřejňování představuje riziko. Také by bylo záhodno nedůvěřovat slepě řetězci znaků z příchozího požadavku. Což nás přivádí k následující problematice: ■

16.5 Bezpečnost

Programy spouštěné na serveru představují jeho potenciální ohrožení. Dovolují totiž externímu uživateli spouštět program na vašem počítači. Pravda, program je pod vaší kontrolou, ale data mu

předkládá druhá strana. Základní forma útoku spočívá v tom, že se potenciální vetřelec snaží váš program zmást a podstrčit mu taková data, která způsobí činnost podle jeho gusta.

Nebezpečné jsou především ty hodnoty, které váš program využívá k sestavování příkazů, příkazových řádků či jmen souborů. Útočník se snaží podstrčit svůj příkaz či soubor. Proto je záhodno povolit v takto používaných vstupech jen „obyčejné znaky" – písmena, číslice, podtržítko, pomlčku a tečku. Žádné mezery, středníky, zpětné apostrofy a další vychytávky.

Perl speciálně pro takovéto případy nabízí tak zvaný nakažený (tainted) režim. Aktivujete jej volbou `-T` při startu interpretu. V něm se chová jako těžký paranoik a nedůvěřuje žádným datům, která přicházejí zvenčí – v parametrech, proměnných prostředí, souborech či výpisech adresářů. Pokud se takto získaná data pokusíte použít například jako jméno souboru, skončíte s chybovým hlášením.

Jediná cesta k dezinfekci dat vede přes zapamatování v regulárních výrazech. Například následující postup vede k „čisté" hodnotě proměnné *$jmenosoub*:

```
$jmenosoub = param('soubor');
$jmenosoub =~ /^([-.\w]*)/;
$jmenosoub = $1;
```

Nyní *$jmenosoub* obsahuje extrakt z položky *soubor* od začátku až před první nepovolený znak. Přísnější verze na konec regulárního výrazu vkládá ještě $, takže pokud uživatel zadal nepovolený znak, vstup se místo oříznutí kompletně zamítne:

kontrola.pl

```
my $jmenosoub = param('soubor');
if ( $jmenosoub =~ /^([-.\w]+)$/ ) {
    $jmenosoub = $1;
} else {
    die "Co to na mě zkoušíš?\n";
}
```

Je velmi vhodné spouštět všechny programy pro web v nakaženém režimu. Při práci s proměnnými to může vést k jistému nepohodlí, ale můžete se tak ochránit před nepříjemným překvapením. Stačí úvodní řádek každého programu upravit na:

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

Zásadně se nespoléhejte na žádné kontroly, které probíhají mimo program. Například že obsah skryté položky formuláře nelze změnit. Nebo že jste na stránku umístili JavaScript, který zabrání vkládání nepovolených znaků. Protokol HTTP, kterým se data přenášejí, je otevřený. Kdoko-

li si přečte jeho volně dostupnou definici, může obyčejným Telnetem simulovat WWW klienta a zadat data, jaká chce. Nerad to říkám, ale tomu, co přichází od uživatele, se skutečně nedá věřit.

Dalším prvkem, kterému je záhodno se vyhýbat, je změna uživatele jehož přístupovými právy program disponuje (setuid). Snažte se raději nastavit přístupová práva tak, aby serverem spouštěný program mohl běžet se standardními právy. *Za žádnou cenu nespouštějte z webu programy s oprávněním správce systému.*

Část IV

Přílohy

Priorita operátorů

asociativita	operátory	popis
zleva	termy a levé seznamové operátory	
zleva	–>	*volání metody, dereference*
ne	++ ––	*zvětšení, zmenšení*
zprava	**	*mocnina*
zprava	! ~ \ unární + a –	*logické not, binární not, odkaz*
zleva	=~ =!	*srovnání*
zleva	* / % x	*násobení, dělení, modulo, opakování řetězce*
zleva	+ – .	*sčítání, odčítání, zřetězení*
zleva	<< >>	*bitové posuny*
ne	pojmenované unární operátory a testy souborů	
ne	< > <= >= **lt gt le ge**	*porovnání*
ne	== != <=> **eq ne cmp**	*porovnání*
zleva	&	*bitové and*
zleva	| ^	*bitové or, xor*
zleva	&&	*logické and*
zleva	||	*logické or*
ne	.. ...	*rozsah*
zprava	?:	*ternární podmíněný*
zprava	= += –= a další	*přiřazení*
zleva	, =>	*oddělovač v seznamech*
ne	pravé seznamové operátory	
zprava	**not**	*logické not*
zleva	**and**	*logické and*
zleva	**or xor**	*logické or, xor*

17 Řešení ke cvičením

Tato příloha obsahuje řešení k většině příkladů uvedených v textu. Pokud je řešením program, neberte jej prosím jako dogma. Při mnohotvárnosti Perlu nepochybně existuje řada alternativ, jak dosáhnout kýženého výsledku. Zdejší programy zkrátka představují způsob, kterým bych danou úlohu řešil já.

Řešení 2.1 dlouhá verze

```
$zisk = $zisk * 2
$sklad = $sklad – $prodano
```

krátká verze

```
$zisk *= 2
$sklad –= $prodano
```

Řešení 2.2 Napadají mne dvě varianty. Buď:

```
$c = sqrt( $a**2 + $b**2 )
```

nebo lze druhou mocninu rozepsat pomocí násobení:

```
$c = sqrt( $a*$a + $b*$b )
```

Řešení 2.3 Vypadá to možná hrozivě, ale řešení je celkem snadné. Stačí od proměnné *$x* odečíst zbytek po jejím dělení deseti. Ve zkráceném tvaru by to bylo:

```
$x –= $x % 10
```

Řešení 2.4 Umírněné řešení by mohlo vypadat takto:

```
$prvni = substr($slovo, 0, 1, "");
$posledni = substr($slovo, –1, 1, "");
$slovo = $posledni . $slovo . $prvni;
```

Odeberu z řetězce první a poslední znak a následně vše složím v požadovaném pořadí. Dá se to však udělat i jediným příkazem:

```
$slovo .=   substr($slovo, 0, 1, substr($slovo, –1, 1, "") );
```

Řešení 3.1 Výsledkem bude pravda. *$pokus* sice po provedení **substr** obsahuje prázdný řetězec (tedy nepravdivou hodnotu), ale podmínkou je výsledek volání funkce. Tím je vymezený podřetězec, v našem případě „nazdar“, což je hodnota pravdivá.

Řešení 3.2

1. Délku řetězce zjišťuje funkce **length**, takže:
   $r **ne** "" **and** **length**(*$r*)<**length**(*$s*)
   U první podmínky testující neprázdnost *$r* to svádí použít samotné *$r*. Ovšem taková podmínka by nefungovala dobře, pokud by *$r* obsahovalo řetězec "0". Proto raději použijte naznačený test.
2. Tohle byl trochu chyták, protože logické operace zde nejsou potřeba. Je-li jedno číslo kladné a druhé záporné, je jejich součin jistě záporný. Proto stačí podmínka:
   $a * *$b* < 0
   Lze to však vyjádřit i po lopatě:
   $a<0 **and** *$b*>0 **or** *$a*>0 **and** *$b*<0
   Díky vhodně nastaveným prioritám není třeba používat závorky.

Řešení 3.3 Maximum ze dvou hodnot určíte celkem snadno:

```
if ( $a > $b ) {
   $max = $a;
} else {
   $max = $b;
}
```

Největší ze tří hodnot lze vybrat několika způsoby. Mému srdci nejvíce lahodí tento:

```
if ( $a > $b ) {
   $max = $a;
} else {
   $max = $b;
}
if ( $c > $max ) { $max = $c; }
```

Kompaktní a docela systematická je i tato varianta:

```
$max = $a;
if ( $b > $max ) { $max = $b; }
if ( $c > $max ) { $max = $c; }
```

Řešení 3.4 Program pro řešení kvadratické rovnice může vypadat třeba takto:

kvadrov.pl

```
print "Řešení kvadratické rovnice a*x*x + b*x + c = 0\n\n";
print "zadejte a: "; my $a = <STDIN>; chomp($a);
```

```
print "zadejte b: "; my $b = <STDIN>; chomp($b);
print "zadejte c: "; my $c = <STDIN>; chomp($c);
print "\n";
if ( $a == 0 ) {
    print "To není kvadratická rovnice (a=0)\n";
} else {
    $d = $b*$b - 4*$a*$c;
    if ( $d > 0 ) {
        $d = sqrt($d);
        print "x1 = ", (-$b + $d) / (2*$a), "\n";
        print "x2 = ", (-$b - $d) / (2*$a), "\n";
    } elsif ( $d == 0 ) {
        print "x1 = x2 = ", -$b / (2*$a), "\n";
    } else {
        print "Rovnice nemá reálné řešení.\n"
    }
}
```

Všimněte si kontroly nenulovosti *$a* – jinak by při výpočtu hrozilo dělení nulou.

Řešení 3.5 Je třeba nezapomenout na přípravné práce – převést čísla na jejich absolutní hodnoty (na děliteli to nic nemění a záporné číslo by s algoritmem nepěkně zacloumalo) a ověření nenulovosti:

nsd.pl

```
2   if ( $a * $b == 0 ) {
3       print "Hohó! Nula nemá dělitele.\n";
4   } else {
5       if ( $a < 0 ) { $a = -$a; }
6       if ( $b < 0 ) { $b = -$b; }
7       while ( $a != $b ) {
8           if ( $a < $b ) {
9               $b -= $a;
10          } else {
11              $a -= $b;
12          }
13      }
14      $nsd = $a;
15  }
```

Řešení 5.1 K převodu čísla měsíce na jméno není pole potřeba. Lze to zvládnout sadou podmíněných příkazů (jestliže je číslo rovno 1, vypiš „leden“, jinak…). Takové řešení je však dost

neohrabané, dá vám spoustu psaní a nebude z nejrychlejších. Jedinou jeho výhodou je, že nemusíte umět pracovat s polem. Jakmile tuto znalost zvládnete, jistě dáte přednost zhruba následující variantě:

jmenomes.pl

```
my @mesic = ( "leden", "únor", "březen", "duben",
              "květen", "červen", "červenec", "srpen",
              "září", "říjen", "listopad", "prosinec" );
print "Zadejte číslo měsíce: ";
my $cislo = <STDIN>;
chomp($cislo);
print "Pořadové číslo $cislo má $mesic[$cislo-1].\n";
```

Jelikož jsou názvy měsíců uloženy od indexu 0, musím od zadaného čísla odečíst jedničku. Pokud bych se chtěl této vlastnosti vyhnout, musel bych seznam v prvním přiřazovacím příkazu zahájit ("", "leden"…

Řešení 5.2 První příkaz navzájem vymění hodnoty prvních dvou prvků pole *@test* – zde se právě projeví, že se nejprve určí přiřazované hodnoty a jejich cíle a teprve potom se provede přiřazení. Kdybych použil dvojici příkazů:

```
$test[0] = $test[1];
$test[1] = $test[0];
```

Přiřadila by se do obou prvků pole hodnota 1. Při provádění druhého přiřazení by totiž původní hodnota *$test*[0] již byla přepsána. Závěrečný příkaz pak do prvků *$test*[2], *$test*[3] a *$test*[4] zkopíruje hodnoty z prvků s indexem o jedničku vyšším.

Výsledná hodnota pole *@test* tedy bude (1, 0, 3, 4, 5, 5).

Řešení 5.3 Algoritmus je prostý: budu si pamatovat měsíc s největší dosud nalezenou výrobou (za výchozí hodnotu vezmu leden). Projdu zbytek pole a pokud najdu měsíc s větší výrobou, zapamatuji si nového rekordmana. Po zpracování celého pole mám nalezeno absolutní maximum.

maxvyr.pl

```
my $max = 1;
foreach my $mesic ( 2..12 ) {
    if ( $vyroba[$mesic] > $vyroba[$max] ) {
        $max = $mesic;
    }
}
print "Největší výroba ($vyroba[$max]) ";
print "byla dosažena v měsíci $max.\n";
```

Řešení 5.4 Základní datovou strukturou pro postfixový kalkulátor je zásobník. Na jeho vrchol se ukládají vstupní hodnoty a mezivýsledky, odebírají se z něj potřebné operandy. K jeho realizaci je ideální dvojice funkcí **push** a **pop** nebo **shift** a **unshift**. Použiji třeba druhou z nich.

Zásobník bude realizován polem *@zasobnik* a jeho vrchol se bude nacházet na začátku (index 0). Základní algoritmus: načtu vstup. Jedná-li se o operátor, vyzvednu dva prvky z vrcholu zásobníku, provedu potřebnou operaci a výsledek opět uložím na zásobník. Jestliže vstupem není operátor, považuji jej za číslo a uložím na vrchol zásobníku. To celé opakuji, dokud nevstoupí prázdný řetězec. Navrhuji následující implementaci této myšlenky:

revkalk.pl

```
print "Kalkulátor pro polskou reverzní notaci\n";
my @zasobnik = ();
my ( $vysledek, $vstup );
do {
    print "vstup (prázdný ukončí): ";
    $vstup = <STDIN>;
    chomp($vstup);
    if ( $vstup eq "+" ) {
        $vysledek = shift(@zasobnik) + shift(@zasobnik);
        unshift(@zasobnik, $vysledek);
    } elsif ( $vstup eq "–" ) {
        $vysledek = splice(@zasobnik,1,1) – shift(@zasobnik);
        unshift(@zasobnik, $vysledek);
    } elsif ( $vstup eq "*" ) {
        $vysledek = shift(@zasobnik) * shift(@zasobnik);
        unshift(@zasobnik, $vysledek);
    } elsif ( $vstup eq "/" ) {
        $vysledek = splice(@zasobnik,1,1) / shift(@zasobnik);
        unshift(@zasobnik, $vysledek);
    } elsif ( $vstup ne "" ) {
        $vysledek = $vstup;
        unshift(@zasobnik, $vstup);
    }
    print "> $vysledek\n";
} while ( $vstup ne "" );
```

Všimněte si odečítání a dělení. U nich záleží na pořadí operandů. Jelikož uživatel zadal prvního dělence a dělitele až po něm, leží dělitel na vrcholu zásobníku. Já však jako prvního potřebuji dělence. Proto si nejprve vyzvednu druhý prvek pole *@zasobnik* (pomocí funkce **splice**) a jinde obvyklý **shift** použiji jen pro druhý operand.

Řešení 5.5 V *$a* je uložen řetězec „zelená“ (poslední hodnota seznamu) a v *$b* číslo 3 (počet prvků pole).

Řešení 6.1 Všechny, které jsem našel, jsem také opravil. Ale určitě nějaká zbyla…

Řešení 6.2 Rodné číslo je tvořeno šesti číslicemi, za kterými následuje lomítko a další tři nebo čtyři číslice. Příslušný regulární výraz bude zřejmě silně záviset na vaší znalosti Perlu. Zde se pokusím vymezit tři stupně poznání:

začátečník	[0–9][0–9][0–9][0–9][0–9][0–9]/[0–9][0–9][0–9][0–9]?
středně pokročilý	\d\d\d\d\d\d/\d\d\d\d?
regulární Mistr	\d{6}/\d{3,4}

Řešení 6.3 Prvnímu „.*“ bude odpovídat „Popokatepetl“ a druhému prázdný řetězec. Regulární výrazy jsou hladové a postupují od začátku. Vyhodnocení začne tím, že zkusí prvnímu „.*“ přiřadit co nejvíce, tedy celý zkoumaný řetězec. Následně se podívá, zda dokáže regulární výraz dokončit. V něm zbývá už jen druhé „.*“. Tomu odpovídá libovolný řetězec znaků, také prázdný. Jelikož ve zkoumaném řetězci už nic nezbývá, použije právě prázdný řetězec. Tím se podařilo úspěšně dojít na konec vzoru.

Řešení 6.4 Pokud se ve zkoumaném textu objeví několik řetězců v uvozovkách, uvedený regulární výraz selže. Díky hladovosti „.*“ se roztáhne od první uvozovky až po poslední – například:

Kdo řekl ”A”, musí říci i ”B”.

což není to, co jsem chtěl. Aby se choval správně, musí se místo „libovolný řetězec“ použít „libovolný počet znaků různých od uvozovek“. Řešením tedy je

”[^”]*”

Řešení 6.5 Oběma „.*?“ bude odpovídat prázdný řetězec. Nikde totiž není řečeno, že vzor se musí roztáhnout na celý zkoumaný text. Zde Perl zkusí nejprve přidělit oběma minimum (prázdný řetězec) a vzápětí zjistí, že se mu podařilo úspěšně dorazit na konec vzoru.

Pro hledání řetězců v uvozovkách by se dalo použít:

”.*?”

Z hlediska konečného výsledku je toto řešení rovnocenné řešení minulého cvičení. Doba provádění však bude poněkud delší, protože se vyhledávací stroj bude opakovaně vracet a prodlužovat řetězec odpovídající „.*?“. Naproti tomu ”[^”]*” je zpracováno jediným průchodem bez návratů.

Řešení 6.6

1. Řádek obsahující alespoň jednu číslici (kdekoli).
2. Řádek začínající číslicí.
3. Řádek začínající číslicí, před kterou však smí být prázdné místo (mezery, tabulátory).
4. Libovolný řádek. Hledat samotné \d* (\w*, A* atd.) je chyba, protože libovolný počet opakování připouští i nulový počet opakování. To znamená, že vašemu požadavku vyhoví i prázdný řetězec a ten je obsažen všude. Na tuto chybu si ve svých regulárních začátcích naběhl snad každý.
5. Řádek obsahující pouze číslice, nikoli však prázdný řádek. Kdybych připustil ještě znaménko a prázdné místo na začátku i na konci, vznikla by podmínka pro „řádek s jedním celým číslem“. Zkuste si takový regulární výraz vytvořit a otestovat jej.

Řešení 6.7 Stačí nahradit dotyčné slovo správným zápisem. Pomocí \b na začátku i na konci zajistím, že se skutečně bude jednat o samostatné slovo. A je třeba to provést pro všechny výskyty:

```
$radek =~ s/\bpavel\b/Pavel/g;
$radek =~ s/\bsatrapa\b/Satrapa/g;
```

Řešení 6.8 Jedno z možných řešení funguje takto: Po načtení vyhledám v řádku první dvě slova a vyměním jejich pořadí. Výsledek uložím do pole. Když je celý vstup načten, nechám pole uspořádat pomocí funkce **sort** a vypíši výsledek:

seznamjm.pl

```
use locale;

my @jmena = ();
while ( my $radek = <> ) {
    chomp($radek);
    # vyměním první dvě slova
    $radek =~ s/\s*(\w+)\s+(\w+)/$2 $1/;
    # a uložím do pole
    push(@jmena, $radek);
}

foreach my $radek ( sort(@jmena) ) {
    print "$radek\n";
}
```

Mimochodem – kdyby měl být výstup opět ve tvaru *jméno příjmení*, stačilo by na začátku těla **foreach** cyklu zopakovat stejný substituční příkaz, který by opět vyměnil pořadí prvních dvou slov. Seznam jmen by se tak vypisoval v původním tvaru, ale v pořadí podle příjmení.

Řešení 6.9 Nejprve funkcí **split** rozdělím řádek z */etc/passwd* na jednotlivé položky. Oddělovačem je dvojtečka. Nepříjemným požadavkem je, že uživatelé mají být uspořádáni podle příjmení. Proto musím vyměnit pořadí slov ve jméně uživatele a takto pozměněný řádek si uložím do pole *@uzivatele*. To si nechám uspořádat funkcí **sort** a těsně před tiskem řádku slova opět vrátím do původního pořadí.

uziv.pl

```
use locale;

my @uzivatele = ();
while ( my $radek = <> ) {
    chomp($radek);
    my @polozky = split( /:/, $radek );
    # mám raději názvy než nicneříkající indexy
    my $uziv = $polozky[0];
    my $jmeno = $polozky[4];
    if ( $jmeno =~ /\w+\s+\w+/ ) {
        # je to reálný uživatel
        # prohodím jméno s příjmením a uložím
        $jmeno =~ s/(\w+)\s+(\w+)/$2 $1/;
        push(@uzivatele, "$jmeno ($uziv)");
    }
}

foreach my $radek ( sort(@uzivatele) ) {
    $radek =~ s/(\w+)\s+(\w+)/$2 $1/;
    print "$radek\n";
}
```

Řešení 7.1 Počty výskytů jednotlivých slov budu ukládat do asociativního pole *%pocty*, které je pro takovýto účel jako stvořené. Zpracování řádků bude probíhat tak, že vždy nejprve odříznu počáteční neslovní znaky a pokud něco zbylo, odstraním a zapamatuji si slovo, kterým řádek začíná. Zapamatovanému slovu pak v poli *%pocty* zvětším počet výskytů o jedničku.

slova.pl

```
use locale;

my %pocty = ();
while ( my $radek = <> ) {
    chomp($radek);
    while ( $radek ne "" ) {
        $radek =~ s/^\W*//;
```

```
            if ( $radek =~ s/^(\w+)// ) {
                $pocty{$1}++;
            }
            print "$radek\n";
        }
    }

    foreach my $slovo ( sort keys %pocty ) {
        print "$slovo ... $pocty{$slovo}\n";
    }
```

Řešení 8.1 Výstupem bude:

```
a = 20
a = 10
```

Vznikne takto: Na řádku 1 je založena globální proměnná *$a*, které se přiřadí hodnota 10. Následuje volání podprogramu *raz*. Ten na řádku 6 odloží stávající hodnotu proměnné *$a* a nahradí ji číslem 20. Nevzniká však žádná nová proměnná, dále se pracuje s globální. Na dalším řádku se volá podprogram *dva*, který zobrazí současnou hodnotu globální proměnné *$a*, tedy 20. Následuje ukončení podprogramu *dva* a jelikož jeho volání bylo posledním příkazem podprogramu *raz*, vzápětí (na řádku 8) končí i ten. V rámci ukončení se proměnné *$a* vrátí původní hodnota 10, kterou následně vypíše příkaz na řádku 3.

Pokud bych na šestém řádku místo **local** použil **my**, ohlásí program dvakrát hodnotu 10. V tomto případě totiž na řádku 6 vznikne zcela nová, lokální proměnná *$a*, která je dostupná jen v rámci podprogramu *raz*. Do ní se uloží číslo 20. Následně volaný podprogram *dva* nemá k této lokální proměnné přístup a proto zobrazí nezměněnou hodnotu globální proměnné *$a*, čili 10.

Řešení 8.2 Úprava by mohla vypadat třeba takto:

overeni.pl

```
sub vstup_hodnoty {
    my $hodnota;
    do {
        print "hodnota: ";
        $hodnota = <STDIN>;
        chomp($hodnota);
    } until platny_vstup($hodnota);
    return $hodnota;
}
```

```
sub platny_vstup {
# ověří, zda uživatel zadal číslo nebo prázdný řetězec
# vstup: uživatelem zadaná hodnota
# výstup: 1 je–li vstup platný, 0 jinak
    my ( $hodnota ) = @_;
    if ( $hodnota eq "" ) {
        return 1;
    } elsif ( $hodnota =~ /^\s*[-+]?\d+\s*$/ ) {
        return 1;
    } else {
        print "Zadejte celé číslo nebo prázdný řetězec!\n";
        return 0;
    }
}
```

Řešení 8.3 Kontrola vstupních dat je v podstatě součástí vstupní procedury. Důkazem je i to, že ji lze realizovat samostatným programem, který data připraví ještě před zpracováním hlavním slosovadlem. Proto se nabízí doplnit kontrolu jako součást vstupního podprogramu *dalsi_los*.

Situace se komplikuje v tom, že načtení řádku nemusí stačit k získání platného losu. Občas jej kontrola vyřadí a bude se muset číst dál. Proto se ve funkci *dalsi_los* objevuje cyklus, který skončí buď koncem vstupních dat nebo nalezením platného losu. Proti původnímu znění programu se změnilo následující:

```
# globální proměnné                                                    losyb.pl
my $pocetcisel = 5; # počet losovaných čísel
my $maxcislo = 50; # největší losované číslo
my (%tazeno, %zpracovano); # tažená čísla, zpracované losy
my (@prvni, @druzi, @treti); # výherci

inicializace();
vylosuj_cisla();
while ( my @los = dalsi_los() ) {
    my $poradi = urci_poradi(@los);
    if ( $poradi <= 3 ) {
        zarad_viteze($los[0], $poradi);
    }
}
vypis_viteze();
```

```
sub inicializace {
    srand;
}

sub vylosuj_cisla {
    my ( $cislo, $vylosovano ) = ( 0, 0 );
    while ( $vylosovano < $pocetcisel ) {
        $cislo = int( rand($maxcislo) ) + 1;
        if ( not $tazeno{$cislo} ) {
            $tazeno{$cislo} = 1;
            $vylosovano++;
        }
    }
    print "Tažena byla čísla: ";
    foreach $cislo ( sort {$a <=> $b} keys %tazeno ) {
        print "$cislo ";
    }
    print "\n";
}

sub dalsi_los {
    my @los = ();
    while ( my $radek = <> ) {
        chomp($radek);
        @los = kontrola_dat( split(/\s+/, $radek) );
        if ( @los ) { last; }
    }
    return @los;
}

sub kontrola_dat {
    my @los = @_;
    my $cislo = shift(@los);
    if ( cislo_uz_bylo($cislo) ) { return (); }
    @los = prover_cisla(@los);
    if ( @los ) {
        return ( $cislo, @los );
    } else {
        return ();
    }
}
```

```
sub cislo_uz_bylo {
    my ( $cislo ) = @_;
    if ( $zpracovano{$cislo} ) {
        return 1;
    } else {
        $zpracovano{$cislo} = 1;
        return 0;
    }
}

sub prover_cisla {
    my %cisla;
    foreach my $cislo ( @_ ) {
        $cisla{$cislo} = 1;
    }
    if ( (keys %cisla) > $pocetcisel ) {
        return ();
    } else {
        return sort {$a <=> $b} keys %cisla;
    }
}

sub urci_poradi {
    my $uhodl = 0;
    shift(@_); # zbavím se čísla losu
    foreach my $cislo ( @_ ) {
        if ( $tazeno{$cislo} ) { $uhodl++; }
    }
    return $pocetcisel + 1 - $uhodl;
}

sub zarad_viteze {
    my ( $los, $poradi ) = @_;
    if ( $poradi == 1 ) { push(@prvni, $los); }
    elsif ( $poradi == 2 ) { push(@druzi, $los); }
    elsif ( $poradi == 3 ) { push(@treti, $los); }
}

sub vypis_viteze {
    print "Výherci první ceny:\n";
```

```
    vypis_seznam(@prvni);
    print "\nVýherci druhé ceny:\n";
    vypis_seznam(@druzi);
    print "\nVýherci třetí ceny:\n";
    vypis_seznam(@treti);
}

sub vypis_seznam {
    foreach my $los ( sort {$a <=> $b} @_ ) {
        print "$los\n";
    }
}
```

Jak je vidět, kontrola vstupních dat je rozložena do dvou kroků. Prvním je, zda se číslo losu vyskytuje poprvé (funkce *cislo_uz_bylo*). Jedná se o přímočarou aplikaci asociativního pole *%zpracovano*, do kterého ukládám čísla zpracovaných losů.

Ve druhém kroku kontroluji tipovaná čísla. Abych si nemusel dělat hlavu s jejich pořadím na losu, prostě každému tipovanému číslu přiřadím jedničku v asociativním poli *%cisla*. Případné opakované výskyty téhož čísla se tak scvrknou do jediné položky. Na závěr vyzvednu všechny klíče z tohoto pole a běda, když jich bude více než losovaných čísel…

Řešení 9.1 Řešení spočívá v přímočarém použití funkce **printf**. Stačí si uvědomit, že názvy výrobků musí být zarovnány nalevo a že je vhodné se pojistit maximální délkou proti případnému dlouhému názvu. Pokud čísla zarovnáte napravo a stanovíte pevný počet číslic za desetinnou tečkou, vzniknou sloupce právě v požadovaném tvaru.

cenik.pl

```
while ( my $radek = <> ) {
    chomp($radek);
    my ($nazev,$cena) = split( /#/, $radek );
    printf "%-40.40s %9.2f Kč %9.2f Kč\n", $nazev, $cena, $cena*1.21;
}
```

Řešení 9.2 Pokud místo skládání cest použiji přepínání aktuálního adresáře, leccos se zjednoduší. Například již nemusím podprogramu předávat jméno adresáře, pracuje se vždy s aktuálním. Jen jej musím před voláním podprogramu změnit a po dokončení zase vrátit (řádky 12 a 14).

tree2.pl

```
1   zpracuj_adr( "" );
2
3   sub zpracuj_adr {
4       my ( $mezery ) = @_;
```

```
5         if ( not opendir(ADR, ".") ) { return 0; }
6         my @soubory = readdir(ADR);
7         closedir(ADR);
8         foreach my $jmeno ( sort @soubory ) {
9             if ( $jmeno =~ /^\./ ) { next; }
10            if ( -d $jmeno ) {
11                print "$mezery$jmeno/\n";
12                chdir($jmeno);
13                zpracuj_adr($mezery." ");
14                chdir("..");
15            } else {
16                print "$mezery$jmeno\n";
17            }
18        }
19    }
```

Řešení 11.1

tři
chybové hlášení (*$ohash* je odkaz na asociativní pole, ne na obyčejné)
dva
nic (asociativní pole neobsahuje klíč „dva“)

Řešení 11.2 Celkem jednoduché řešení představuje použití dvou proměnných. Do jedné uložím index prvního a do druhé posledního prvku pole. Postupně vždy vyměním dvojici prvků a posunu indexy na následující/předchozí prvek, dokud se nesejdou.

obrat.pl

```
sub obrat {
    my ( $pole ) = @_;
    my ( $predni, $zadni ) = ( 0, @$pole-1 );
    while ( $predni < $zadni ) {
        ( $pole->[$predni], $pole->[$zadni] ) =
            ( $pole->[$zadni], $pole->[$predni] );
        $predni++; $zadni--;
    }
}
```

Řešení 11.3 Počet figurek na šachovnici získáte třeba takto:

sachy.pl

```
sub spocitej_figurky {
    my ( $sachovnice ) = @_;
    my $pocet = 0;
```

```
        foreach my $radek (0..7) {
            foreach my $sloupec (0..7) {
                if ( $sachovnice->[$radek]->[$sloupec] ) {
                    $pocet++;
                }
            }
        }
        return $pocet;
    }
```

V normálním případě bych pravděpodobně cykly **foreach** procházel hodnoty místo indexů, protože je s tím o chlup méně psaní. Chtěl jsem však předvést použití dvou indexů ve dvojrozměrném poli.

Řešení 12.1 Pro otočení pořadí řádků v souboru bohužel neexistuje lepší recept, než celý soubor načíst do pole a to pak projít v obráceném pořadí. Pokusil jsem se jej vylepšit alespoň tak, aby načetl vždy jen jeden soubor, ne všechny najednou. Má-li se program chovat skutečně jako *cat*, musím rozlišit tři případy:

1. Nebylo zadáno `-r` – pak stačí prostě opsat vstup.
2. Bylo zadáno `-r` a seznam souborů – pak musím tento seznam postupně procházet od konce a každý soubor otočit.
3. Bylo zadáno `-r`, ale žádné soubory – musím obrátit obsah standardního vstupu.

Program by mohl vypadat třeba takto:

perlcat.pl

```
use Getopt::Std;

getopts("r");
if ( not $opt_r ) {
# bez –r jen vše opíšeme
    while ( my $radek = <> ) { print $radek; }
} elsif ( @ARGV ) {
# máme zadány soubory
    foreach my $soubor ( reverse @ARGV ) {
        open(DATA, $soubor) or next;
        my @data = <DATA>;
        close(DATA);
        foreach my $radek ( reverse @data ) { print $radek; }
    }
} else {
# prázdné @ARGV – obracíme řádky stand. vstupu
```

```
    my @data = <>;
    foreach my $radek ( reverse @data ) { print $radek; }
}
```

Kdybych si chtěl ušetřit práci a příliš se neohlížel na spotřebu paměti (což je standardní postup některých softwarových firem), mohl bych jednoduše vypustit prostřední větev (**elsif** a jeho blok).

Řešení 12.2 Jedinou rafinovaností je ukládání do souboru. Rourou živím *gzip*, který své výsledky posílá do standardního výstupu. Součástí „jména souboru" ve funkci **open** však může být i přesměrování tohoto standardního výstupu ze spouštěného programu do kýženého souboru.

compress.pl

```
sub uloz_pole {
    my ( $jmenodat ) = @_;
    open( my $PACK, "| gzip – > $jmenodat" )
        or die "Data nelze uložit.\n";
    foreach my $prvek ( @pole ) { print $PACK "$prvek\n"; }
    close($PACK);
}

sub nacti_pole {
    my ( $jmenodat ) = @_;
    open( my $PACK, "gunzip –c $jmenodat |" )
        or die "Data nelze načíst.\n";
    @pole = ();
    while ( my $prvek = <$PACK> ) {
        chomp($prvek);
        push(@pole, $prvek);
    }
    close($PACK);
}
```

Řešení 13.1 Definice třídy *Zak* s požadovanými metodami by mohla vypadat následovně:

Zak.pm

```
package Zak;

use Moose;

has jmeno => ( is => 'ro', isa => 'Str' );
has prospech => ( is => 'rw', isa => 'Num', default => 0 );
```

```
    sub vypis {
        my $self = shift;
        printf "%-30s %0.2f\n", $self->jmeno, $self->prospech;
    }

    1;
```

a podprogram *pridej_zaka*:

```
    sub pridej_zaka {
        print "Příjmení a jméno: ";
        my $jmeno = <STDIN>; chomp( $jmeno );
        print "Průměr: ";
        my $prumer = <STDIN>; chomp( $prumer );
        my $zak = Zak->new( jmeno=>$jmeno, prumer=>$prumer );
        push( @_, $zak );
        return @_;
    }
```

S využitím objektově orientovaného programování se nabízí celou třídní agendu od základů předělat a reprezentovat také školní třídu pomocí objektu. Ten by nabízel metody jako *seznam_abecedne* či *vloz_zaka*. Mohl by vypadat třeba takto:

Trida.pm

```
package Trida;
# balík implementující školní třídu
# obsahuje odkaz na pole odkazů na objekty typu Zak
#
# metody:
# vloz_zaka(žák) přidá žáka do třídy (žák je odkaz na objekt typu Zak)
# hledej_zaka("jméno") vydá odkaz na žáka daného jména nebo undef
# vyrad_zaka(žák) vyřadí žáka (dostane odkaz) ze třídy
# seznam_abecedne() vypíše žáky ve třídě v abecedním pořadí
# seznam_prospech() vypíše žáky ve třídě podle prospěchu
# nacti("jméno") načte třídu z diskového souboru
# uloz("jméno") uloží třídu do souboru

use locale;
use Moose;
use Zak;

has zaci => ( is => 'rw', isa => 'ArrayRef', default => sub { [] } );
```

```
sub vloz_zaka {
    my $self = shift;
    my $zak = shift;
    push( @{$self->zaci}, $zak );
}

sub vyrad_zaka {
    my $self = shift;
    my $zak = shift;
    for ( my $i=0; $i<@{$self->zaci}; $i++ ) {
        if ( $zak == $self->zaci->[$i] ) {
            splice( @{$self->zaci}, $i, 1 );
        }
    }
}

sub hledej_zaka {
    my $self = shift;
    my $jmeno = shift;
    foreach my $zak ( @{$self->zaci} ) {
        if ( $zak->jmeno eq $jmeno ) {
            return $zak;
        }
    }
    return undef;
}

sub seznam_abecedne {
    my $self = shift;
    foreach my $zak ( sort srovnej_jmena @{$self->zaci} ) {
        $zak->vypis();
    }
}

sub srovnej_jmena {
    return ( $a->jmeno cmp $b->jmeno );
}

sub seznam_prospech {
    my $self = shift;
```

```
    foreach my $zak ( sort srovnej_prospech @{$self->zaci} ) {
        $zak->vypis();
    }
}

sub srovnej_prospech {
    return ( $a->prospech <=> $b->prospech );
}

sub nacti {
    my $self = shift;
    my $soubor = shift;
    $self->zaci( [] );

    my ( $radek, $zak, $jmeno, $prospech, $DATA );
    if ( open($DATA, $soubor) ) {
        while ( $radek = <$DATA> ) {
            chomp($radek);
            ($jmeno,$prospech) = split( /:/, $radek );
            $zak = Zak->new(jmeno=>$jmeno, prospech=>$prospech);
            $self->vloz_zaka($zak);
        }
        close($DATA);
        return 1;
    } else {
        print STDERR "Nelze otevřít soubor $soubor!\n";
        return 0;
    }
}

sub uloz {
    my $self = shift;
    my $soubor = shift;
    my $DATA;
    if ( open($DATA, ">$soubor") ) {
        foreach my $zak ( @{$self->zaci} ) {
            print $DATA $zak->jmeno, ":", $zak->prospech, "\n";
        }
        close($DATA);
        return 1;
    } else {
```

```
        print STDERR "Nemohu zapisovat do souboru $soubor!\n";
        return 0;
    }
}

1;
```

Situace se poněkud zkomplikovala, protože jako datovou položku potřebuji pole, resp. odkaz na pole – typ *ArrayRef*. Jako výchozí hodnotu ovšem modul *Moose* nepřipouští [], je nutno je zabalit do funkce, která odkaz na prázdné pole vrátí jako svůj výsledek. Proto je atribut *default* poněkud krkolomný.

Hlavní program pak zajistí zejména uživatelské rozhraní, ve kterém volá metody definovaných objektů:

objtrida.pl

```
use locale;
use lib ".";
use Zak;
use Trida;

my $trida = Trida->new();

while ( my $pokyn = dej_znak() ) {
    if ( $pokyn =~ /n/i ) { pridej_zaka(); }
    elsif ( $pokyn =~ /d/i ) { smaz_zaka(); }
    elsif ( $pokyn =~ /j/i ) { $trida->seznam_abecedne(); }
    elsif ( $pokyn =~ /p/i ) { $trida->seznam_prospech(); }
    elsif ( $pokyn =~ /r/i ) { nacti_tridu(); }
    elsif ( $pokyn =~ /s/i ) { uloz_tridu(); }
    elsif ( $pokyn =~ /q/i ) { last; }
    else { print "Neplatný příkaz!"; }
}

sub dej_znak {
    print "\nZadejte příkaz\n";
    print " n ... přidat žáka\n";
    print " d ... vymazat žáka\n";
    print " j ... seznam žáků abecedně\n";
    print " p ... seznam žáků podle prospěchu\n";
    print " r ... načíst z disku\n";
    print " s ... uložit na disk\n";
```

```
    print " q ... konec\n";
    print "> ";
    my $vstup = <STDIN>;
    return substr( $vstup, 0, 1 );
}

sub pridej_zaka {
    print "Příjmení a jméno: ";
    my $jmeno = <STDIN>; chomp($jmeno);
    print "Průměr: ";
    my $prumer = <STDIN>; chomp($prumer);
    my $zak = Zak->new(jmeno=>$jmeno, prospech=>$prumer);
    $trida->vloz_zaka($zak);
}

sub smaz_zaka {
    print "Příjmení a jméno: ";
    my $jmeno = <STDIN>; chomp($jmeno);
    if ( my $zak = $trida->hledej_zaka($jmeno) ) {
        $trida->vyrad_zaka($zak);
        print "Žák $jmeno byl vyřazen.\n";
    } else {
        print "Žák $jmeno není zařazen do třídy.\n";
    }
}

sub nacti_tridu {
    print "Jméno souboru: ";
    my $jmeno = <STDIN>; chomp($jmeno);
    $trida->nacti($jmeno);
}

sub uloz_tridu {
    print "Jméno souboru: ";
    my $jmeno = <STDIN>; chomp($jmeno);
    $trida->uloz($jmeno);
}
```

Řešení 14.1 Tělo funkce se podobá filtru, jen je o něco jednodušší. Nemusí sbírat do pole vyhovující hodnoty, stačí zastavit a ohlásit neúspěch, jakmile narazí na nevyhovující hodnotu (všimněte

si, že kritérium je negováno). Pokud po zpracování všech hodnot funkce stále pokračuje, znamená to, že všechny vyhověly a je na čase ohlásit úspěch.

vsechny.pl

```
sub vsechny {
    my $kriterium = shift;
    foreach my $hodnota ( @_ ) {
        if ( not $kriterium->($hodnota) ) {
            return 0;
        }
    }
    return 1;
}
```

Řešení 14.2 Jedná se o celkem přímočarý přepis uvedeného vzorce. Zde i s příkladem použití:

derivace.pl

```
use constant EPSILON => 0.001;

sub derivace {
    my $f = shift;
    my $x = shift;
    return ($f->($x+EPSILON) - $f->($x-EPSILON))
            / (2*EPSILON);
}

sub kvadrat { my $x = shift; return $x*$x; }
print derivace(\&kvadrat, 3), "\n";
```

Řešení 14.3 Vlastní tělo se proti minulému cvičení příliš nezměnilo, jen se parametr *$f* odstěhoval do generátoru:

genderivaci.pl

```
use constant EPSILON => 0.001;

sub derivuj {
    my $f = shift;
    return sub {
        my $x = shift;
        return ($f->($x+EPSILON) - $f->($x-EPSILON))
                / (2*EPSILON);
    }
}
```

18 Instalace Perlu a modulů

Tato kapitola vám poradí, jak si nainstalovat Perl vlastníma rukama, pokud jej váš operační systém neobsahuje. Také se zde dozvíte, kde a jak si opatřovat různé doplňky (čili moduly) a co s nimi dělat.

Základní pokladnicí Perlu je *Comprehensive Perl Archive Network (CPAN)*. Na adrese:

https://www.cpan.org/

najdete obludné množství modulů – v polovině roku 2018 jich bylo 175 tisíc. Vyznat se v nich není zcela snadné, jsou organizovány několika alternativními způsoby (tematicky, podle autorů,…). Naštěstí je k mání vyhledávání. Nenechte se zmást tím, že vyhledávač má svou vlastní doménu:

https://metacpan.org/

i poněkud odlišný vzhled. Nejedná se o žádný trucprojekt, ale o zcela oficiální nadstavbu vlastního CPAN. Kromě toho pokud použijete některý ze standardních webových vyhledávačů a necháte si najít třeba „Perl Moose“, první odkaz obvykle vede na (Meta)CPAN.

18.1 Instalace Perlu v Unixu

V Unixu či Linuxu je skoro jisté, že se Perl nainstaluje automaticky jako součást systému. Pokud by snad nebyl nainstalován, bude k mání v podobě balíčku ve standardním úložišti aplikací dané distribuce. Na adrese:

https://www.cpan.org/ports/binaries.html

najdete přehled, jaká verze Perlu je standardně distribuována v různých systémech. Pokud byste se z jakéhokoli důvodu rozhodli překládat si jej sami, zdrojové texty jsou k dispozici na adrese:

https://www.cpan.org/src/

Vyberte si distribuční soubor a uložte jej do vhodného adresáře. Rozbalte příkazem:

```
tar xzvf perl-X.Y.Z.tar.gz
```

Měl by se vytvořit podadresář *perl-X.Y.Z* (kde *X*, *Y* a *Z* závisí na aktuální verzi interpretu). Přejděte do něj a spusťte:

```
sh Configure -de
```

Volba `-d` je důležitá, protože přiměje konfigurační program, aby si na své otázky odpovídal sám podle nalezených informací a neobtěžoval s tím vás. Pokud spustíte `Configure` bez parametrů, budete zavaleni mračnem dotazů až po velikost nohy, otáčky ventilátoru vašeho počítače či počet vlasů, které jste si vyrvali od jeho spuštění. Vřele doporučuji se netrápit a použít `-d`. S ním se konfigurační skript neptal vůbec na nic, sám si všechno zjistil a nastavil.

Po konfiguraci následuje obvyklé:

```
make
```

které přeloží interpret. Dopadne-li vše dobře, což by mělo, můžete jej prověřit pomocí:

```
make test
```

I tentokrát snad vše dopadne bez problémů a budete moci přistoupit k:

```
make install
```

Tento krok musíte provádět jako správce systému (root), protože se instaluje do sdílených adresářů (implicitně */usr/local/…*). Vřele doporučuji, abyste instalaci Perlu zakončili konverzí hlavičkových souborů. Není sice nutná, ale řada modulů bez ní nepoběží. Zajistí ji dvojice příkazů:

```
cd /usr/include
h2ph *.h sys/*.ph
```

Program *h2ph* vytvoří perlivou verzi hlavičkových souborů a umístí ji do adresáře, kde ji Perl dokáže najít. Tím je instalace skončena a vše by mělo fungovat, jak má.

18.2 Instalace modulů v Unixu

Svět různých distribucí Unixů a Linuxů je velmi pestrý a těžko se dávají univerzálně platná doporučení. Populární a často používané moduly bývají k dispozici ve formě balíčků pro standardní instalační systém. Instalují se buď automaticky s Perlem, nebo na žádost. Doporučuji jednoduše zadat do webového vyhledávače „*»distribuce«* perl *»modul«*“ a obvykle se rychle dozvíte, jak jej instalovat.

Opačný pól proti modulům připraveným v balíčkovacím systému představuje překlad ze zdrojových kódů. Ten je k dispozici vždy, stačí stáhnout zdrojový kód modulu z CPAN, rozbalit a postupovat podle instrukcí v souboru *INSTALL* nebo *README*. Obvykle to bývá něco jako:

```
perl Makefile.PL
make
make test
make install
```

První krok vytvoří konfiguraci, další pak sestaví modul a vše potřebné, prověří funkčnost výsledku a nainstalují do systému. Závěrečný krok musí provádět správce systému (root). Po jeho dokončení je modul připraven k použití.

Pro ty z nás, kteří si potrpí na pohodlí, je určen modul *CPAN*. Jeho cílem je automatizovat instalaci modulů a zahrnout uživatele veškerým myslitelným pohodlím. Dá se říci, že *CPAN* je posledním modulem, který budete instalovat výše popsaným způsobem. Máte-li jej k dispozici, vyvolejte jeho interaktivní rozhraní pomocí:

```
perl -MCPAN -e shell
```

V některých distribucích je k dispozici i příkaz:

```
cpan
```

který dělá totéž. Při prvním spuštění proběhne nejprve konfigurace. Modul si najde vhodné programy pro překlad a instalaci, případně mu je můžete zadávat i ručně, pokud se zrovna nudíte. Jakmile skončí tato fáze (vše si uloží, takže příště se již nebude opakovat), obdržíte výzvu k zadání příkazu. Jelikož modul komunikuje s CPAN, musíte být při jeho používání připojeni k Internetu.

Nejzákladnějším příkazem je `h`, který poskytne stručnou nápovědu. Je ovšem stručná až příliš. Raději vaší pozornosti doporučuji manuálovou stránku *man CPAN*, kde se dozvíte mnohem více.

Dvě nejčastější činnosti, které budete s modulem *CPAN* provádět, jsou hledání vhodných modulů a jejich instalace. Začněme u ní, protože je zcela jednoduchá. Prostě napište:

```
install »modul«
```

a pak už se jen dívejte. *CPAN* obstará vše potřebné: zjistí, zda náhodou už nemáte instalovánu aktuální verzi. Pokud ne, přenese potřebné soubory na váš počítač, přeloží, otestuje a pokud vše dobře dopadne, nainstaluje. Pochopitelně musíte oplývat potřebnými právy. *CPAN* si poradí i se závislostmi a nainstaluje také všechny moduly, které instalovaný ke své činnosti potřebuje.

Základ vyhledávání tvoří pětice příkazů z tabulky 18.1. Použitý příkaz rozhoduje o tom, zda hledáte autora, modul či něco jiného. Pokud jako parametr zadáte běžný řetězec znaků, vyhoví mu

`a`	autor
`b`	balík (bundle)
`d`	distribuce
`m`	modul
`i`	cokoli z výše uvedených

Tabulka 18.1: Vyhledávací příkazy modulu *CPAN*

jen přesná shoda. Když však parametr uzavřete mezi dvě lomítka, interpretuje se jako regulární výraz. Hledání nerozlišuje malá písmena od velkých.

Pokud disponujete modulem *CPAN::WAIT* (ideální příležitost vyzkoušet `install CPAN::WAIT`), máte navíc k dispozici fulltextové vyhledávání pokrývající všechny dokumenty z autorských adresářů CPAN. Základním vyhledávacím příkazem je `wq »řetězec«`, který vám vydá seznam všech nálezů. Jsou označeny pořadovými čísly. Pomocí `wd »číslo«` si pak můžete zjistit podrobnější informace o daném nálezu a příkazem `wr »číslo«` zobrazit jeho *README*.

18.3 Instalace Perlu v MS Windows

MS Windows je jeden ze systémů, kde Perl není součástí standardní distribuce. Zatím. Kupodivu implementací Perlu pro Windows není nijak oslňující množství. O programátorskou přízeň se přetahují dvě alternativy.

ActivePerl firmy ActiveState Tool Corporation se dlouho těšil pověsti nejzdařilejší implementace Perlu pro MS Windows. Lze jej používat zdarma pro komerční[1] i nekomerční účely, takže instalaci nic nebrání. Distribuční soubor najdete na stránce:

https://www.activestate.com/activeperl

Má podobu samoinstalujícího se programu, takže stačí spustit získaný *.exe* soubor. Instalace probíhá obvyklým průvodcovským způsobem. Nechá si potvrdit licenci, dovolí vám vybrat si cílový adresář a instalované komponenty. Nabídne také, že přiřadí *ActivePerl* jako zpracovávající aplikaci pro příponu *.pl*.

1: Licence volné verze je poněkud šalamounská. Připouští komerční využití, nikoli však produkční využití, které je definováno jako cokoli, co přesahuje vývoj programů. Máte-li s *ActivePerlem* komerční záměry, prostudujte si ji a případně zakupte edici s širší licencí.

Poslední dobou se ovšem množí názory, že autoři si až příliš hledí svých komerčních zájmů a doporučení ohledně Perlu pro Windows se kloní spíše na stranu implementace *Strawberry Perl*, kterou lze získat z webu:

http://strawberryperl.com/

Instalace opět probíhá standardní sérií dialogů, až na to, že tentokrát nestahujete spustitelný soubor typu *exe*, ale instalační balíček *msi*. To je ovšem detail ryze technický, na postup instalace nemá žádný vliv.

Osobně se domnívám, že je celkem jedno, kterou variantu zvolíte. Obě poslouží k naprosté spokojenosti. *Strawberry Perl* je purističtější a více se drží kořenů Perlu. *ActivePerl* se snaží strohý základ alespoň částečně obalit uživatelsky přívětivější nadstavbou (viz třeba instalace modulů popsaná v následující části) a jemně vás vést k nákupu komerčních produktů.

Bez ohledu na to, kterou implementaci jste zvolili, po dokončení instalace můžete spouštět perlivé programy. Buď ručně v okně příkazového řádku tradičním:

```
perl »soubor«
```

nebo poklepáním na soubory s příponou *.pl* v grafickém rozhraní. Otevře se pro ně nové okno, v němž program pracuje. Bohužel se okamžitě po dokončení programu zase zavře a veškeré výstupy s sebou vezme do hrobu. Takže doporučuji při práci v MS Windows přidat vždy na konec programu příkaz:

```
<STDIN>
```

případně:

```
system( "pause")
```

který počká, dokud nestisknete **Enter**. Ovšem u cizích programů nebo při zhroucení po chybě je tato vlastnost krajně nemilá. Nepřišel jsem na to, jak se jí rozumně bránit. Ještě že Perl v MS Windows nepoužívám…

Zajímavou alternativu k tradičním implementacím Perlu mohou představovat balíčky unixových nástrojů pro MS Windows jako je *Cygwin* (*www.cygwin.org*) nebo *Git for Windows* (*git-scm.com*), v nichž bývá Perl přibalen jako jeden ze sady programů. Pokud některý z nich používáte, můžete být překvapeni, že Perl máte již k dispozici.

Pro jednoduché použití taková varianta postačí. Jestliže to s Perlem myslíte vážně a hodláte v něm vyvíjet programy, doporučuji dát přednost plnohodnotné instalaci jedné ze dvou výše uvedených variant.

18.4 Instalace modulů v MS Windows

Jak *ActivePerl*, tak *Strawberry Perl* obsahují příkaz *cpan*, kterým lze v textovém režimu vyhledávat a instaloval moduly. Stačí otevřít příkazový řádek, zadat:

```
cpan
```

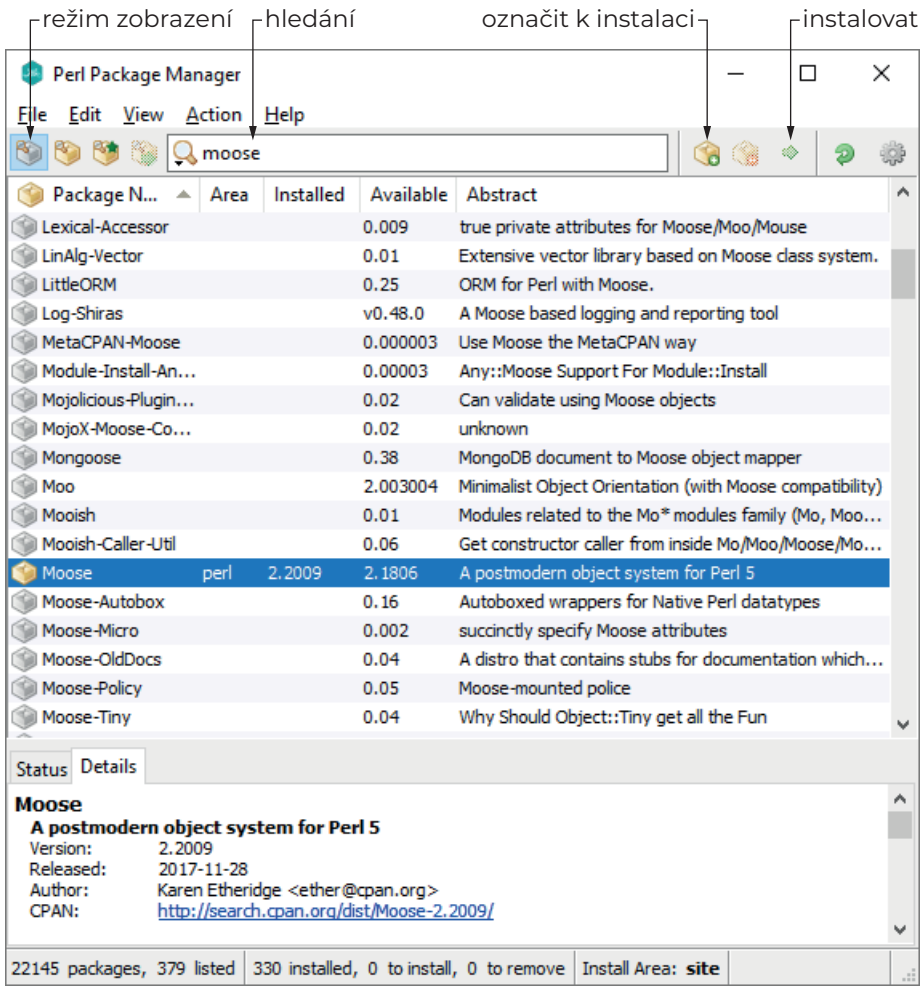
a dále postupovat, jak jsem popsal na straně 277. Případně lze rovnou instalovat konkrétní modul zadáním:

```
cpan »modul«
```

ActivePerl navíc nabízí grafický nástroj pro práci s balíky. Najdete jej normálně v nabídce **Start** pod názvem **Perl Package Manager**. Jeho podobu vidíte na obrázku 18.1.

Horní části okna vévodí pole pro vyhledávání. Nalevo od něj najdete ikony pro přepínání zobrazeného seznamu – umí všechny moduly, instalované moduly, moduly s novější verzí a moduly vybrané k instalaci. Podobné služby nabídne i menu **View**.

Instalaci modulu nařídíte klepnutím na ikonu se zeleným plus napravo od vyhledávacího pole nebo volbou **Install** z kontextového menu, které se objeví po klepnutí pravým tlačítkem na jméno modulu. Neinstaluje se hned, program si jen poznamená, že má něco provést. Ke spuštění naplánovaných akcí stiskněte tlačítko se zelenou šipkou napravo od vyhledávacího řádku nebo vyberte z menu **File / Run Marked Actions**. Jestliže zapomenete a program ukončíte, nijak vás neupozorní a s ledovým klidem skončí, aniž by cokoli nainstaloval. Takže vždy ve střehu.



Obrázek 18.1: Perl Package Manager *ActivePerlu*

Literatura

[1] Larry Wall, Tom Christiansen, Randal L. Schwartz: **Programování v jazyce Perl.**
Computer Press, 1997, ISBN 80-85896-95-8

Referenční příručka Perlu. V podstatě se jedná o oficiální definici jazyka. Kniha se hodí spíše k vyhledávání konkrétních informací, než k souvislému čtení od začátku do konce. Pokud to s Perlem myslíte vážně, určitě by ve vaší knihovničce neměla chybět. Navíc se prodává za velice rozumnou cenu. Anglický originál *Programming Perl* vyšel u O'Reilly v roce 2012 již ve čtvrtém vydání.

[2] Tom Phoenix, Randal L. Schwartz, brian d foy: **Learning Perl.**
7. vydání, O'Reilly, 2016, ISBN 978-1-4919-5432-4

Výborná učebnice Perlu pro začátečníky. Z této knihy jsem se naučil základy Perlu a moc se mi líbila. Kdyby existoval její český překlad, pravděpodobně by *Perl pro zelenáče* nikdy nevznikl.

[3] Tom Christiansen, Nathan Torkington: **Perl Cookbook.**
O'Reilly, 1998, ISBN 1-56592-243-3

Vynikající a mimořádně užitečná kniha. Jedná se o sbírku receptů v Perlu – kolekci problémů, k nimž je popsáno a zdůvodněno řešení (zpravidla v několika alternativách). Ve valné většině případů zde pro své problémy najdete buď hotové řešení nebo cennou inspiraci.

[4] Nathan Patwardhan, Ellen Siever, Stephen Spainhour: **Perl in a Nutshell.**
2. vydání, O'Reilly, 2009, ISBN 0-596-00241-6

Perl v kostce. Stručná referenční příručka popisující vlastní jazyk, standardní moduly a některé nestandardní (např. pro grafické uživatelské rozhraní, síťovou komunikaci či vytváření CGI programů). Ve srovnání s [1] je stručnější, ale má širší záběr. Je užitečné mít ji při programování po ruce.

[5] Sriram Srinivasan: **Programování v Perlu pro pokročilé.**
Computer Press, 1998, ISBN 80–7226–079–0

Název nepřehání! Dosti drsná záležitost popisující složitější partie Perlu. Zahrnuje odkazy, vytváření dynamických datových struktur či popis vnitřního života interpretu. Z mírumilovnějších končin pochází popis síťové komunikace a grafického uživatelského rozhraní. Neočekávejte, že vše pochopíte již při prvním čtení. Originál vydalo nakladatelství O'Reilly v roce 1997 pod názvem *Advanced Perl Programming*.

[6] Jeffrey E. F. Friedl: **Mastering Regular Expressions.**
O'Reilly, 1997, ISBN 1–56592–257–3

Regulární výrazy pro pokročilé. Zabývá se velice podrobně zvyšováním efektivity, konstruováním speciálních regulárních výrazů, vnitřními mechanismy, srovnává různé odrůdy a podobně. Udělám-li analogii s motoristickými příručkami, tahle neřeší otázku „kde najít plynový pedál", ale radí pilotům F1 „jak efektivně najíždět do zatáčky v rychlosti 250 km/h".

[7] Jerry Peek, Tim O'Reilly, Mike Loukides: **Unix Power Tools.**
2. vydání, O'Reilly, 1997, ISBN 1–56592–260–3

Snad nejužitečnější knížka o Unixu, kterou jsem kdy četl. Sbírka tipů a návodů, které proklatě prohloubí vaše znalosti.

[8] Bruce Hyslop, Elizabeth Castro: **HTML5 a CSS3**
Computer Press, 2012, ISBN 978-80-251-3733-8

Dost zevrubný popis dvou klíčových jazyků pro vytváření webových stránek – aktuálních verzí HTML pro obsah a CSS pro vzhled.

[9] Jiří Kosek: **Aplikace na webu.**
Computerworld, 1998,
http://www.kosek.cz/clanky/iweb/index.html

Soubor článků původně publikovaných v časopise Computerworld, které popisují technologie a metody pro tvorbu webových aplikací. Text je letitý, současné webové aplikace už jsou mnohem dál. Nicméně pro pochopení základů CGI, pro něž se Perl dosud používá, poslouží dobře.

Rejstřík

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 978-80-88168-35-5 (tištěná verze)
ISBN 978-80-88168-36-2 (ve formátu EPUB)
ISBN 978-80-88168-37-9 (ve formátu MOBI)
ISBN 978-80-88168-38-6 (ve formátu PDF)

CZ.NIC | SPRÁVCE DOMÉNY CZ

Edice CZ.NIC

O knize Programovací jazyk Perl je silný, elegantní a trochu nebezpečný. Cílem knihy je uvést čtenáře do jeho zajímavého světa. Začíná zlehka od základních konstrukcí a postupně se přes nejcennější klenoty, jako jsou regulární výrazy či asociativní pole, propracuje až k obecnějším tématům. Perl totiž má co nabídnout i pro objektově orientované a funkcionální programování, práci s databázemi nebo aplikace pro web. Text je doprovázen četnými příklady a cvičeními, na kterých si může čtenář prověřit své schopnosti.

O autorovi Pavel Satrapa vyučuje na Technické univerzitě v Liberci. Specializuje se zejména na počítačové sítě a programování a v těchto oblastech i často publikuje. Toto vydání Perlu pro zelenáče je patnáctým členem řady knih, vinoucí se až do roku 1996 k World-Wide Web pro čtenáře, autory a misionáře, první české knize o webu. Jeho články najdete zejména na serverech Root.cz a Lupa.cz.

O edici Edice CZ.NIC je jednou z osvětových aktivit správce české národní domény. Ediční program je zaměřen na vydávání odborných, ale i populárně naučných publikací spojených s Internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz.

knihy.nic.cz

ISBN 978-80-88168-38-6

9 788088 168386 >