FRANTIŠEK DAŘENA

myslíme v jazyku PERL

knihovna programátora

- Stavba, charakter a prvky jazyka
- Deklarace a definice podprogramu, práce se soubory a adresáři
- Složitější datové struktury, balíky a moduly
- Objektově orientované programování
- CGI programování a práce s databází
- Ladění skriptů

GRADA

Upozornění pro čtenáře a uživatele této knihy

GRADA®

Obsah

KAPITOLA 1.

ÚVOD

- Stručná historie jazyka
- Základní vlastnosti Perlu
- Zdroje informací použitelných pro programování v Perlu
- Instalace Perlu a modulů
- Typografické konvence knihy

Autorem jazyka Perl je svého času systémový programátor a příležitostný lingvista *Larry Wall* (jeho webová stránka má adresu *www.wall.org/~larry*). Ten navrhl jazyk původně pro svoji potřebu již v roce 1986. Perl měl Larrymu nahradit v té době nepostačující prostředky pro zpracování textu a dálkové řízení počítačů — existující programové vybavení nebylo schopné splnit všechny požadavky a tak vznikl místo nového programu nový programovací jazyk. Jako jméno bylo po dlouhém zvažování vybráno slovo Perl. Je to zkratka z anglických slov Practical Extraction and Report Language nebo také Pathologicaly Eclectic Rubbish Lister.*

Uvolnění interpretu jazyka pro veřejnost v roce 1987 se setkalo s nečekaným ohlasem. Tato odezva vedla Larryho a jeho spolupracovníky k dalšímu obohacování jazyka — z nástroje pro zpracování textů se vyvinul „skutečný programovací jazyk" s ladícími nástroji (debuggery), kompilátory, rozsáhlou sítí knihoven, dokumentace, podpůrných programů atd. V současnosti se vývoji a rozšiřování jazyka věnuje kolektiv lidí, kteří si říkají Perl Porters, a Larry Wall stojí v jeho čele.

Původně byl Perl navržen pro platformu s operačním systémem UNIX, ale v současné době běží také na jiných systémech, jako je např. MS Windows, VMS, OS/2, Macintosh, MS DOS, Atari, Novell aj.† Patří k nejpřenositelnějším jazykům, které v současnosti existují, a dnes patří také ke všeobecně uznávanému standardu pro vyvíjení webových aplikací.

Programátoři jazyka C ocení poměrně velkou podobnost s tímto jazykem, prvky svých programovacích jazyků zde naleznou také programátoři v jiných jazycích nebo tvůrci shell skriptů. Funkčnost konstrukcí v Perlu a ostatních programovacích jazycích je většinou stejná nebo velmi podobná.‡

V současnosti je rozšířená verze Perlu řady 5 (v červenci roku 2004 byla uvolněna verze 5.8.5.§), která umožňuje mj. objektově orientované programování, programování ve více souběžných vláknech či rozšiřování jazyka formou samostatných modulů. Již několik let probíhá paralelně s vývojem Perlu řady 5 nový Perl verze 6. Ten není pouhým přepsáním Perlu 5, dojde ke změně některých přístupů, ale především se zde uplatňuje nová filosofie — jsou od sebe odděleny parser, kompilátor a běh programu. Bude tak možné, aby spolupracovaly různé programovací jazyky. Součástí strategie tohoto nového jazyka je platformově nezávislý interpret, známý pod jménem Parrot.

V mnoha směrech je Perl velmi jednoduchý jazyk a není třeba znát mnoho, aby bylo možné psát různé i poměrně obsáhlé programy s mnoha funkcemi. Tento fakt však nevylučuje, že je možné ho využít i pro psaní velmi složitých a velmi rozsáhlých

*Původně se jazyk jmenoval Gloria po Larryho manželce, ale Larry od tohoto pojmenování ustoupil. Dalším jménem bylo slovo Pearl, ovšem i tento pokus byl zamítnut, protože jazyk tohoto jména již existoval.

†Úplný seznam operačních systémů, pro které je Perl dostupný, je možné získat na webové adrese *http://www.cpan.org/ports/index.html*.

‡Existuje však několik rozdílů, na které je třeba si dávat pozor. Jejich seznam, roztříděný podle podobnosti k jiným programovacím jazykům, se nachází v manuálové stránce *perltrap*.

§Při psaní této knihy bylo pracováno s verzemi Perlu 5.8 a vyššími (konkrétně verze 5.8.0 pro sun4-solaris-64int a 5.8.4 pro MSWin32-x86-multi-thread) a i pro verzi 5.8.5 by měly být veškeré příklady platné.

aplikací, podporujících práci s grafickým uživatelským rozhraním, práci s databází či síťovou komunikaci. Je to také jazyk vhodný pro systémové administrátory (umožňuje jednoduše a efektivně provádět úkoly zahrnující zpracování textu, přístup k databázím a síťovou komunikaci) a díky dostupnosti na mnoha platformách je vhodný i pro správu víceplatformového systému.

Velmi silným nástrojem, který není běžnou součástí ostatních programovacích jazyků, jsou regulární výrazy. Ty se používají např. v jednoúčelových nástrojích pro zpracování textu, jakými je například program *awk* či v různých textových editorech (například nejslavnější UNIXový editor *vi*). Perlové regulární výrazy ve spojení s konstrukcemi nejznámějších programovacích jazyků jsou však daleko za možnostmi těchto nástrojů. Další zajímavostí Perlu jsou asociativní pole (běžně nazývané hash), což jsou mimořádně výkonné struktury, v nichž se k příslušným hodnotám přistupuje dle programátorem zvoleného řetězcového klíče.

Perl má samozřejmě i své stinné stránky. Především v tomto jazyce neexistuje prakticky žádná pevně definovaná syntaxe, proměnné se nemusí deklarovat, nelze si definovat vlastní datové typy (jazyk je netypovaný), neexistuje typová kontrola. Není to tedy ani vhodný jazyk, jehož by bylo možné doporučit k výuce programování.

Perl je obvykle implementován pomocí interpretu, pro spuštění programu v Perlu tedy potřebujete přítomnost interpretu jazyka Perl. Při interpretaci dochází nejprve k rychlému překladu zdrojového kódu do paměti interpretu a teprve potom k následnému provedení kódu. Tento způsob interpretování umožňuje dosáhnout vysoké prováděcí rychlosti při zachování běžných výhod interpretovaných jazyků — rychlý vývoj, rychlé změny zdrojového kódu, okamžitě patrný výsledek. Znamená to však, že je třeba mít nainstalován další prostředek, který program vykoná — příslušný interpret. Při vlastní předkompilaci do paměti se provádí kontrola na syntaktické chyby — a programátor může odhalit potenciální problém i v těch částech programu, které interpret obvykle nebude provádět. I tento přístup zrychluje hladký průběh vývoje, což je současně největší požadavek na moderní programovací nástroje. V současné době existuje i překladač Perlu, který vytvoří binární program nevyžadující interpret.

Distribuce Perlu obsahuje velké množství dokumentace ve formě tzv. manuálových stránek. V operačních systémech třídy UNIX je prohlížíme nejčastěji příkazem *man*, ve Windows je dokumentace dostupná ve formě HTML stránek. Základní manuálovou stránkou je stránka s názvem *perl*, která obsahuje tématicky rozčleněný seznam dalších manuálových stránek. V začátcích programování v Perlu nám může pomoci manuálová stránka *perlcheat*, která obsahuje velmi stručný popis nejdůležitějších konstrukcí jazyka (práce s odkazy, metasymboly regulárních výrazů, speciální proměnné apod.).

Samotný Perl a většina programového vybavení je zcela bezplatná.* Lze je nalézt na celosvětově sdílené počítačové síti s názvem *Comprehensive Perl Archive Network* (CPAN), jejíž základní stránkou je *www.cpan.org*. V naší zemi se nachází několik zrcadel sítě CPAN, můžeme zmínit například server MZLU v Brně *ftp.mendelu.cz/perl*

*Perl podléhá kombinaci Artistic License a General Public Licence — více informací lze nalézt v manuálových stránkách *perlartistic* a *perlgpl*.

nebo server Masarykovy univerzity *ftp.fi.muni.cz/pub/CPAN*. Na této síti naleznete nejen interpret Perlu pro většinu běžných platforem, ale také množství doplňkového software v podobě knihoven, demoprogramů, dokumentace apod.

Pro platformu Windows existuje i polokomerční verze jazyka Perl s názvem ActiveState Perl, jejíž základní verze je zdarma a je dostupná na serveru *www.activestate.com*.

1.1 Motto

Motto jazyka Perl, které vystihuje jeho podstatu, zní:

„Existuje více způsobů, jak něčeho dosáhnout."
(anglicky „There's more than one way to do it.")

Perl zároveň podporuje tři základní vlastnosti programátora, kterými jsou*:

- lenost,
- netrpělivost,
- přílišné sebevědomí.

1.2 Informační zdroje

Úvodní branou ke zdrojům Perlu může být server *www.perl.com*. Tento web je udržován Tomem Christiansenem, jednou z hlavních osobností spojovanou s Perlem, a nakladatelstvím O'Reilly & Associates, které vydalo velké množství literatury týkající se jazyka Perl. Jiným zdrojem je web *www.perl.org* udržovaný organizací The Perl Foundation. Nacházejí se zde informace týkající se jazyka Perl, jeho historie a současného vývoje, událostí vztahující se k jazyku či odkazy na dokumentaci, knihy a články.

Jazyk Perl má i svůj vlastní časopis. Jmenuje se The Perl Journal, je vydáván Jonem Orwantem (jeden z autorů knihy Programming Perl), vychází čtvrtletně a bližší informace k němu je možné nalézt na adrese *www.tpj.com*. Každoročně je také pořádáno několik mezinárodních konferencí† zaměření na vývoj či použití jazyka.

Dokumentace k Perlu, společně s odkazy na další zdroje, se nachází na adrese *www.perldoc.com*.

K problematice Perlu vyšla rovněž řada knih. Asi nejznámější je kniha *Programming Perl* od autorů Larry Walla, Toma Christiansena a Jona Orwanta (dnes existuje

*Podle knihy *Programming Perl*.
†Viz například *http://www.yapc.org/*

již její třetí vydání). Objevila poprvé ve verzi 4 a jedná se v podstatě o komentovanou verzi manuálových stránek. Velbloud, zobrazený na titulní stránce knihy, se stal symbolem jazyka Perl (v anglickém jazyce se o této knize hovoří jako o „camel book“, česky „velbloudí kniha“).

Perl Golf

Již několik let probíhá na internetu zajímavá soutěž s názvem *perl golf*. Spočívá v tom, že soutěžící mají zapsat v Perlu daný algoritmus tak, aby zabíral co nejméně znaků. Výsledky jednotlivých kol soutěže se sčítají a hráči tak mají i svůj žebříček.

Některé algoritmy jsou velmi prosté, jiné jsou poněkud obtížnější. Mezi mnoho úloh, které zde byly řešeny, patří například vypsání určitých znaků ASCII tabulky na řádky po 16 znacích, algoritmy z různých druhů her, křížovky, kreslení různých obrazců, nalezení nejkratší cesty v grafu, načtení či vypsání digitální číslice apod.

Po každém kole jsou výsledky všech soutěžících zveřejněny a mohou tak být dobrou inspirací nejen pro kola další, ale je v nich možné nalézt i některé ze zajímavostí Perlu. Pro začínající, ale mnohdy i pokročilé programátory, jsou však tato řešení nebo jejich části naprosto nesrozumitelné. Je však velkou výzvou a i zábavou, porovnat své znalosti s těmi nejlepšími, jako jsou Ton Hospel, MTV Europe, Rick Klement, Jasper McCrea, Daniel Tuijnman a další.

Soutež původně běžela na adrese *perlgolf.sourceforge.net*, ale tamní aktivity již před delší dobou utichly. Veškeré dění se nyní odehrává na stránkách Terje Kristensena, jejichž adresa je *terje.perlgolf.org* (soutěži se říká Minigolf). V poslední době se podobná soutěž odehrává i na polských atránkách *www.kernelpanic.pl*.

1.3 Instalace Perlu

Pokud máme na počítači operační systém třídy UNIX, je možné, že je Perl jeho standardní součástí. Pokud tomu tak není, je třeba zdrojové kódy nejprve stáhnout (např. ze sítě CPAN). Celý Perl se nachází jako jeden archív, který je nejprve třeba odpovídajícím způsobem rozbalit (záleží na metodě, která byla pro vytvoření archívu použita). Potom je dobré přečíst si soubory *README* a *INSTALL*, kde se nacházejí instrukce, kterých je třeba se držet. Typicky spustíme program *Configure*, který zjistí informace o systému a vytváří soubor *Config.pm*, a dále spustíme program *make*. Je vhodné zkontrolovat instalaci příkazem *make test* a nakonec spustit *make install* pro zkopírování souborů na správné místo.

Pokud se rozhodneme nainstalovat Perl na operačním systému Windows, můžeme stáhnout instalační archív ze stránek společnosti ActiveState. Ten se nachází buď ve formě instalátoru Windows (soubor s příponou *.msi*) nebo pouze ve formě umožňující instalaci bez možnosti odinstalování. Při instalaci se kromě vlastního ko-

pírování Perlu provede asociace souborů s příponou *.pl* a také nastavení některých proměnných prostředí.

Ve všech dostupných operačních systémech můžeme Perl v podstatě nainstalovat ze zdrojových kódů,* pouze se musíme držet instalačních pokynů pro konkrétní operační systém (např. pro Windows se nacházejí v souboru *README.win32*).

Pokud bychom chtěli ze sítě CPAN nainstalovat modul, je opět třeba brát v úvahu operační systém, na kterém Perl běží. Instrukce, kterých bychom se měli držet, je možné nalézt na adrese *http://www.cpan.org/modules/INSTALL.html*. Můžeme zmínit, že jakmile jednou máme nainstalovaný standardní modul `CPAN`, můžeme s jeho pomocí instalovat moduly jednodušším způsobem. Ve Windows si můžeme vzít na pomoc program *ppm* (Programmer's Package Manager, dříve Perl Package Manager).

1.4 Typografické konvence této knihy

Pro odlišení textu různého obsahu se v této knize používají následující typografické konvence:

- Souvislá část zdrojového kódu programu je vysázena jako samostatný odstavec `neproporcionálním písmem`.

```
print 'Zadejte dvě čísla:';
my $x = <STDIN>;
my $y = <STDIN>;
print 'Součet je ', $x+$y;
```

- Části kódu programu, které jsou součástí běžného textu, jsou rovněž vysázené `neproporcionálním písmem`.

  Proměnné, které mohou obsahovat několik hodnot, se nazývají pole. Jméno takové proměnné začíná znakem `@` (např. `@hodnoty`) a k jednotlivým hodnotám se přistupuje prostřednictvím číselných indexů. Ty píšeme mezi dvojici hranatých závorek, před jménem proměnné pak píšeme znak `$` (např. `$hodnoty[1]`).

- Pokud se v textu objeví nějaké důležité slovo či pojem, bude na to upozorněno *skloněným písmem*.

  Velkou část programu v Perlu tvoří *výrazy*. Výraz je posloupnost *operandů* a *operátorů*, a má po svém vyhodnocení vždy určitou hodnotu (pokud ne, tato hodnota se přímo nazývá *nedefinovaná hodnota*).

- V textu se někdy odvoláváme na externí příkazy, systémová volání či soubory. Jejich jména jsou vysázena *skloněným bezpatkovým písmem*.

*Dostupné na adrese *http://www.cpan.org/src/README.html*.

Funkce uzavře soubor */etc/passwd* (k tomu je použito systémové volání *endpwent*). V případě úspěchu je vrácena pravdivá hodnota.

- Pokud se v textu odkazujeme na jinou kapitolu nebo na její část, je použito *písmo skloněné*.

  Podrobně je práce se seznamovými proměnnými a seznamovými hodnotami popsána v kapitole *4. Datové typy, proměnné*, v částech *Pole a seznamy* a *Hashe (asociativní pole)*.

- Adresy dokumentů na Internetu jsou vysázeny *širším skloněným písmem*.

  Seznam operačních systémů, pro které je Perl dostupný, je možné získat na webové adrese *http://www.cpan.org/ports/index.html*.

KAPITOLA 2.

STAVBA JAZYKA

- Charakter jazyka
- Struktura programu v Perlu
- Stručný přehled nejdůležitejších prvků jazyka

V následující kapitole se seznámíme s prvky jazyka Perl, s jeho základními konstrukcemi a způsobem zápisu perlového programu. Všechno je samozřejmě mnohem detailněji zmíněno v následujících částech knihy, účelem této kapitoly je pouze to, aby čtenář získal o jazyku alespoň stručný přehled.

2.1 Charakter jazyka

Perl byl navržen Larry Wallem, příležitostným lingvistou, proto má určitou podobnost s přirozeným jazykem (angličtinou). Zápis programu v Perlu je proto většinou poměrně snadno čitelný i pro obyčejného člověka, který se programováním nezabývá.

Tak jako v přirozeném jazyce existují slovní druhy (podstatná jména, přídavná jména, slovesa), stejně tak i v Perlu existují jejich „ekvivalenty“ — podstatná jména reprezentují proměnné, přídavná jména operátory vymezující platnost proměnných, slovesa tvoří volání podprogramů apod. Podobně, jako lze v přirozeném jazyce jednu skutečnost vyjádřit několika způsoby a jedno sdělení může mít různý význam v závislosti na kontextu, v jakém bylo proneseno, i v Perlu je tomu rovněž tak. Můžeme říci: „Jestliže dělitel není nula, proveď podíl.“ Můžeme ale také říci: „Proveď dělení, ale pouze v případě, že je dělitel různý od nuly.“ Výsledek je pokaždé stejný, ale sdělení bylo řečeno pokaždé trochu jinak. Můžeme také říci: „Načti data ze vstupu,“ ale jednou nás zajímá jeden řádek, jindy chceme vše. Neexistuje striktně daná syntaxe, je možné používat různá zjednodušení a zkracování zápisu. Perl se vždy snaží porozumět tomu, co jsme napsali.* Nic ale nebrání tomu, abychom dodržovali určitá pravidla, která učiní kód snadno čitelný a podobný jazykům s přesně danou strukturou a syntaxí.

Podobně jako v přirozeném jazyce je možné v Perlu věci vyjadřovat přímo, bez zbytečných přívlastků, které zamlžují původní sdělení. Podobně jako v přirozeném jazyce lze věci vyjádřit velmi komplikovaně tak, že je téměř nemožné zápis programu rozluštit.

Existuje řada mechanismů, které se provádějí implicitně či jsou předpokládány nebo očekávány. Není ale chybou, pokud tyto věci sdělíme Perlu explicitně. Je to spíše otázkou osobního vkusu a lenosti. Není například nutné provádět deklarace proměnných, nemusíme nutně psát při každém volání podprogramu závorky okolo argumentů, v řadě míst se implicitně používají předdefinované proměnné. V některých jiných programovacích jazycích je naopak dodržování podobných pravidel nezbytně nutné pro syntaktickou správnost programu (např. jazyk Pascal, C). V Perlu nikoliv.

Perl může být jazykem jednoduchým, snadným pro vyjadřování. Není třeba znát mnoho, aby bylo možné psát jednoduché i méně složité programy. Při dalším postupném rozšiřování znalostí je možné pouštět se do programů složitých, které

*Pro tuto vlastnost existuje zkratka DWIM — „Do What I Mean“.

pracují s databází, využívají síťovou komunikaci či komunikují s uživatelem pomocí grafického uživatelského rozhraní.

Klasickým příkladem jednoduchosti a přímočarosti jazyka je následující ukázka. Když chcete napsat program, který na obrazovku vytiskne pozdrav „Ahoj“, stačí, aby celý program obsahoval jeden řádek.*

```
print "Ahoj";
```

Úplně stejným způsobem by se toto vyjádřilo v běžném jazyce. Není tedy třeba provádět různé deklarace, říkat, kde začíná a končí tělo programu.

2.2 Prvky jazyka

Pro vytváření programu v Perlu není třeba žádný speciální prostředek (jako např. vývojové prostředí pro Turbo Pascal nebo C). Program v Perlu je obyčejný text obsahující písmena, číslice, bílé znaky a některé speciální symboly. Je možné ho tedy vytvářet v libovolném textovém editoru. V systémech UNIX tedy například editorem *vi*, *pico* či *joe*, ve Windows editorem *Notepad* (Poznámkový blok), *WordPad* či *TextPad*. Je důležité, aby se jednalo o čistý text, který neobsahuje žádné řídící znaky příslušného editoru.

Celý text programu v Perlu se skládá z určitých částí, které jsou překladačem určitým způsobem chápány. V následující části kapitoly se s nimi postupně seznámíme.

Před tím ale zmíníme dva užitečné režimy, v nichž může interpret Perlu pracovat. Jedním z nich je *režim varování*. V tomto režimu, který zapneme přepínačem `-w`, nastavením hodnoty proměnné `$^W` či zavedením pragmatu `warnings`, budeme upozorněni (výstupem na standardní chybový výstup) na podezřelé operace, jakými jsou použití některého jména pouze jedenkrát, tisk nedefinované hodnoty, lichý počet prvků při přiřazení do hashe, práce s neotevřeným ovladačem apod.

Dalším režimem je tzv. *striktní režim*. Ten způsobí chybu v případě, že se pokusíme provádět některé nebezpečné operace — práce se jménem proměnné, u které není zřejmé, jaký je její rozsah platnosti, použití holých slov a práce se symbolickými odkazy. Striktní režim je zapnut zavedením pragmatického modulu `strict` a více prostoru je mu věnováno v kapitole *19. Pragmatické moduly*.

Je dobré oba tyto režimy používat, protože tak odhalíme řadu chyb ještě před tím, než program vůbec spustíme.

*Toto je jeden ze způsobu zápisu tohoto příkazu. Text „Ahoj“ může být uzavřený do uvozovek, apostrofů či jako argument nějakého z řetězcových operátorů, může a nemusí být uzavřen do kulatých závorek, na konci příkazu v tomto případě může a nemusí být středník. Proč tomu tak je a jaký je rozdíl mezi nastíněnými variantami se dozvíme v dalším textu.

Výrazy a příkazy

Velkou část programu v Perlu tvoří *výrazy*. Výraz je posloupnost *operandů* a *operátorů*, a má po svém vyhodnocení vždy určitou hodnotu (pokud ne, nazývá se tato hodnota přímo *nedefinovaná hodnota*). Podle typů operátorů — jejich arity, priority a asociativity — jsou s operandy prováděny různé operace, které ovlivňují výsledek výrazu. Operandy jsou obecně opět výrazy, případě jejich výsledné hodnoty. Nejjednodušším prvkem výrazu je tzv. *term*, což je např. číslo, řetězec či volání podprogramu. Podrobně se termům, operátorům a jejich vyhodnocováním věnuje kapitola *6. Operátory*.

Je-li za výrazem znak středník, stává se z něj *příkaz*. Protože každý příkaz je i výrazem, má vždy nějakou hodnotu bez ohledu na to, zda je tato hodnota potřebná či nikoliv.

Program v Perlu je tedy posloupnost příkazů — výrazů oddělených středníkem. Vyhodnocují se sekvenčně, případně opakovaně či podmíněně v závislosti na použitých příkazech cyklu, větvení či skoku. Některé výrazy mají svůj vedlejší efekt. Již zmíněný příklad `print "Ahoj"` je výrazem, jehož hodnota je `1`.* Jedná se o návratovou hodnotu funkce `print`. Vedlejším efektem tohoto výrazu je to, že se na standardní výstup vytiskne slovo „Ahoj“.

Blok

Příkazy mohou být spojeny do bloku. *Blok* je tedy posloupnost příkazů uzavřených mezi dvojici *složených závorek*. Poslední příkaz v bloku nemusí být ukončen středníkem. Často je to však doporučováno, aby nedošlo k chybě nebo k neočekávaným výsledkům v případě, že bychom za tento poslední příkaz chtěli umístit nějaký další.

Je-li někde očekáván blok, vždy je tím rozuměna posloupnost příkazů uvnitř složených závorek. Je tomu tak i v případě, že by tento blok obsahoval pouze jeden jediný příkaz. Tato poznámka se týká například příkazů větvení či cyklů, kdy se v popisu syntaxe příkazu nachází blok. Například jazyk Pascal nebo C na tomto místě vyžaduje příkaz — ten může být buď jednoduchý, nebo složený s ohraničovacími symboly `begin` a `end` nebo `{` a `}`. Noví programátoři v Perlu zde často chybují.

Blok může mít i své jméno. Jméno se mu může přidělit prostřednictvím tzv. návěští. *Návěští* je řetězec následovaný dvojtečkou. Používá se pro řízení běhu programu pomocí skoků či funkcí pro řízení běhu cyklů. Není nutné, aby návěští pojmenovávalo blok, může se vyskytovat i jinde. Konvencí je, že se jeho jméno píše velkými písmeny, aby nedošlo k záměně s rezervovaných slovem. Více se blokům a návěštím věnuje kapitola *5. Příkazy*.

Blok má i další vlastnosti, se kterými se seznámíme v následujících kapitolách — koncem bloku například končí platnost balíku či lexikálně vymezených proměnných.

*Za předpokladu, že se zadaný text úspěšně vytiskne do zvoleného výstupního souboru (implicitně na obrazovku).

Složené závorky neslouží pouze pro ohraničení bloku, používají se i pro zadávání hashových klíčů, pro vytváření odkazů, někdy i při zápisu jména proměnné nebo jako ohraničovací symbol argumentů některých operátorů. To, zda se jedná o blok nebo jiný způsob užití složených závorek, se pozná podle kontextu, kde jsou příslušné symboly použity. Někdy je třeba kompilátoru „pomoci“, například použitím unárního + či změnou pořadí jednotlivých částí výrazu, kde jsou složené závorky použity. Příklady budou uvedeny dále v kapitole *6. Operátory*.

Operátory

Operátory v Perlu můžeme chápat jako zvláštní podprogramy, které pracují se svými operandy a vracejí určitou hodnotu. Rozdíl mezi podprogramy a operátory je ve způsobu zápisu (podprogramy se zapisují pouze v tzv. prefixovém tvaru, kdy jméno podprogramu předchází svým argumentům, operátory se zapisují v prefixovém, postfixovém (operátor se nachází za operandem) i infixovém tvaru (operátor je mezi operandy)). Každý operátor má daný počet operandů, se kterými pracuje (tato vlastnost se nazývá *arita*), a které jsou prováděné v určitém pořadí (podle jejich *priority*).

Operátorů existuje poměrně velké množství a tabulka určující jejich prioritu je poměrně obsáhlá a složitá na zapamatování. Dobrou zprávou je, že většina operátorů Perlu má svoje ekvivalenty i v jiných programovacích jazycích či programových prostředcích. Detailně se operátorům věnuje kapitola *6. Operátory*.

Příkazy

Jednoduchý příkaz je výraz ukončený středníkem. Každý takový příkaz může navíc obsahovat modifikátor, který upřesňuje kolikrát a zda vůbec má být příkaz proveden.

```
# jednoduchý příkaz bez modifikátoru
print ”$x / $y = ”, $x/$y;

# jednoduchý příkaz s modifikátorem, provede se pouze
# v případě, že hodnota proměnné $y je různá od 0
print ”$x / $y = ”, $x/$y if $y != 0;

# vytiskne druhé mocniny čísel od 1 do 10
$x = 0;
print ”$x * $x = ”, $x*$x, ”\n” while ++$x <= 10;
```

Jednotlivé příkazy programu nemusejí být samozřejmě prováděny pouze sekvenčně, ale také opakovaně či vůbec při nesplnění určité podmínky. Aby to bylo umožněno, Perl nabízí tzv. řídicí příkazy.

Jako většina programovacích jazyků má Perl implementováno větvení pomocí příkazu `if` nebo `unless`, cyklus s podmínkou na začátku `while` a `for` určený

primárně pro předem známý počet opakování. Existuje navíc cyklus `foreach`, který umožňuje provádět jednotlivé iterace pro každou položku ze zadaného seznamu. Pro cyklus s podmínkou na konci neexistuje žádný speciální příkaz, ale je možné ho vytvořit s využitím ostatních konstrukcí. Příkazy a jejich syntaxe jsou náplní kapitoly *5. Příkazy*.

```
if ($x == 0) {
    print "$x / $y = ", $x/$y;
} else {
    print "Nelze dělit nulou.";
}

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

Při rozhodování o tom, zda se určitá část kódu provede nebo neprovede, je třeba v logickém kontextu vyhodnotit výraz zvaný podmínka — rozhodnutí se provádí na základě její pravdivosti či nepravdivosti. Pravdivé jsou v Perlu všechny výrazy, kromě nedefinované hodnoty, hodnoty `0`, prázdného řetězce a řetězce obsahujícího pouze znak `0`. Podrobně je pravdivost rozebrána v kapitole *3. Kontext*.

Bílé znaky

Důležitou součástí programu v Perlu jsou i *bílé znaky*. Jedná se o znaky jako mezera, tabulátor, znak nového řádku. Tyto znaky jsou při interpretaci ignorovány, takže nezáleží na tom, zda je někde uveden jeden znak nového řádku nebo je jich sto (výjimkou je použití těchto znaků např. v řetězcovém literálu). Slouží k oddělení symbolů, které by jinak mohly být chápány jako celek — Perl se totiž vždy snaží nalézt co největší část kódu, která by mohla tvořit jeden nedělitelný celek. Objeví-li se například v programu řetězec `$promenna`, Perl to vždy bude považovat za jméno proměnné, která se jmenuje `$promenna`. I když by správný název byl i `$p`, `$pr` atd.

Bílé znaky mohou také výrazně zvyšovat čitelnost programu. Program v Perlu může být klidně napsán na jednom řádku pouze s tolika bílými znaky, kolik je nezbytně nutné. Vhodné odřádkování a odsazování však může podstatným způsobem program zpřehlednit a zjednodušit tak případné úpravy.

Komentáře

Další součástí programu jsou *komentáře*. Nejsou nezbytné, ale jejich použití je nanejvýš vhodné. Nejsou přímou součástí programu, při interpretaci se stejně jako bílé znaky

ignorují. Zvyšují přehlednost programového kódu a vysvětlují význam použitých konstrukcí. Začínají znakem # a pokračují až do konce řádku.

Zvláštním druhem komentáře je tzv. *shebang*.* Objevuje se na prvním řádku skriptu a začíná symboly `#!` následovanými jménem souboru reprezentujícího interpret Perlu (mezi `#!` a jménem souboru mohou nebo nemusejí být mezery). Má význam u skriptů na operačních systémech UNIX. Říká shellu, který program se má pro vykonání skriptu spustit v případě, že je soubor spuštěn jako spustitelný soubor. Shebang může mít následující podobu:

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

Tím je řečeno, že soubor má být předán programu *perl* v adresáři */usr/bin*. Má být rovněž spuštěn s přepínačem `-w` (znamená to vypisování varovných hlášení). Tento řetězec samozřejmě záleží na skutečném umístění interpretu a na tom, jaké přepínače Perlu chceme použít.

V systémech Windows shebang není třeba používat, protože interpret Perlu je možné asociovat k souborům reprezentujícím perlové skripty (na základě jejich přípony†). Při otevření takového souboru se pak automaticky zavolá interpret, jemuž se obsah souboru předá.

Víceřádkové komentáře

Víceřádkové komentáře začínají řádkem, jehož první znak `=` je následován alespoň jedním písmenem a dále jakýmikoliv znaky, a končící řádkem začínajícím na `=cut`. Oba tyto řádky a veškerý text mezi nimi je interpretem ignorován.

```
print 'ahoj'; # toto je komentář - příkaz vytiskne 'ahoj'

=komentar
Toto je
víceřádkový komentář
=cut
```

Tento typ komentářů je využit i v rámci dokumentace zvané Plain Old Documentation. Používá se nejčastěji k dokumentaci modulů a její podrobnější popis se nachází v kapitole *14. Moduly*.

*Název shebang vznikl z výslovnosti názvů obou těchto symbolů — znak # se někdy nazývá sharp a znak ! bang.

†Při instalaci ActiveState Perlu pro Windows je nabízena přípona *.pl*, ale někteří autoři doporučují zvolit příponu *.plx*, protože přípona *.pl* je určena pro knihovny (Perl Library).

Literálové symboly

Identifikátory začínající a končící dvěma podtržítky jsou v Perlu použity pro zvláštní účely. Literálový symbol `__FILE__` obsahuje jméno aktuálně prováděného souboru, `__LINE__` číslo aktuálně prováděné řádky a `__PACKAGE__` balík používaný v okamžiku zjišťování hodnoty tohoto symbolu. Literál `__END__` logicky ukončuje skript ještě před koncem souboru. Následující obsah je ignorován, ale je přístupný pomocí ovladače `DATA`. Podobnou funkci má symbol `__DATA__`, který otevírá ovladač `DATA` ve jmenném prostoru aktuálně používaného balíku. Při použití těchto symbolů uvnitř řetězce v uvozovkách nedojde ke vložení jejich hodnot.

Identifikátory

Identifikátory slouží k pojmenování jednotlivých objektů programu, čímž dochází k jejich odlišení od ostatních. Důležitou vlastností jazyka je, že rozlišuje mezi malými a velkými písmeny (říkáme, že Perl je case-sensitive). Děje se tak nejen v řetězcích či řetězcových literálech, ale i u identifikátorů (tzn. u jmen proměnných, podprogramů, modulů apod.). Pokud tedy někde v programu použijeme proměnnou `$Slovo` a o kousek dále chceme použít tu stejnou proměnnou, ale napíšeme `$slovo`, můžeme dostat pro nás neočekávané výsledky. Perl totiž žádnou chybu nezahlásí,* ale bude pracovat s novou proměnnou, která má jinou hodnotu, než bychom potřebovali. Je to proto, že deklarace proměnných se neprovádí — proměnné vznikají až v místě svého prvního použití.

Abychom se vyhnuli takovým problémům, dodržují se určité konvence. Jména proměnných se píší malými písmeny, jména modulů s prvním písmenem velkým, ovladače velkými písmeny. Jména, která jsou přesně daná a jsou očekávána mechanismy Perlu nebo některými moduly a jsou často využívána implicitně, se většinou píší velkými písmeny (`AUTOLOAD`, `BEGIN`, `@EXPORT` apod.). Není to nutnost, ale může to odstranit řadu problémů a zvýšit čitelnost programu.

Identifikátory začínající písmenem nebo podtržítkem mohou mít libovolnou délku (minimálně je zaručeno 255 znaků) a mohou obsahovat písmena, číslice a podtržítka. Identifikátory začínající číslicí mohou obsahovat pouze další číslice. Jestliže jméno proměnné začíná jiným znakem, omezuje se často pouze na tento jediný znak. Tyto proměnné mají v Perlu většinou určitý speciální význam (např. `$$` — identifikátor procesu, `$"` — oddělovač seznamových hodnot při vkládání do řetězců . . .) a jsou jako jediné považovány za globální (není nutná plná specifikace jejich jména). Protože tato jména jsou špatně zapamatovatelná, byl vytvořen modul `English`, jehož zavedení umožňuje pojmenovávat tyto proměnné alternativními jmény (místo `$$` použijeme `$PID` apod.).

Jméno může obsahovat i symbol `::`. Tímto symbolem se oddělují jména balíků a jméno balíku od jména proměnné či podprogramu. Konstrukce `$Kniha::nazev` označuje proměnnou `$nazev` z balíku `Kniha`.

*Pokud zapneme výpis varovných hlášení, je možné, že tuto chybu odhalíme. Pokud použijeme striktní režim pomocí `use strict`, je větší pravděpodobnost, že tuto chybu odhalíme již v době překladu.

```
# platné identifikátory
$x;
%_abc;
@12345;
$dlouhe_jmeno_promenne;
$Balik::jmeno;

# neplatný identifikátor
$#^%;
```

Zvláštním případem je práce se jménem jako s klíčem tabulky symbolů, kdy jméno může být tvořeno libovolnými znaky (tento příklad je ukázán v kapitole *13. Balíky*).

Proměnné, ovladače, formáty, typegloby, podprogramy

Proměnné jsou nositelem různých vlastností (hodnot) a tvoří datovou podstatu programu. Pomocí jména proměnné se odkazujeme do operační paměti, kde jsou uložena určitá data. Proměnné mohou obsahovat jednoduchou — skalární hodnotu (nazývají se skaláry), nebo hodnoty násobné — seznamové (pole a asociativní pole neboli *hash*).

Skalární proměnná může obsahovat buď číslo nebo řetězec (ještě může obsahovat odkaz, ale to je trochu jiná kategorie). Není však možné říci, že nějaká konkrétní proměnná obsahuje číslo a jiná řetězec. Obsah bude chápán podle toho, kde bude proměnná použita. Bude-li použita operátorem pro sčítání pracujícím s čísly, bude s hodnotou proměnné pracováno jako s číslem, pokud použijeme proměnou ve výrazu s řetězcovým operátorem, bude chápána jako řetězec. Proměnnou můžeme použít také v logickém výrazu, potom na její hodnotu pode pohlíženo jako na logickou hodnotu. I když bude obsah proměnné ve všech třech případech stejný, nebude to chyba, provedou se totiž automatické konverze.

Ovladače jsou objekty, které nám umožňují pracovat se soubory, adresáři, rourami či sockety. Ovladač nejčastěji spojujeme se souborem a používáme k tomu funkci `open`.

```
open SOUBOR, 'data.txt';  # otevře soubor data.txt pro čtení,
                          # pro přístup k souboru bude použit
                          # ovladač SOUBOR
```

Pomocí *formátů* je možné definovat vzhled výstupu z programu. Formát je jakási šablona přidružená k ovladači, pomocí které se veškerý výstup upravuje (pro zápis používáme funkci `write`).

Zvláštním útvarem jsou *typegloby*, jež pojmenovávají všechno, co se jmenuje stejně jako typeglob. Prostřednictvím typeglobu můžeme přistupovat ke skalárům, polím, hashům, ovladačům, formátům a podprogramům.

Podprogramy — ať už vestavěné, nebo uživatelem definované — vykonávají určité akce a často pracují s proměnnými. Tvoří podstatu programu z hlediska prováděných činností. V terminologii jiných programovacích jazyků bývají podprogramy bez návratové hodnoty označeny jako *procedury*, podprogramy s návratovou hodnotou jako *funkce*. V Perlu není žádný rozdíl mezi procedurami a funkcemi, neboť v Perlu má všechno nějakou hodnotu. Procedury v pravém slova smyslu tedy neexistují, všechny podprogramy jsou nazývány funkcemi.* Záleží pouze na programátorovi, zda návratovou hodnotu podprogramu využije či nikoliv.

Každá proměnná i funkce mají své pojmenování. Pravidla pro vytváření jmen jsou zmíněna výše. Perl má tu vlastnost, že je možné zvolit stejné jméno pro skalární proměnnou, proměnnou typu pole, podprogram atd. Aby se jednotlivé druhy objektů od sebe odlišily, před jejich jméno píšeme ještě jeden znak specifikující typ objektu. Jméno skalární proměnné začíná znakem `$` (`$` je podobné písmenu S jako Skalár), jméno pole znakem `@` (`@` znamená A jako Array), hash `%` (`%` připomíná písmeno H jako Hash), jménu typeglobu předchází znak `*` (`*` je často zástupným symbolem pro jakýkoliv znak) a před jmény funkcí se píše `&`. Ovladače ani formáty žádný prefix nemají.

Typickým znakem Perlu je to, že proměnné či ovladače se nemusejí nikde deklarovat ani inicializovat, vznikají až v okamžiku potřeby při jejich prvním použití. Neznamená to, že to v žádném případě nesmíte udělat. Je to ale spíše zvyk z jiných programovacích jazyků a brzy určitě přejde.

Podrobně je práce se skalárními a seznamovými proměnnými a skalárními a seznamovými hodnotami popsána v kapitole *4. Datové typy, proměnné*, práce s podprogramy v kapitole *9. Podprogramy* a formáty se zabývá kapitola *8. Formáty*. Ovladače jsou zmíněny v kapitole *17. Práce se soubory a adresáři* a u popisu funkce `open` v kapitole *10. Standardní funkce*. Typegloby jsou detailněji rozebrány v kapitole *13. Balíky*.

Balíky, tabulky symbolů, moduly

Jména proměnných, podprogramů, ovladačů apod. patří do určitého prostoru — jejich platnost je omezená, nejsou viditelné všude (výjimku tvoří několik speciálních proměnných a ovladačů, jejich seznam a význam se nachází v kapitole *4. Datové typy, proměnné*). Říkáme, že jejich *rozsah platnosti* je vymezený na určitý *jmenný prostor*.

Takový jmenný prostor se nazývá *balík*. Každý balík má svoji *tabulku symbolů*, kde jsou uchovávána jména všech objektů a odkazy na jejich hodnoty. Implicitně používaným balíkem (pokud neřekneme jinak) je balík `main` (tam se zároveň nacházejí ony zmíněné speciální proměnné, které jsou vždy globální). Ten zároveň obsahuje odkazy na všechny ostatní tabulky symbolů, takže jakýkoliv objekt nacházející se v tabulce symbolů je v podstatě globální (je možné se k němu vždy dostat přes tabulku symbolů balíku `main`).

*Protože existuje podobnost mezi funkcemi a operátory, především seznamovými, bývá také někdy zaměňován pojem funkce a operátor.

Platnost balíku vyznačujeme pomocí funkce `package`, které jako argument předáme jméno balíku, který má být aktivní.

Jméno každého objektu se navíc kromě svého jména skládá i ze jména balíku, do kterého patří. Obě tyto části jsou od sebe odděleny znakem `::`. Pokud jméno balíku neuvedeme, předpokládá se, že objekt náleží do balíku aktuálního.

```
package A;
$x = '123';   # nastavení proměnné $x v balíku A

package B;
print $x;     # tisk proměnné $x z balíku B, nevytiskne
              # nic, protože proměnná má nedefinovanou hodnotu
print $A::x;  # vytiskne '123'
```

U proměnných máme navíc možnost specifikovat jejich rozsah platnosti. Existují tři způsoby — *dynamické vymezení platnosti* hodnoty proměnné pomocí operátoru `local`, *lexikální vymezení jména globální proměnné* operátorem `our` a *lexikální vymezení proměnné* za pomoci operátoru `my`.

Poslední jmenovaný způsob umisťuje proměnnou místo do tabulky symbolů do speciálního prostoru, takže se pak jedná o skutečně lokální proměnnou. Po opuštění rozsahu platnosti je zrušena, takže dál nezabírá paměť. V příkladech této knihy bude tento způsob občas použit, protože se jedná o dobrý zvyk (takovéto lokální proměnné nenarušují své okolí a práce s nimi je rychlejší). Dynamické vymezení platnosti pouze pro daný rozsah platnosti zastíní hodnotu proměnné a po opuštění rozsahu platnosti je tato hodnota obnovena zpět. Lexikální vymezení jména globální proměnné způsobí vytvoření aliasu pro jméno globální proměnné, a ta se pak chová jako lokální proměnná.

Rozsah platnosti v tomto případě není omezen balíkem, ale blokem, kde byla příslušná deklarace provedena.

Kromě způsobu uschování proměnné v paměti (tabulka symbolů nebo lexikální prostor) ovlivňuje platnost proměnné také proces vyhledání proměnné podle jména.

Alespoň nastínit problematiku vymezování platnosti proměnných má za úkol následující příklad.

```
$globalni = 'globalni';
{
 print $globalni;    # vytiskne 'globalni'
 $lokalni = 123;
 my $globalni = 123;
 my $lexikalni = 'lexikalni'
 {
  local $lokalni = 'lokalni';
  print $globalni;   # vytiskne '123'
```

```
  our $globalni;
  print $globalni;   # vytiskne 'globalni'
 }
 print $globalni;    # vytiskne '123'
 print $lokalni;     # vytiskne '123'
}
print $lexikalni;    # nevytiskne nic
```

Balíky, tabulky symbolů a rozsahy platnosti proměnných jsou náplní kapitoly *13. Balíky*.

Pokud umístíme balík do souboru, který se jmenuje stejně jako balík a má příponu *.pm*, hovoříme o *modulu*. Perl umožňuje vtahovat jakékoliv soubory do programu (efekt je stejný, jako kdybychom obsah souboru překopírovali na příslušné místo), ale moduly poskytují i něco navíc. Pomocí tzv. mechanismu *importu symbolů* je umožněno, aby v jednom balíku byla používána jména objektů z jiných balíků bez nutnosti plné specifikace jejich jména (tzv. nemusíme do jejich jména zahrnout název balíku).

Podrobně se tvorbou a zaváděním modulů a exportem a importem symbolů zabývá kapitola *14. Moduly*.

Hlavním důvodem používání modulů je znovupoužitelnost již jednou vytvořeného kódu. Programátor pak nemusí stále dokola psát stejné části programu, ale umístí je do funkcí, které opakovaně používá. V distribuci Perlu je k dispozici celá řada standardních modulů (jejich seznam a popis je v kapitole *18. Standardní moduly*), je taktéž možné najít a stáhnout si moduly ze sítě CPAN.

Zvláštním typem modulů jsou tzv. *pragmatické moduly*, které ovlivňují proces překladu zdrojového kódu programu. Asi nejčastěji používaným pragmatem je modul `strict`, který vynucuje při psaní skriptu dodržování určitých pravidel, která jsou přísnější než bez tohoto modulu. Pragmatickými moduly se podrobněji zabývá kapitola *19. Pragmatické moduly*.

Regulární výrazy

Velmi mocným nástrojem jazyka Perl jsou *regulární výrazy*. Ty jsou prostředkem, který umožňuje prohledávat textová data a nelézat v nich části odpovídající zadanému vzoru. Výsledkem takového prohledávání je nejčastěji informace o tom, zda text vzoru odpovídá nebo neodpovídá.

Regulární výraz je řetězec, který je tvořen podle určitých pravidel. Některé symboly odpovídají samy sobě, jiné mají speciální význam — hovoříme o metaznacích a metasymbolech (např. symbol `\n` odpovídá znaku konce řádku, `\d` je označení pro třídu znaků a odpovídá právě jedné jakékoliv číslici, `+` říká, že předcházející část se má v řetězci nacházet jednou a vícekrát). Pomocí tzv. kvantifikátorů říkáme, kolikrát se mají určité části vzoru opakovat. Můžeme používat i varianty vzorů či jednotlivé nalezené části zapamatovat a uložit.

Pro práci s regulárními výrazy slouží speciální operátory. Nejčastěji používaným je operátor `m`, který pouze nalezne část řetězce odpovídající vzoru, a operátor `s`, který umí nalezené podřetězce nahradit jiným textem.

```
if ($text =~ m/\d+/) {
    print "Hodnota $text je číslo";
else {
    print "Hodnota $text není číslo";
}
```

Všechny podrobnosti týkající se regulárních výrazů se nacházejí v kapitole *7. Regulární výrazy*.

KAPITOLA 3.

KONTEXT

- Co se rozumí pod pojmem kontext
- Všechny druhy kontextů
- Pravdivost a nepravdivost v Perlu

Pojem kontext je znám z přirozeného jazyka. Můžeme ho chápat jako okolí určitého sdělení, jedno sdělení v různém kontextu může mít zcela rozdílný význam. Stejně tak každá operace v Perlu má svůj *kontext* — vyhodnocuje se v určitém kontextu. Znamená to, že na operátory i operandy jsou kladeny určité požadavky a ony se potom podle nich chovají. Některé operátory či výrazy se chovají stále stejně nezávisle na kontextu, ve kterém se vyskytují, jindy jejich chování na kontextu závisí. Uvědomění si, v jakém kontextu se právě nacházíme, je často velmi důležité. Můžeme totiž dostávat zdánlivě chybné výsledky plynoucí právě z toho, že jsme špatně rozhodli, v jakém kontextu se nacházíme.

3.1 Skalární a seznamový kontext

Základními dvěma kontexty jsou *kontext skalární* a *kontext seznamový*. Jestliže operátor očekává skalární operand nebo funkce očekává skalární operand, říkáme, že vynucuje skalární kontext, a tento operand je vyhodnocen ve skalárním kontextu. Například přiřazení do skalární proměnné vyhodnocuje pravou stranu přiřazovacího příkazu ve skalárním kontextu (je očekávána jedna hodnota), zatímco přiřazení do pole vyhodnocuje pravou stranu v seznamovém kontextu (očekává se seznam hodnot). Některé operátory vracejí v každém z těchto kontextů různé hodnoty.

```
# pravé strany přiřazení vyhodnoceny ve skalárním kontextu
$x = 1;                # výsledek '1'
$x = ('a', 'b', 'c');  # výsledek 'c'
$x = 'a', 'b', 'c';    # výsledek 'a', = má větší prioritu než čárka
$x = @pole;            # výsledek je počet prvků @pole

# některé operátory pracují pouze se skalárními operandy
@x = ('a', 'b'); @y = (1, 2, 3);
$soucet = @x + @y;   # $soucet obsahuje hodnotu 5
```

Vyhodnocením seznamu (kolekce skalárních hodnot uzavřených v kulatých závorkách) ve skalárním kontextu je poslední prvek tohoto seznamu. Vyhodnocením proměnné typu pole ve skalárním kontextu je délka tohoto pole (tzn. počet prvků pole). Vyhodnocením proměnné typu asociativní pole ve skalárním kontextu je hodnota vyjadřující obsazenost hashovací tabulky pro uchování klíčů a hodnot této struktury.

Je-li na levé straně přiřazení seznamová hodnota nebo seznamová proměnná, je pravá strana vyhodnocena v seznamovém kontextu. Taktéž v rámci seznamu jsou všechny prvky seznamu vyhodnocovány v seznamovém kontextu.

```
# pravé strany přiřazení vyhodnoceny v seznamovém kontextu
($x) = 1;
@pole = 1;
```

```
@pole = podprogram();
%hash = (1 => 'a');
@pole[1..3] = 'a', 'b', 'c';

# argumenty většiny funkcí jsou vyhodnoceny v seznamovém kontextu
print 1, 2, @pole, funkce;
podprogram(1, 2, 'abc', @pole, funkce());
```

Skalární kontext lze vynutit použitím funkce `scalar`. Je třeba dávat pozor na to, že se jedná o unární operátor, proto se vztahuje pouze k bezprostředně následujícímu výrazu. Pro vynucení seznamového kontextu obdobná funkce neexistuje.

```
@pole = ('a', 'b', 'c', 'd');

# vynucený skalární kontext
@a = scalar @pole;  # @a obsahuje jeden prvek - hodnotu 4,
                    # což je počet prvků @pole
podprogram(1, 2, 'abc', scalar @pole, funkce());
# tady se scalar vztahuje pouze na @pole
```

Uvnitř podprogramu můžeme zjistit kontext, v jakém byl podprogram volán, pomocí funkce `wantarray`. Ten vrací pravdivou hodnotu v případě, že podprogram byl volán v seznamovém kontextu, v ostatních případech vrací nepravdivou hodnotu.

V těch částech knihy, kde bude popisováno chování jednotlivých prvků jazyka Perl závisejících na kontextu (volání standardních podprogramů, práce operátorů, chování příkazů apod.), bude na rozdílné chování v různých kontextech upozorněno.

3.2 Logický kontext, pravdivost

V logickém kontextu jsou výrazy vyhodnocovány v případě, že je třeba získat hodnotu pravda nebo nepravda. Vyhodnocování pravdivosti se děje vždy ve skalárním kontextu — operátory vracející pravdivostní hodnotu pracují se skalárními operandy, příkazy `if`, `while` apod. vyhodnocují podmínku ve skalárním kontextu. Proto je třeba na pravdivostní výraz pohlížet ve skalárním kontextu a odlišit ho od kontextu seznamového.

```
if (@pole) {
    # proběhne pouze v případě, že @pole obsahuje nějaké prvky,
    # tzn. jeho délka je různá od 0, což je nepravdivá hodnota
    ...
}
```

```
die 'Nelze dělit nulou' unless $y;
print $x/$y;

@a && @b && @c && print 'Všechna tři pole obsahují nějaké prvky';
```

A jak vlastně poznáme, zda je hodnota pravdivá nebo nepravdivá? Stačí si zapamatovat následující pravidlo. Pravdivé jsou všechny řetězce kromě `''` (prázdný řetězec) a `'0'`, všechna čísla kromě `0` a všechny odkazy, nepravdivá je jakákoliv nedefinovaná hodnota (nemá hodnotu).

Tab. 3–1: Příklady pravdivých a nepravdivých hodnot

Hodnota	Pravdivost
`0`	nepravdivá
`0.0`	nepravdivá
`-0e0`	nepravdivá
`000`	nepravdivá
`0x0`	nepravdivá
`"0"`	nepravdivá
`"0.0"`	pravdivá
`"0e0"`	pravdivá

Perl nepodporuje logický datový typ (jako je například boolean v Pascalu). Pokud chceme uchovat pravdivostní hodnotu, musíme použít buď skalární nebo seznamovou proměnnou.

Budeme-li chtít pro uchování logické hodnoty použít skalární proměnnou, pro nastavení nepravdivé hodnoty do ní uložíme hodnotu `0`, prázdný řetězec nebo nedefinovanou hodnotu, pro hodnotu pravdivou cokoliv jiného.

```
$podminka = 0;
...
if ($podminka) {
    # $podminka obsahuje pravdivou hodnotu
    ...
} else {
    # $podminka obsahuje nepravdivou hodnotu
    ...
}
```

Chceme-li použít proměnnou seznamovou (pole, hash) a přiřadíme-li do ní jednu z nepravdivých hodnot, nedostaneme výsledek, jaký bychom potřebovali.

```
@podminka = 0;
...
if (@podminka) {
    # výraz @podminka je pravdivý
    ...
}
```

Vyhodnocením hodnoty `0` v seznamovém kontextu je totiž jednoprvkový seznam a vyhodnocením tohoto seznamu ve skalárním kontextu je hodnota `1` (délka seznamu), což je hodnota pravdivá. Proto je jasné, že jedinou seznamovou hodnotou, která bude v logickém (a tedy skalárním) kontextu vyhodnocena jako hodnota nepravdivá, je prázdný seznam.

3.3 Prázdný kontext

Vzniká tam, kde není požadována žádná návratová hodnota nebo nezáleží na jejím typu. Příkladem je volání podprogramu jako jediného v celém výrazu nebo použití řetězce jako příkazu (při překladu se zapnutými varovnými hlášeními to vede k varování, že byla použita konstanta v prázdném kontextu).

```
podprogram();
"abcd";
```

3.4 Vkládací kontext

Vzniká při vkládání hodnot proměnných a symbolů začínajících znakem obráceného lomítka uvnitř řetězců ohraničených uvozovkami a v některých dalších případech (operátory porovnání podle vzoru, nahrazení atd.). V podstatě existují dva základní druhy kontextů — jeden, kde vkládání probíhá, a druhý, kde vkládání neprobíhá. Podrobně je princip vkládání popsán v kapitole *4. Datové typy, proměnné* a *6. Operátory*.

KAPITOLA 4.

DATOVÉ TYPY, PROMĚNNÉ

- Datové typy podporované v Perlu
- Práce s čísly a řetězci, vkládání do řetězců
- Typy proměnných a práce s nimi
- Standardní funkce pro práci se skalárními a seznamovými hodnotami
- Seznam a popis předdefinovaných jmen

Na rozdíl od některých programovacích jazyků má Perl relativně malý počet datových typů. Neumožňuje také vytvářet nové pojmenované abstraktní datové typy jako například v Pascalu nebo v C. Neznamená to však, že takové věci není možné provést jiným způsobem — je možné například použít objektově orientovaného přístupu. I tak se ale v podstatě nejedná o vytvoření nového typu, ale o využití několika málo běžně používaných mechanismů.

Datové typy se vztahují jak k literálovým hodnotám, tak i k proměnným. Často je možné hodnoty různých typů zaměňovat bez toho, že by překladač hlásil chybu, jak je tomu v přísně typovaných programovacích jazycích.

Perl implementuje pouze tři základní datové typy — skaláry, pole skalárů a hashe skalárů (asociativní pole).

Skaláry jsou základním datovým typem a tvoří se z nich ostatní, složitější datové typy. Uchovávají jednu jedinou hodnotu, kterou může být řetězec, číslo nebo odkaz. *Pole* je uspořádaný seznam skalárů, k jednotlivým hodnotám se přistupuje pomocí číselného indexu (indexy začínají implicitně od `0`). *Hash* je neuspořádaná množina dvojic klíč/hodnota. K hodnotám se zde přistupuje pomocí řetězcových indexů zvaných *klíče*. Skalární proměnná může obsahovat skalární hodnotu, proměnná typu pole obsahuje seznamovou hodnotu, která je chápána jako posloupnost jednoduchých hodnot, a proměnná typu hash obsahuje seznamovou hodnotu, jež je interpretována jako seznam neuspořádaných dvojic klíč/hodnota.

Identifikátor proměnné každého typu je uvozen zvláštním znakem. Skalární proměnné začínají znakem `$` (připomíná S jako Scalar), seznamy začínají znakem `@` (připomíná a jako array) a hashe začínají znakem `%` (připomíná H jako Hash). Každý typ proměnné má svůj vlastní jmenný prostor. Znamená to, že pro skalární proměnnou je možné použít stejné jméno jako pro proměnnou typu pole nebo hash. Tato jména se pak budou lišit pouze svým prvním znakem.

Díky tomu, že jména proměnných začínají speciálním znakem, se nemůže stát, že by se pletla s rezervovanými slovy, jak tomu může být v jiných programovacích jazycích. Tento problém však může nastat např. u jmen podprogramů či ovladačů.

Jméno proměnné nemusí být vždy známo v době překladu, ale může být určeno až za běhu programu. Toho lze docílit tak, že místo jména proměnné lze použít blok vracející řetězec nebo odkaz. Pokud blok vrací odkaz, hovoříme o dereferenci pevného odkazu, pokud blok vrací řetězec, hovoříme o symbolickém odkazu.

```
${'x'} = 1;         # pracuje se s proměnnou $x

sub jmeno { return 'y'};
${ jmeno() } = 2;   # pracuje se s proměnnou $y

$rx = \$x;          # odkaz na proměnnou $x
${ $rx } = 123;     # pracuje se s proměnnou $x
```

Jména proměnných a odkazy na jejich hodnoty jsou ukládány ve dvou typech paměťových struktur. Ty se nazývají tabulky symbolů a lexikální prostory. Tabulky symbolů jsou vztaženy vždy k určitému balíku, lexikální prostory se vztahují k bloku a nemají s tabulkami symbolů nic společného. To, kde bude proměnná umístěna, je určeno v okamžiku její deklarace. Jestliže proměnnou deklarujeme pomocí funkce `my`, jedná se o lokální proměnnou a bude umístěna do lexikálního prostoru, v ostatních případech se jedná o globální proměnnou náležející do tabulky symbolů. Více informací týkajících se tabulek symbolů, lexikálních prostorů, deklarací a práce se jmény proměnných se nachází v kapitole *13. Balíky*.

4.1 Skalární hodnoty a skalární proměnné

Skaláry obsahují vždy jednu jedinou hodnotu, kterou je řetězec, číslo nebo odkaz na jiná data. Reprezentuje tedy všechny celočíselné typy, typy pracující v pevné i pohyblivé řádové čárce, typy znakové i řetězcové.

Ve skutečnosti nelze říci, že skalár obsahuje hodnotu nějakého konkrétního typu. Není tedy například možné říci, že proměnná `$x` je celočíselná proměnná nebo proměnná typu znak. Se skalární proměnnou nebo hodnotou se zachází podle toho, jakým způsobem ji programátor použije (záleží na operátorech ve výrazu, kde se proměnná objevuje). Znamená to, že v jednom okamžiku je možné s takovou proměnnou zacházet jako s číslem a jindy jako s řetězcem. Bez ohledu na obsah proměnné nebude překladačem ani v jenom případě zahlášena chyba a záleží na programátorovi, aby si pohlídal, jaká data proměnná obsahuje a k jakému účelu jsou tato data využívána. Případné konverze z čísla na řetězec a naopak se provádějí automaticky bez upozornění. To může být v některých případech zdrojem chyb, jež nejsou na první pohled patrné, a proto se těžko odhalují.

Poněkud komplikovanější situace je při práci s odkazy. Je sice možné provést automatickou konverzi odkazu na číslo nebo řetězec, ale kromě možnosti porovnání takových hodnot čí zjištění typu odkazu to nemá smysl. Použití odkazu jako řetězce vede k tomu, že je použita řetězcová reprezentace odkazu (např. `SCALAR(0x1a950e0)`), jestliže odkaz použijeme jako číslo, získáme několikamístné celé číslo. Je samozřejmé, že pro dva různé odkazy dostaneme vždy dvě různé číselné nebo řetězcové reprezentace. Zpětná konverze takovýchto hodnot na odkaz je však nemožná.

V Perlu neexistuje nic jako logický typ (např. boolean v Pascalu). Pro uchování logické hodnoty lze použít proměnnou jakéhokoliv typu. Záleží na tom, v jakém výrazu a kontextu tuto proměnnou použijeme. Chceme-li z nějaké proměnné logickou hodnotu (pravda nebo nepravda) získat, řídí se vše pravidly o pravdivosti. Použijeme-li k uchování či zjištění logické hodnoty skalár, pro nepravdivou hodnotu do něj uložíme hodnotu nula, prázdný řetězec, řetězec obsahující znak nula či nedefinovanou hodnotu. Pravdivá hodnota bude všechno ostatní. Použijeme-li pro logickou hodnotu pole, nepravdivé bude vyhodnocení prázdného pole ve skalárním kontextu (hodnota `0`, což je délka pole, viz dále).

```
$x = 1;        # $x je pravdivé
$x = 0;        # $x je nepravdivé
$x = "0";      # $x je nepravdivé
$x = undef;    # $x je nepravdivé
@pole = ();    # @pole ve skalárním kontextu je nepravdivé
@pole = undef; # @pole ve skalárním kontextu je pravdivé,
               # protože obsahuje jeden prvek
```

Nedefinovaná hodnota

Je-li použita skalární proměnná poprvé a není jí přiřazena žádná hodnota, říkáme, že obsahuje *nedefinovanou hodnotu*. Nedefinovaná hodnota také indikuje, že pro něco skutečná hodnota neexistuje a někdy se používá pro označení chybového stavu. Je tomu tak například v případě, kdy chceme pracovat s neexistujícím prvkem pole nebo hashe nebo jsme při čtení souboru došli na jeho konec.

K tomu, aby bylo možné zjistit, zda je hodnota skalární proměnné definována, slouží unární operátor `defined`, kterému jako argument předáme příslušnou proměnnou (výraz). Při práci s proměnnou typu pole nebo hash jako s celkem nemá žádný význam o definovanosti uvažovat, protože to jsou kolekce skalárů. Pokud zjišťujeme definovanost pole nebo hashe, závisí výsledek na tom, zda byl pro tuto proměnnou alokován prostor v paměti. Přiřazení nedefinované hodnoty do pole má za následek vznik pole o jednom prvku, jehož hodnota je nedefinovaná. Při přiřazení do hashe vznikne hash o jedné dvojici nedefinovaných hodnot. Navíc je očekáván sudý počet hodnot (dvojic klíč/hodnota), jedna hodnota by tedy chyběla. Oba dva případy by způsobily varování.

Vestavěná funkce `undef` vrací nedefinovanou hodnotu a tu je možné použít například v přiřazení pro zrušení hodnoty proměnné nebo jako návratovou hodnotu funkce, která bude indikovat neúspěch.

Čísla

V Perlu rozlišujeme čísla celá a čísla v pohyblivé řádové čárce. Interně Perl ukládá čísla jako čísla v pohyblivé řádové čárce s dvojnásobnou přesností (pokud ovšem nezakážeme používání reálných čísel pomocí `use integer`). Číselné hodnoty lze zapisovat několika způsoby.

Tab. 4–1: Příklady zápisů číselných hodnot

Hodnota	Význam
`12345`	celé číslo kladné
`-123`	celé číslo záporné

Hodnota	Význam
`123.456`	číslo s desetinnou čárkou kladné
`-123.456`	číslo s desetinnou čárkou záporné
`1.23E16`	v semilogaritmickém tvaru ($1.23 * 10^{16}$)
`-12e34`	velké záporné číslo ($-12 * 10^{34}$)
`-12e-34`	velmi malé záporné číslo ($-12 * 10^{-34}$)
`0xFF,0X2e`	šestnáctkové číslo (může být malé `x` i velké `X`)
`0247`	osmičkové číslo (`0` na začátku)
`0b111000`	binární číslo (pouze malé `b`)
`0b111_000`	binární číslo s podtržítkem
`1_123_456_789`	s podtržítky pro zvýšení čitelnosti
`0xFF_FF_AA`	s podtržítky pro zvýšení čitelnosti
`1__23_4e12_34`	s podtržítky pro snížení čitelnosti :-)

Pro oddělení celé a desetinné části se používá tečka, čárka je operátor oddělující jednotlivé prvky seznamu. Kdekoliv mimo začátek literálu je možné pro zvýšení čitelnosti použít znaku podtržítko (jeden nebo více výskytů). Této formy lze používat pouze u konstant při psaní programu. Pokud jsou tato čísla získána například jako vstup programu čtením ze souboru (což jsou v podstatě řetězce), bude jako číslo braná část pouze po první podtržítko (viz *Automatické konverze mezi řetězci a čísly*). Stejně tak šestnáctková a osmičková čísla na začátku s `0x` nebo `0` jsou rozpoznána pouze v případě, že se jedná o literály. Chceme-li pracovat s takovými čísly zadanými ve formě řetězce jako s čísly v dané soustavě, musíme použít konverzní funkce `hex` nebo `oct`.

Pro práci s čísly je možné použít standardní moduly `Math::*`. Ty umožňují pracovat s libovolně velkými celými i desetinnými čísly, racionálními čísly (zlomky) či komplexními čísly.

Řetězce

Řetězce jsou posloupnosti znaků chápané dohromady jako jeden skalár a není možné přímo přistupovat ke každému jednotlivému znaku.* Minimální délka řetězce je nulová a maximální délka je omezená velikostí dostupné paměti. Aktuální délka řetězce se dynamicky mění podle toho, jak s ním pracujeme.

Záleží na použité znakové sadě, kolik místa v paměti bude zabírat jeden znak. Ve znakové sadě o délce znaku 8 bitů (ASCII) bude délka jednoho znaku jeden byte. Znaky Unicode mohou zabírat bytů více (znak v UTF8 jeden až šest bytů). Pokud bychom i s takovými řetězci chtěli pracovat po bytech, musíme zavést pragmatický modul `bytes`.

*Některé programovací jazyky chápou řetězce jako složený datový typ a jednotlivé znaky jsou brány samostatně. Také proto je k nim možné přistupovat pomocí číselných indexů.

Pokud se v programu objevují řetězcové literály, zadávají se pomocí řetězcových operátorů. Rozlišujeme dvě základní skupiny těchto operátorů — tam, kde probíhá vkládání hodnot proměnných a speciálních sekvencí, a tam, kde neprobíhá. Základními verzemi těchto operátorů jsou uvozovky (zde vkládání probíhá) a apostrofy (zde vkládání neprobíhá)*. Dále existují obdoby těchto operátorů, které však mají stejné chování, liší se pouze vzhledem. Jak jsou kompilátorem řetězcové konstanty zpracovávány, jak probíhá vkládání hodnot apod. je podrobně popsáno v kapitole *6. Operátory* v části *Termy a seznamové operátory (vlevo)*.

Je-li někde zapsán literál začínající písmenem `v` následovaný posloupností čísel oddělených tečkami, jedná se o zápis řetězce pomocí ordinálních čísel jeho jednotlivých znaků (tyto řetězce se nazývají Version strings, nebo také v-strings). Obsahuje-li literál dvě a více teček, není nutné počáteční `v` psát (v opačném případě by tento zápis byl považován za desetinné číslo). Takto zapsaný řetězec se nepíše do uvozovek.

```
v97.105.111.107              # to samé jako 'ahoj'
97.105.111.107        # to samé jako v97.105.111.107
```

Po verzi Perlu 5.8 by tento způsob zadávání řetězcových literálů neměl být možný.

Vkládání do řetězců

Řetězce v uvozovkách podléhají vkládání hodnot proměnných a speciálních sekvencí začínajících obrácenými lomítky podobně jako v UNIXovém shellu. Některé znaky ve spojení s obráceným lomítkem získávají speciální význam — jsou známy jako *escape sekvence* či *metaznaky*. Obsahuje-li proměnná `$den` hodnotu `pondělí`, příkaz

```
print "Dnes je $den\n";
```

vypíše text `Dnes je pondělí` a odřádkuje.

Díky tomu, že jméno proměnné začíná speciálním symbolem, není třeba při vkládání její hodnody do řetězce používat zvláštní syntaxi.

Je-li nalezen znak `$`, považuje se za začátek jména proměnné. Potom se vezme všechno, co následuje a vypadá jako jméno proměnné, tedy řetězec `$den` (znak `\` už nemůže být součástí jména proměnné) a hodnota této proměnné se do řetězce vloží. Sekvence `\n` znamená přechod na další řádek (je to označení znaku konce řádku).

Chceme-li ovlivnit způsob, jakým Perl vyhledává jméno proměnné při vkládání její hodnoty do řetězce, můžeme použít dvojice znaků `{` a `}`.

```
$x = 3; $y = 5;
print "${x}x$y=", $x*$y;  # vytiskne 3x5=15
```

*V originální anglické dokumentaci se řetězce v uvozovkách nazývají „double-quoted“ a řetězce v apostrofech „single-quoted“.

Bez použití složených závorek by se za jméno proměnné považoval řetězec `$xx`. Taková proměnná v našem případě neexistuje, proto bychom nedostali požadovaný výstup.

Do řetězců je možné vkládat i hodnoty z proměnných typu pole. V takovém případě je mezi jednotlivé prvky pole vložen obsah proměnné `$"` (`$LIST_SEPARATOR`) — implicitně je tam uložena jedna mezera — a vzniklý řetězec je potom do řetězce vložen. Přiřazením jiné hodnoty do této proměnné je možné způsob vkládání změnit.

```
@pole = (1, 2, 3);
print "Obsah pole je: @pole";  # vytiskne 'obsah pole je: 1 2 3'

$" = '+';                      # změna oddělovače prvků pole
print "@pole";                 # vytiskne '1+2+3'
```

Je třeba rozlišit případ, kdy je funkcí `print` vypisováno pole (nebo seznam) bez vložení do řetězce. Prvky pole jsou v tomto případě odděleny řetězcem uloženým v proměnné `$,` (`$OUTPUT_FIELD_SEPARATOR`).

Kromě hodnot proměnných je možné do řetězců vkládat i znaky popsané určitou sekvencí znaků, označovanou jako escape sekvence (metaznak) — jejich seznam se nachází v následující tabulce.

Tab. 4–2: Escape sekvence při vkládání do řetězců

Symbol	Význam
`\n`	nový řádek
`\r`	návrat vozíku
`\t`	tabulátor
`\f`	nová stránka
`\b`	backspace
`\a`	pípnutí
`\e`	znak ESC
`\101,012`	znaky s ASCII hodnotou osmičkově (znaky `A` a `\n`)
`\x61`	znak s ASCII hodnotou šestnáctkově (písmeno `a`)
`\cC`	znak CTRL+C
`\x{263a}`	znak ve znakové sadě Unicode
`\N{JMÉNO}`	znak se zadaným jménem*
`\\`	znak `\`

*Více v kapitole *19. Pragmatické moduly*, modul `charnames`.

Dále existují některé sekvence, které ovlivňují reprezentaci následují znaků. Tyto sekvence se nazývají *modifikátory*.

Tab. 4–3: Modifikátory při vkládání do řetězců

Symbol	Význam
`\u`	převede následující znak na velké písmeno
`\l`	převede následující znak na malé písmeno
`\U`	převede následující znaky na velká písmena
`\L`	převede následující znaky na malá písmena
`\Q`	následující nealfanumerické znaky uvede obráceným lomítkem
`\E`	ukončuje `\U`, `\L`, `\Q`

Modifikátor `\E` se vztahuje vždy k nejbližšímu předchozímu modifikátoru. Neruší tedy platnost všech ostatních. Je to vidět na posledním řádku následujícího příkladu.

```
$x = 'abcDEF'; $y = 'ABCdef'; $z = 'a+b=1';
print "\u$x \l$y";          # vytiskne 'AbcDEF aBCdef'
print "\U$x \L$y";          # vytiskne 'ABCDEF abcdef'
print "\Q$z";               # vytiskne 'a\+b\=1'
print "\Q\U$z \E$z \E$z";   # vytiskne 'A\+B\=1\ a\+b\=1\ a+b=1'
```

Do části řetězce, kde platí modifikátor `\Q`, není možné vložit literál `$` nebo `@`. Na tyto symboly je totiž pohlíženo jako na začátky jmen proměnných, jejichž hodnoty budou do řetězce vkládány. Umístíme-li před tyto symboly obrácené lomítko, bude tento jejich význam potlačen, avšak pomocí sekvence `\Q` se před ně doplní obrácené lomítko. Řešením není ani vložení tohoto symbolu jako hodnoty jiné proměnné či pomocí ordinální hodnoty.

```
$x = "1+2";
$y = "aaa\Q$x\Eaaa";  # $y obsahuje 'aaa1\+2aaa'
$y = "aaa\Q\$x\Eaaa"; # $y obsahuje 'aaa\$xaaa';
$dolar = '$';
$y = "aaa\Q$dolar";   # $y obsahuje 'aaa\$';
$y = "aaa\Q\44\Eaaa"; # $y obsahuje 'aaa\$xaaa';
```

Řetězec v obrácených apostrofech

Řetězec uvedený v obrácených apostrofech není řetězcovým literálem, ale je zápisem pro získání výstupu z provedení externího příkazu, jehož jméno je uvnitř apostrofů uvedeno (stejně jako v UNIXovém shellu). I v těchto řetězcích však probíhá vkládání hodnot proměnných a escape sekvencí.

```
print ‘\144\151\162‘;   # to samé jako print ‘dir‘,
                        # ve Windows vypíše obsah adresáře
```

Více se tomuto operátoru budeme věnovat v kapitole *20. Komunikace s příkazovým interpretem a ostatními procesy*.

Řetězce v apostrofech

V řetězcových literálech ohraničených jednoduchými apostrofy vkládaní neprobíhá, jedinými výjimkami je vkládání znaku `'` pomocí sekvence `\'`* a znaku `\` pomocí `\\`.

Vlastní způsob ohraničení řetězce

Perl umožňuje vlastní volbu ohraničení řetězce (jinak než pomocí operátorů `""`, `''` a ` `` `). K tomu slouží předdefinované funkce (operátory), jejichž argumentem je řetězec ohraničený dvěma ohraničovacími symboly. Tyto symboly mohou být libovolné znaky, kromě bílých znaků. Následující tabulka ukazuje způsoby ohraničení řetězců (a objektů, které jsou zpracovávány podobně jako řetězce) a jejich ekvivalenty zapsané pomocí operátorů s nejběžněji používanými ohraničovacími znaky.

Tab. 4–4: Způsoby ohraničení řetězce

Obyčejně	Jinak	Význam	Vkládání
`''`	`q//`	literálový řetězec	ne
`""`	`qq//`	literálový řetězec s vkládáním	ano
`()`	`qw//`	seznam řetězců	ne
`//`	`m//`	nalezení vzoru	ano
`s///`	`s///`	nahrazení	ano
`y///`	`tr///`	překlad	ne
`""`	`qr//`	regulární výraz	ano

Výhodou použití vlastního ohraničovacího symbolu je to, že není třeba před každým symbolem, který by jinak znamenal ukončení řetězce, používat obrácené lomítko. Pomocí tohoto způsobu zápisu je např. možné docílit toho, že není nutné před každým apostrofem (případně uvozovkou) v řetězci psát obrácené lomítko proto, aby bylo jasné, že tímto znakem není řetězec ukončen. Tím se zvýší čitelnost a sníží se riziko toho, že někde obrácené lomítko zapomeneme uvést. Proto jako

*Toto pravidlo se přesněji řečeno týká symbolu, který je použit pro ohraničení řetězce. Ve velké části to je apostrof. Více viz kapitola *6. Operátory*, část *Termy a seznamové operátory (vlevo)*.

ohraničovací symbol volíme takový znak, který se v řetězci vyskytne co nejméněkrát, nejlépe vůbec.*

Mezi jménem operátoru a počátečním ohraničovacím symbolem mohou být bílé znaky. Výjimku tvoří znak #. Ten bude chápán jako začátek komentáře a jedná se tedy o chybu.

```
$x = ”Petr pozdravil: \”Ahoj!\””;
$x = qq/Petr pozdravil: ”Ahoj!”/;  # jinak, přehledněji
$x = qq /Petr pozdravil: ”Ahoj!”/; # jinak, s mezerou
$x = qq#Petr pozdravil: ”Ahoj!”#;  # s jiným ohraničovacím symbolem

$x = q #abc#;      # řetězec #abc# je chápán jako koměntář - chyba
```

Jako ohraničovací znak může být použito i písmeno, číslice nebo podtržítko (symboly, které tvoří identifikátory). V takovém případě ale musí být mezi jménem operátoru a prvním ohraničovacím symbolem mezera, jinak by bylo vše chápáno jako jeden celek (jméno operátoru). Je tedy vhodnější používat nealfanumerické znaky, protože se zvýší čitelnost programu.

```
$y = ’111’;
$x = qq xabcx;         # to samé jako ”abc”
$x = qq xa${y}bx;      # to samé jako ”a${y}b”
$x = qqxabcx;          # to samé jako $x = qqxabcx,
                       # tzn. použití holého slova
$text =~ m m\m$m;      # to samé jako m/m$/m, tzn. hledáme
                       # ’m’ na konci řádku
```

Použijeme-li jako počáteční ohraničovací znak otevírací závorku (kulatou (()), složenou ({), hranatou ([) nebo špičatou (<)), bude jako ukončovací znak očekávána uzavírací závorka stejného typu.

```
$x = q[abc123];       # $x obsahuje ’abc’
$x =~ s{\d}<X>g;      # nahradí číslice znakem X,
                      # $x teď obsahuje ’abcXXX’
```

Obsahuje-li řetězec i ohraničovací symbol, musí být uvozen znakem obráceného lomítka, jinak bude považován za konec řetězce. Použijeme-li jako ohraničovacího znaku libovolný typ závorek a řetězec bude obsahovat vnořené závorky stejného typu (ale vždy v páru), není nutné před nimi obrácené lomítko uvádět (není to ale chyba).

*Blíže je o zpracování řetězcových a podobných termů pojednáno v kapitole *6. Operátory* (část *Termy a seznamové operátory (vlevo)*) a v kapitole *7. Regulární výrazy* (část *Zpracování vzoru – převedení do interní formy*).

```
$x = q$aaa\$bbb$;   # 'aaa$bbb'
$x = q{a{b}c};      # 'a{b}c'
$x = q{a\{b\}c};    # 'a{b}c'
$x = q{a{b\}c};     # chyba
```

Na zvoleném ohraničovacím symbolu také závisí to, zdali bude uvnitř řetězce probíhat vkládání hodnot proměnných a escape sekvencí, nebo nebude. U operátoru `q//` nebude probíhat nikdy, u ostatních operátorů v případech, že jako ohraničovací znak nebude použit apostrof.

Víceřádkové řetězce

V Perlu je umožněno pracovat s několikařádkovým textem jako s řetězcem. Počátečním ohraničovacím symbolem je `<<`. Bezprostředně potom následuje posloupnost znaků, kterou celý text bude končit. Tato posloupnost pak musí začínat na novém řádku a za ní musí následovat znak konce řádku. Je-li řetězec za `<<` v uvozovkách nebo bez uvozovek, pracuje se s celým textem jako s řetězcem v uvozovkách. Je-li posloupnost znaků v apostrofech, s textem se pracuje jako s řetězcem v apostrofech. Text v obrácených apostrofech je považován za externí příkazy, které jsou postupně vykonány.

```
print <<KONEC_TEXTU;
první řádek textu
druhý řádek textu
KONEC_TEXTU

print <<"KONEC_TEXTU";
první řádek textu
druhý řádek textu
KONEC_TEXTU

# oba případy fungují stejně, uvnitř dochází
# ke vkládání hodnot proměnných apod.

print <<'KONEC_TEXTU';
první řádek textu
druhý řádek textu
KONEC_TEXTU

# uvnitř nedochází ke vkládání hodnot proměnných apod.

print <<`KONEC_PRIKAZU`;
echo 'Aktualni adresar'
pwd
```

```
echo 'Obsach adresare:'
ls -l
KONEC_PRIKAZU

# provede uvedené příkazy
```

Je-li víceřádkový řetězec součástí výrazu nebo následuje-li další příkaz, musí být správně oddělen (středníkem, čárkou, operátorem apod.). Tento oddělovač následuje za << a slovem označujícím ukončení celého řetězce nebo za celým řetězcem (je třeba ale dávat pozor na to, že oddělovač musí být bezprostředně následován znakem nového řádku.

```
print <<KONEC;   # správně
text
KONEC

print <<KONEC    # správně
text
KONEC
;

print <<KONEC    # špatně
text
KONEC;
```

Koncový oddělovač textu, který není uveden v uvozovkách, apostrofech nebo obrácených apostrofech, musí následovat bezprostředně za symbolem <<. Není-li tomu tak, za koncový oddělovač bude považován prázdný znak, což znamená, že text bude končit prvním řádkem, na kterém bude pouze znak konce řádku. Při zapnutých varováních je na toto upozorněno.

```
print << KONEC    # chyba
první řádek textu
druhý řádek textu
KONEC

print <<
první řádek textu
druhý řádek textu

# ^^^ tady končí celý řetězec
```

Řetězce vytvořené tímto způsobem mohou normálně figurovat jako součást výrazů.

```
print uc <<KONEC, 1 x 5;
první řádek textu
druhý řádek textu
KONEC

# vytiskne
PRVNÍ ŘÁDEK TEXTU
DRUHÝ ŘÁDEK TEXTU
11111

@cisla = ($cisla = <<KONEC) =~ /\d+/gm;
5 12
3 8
7
KONEC

print join ', ',@cisla;   # vytiskne '5, 12, 3, 8, 7'
```

Řetězce vzniklé z víceřádkového textu se mohou skládat za sebe.

```
print <<PRVNI, "123\n" , <<DRUHY x 3;
abc
PRVNI
xyz
DRUHY

# vytiskne
abc
123
xyz
xyz
xyz
```

Automatické konverze mezi řetězci a čísly

Pokud se číselná hodnota použije jako řetězec, automaticky se tato hodnota převede na řetězec, který obsahuje zápis příslušného čísla. V opačném případě interpret přeskočí počáteční bílé znaky (mezery, tabulátory, konce řádků), vezme nejdelší souvislou část řetězce představující platný zápis čísla a zbytek řetězce ignoruje. Pokud řetězec nezačíná zápisem čísla, je vyhodnocen jako hodnota 0.

```
print 'a'.123;            # vytiskne 'a123'
print '12a' + 3;          # vytiskne '15' (12+3)
```

```
print '-1.2e2abc' * '2';  # vytiskne '-240' (-120*2)
print 'abc'>>3;           # vytiskne '0' (0>>3)
```

Pozor, Perl nerozpoznává čísla zadané hexadecimálně nebo oktalově, na převod je třeba volat funkce `hex` nebo `oct`. Rovněž případná podtržítka nejsou chápána tak, jako při zápisu číselného literálu.

```
print 2 * '0xf';          # vytiskne '0' (2*0)
print 2 * hex '0xf';      # vytiskne '30' (2*15)
```

Funkce pracující se skaláry

Pro práci se skalárními hodnotami existuje řada vestavěných funkcí (jejich seznam je možný najít v kapitole *10. Standardní funkce*). Navíc se některé další funkce, které mohou být užitečné pro práci se skalárními hodnotami, ale nepatří zatím mezi standardní funkce, nacházejí ve standardním modulu `Scalar::Util`.

Holá slova

Je-li někde v programu použito slovo, které nemá žádný význam (není to funkce, návěští ani název ovladače souboru apod.), je automaticky považováno za řetězec. Tyto řetězce jsou označovány jako *holá slova* (anglicky barewords). Mohou být nebezpečná v tom, že v budoucích verzích Perlu by mohla přibýt takováto rezervovaná slova. Při zapnutých varováních je na ně upozorněno, ale nejsou chybou. Je-li zaveden pragmatický modul `strict` (striktní režim) s parametrem `'subs'`, jsou považována za chybu již při kompilaci.

Holá slova nepodléhají vkládání a nemohou obsahovat některé znaky jako řetězce s ohraničovacímy symboly.

```
abc-xyz;            # použití nečíselných hodnot při odečítání

$x = 1; abc${x}yz;  # chyba, nelze provést vkládání
```

4.2 Pole a seznamy

V následující podkapitole se seznámíme s prvním ze dvou seznamových datových typů. Nejprve je ale třeba definovat dva pojmy — seznam a pole. Zatímco *seznam* je pouze uspořádaná posloupnost skalárů, *pole* je proměnná obsahující seznamovou hodnotu. Je-li někde vyžadována seznamová hodnota, je tam možné použít pole i seznam. Pouze v případě, že je vyžadováno přímo pole (například u funkcí, které modifikují pole), záměnu provést možné není.

K jednotlivým prvkům seznamu či pole lze přistupovat pomocí *číselných indexů*. Můžeme říci, že číselné indexy začínají od nuly a není možné stanovit si vlastní rozsah pole jako například v Pascalu. Ve skutečnosti je vždy indexem prvního prvku pole hodnota, kterou obsahuje proměnná `$[`* a ta je implicitně `0`. Pokud tuto hodnotu změníme, změní se i indexování prvků pole. Není to však doporučováno, protože na této vlastnosti může záviset řada dalších věcí.

Prvkem pole může být pouze skalární hodnota (číslo, řetězec nebo odkaz) a každý z těchto prvků má přesně dané pořadí (je označeno indexem). Nejmenší pole neobsahuje žádné hodnoty a nazývá se prázdné pole (takový literál se označuje `()`), největší pole zaplní celou paměť. Nikde se předem nedefinuje délka pole, ani není třeba alokovat pro prvky v poli paměť. Délka pole, a tím i velikost pole v paměti, se dynamicky mění podle toho, jak prvky do pole přidáváme či odebíráme.

Seznamové literály

Prvky seznamového literálu mohou být výrazy vracející jak skalární, tak i seznamové hodnoty. Jednotlivé hodnoty jsou odděleny operátorem čárka. Celý literál se v případě nutnosti ohraničuje kulatými závorkami. Tato nutnost je dána preferencemi operátorů ve výrazu, kde je seznamový literál použit.

```
@pole = (1, 2, 'abc', @a, funkce());
# závorky nutné, bez nich to samé jako
# (@pole = 1), 2, 'abc', @a, funkce();
# operátor = má vyšší preferenci než operátor čárka

print 1, 2, 'abc', @a, funkce();
# závorky nejsou nutné, jejich použití není chyba
# to samé jako print (1, 2, 'abc', @a, funkce())
# operátor čárka má větší prioritu než print
```

Na konci seznamového literálu může být umístěna čárka, která není následována žádnou hodnotou. Není to chyba, je to vhodné v případě, že v budoucnu budeme do literálu přidávat další prvky.

```
@pole = (1,2,3,);  # oba literály mají pouze
print 1,2,3,;      # tři hodnoty
```

V případě, že seznamový literál obsahuje jednoslovné řetězce (řetězce neobsahující bílé znaky), je možné takový seznam vytvořit pomocí operátoru `qw`.† Operátor

*Znak [ve jménu proměnné $[má znázorňovat příslušnost k polím v tom smyslu, že indexy polí se zapisují do hranatých závorek.

†Z této kapitoly už víme, že pro ohraničení argumentu operátoru `qw` můžeme zvolit libovolný znak (kromě bílých znaků) nebo dvojici závorek.

`qw` je zkratkou ze slov „quote words“ a provede to, že ze svého argumentu odstraní bílé znaky a vrátí seznam složený z takto vzniklých částí (jsou chápány jako řetězce v apostrofech).

```
@pole = qw(aaa bbb ccc ddd);

# nebo

@pole = qw(
        aaa
        bbb
        ccc
        ddd
);

# @pole obsahuje hodnoty ('aaa','bbb','ccc','ddd')
```

Jestliže se v seznamovém literálu objeví operátor konstruktoru seznamu (`..`), automaticky se vytvoří seznam začínající hodnotou na levé straně a končící hodnotou na pravé straně tohoto operátoru. Inkrementace se provádí o hodnotu jedna a probíhá pro čísla i pro řetězce. Větší prostor je tomuto operátoru věnován v kapitole *6. Operátory*.

```
(1 .. 5);         # to stejné jako (1, 2, 3, 4, 5)
(1.1 .. 5.1);     # to stejné jako (1, 2, 3, 4, 5)
(1 .. 5, 10, 15); # to stejné jako (1, 2, 3, 4, 5, 10, 15)
('a' .. 'z');     # seznam obsahující všechna malá písmena
($x .. $y);       # záleží na hodnotách proměnných $x a $y
```

Délka pole

Počet prvků v poli (tzv. délku pole) je možné získat vyhodnocením proměnné ve skalárním kontextu. Explicitně je toho možné dosáhnout s využitím funkce `scalar` jako `scalar @pole`. Stejný výsledek získáme i v jinak vzniklém skalárním kontextu.

Index posledního prvku pole je obsažen v proměnné `$#pole` (pokud indexy pole číslujeme od nuly, je tato hodnota o jedna menší, než velikost pole). Přiřazením do této proměnné se mění velikost pole. Je-li přiřazovaná hodnota větší než délka pole, je počet prvků zvýšen (doplněné prvky budou obsahovat nedefinovanou hodnotu). Je-li přiřazovaná hodnota menší než délka pole, je pole zkráceno a přebytečné prvky ztraceny. Zkrátit pole na nulovou délku je tedy možné následujícími způsoby.

```
@pole = ();
$#pole = -1;
```

Uvolnit paměť zabranou polem je možné pomocí `undef @pole`. Pouhé zkrácení pole na nulovou délku toto neučiní.

Vyhodnocování polí a seznamových literálů v různém kontextu

V seznamových literálech jsou seznamové hodnoty odděleny čárkami. Když je to vyžadováno, je tento seznam hodnot oddělen od zbytku textu kulatými závorkami. Ty slouží k tomu, aby bylo jasné, kde seznam začíná a končí. V seznamovém kontextu je hodnotou seznamového literálu posloupnost všech prvků seznamu v daném pořadí. Ve skalárním kontextu je hodnotou seznamového literálu poslední prvek seznamu.

```
@pole = ('a', 'b', 'c');
```

Seznam `@pole` obsahuje tři hodnoty v pořadí tak, jak je zadáno v kulatých závorkách.

```
$prvek = ('a', 'b', 'c');
```

Takto je seznamový literál vyhodnocen ve skalárním kontextu. Proměnná `$prvek` tedy obsahuje hodnotu `c`, což je poslední hodnota seznamového literálu na pravé straně.

Na druhou stranu je třeba odlišit případ, kdy ve skalárním kontextu vyhodnocujeme proměnnou typu pole. Tehdy je výsledkem takového výrazu nikoliv poslední prvek pole, ale jeho délka.

```
@pole = ('a', 'b', 'c');

$pocet = @pole;          # v obou případech proměnná $pocet
$pocet = scalar @pole;   # obsahuje počet prvků pole
```

Pole i seznamy mohou obsahovat libovolné výrazy vracející hodnotu. Hodnoty mohou být skalární (řetězce, čísla, odkazy) nebo seznamové. Obsahuje-li seznamový literál seznamovou hodnotu nebo proměnnou, dojde postupně ke vložení všech prvků těchto struktur do seznamu (dojde k jejich vyhodnocení v seznamovém kontextu). Seznamové proměnné tak ztrácejí svoji identitu. Prázdné seznamy jsou ignorovány.

```
@pole1 = (1, 2, 3);
@pole2 = ('a', 'b', @pole1, 'c')'
# @pole2 obsahuje prvky 'a', 'b', 1, 2, 3, 'c'
```

Ve skutečnosti jsou všechny prvky v seznamu vyhodnocovány v seznamovém kontextu pouze v tom případě, že se celý seznam vyhodnocuje v seznamovém kontextu. Je-li celý seznam vyhodnocován ve skalárním kontextu, jsou jeho jednotlivé prvky vyhodnoceny ve skalárním kontextu.

```
@a = ('a', 'b', 'c');
@x = (1, 2, 3, @a);  # @x obsajuje 1, 2, 3, 'a', 'b', 'c', pole @a
                     # bylo vyhodnoceno v seznamovém kontextu
$x = (1, 2, 3, @a);  # $x obsahuje 3, což je délka pole @a,
                     # které bylo vyhodnoceno ve skalárním kontextu
```

Přístup k prvkům seznamu

Pro přístup k jednotlivým prvkům pole se používají indexy. Indexy jsou číselné hodnoty začínající od hodnoty proměnné `$[` (implicitně `0`) a zadávají se do hranatých závorek za jméno proměnné typu pole.

Jednotlivé hodnoty pole jsou skalární hodnoty, jejich označení tedy není uvozeno znakem `@`, ale znakem `$`. Proměnná `$pole[0]` je prvním prvkem pole `@pole`, proměnná `$pole[1]` druhým atd. a proměnná `$pole[$#pole]` posledním.

K prvkům pole je možné přistupovat i od konce. K tomu se používají záporné indexy. `$pole[-1]` tedy znamená první prvek pole od konce, neboli poslední prvek tohoto pole, `$pole[-2]` prvek předposlední atd.

Nezáleží na tom, zda přistupujeme k prvkům pole nebo k prvkům seznamového literálu, v obou případech se používají hranaté závorky a číselné indexy.

```
$pole[1] = 'abc';  # nastavení prvku pole
print $pole[1];    # zjištění hodnoty prvku pole

$x = (1, 2, 3)[2];            # $x obsahuje hodnotu 3
$rok = 1900 + (localtime)[5];  # správně, musí být použity závorky
$rok = 1900 + localtime[5];    # chybí závorky
```

Číselné indexy samozřejmě nemusejí být pouze konstanty, ale i výrazy vracející hodnotu. Tyto výrazy zadané v hranatých závorkách jsou vyhodnocovány vždy ve skalárním kontextu a hodnota je brána jako číslo. Je-li to hodnota řetězcová, převede se nejprve na hodnotu číselnou.

```
@pole = (1,2,3);
$x = 1;
$pole[$x] = 'x'      # přiřazení do $pole[1]
$pole[@pole] = '4';  # přiřazení do $pole[3]
$pole['2abc'] = 3;   # přiřazení do $pole[2]
$pole['abc'] = 'y';  # přiřazení do $pole[0]
$pole[undef] = 'z';  # přiřazení do $pole[0]
```

Rozsah polí není kontrolován, protože není předem dán. Pokoušíme-li se modifikovat prvek pole, který neexistuje, tento prvek se vytvoří a případná mezera mezi

posledním prvkem původního pole a tímto novým prvkem je vyplněna nedefinovanými hodnotami. Jediným problémem je přiřazení do prvku, jehož index je menší než 0. Jinými slovy, když se pokoušíme modifikovat prvek, který je od konce pole vzdálen více než je délka pole.

Práce s více prvky seznamu nebo pole najednou

Perl umožňuje pracovat s jednotlivými prvky pole, s jeho částí i s polem jako s celkem. Jediným přiřazovacím příkazem lze jedno pole nebo seznam zkopírovat do jiného (prvky seznamu musejí být v takovém případě modifikovatelné, tzn. obsahují proměnné či jiné l-hodnoty). Přiřazování probíhá tak, že si jsou přiřazovány prvky na odpovídajících si pozicích.

```
($x, $y, $z) = (1, 2, 3);
($x, $y, $z) = (1, 2);
($x, $y)     = (1, 2, 3);
```

V prvním případě mají všechny tři proměnné hodnoty 1, 2 a 3. Ve druhém případě se provede přiřazení pouze do proměnných $x a $y. Do proměnné $z bude uložena nedefinovaná hodnota, protože nebyla nalezena odpovídající hodnota v seznamu na pravé straně přiřazení. V posledním případě budou nastaveny obě proměnné a hodnota 3 bude zapomenuta. Nebylo pro ni totiž nalezeno místo na levé straně přiřazení.

```
@pole1 = @pole2;
```

Tímto příkazem se provede zkopírování všech prvků z prvního pole do druhého. Obě pole potom mají stejnou i délku. Bylo-li první pole před přiřazením delší, přebytečné prvky jsou zapomenuty. Bylo-li kratší, je jeho délka zvětšena.

Nepotřebujeme-li některé hodnoty uložit, můžeme na levé straně přiřazení na tomto místě uvést undef.

```
(undef, undef, undef, $den, $mesic, $rok) = localtime(time);
# funkce localtime vrací seznam devíti hodnot, nás z nich
# zajímá pouze aktuální datum (den, měsíc a rok na čtvrté
# až šesté pozici)
```

Je-li na levé straně přiřazení seznamová proměnná, jsou do této proměnné uloženy všechny zbývající hodnoty na straně pravé.

```
($x, $y, @zbytek) = (1, 2, 3, 4, 5);
# $x obsahuje 1, $y obsahuje 2 a @zbytek obsahuje (3, 4, 5)
```

```
($x, @zbytek, $y) = (1, 2, 3, 4, 5);
# $x obsahuje 1, @zbytek obsahuje (2, 3, 4, 5),
# $y obsahuje nedefinovanou hodnotu
```

Přiřazením do pole či seznamu vzniká na pravé straně přiřazovacího operátoru seznamový kontext, čehož je možné využít k nepřímému vynucení seznamového kontextu.

```
@pole = qw(a b c);
$x = (1, 2, @pole);         # literál vyhodnocen ve skalárním
                            # kontextu, $x obsahuje 3
$x = @x = ( 1, 2, @pole);   # literál vyhodnocen v seznamovém
                            # kontextu, $x obsahuje 5 (délka @x)
$x = (@x) = ( 1, 2, @pole); # to samé
$x = () = ( 1, 2, @pole);   # to samé, hodnoty zapomenuty
```

Hodnota přiřazení seznamu je počet prvků na pravé straně tohoto přiřazení, proto v posledních třech případech byla hodnota proměnné `$x` vždy `5`. Byla to totiž délka seznamového literálu na pravé straně a počet skutečně přiřazených hodnot se nebral v úvahu.

Tohoto je možné využít v logickém kontextu, kdy se testuje návratová hodnota funkce vracející seznam. Výraz bude nepravdivý, když funkce vrátí prázdný seznam, jinak bude pravdivý. Vrátí-li funkce hodnotu `0`, je ve skalárním kontextu nepravdivá, ale v seznamovém kontextu je to jednoprvkový seznam.

```
while (($hodnota) = dej_hodnotu ) {
   print $hodnota;
}
```

Kdybychom napsali pouze `$hodnota = dej_hodnotu`, celý výraz by byl nepravdivý v případě, že by funkce vrátila některou z nepravdivých hodnot (hodnotu `0`, `""`, `"0"` či `undef`).

Přiřazením do seznamu je možné vynutit volání funkce v seznamovém kontextu.

```
funkce;          # vyhodnocení funkce v prázdném kontextu,
                 # což je případ skalárního kontextu
(@x) = funkce;   # volání funkce v seznamovém kontextu
() = funkce;     # to samé, návratové hodnoty zapomenuty
```

Úseky polí

S několika prvky pole současně lze pracovat jako s tzv. *úseky* následujícími způsoby:

```
($den, $mesic, $rok) = (localtime)[3, 4, 5];
@pole[5..7] = (5, 6, 7);
foreach (@pole[5..10]) { print };
```

Zápis úseku pole vypadá jako zápis prvku pole, ale na začátku je symbol `@`. Nejedná se totiž o skalární hodnotu, ale o seznamovou hodnotu. S úseky polí se pracuje stejně, jako by to byla normální pole. Napíšeme-li `@pole[1]` (jednoprvkový seznam) a mysleli jsme tím `$pole[1]`, neuděláme velkou chybu. Pouze pokud bychom provedli tuto záměnu na levé straně přiřazovacího příkazu, rozdíl bude v tom, že v prvním případě se pravá strana vyhodnotí ve skalárním kontextu, zatímco ve druhém případě v kontextu seznamovém.

```
@pole = (1,2,3);
$pole[1] = @pole;  # přiřadí se délka @pole
@pole[1] = @pole;  # přiřadí se první prvek @pole
```

Funkce pro práci s poli

Pro práci se seznamy, poli a jejich prvky je v Perlu předdefinována řada funkcí. Pomocí funkce `push` je možné na konec zadaného pole přidat zadaný seznam prvků. Pomocí funkce `unshift` je možné provádět to samé, ale prvky budou přidány na začátek. Délka pole se o daný počet prvků zvýší a je vrácena jako návratová hodnota těchto funkcí.

```
@pole = (1, 2, 3);
print push @pole, 10;    # vytiskne 4
                         # @pole obsahuje 1, 2, 3, 10
print unshift @pole, 20; # vytiskne 5
                         # @pole obsahuje 20, 1, 2, 3, 10
```

Pro odebrání jednoho prvku z pole slouží funkce `pop`, která odebere a vrátí prvek z konce pole, a funkce `shift`, která odebere a vrátí prvek pole ze začátku. V obou případech se tak délka pole zkrátí o jeden prvek.

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

print pop @pole;   # vytiskne 5,
                   # @pole obsahuje 1, 2, 3, 4
print shift @pole; # vytiskne 1,
                   # @pole obsahuje 2, 3, 4
```

Díky existenci těchto funkcí a vlastnostem polí v Perlu lze bez velké námahy implementovat datové struktury fronta a zásobník. Dvojice funkcí `push` a `pop` nebo

`unshift` a `shift` slouží pro práci s polem jako se zásobníkem, dvojice `push` a `shift` nebo `unshift` a `pop` umožňují s polem pracovat jako s frontou.

Funkce `splice` umí pracovat s více prvky pole najednou a tyto prvky se nemusejí nutně nacházet pouze na začátku nebo na konci pole. Funkce ze zadaného pole vezme zadaný počet prvků na zadané pozici a nahradí je zadaným seznamem hodnot. Jsou-li hodnoty udávající počet prvků, které se mají nahradit, nebo počet prvků, které se mají vložit, nulové, dochází pouze k vložení či odebrání prvků z pole.

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

splice @pole, 2, 2, 'a', 'b', 'c';
# z pole @pole odebereme dva prvky (3 a 4) od indexu 2
# a vložíme hodnoty 'a', 'b', 'c'. Pole obsahuje
# hodnoty 1, 2, 'a', 'b', 'c', 5

splice @pole, 3, 1;
# z pole @pole odebereme od indexu 3 jeden prvek ('b')
# a ničím nenahrazujeme. Pole obsahuje
# hodnoty 1, 2, 'a', 'c', 5

splice @pole, 1, 0, 'x', 'y';
# z pole @pole od indexu 1 odstraníme 0 prvku
# a vložíme prvky 'x' a 'y'. Pole obsahuje
# hodnoty 1, 'x', 'y', 2, 'a', 'c', 5
```

Pomocí funkce `splice` je možné s polem pracovat jako s lineárně zřetězeným seznamem.

Pokud chceme seřadit všechny prvky pole, nemusíme vymýšlet vlastní algoritmus (nebo sami implementovat algoritmus již vymyšlený), ale stačí pouze použít vestavěnou funkci `sort`. Ta jako argument očekává seznamovou hodnotu a vrací nový seznam, kde jsou všechny původní prvky seřazeny podle abecedy vzestupně. Díky tomu, že tato funkce umožňuje předefinovat implicitní způsob řazení, je možné implementovat i řazení sestupné, číselné a dokonce i vícekriteriální. Podrobnosti lze nalézt v kapitole *10. Standardní funkce* u popisu funkce `sort`.

Pro postupné zpracování prvků pole se velmi často používá cyklus `foreach` (je možné použít i zkrácený zápis `for`). Ten v jednotlivých iteracích postupně nastavuje řídicí proměnnou cyklu na jednotlivé prvky zadaného seznamu a provádí vlastní tělo cyklu. Pokud není jméno řídící proměnné zadáno, implicitně se použije proměnná `$_`.

```
foreach (@pole) {
   zpracuj_hodnotu($_);
}
```

```
for $x (1..10) {
   print "$x * $x = ", $x*$x, "\n";
}
```

Prvky seznamu v opačném pořadí vrací funkce `reverse`.

Pokud máme řetězec a chceme ho rozdělit na části podle nějakého vzoru, můžeme použít funkcí `split`. Jako argumenty jí zadáme vzor, podle kterého má dojít k rozdělení řetězce, a vlastní řetězec.

```
$radek = <>;   # na řádku předpokládáme jméno, příjmení
               # a telefonní číslo oddělené mezerou
($jmeno, $prijmeni, $telefon) = split ' ', $radek;
```

Opačný proces provádí funkce `join`. Ta jako argument očekává řetězec, kterým spojí prvky seznamu tvořící zbytek argumentů.

```
print join ', ', @hodnoty;  # spojí hodnoty čárkou a vytiskne
```

Pro zjištění, zda pole obsahuje určitou hodnotu můžeme použít funkci `grep`.

```
if (grep {$_ < 0} @pole) {
   print 'Pole obsahuje alespoň jedno záporné číslo.'
}
```

Pro získání pole, u něhož jsou položky modifikovány zadaným způsobem, slouží funkce `map`.

```
@slova_velkymi_pismemy = map {uc} @slova;
```

Některé další funkce, které mohou být užitečné pro práci s poli a seznamy, ale nepatří zatím mezi standardní funkce, jsou definovány ve standardním modulu `List::Util`.

4.3 Hashe (asociativní pole)

Hash je seznamová datová struktura, kde se k jednotlivým prvkům přistupuje pomocí řetězcových indexů zvaných *klíče* (hodnoty jsou asociovány ke klíčům, odtud název asociativní pole). Vnitřně je hash implementován pomocí rozptýlených tabulek a vyhledává se v něm pomocí mapovací funkce. Prvky tedy nemají přesně dané pořadí, jako je tomu u polí, a při postupném získávání hodnot z hashe tyto nebudou pravděpodobně vraceny přesně v tom pořadí, v jakém byly do hashe ukládány.

Jako klíče hashe se vždy používají řetězce nebo výrazy vracející řetězcovou hodnotu (i prázdný řetězec či řetězec s bílými znaky). Klíč je v rámci jednoho hashe jedinečný, a nelze tak uložit dvě nebo více dvojic se stejným klíčem. Při pokusu o přiřazení na již použitou klíčovou pozici se přepíše původní hodnota hodnotou novou.

Klíče hashe nemohou být tvořeny pevnými odkazy.* Při pokusu o použití pevného odkazu jako klíče hashe vždy nejprve dojde k převedení pevného odkazu na řetězcovou reprezentaci (řetězec je pak ve tvaru `SCALAR(0x1ac29f4)`) a teprve tento řetězec se následně použije jako klíč. Zpětná dereference této hodnoty není možná — při pokusu o dereferenci takto vzniklé hodnoty obdržíme varování, protože pracujeme s hodnotou, která vlastně není odkazem.

Hodnoty hashe jsou skalární hodnoty a nemají žádné omezení. Mohou tedy obsahovat čísla, řetězce i pevné odkazy bez jakýchkoliv komplikací.

Vytváření hashů, hashové literály

Hashový literál vypadá stejně jako seznamový literál. Ten by měl obsahovat sudý počet prvků (klíče a hodnoty tvoří dvojice). První hodnota bude brána jako klíč, druhá jako hodnota asociovaná s tímto klíčem, další hodnota opět jako klíč atd.

```
%dny = (’pondeli’, 1 , ’utery’, 2, ’streda’, 3,
        ’ctvrtek’, 4, ’patek’, 5);
```

Tímto příkazem jsme vytvořili hash `%dny` s pěti dvojicemi klíč/hodnota. K řetězci `’pondeli’` je asociována hodnota `1` atd.

Místo operátoru čárka lze použít operátor `=>`, který je synonymem, ale provádí jednu věc navíc. Řetězec, který se nachází nalevo od tohoto operátoru, je automaticky považován za řetězec v uvozovkách. Ušetříme si tedy nějaké psaní a navíc je definice hashe pomocí tohoto operátoru čitelnější a vztah mezi klíčem a hodnotou zřejmější.

```
%dny = (pondeli => 1,
        utery => 2,
        streda => 3,
        ctvrtek => 4,
        patek => 5);
```

Operátor `=>` se vztahuje vždy k bezprostředně předcházejícímu holému slovu, což znamená, že má-li hashový klíč obsahovat nějaký znak, jenž je chápán jako oddělovač slov (mezera, nový řádek) či má jiný speciální význam ($, %, # apod.), musíme explicitně uvést uvozovky nebo apostrofy.

*Toto omezení můžeme obejít navázáním hashe na balík `Tie::RefHash`.