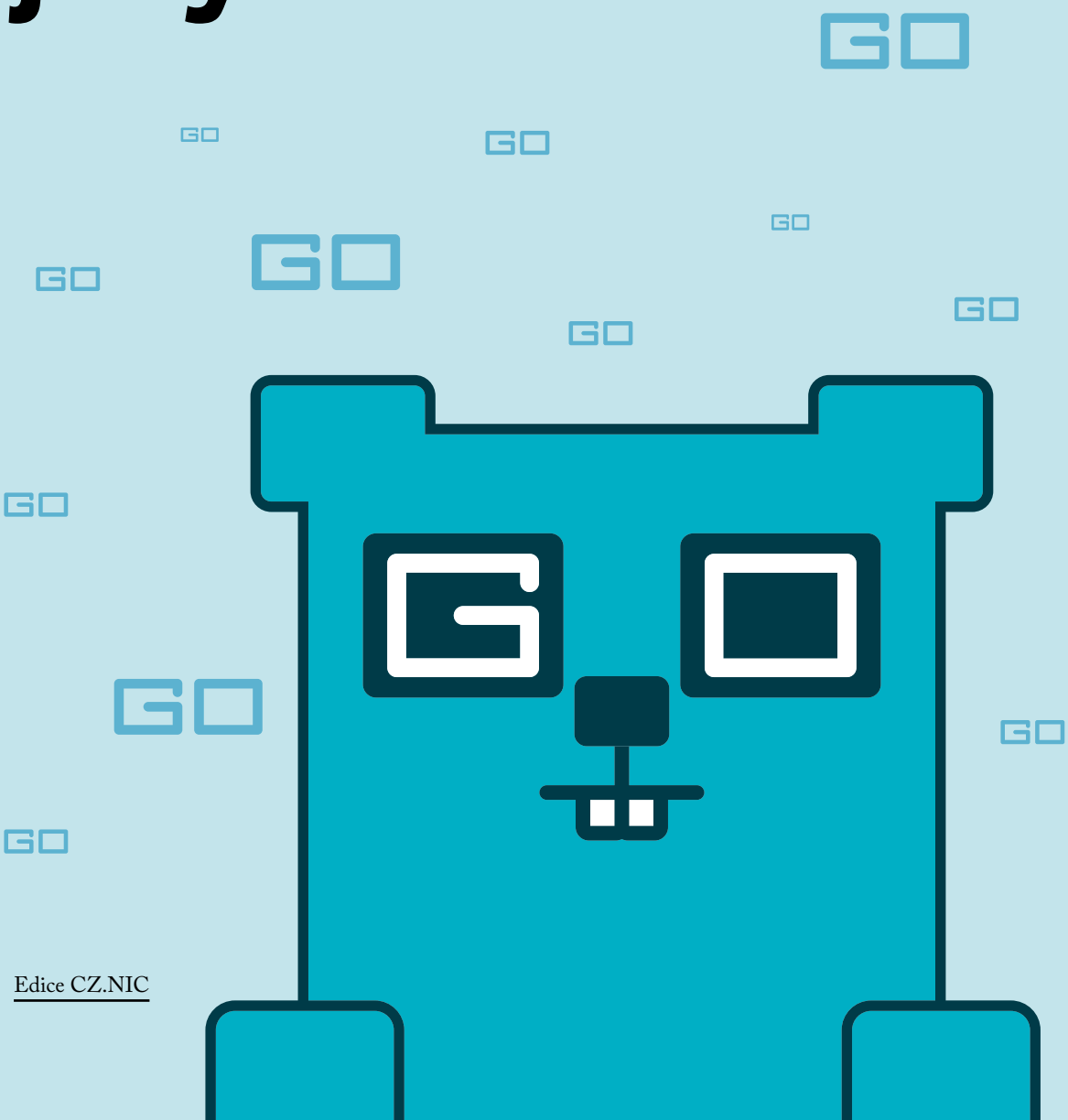
Pavel Tišnovský

Programovací jazyk GO



Edice CZ.NIC

PROGRAMOVACÍ JAZYK GO

Pavel Tišnovský

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

1. vydání, Praha 2025
Kniha vyšla jako 34. publikace v Edici CZ.NIC.
ISBN 978-80-88168-83-6

ISBN 978-80-88168-83-6

— Pavel Tišnovský

Programovací jazyk GO

— Edice CZ.NIC

Předmluva vydavatele

Předmluva vydavatele

Po knize Evoluce Pythonu, která nabídla pohled na vývoj jednoho z nejrozšířenějších programovacích jazyků, přichází autor s novým titulem zaměřeným na jazyk Go. Tentokrát se soustředí na praktický vývoj backendových aplikací, ale opět zůstává u stylu, který jde dál než jen k samotným pravidlům jazyka: snaží se postihnout širší kontext a vývojové souvislosti.

Přestože jazyk Go a Python jsou v mnoha ohledech odlišné, autor přistupuje k tématu podobně – věcně, s důrazem na srozumitelnost a využitelnost v praxi. Text není psán jako učebnice, ale spíš jako průvodce pro vývojáře, kteří hledají praktické odpovědi i prostor pro vlastní úvahy.

Jsme rádi, že můžeme tuto knihu nabídnout čtenářům, kteří se o jazyk Go zajímají nebo s ním již pracují, a zároveň si cení otevřeného pohledu, který autor k tématu přináší.

Za celý tým Edice CZ.NIC přejeme příjemné čtení.
Praha, 19. června 2025

Předmluva

Předmluva

V současnosti se mohou vývojáři setkat s celou řadou programovacích jazyků. Tyto aktivně používané programovací jazyky se od sebe v mnoha oblastech odlišují a taktéž se nasazují v různých oblastech IT. Mezi jeden z nejpopulárnějších programovacích jazyků současnosti patří i programovací jazyk nazvaný Go, s jehož možnostmi se podrobněji seznámíme v této knize. Jedná se o výkonný kompilovaný a taktéž o silně typovaný programovací jazyk, který je navržen takovým způsobem, že základy tohoto jazyka je možné se naučit doslova za několik hodin a již za přibližně týden či čtrnáct dní je v něm možné tvořit aplikace určené pro produkční nasazení. V současnosti tvoří aplikace a nástroje vyvinuté v jazyku Go jádro většiny komponent ekosystému tzv. kontejnerů. V Go je naprogramován například Kubernetes, OpenShift, Podman, Docker, Prometheus, ale i další populární nástroje, mezi které patří NATS, MinIO či NSQ.

Obsah

1 Úvod

1 Úvod

V současnosti se můžeme ve vývojářské praxi setkat s celou řadou programovacích jazyků. Tyto aktivně používané programovací jazyky se od sebe pochopitelně v mnoha oblastech odlišují a taktéž se nasazují v různých oblastech IT (některé na frontendu, další spíše na backendu, v oblasti datové analýzy atd.). Mezi jeden z nejpopulárnějších programovacích jazyků současnosti patří i programovací jazyk nazvaný Go, s jehož možnostmi se podrobněji seznámíme v této knize.

Jedná se o výkonný kompilovaný (překládaný) a taktéž o silně typovaný programovací jazyk, který je navržen takovým způsobem, že základy tohoto jazyka je možné se naučit doslova za několik hodin a již za přibližně týden či čtrnáct dní je v něm možné tvořit aplikace určené pro produkční nasazení. Taktéž se jedná o jazyk, jenž je určen pro vývoj služeb (resp. možná přesněji řečeno mikroslužeb) popř. nástrojů ovládaných z příkazového řádku. V současnosti tvoří aplikace a nástroje vyvinuté v jazyku Go jádro většiny komponent ekosystému tzv. kontejnerů (*containers*). Konkrétně to znamená, že v Go je naprogramován například Kubernetes, OpenShift, Podman, Docker, Prometheus, ale i další populární nástroje, mezi které patří NATS, Minio či NSQ.

1.1 Autoři jazyka Go

Go při svém představení vzbudil ohlas nejenom kvůli svým technickým vlastnostem, ale i díky tomu, že pochází z „dílny" Google a taktéž proto, že za jeho designem stojí mj. i dvě slavné osobnosti z oblasti informatiky: Rob „Commander" Pike a Ken Thompson.

1.2 Go nebo Golang?

Oficiální název tohoto programovacího jazyka je Go. To je ovšem poměrně nešťastně zvolené jméno, protože Go je jak název známé deskové hry, tak i běžné slovo v angličtině. To způsobuje problémy při vyhledávání dalších informací o tomto jazyku a mj. i z tohoto důvodu se poměrně často setkáme i s označením Golang (Go language). V této knize se ovšem budeme držet oficiálního jména Go.

2 Jazyk Go v současnosti

2 Jazyk Go v současnosti

V současnosti nalezneme aplikace a služby psané v jazyce Go především na serverech, tedy na *back endu*. V Go je naprogramována poměrně velká část služeb souvisejících s kontejnery, ale taktéž některé databáze (například CockroachDB), různé formy message brokerů atd. Pro zajímavost se o některých těchto technologiích alespoň ve stručnosti zmíníme.

2.1 Podman

Podman neboli *pod manager* je nástroj určený pro správu kontejnerů. Využívá knihovnu `libpod` a nabízí rozhraní (API) pro správu životního cyklu kontejnerů, jednotlivých podů, obrazů (image) i svazků. Důležité je, že rozhraní Podmanu je kompatibilní s rozhraním známého systému Docker, takže administrátoři, kteří znají Docker, mohou na Podmana snadno přejít. A podobně jako Docker nabízí *Docker Desktop* je možné v případě systému Podman využívat *Podman Desktop*.

2.2 Kubernetes

Kubernetes je systém určený pro takzvanou *orchestraci* virtualizace prováděnou na úrovni operačního systému. Původně jej vyvinula společnost Google (ta stála i za vývojem samotného Go) a jako podřízené nástroje podporuje například známý Docker ale i výše zmíněný Podman. S využitím Kubernetes je možné spouštět vybrané a nakonfigurované úlohy v takzvaných kontejnerech. Díky tomu se velmi často používá v cloudových řešeních, datových centrech atd.

2.3 MinIO

Velmi úspěšným projektem napsaným v jazyce Go je MinIO. Jedná se o sadu několika služeb a nástrojů, které uživatelům poskytují distribuované datové úložiště určené pro ukládání obecných (nestrukturovaných) dat. Typicky se jedná o soubory používané v oblasti AI (Artifical Intelligence) a ML (Machine Learning), ovšem kromě těchto populárních (a vlastně do značné míry i módních) oblastí IT je pochopitelně možné službu MinIO použít i pro ukládání logů, souborů, k nimž je zapotřebí rychle přistupovat z mnoha různých, mnohdy vzájemně vzdálených oblastí (zde využijeme možnost distribuovaného systému), jako centrální úložiště dokumentů, obrázků, videí, pochopitelně i obrazů souborových systémů pro Docker apod. MinIO dosahuje velmi slušné rychlosti přístupu k datům (při vhodně nadimenzované síti, která je většinou limitujícím faktorem) a mj. i díky velmi dobré stabilitě ukazuje přednosti programovacího jazyka Go, ve kterém je celý systém naprogramován.

Jedním z nejdůležitějších a v důsledku i nejpraktičtějších vlastností projektu MinIO je fakt, že se pro přístup k datům používá stejná technologie, jaká je implementována i v populární službě

Amazon S3 či možná přesněji AWS S3. To mj. znamená, že dodávaný MinIO Client SDK může sloužit jak pro přístup k datům uloženým v Miniu, tak i k datům uloženým ve cloudu na S3. Díky tomu lze například snadněji nastavit konfiguraci pro vývoj, konfiguraci CI, zajistit si možnost využití veřejného cloudu (S3) nebo naopak privátního cloudu (založeného na Miniu) atd.

2.3.1 NATS

Systém NATS patří mezi takzvané message brokery, což jsou systémy určené pro doručování, ukládání a redistribuci zpráv. NATS je poměrně úspěšný projekt, jenž je vyvinut právě v programovacím jazyce Go, což mu do značné míry zajišťuje stabilitu i škálovatelnost. To jsou ostatně vlastnosti, které u message brokerů většinou očekáváme. NATS v současnosti poměrně zdárně konkuruje dalším message brokerům, mezi které patří například RabbitMQ, ActiveMQ (Artemis), IBM MQ atd.

2.3.2 NATS JetStream

Součástí projektu NATS je i technologie nazvaná NATS JetStream. S jejím využitím lze realizovat takzvaný streaming zpráv s podobnými vlastnostmi, jaké nalezneme u „konkurenčního" projektu Apache Kafka. NATS JetStream používá klasický systém (server) NATS, který navíc doplňuje o takzvaný storage, tj. o technologii určenou pro ukládání zpráv (někdy nazývaných i záznamy – record) do perzistentního úložiště, kterým může být relační databáze či soubor resp. skupina souborů.

2.3.3 NSQ

V jazyce Go je vytvořen i další message broker nazvaný nsq. Ten se skládá z několika samostatně spouštěných uzlů, které mezi sebou vhodným způsobem komunikují. Výhodou a jednou ze základních vlastností nsq je fakt, že způsob jeho návrhu počítá s tím, že jednotlivé uzly mohou být po nějakou dobu nedostupné a přitom systém jako celek by měl být stále použitelný. Jedná se tedy o architekturu vhodnou pro nasazení do dnešního hybridní prostředí se systémy běžícími v cloudu, lokálně v kontejneru, na dedikovaných serverech atd.

2.3.4 FZF

Jedním z nástrojů psaných v Go, které se těší poměrně značné popularitě, je utilita nazvaná *fzf* neboli „command-line fuzzy finder". Základní funkce *fzf* je ve skutečnosti velmi jednoduchá a vlastně i typicky „unixovská". Nástroj *fzf* totiž na svém standardním vstupu (*stdin*) získá libovolný seznam, který následně zobrazí a nechá uživatele vybrat jednu položku z tohoto seznamu. Výsledek následně předá na svůj standardní výstup (*stdout*) pro další zpracování. To pravděpodobně nezní

moc bombasticky ani příliš užitečně, ovšem samotný výběr položky ze seznamu je plně interaktivní a podporuje takzvaný „fuzzy“ výběr (snadno použitelný je i pro ty uživatele, kteří neznají regulární výrazy). A především – největší síla *fzf* spočívá v tom, že tento nástroj je možné kombinovat s vybranými standardními příkazy popř. dalšími utilitami, všude tam, kde je nutné vybrat hodnotu ze seznamu (platného například v nějakém kontextu).

3 Go a další programovací jazyky

3 Go a další programovací jazyky

Programovací jazyk Go a jeho základní knihovny byly původně navrženy takovým způsobem, aby se v Go daly efektivně psát různé síťové aplikace, které by ideálně neměly trpět potenciálními výkonnostními problémy nebo bezpečnostními dírami typu „buffer overflow" atd. Právě z toho důvodu, že síťové a serverové aplikace musí v typické konfiguraci obsluhovat velké množství souběžně prováděných požadavků, byly do jazyka Go přidány již v úvodní kapitole zmíněné *gorutiny* a *kanály*. A nutno říci, že Go se v této oblasti skutečně velmi často používá. Taktéž se předpokládalo, že se jazyk Go bude používat pro vývoj těch aplikací, které jsou v současnosti psány v jazycích C či C++ (možná i v Rustu). Tento přechod sice skutečně částečně nastal, ovšem prozatím ne v příliš velké míře (a to se bavíme o použití v serverových a desktopových aplikacích, protože v oblasti mikrořadičů se s jazykem Go sice laboruje, ale především u relativně výkonných MCU typicky založených na čipech Cortex-M, nikoli na malých osmibitových a šestnáctibitových mikrořadičích).

3.1 Od Pythonu ke Go?

Několik let po představení tohoto jazyka však nastala zajímavější situace, o níž jsme se opět krátce zmínili v úvodu – programovací jazyk Go a jeho relativní snadnost použití a poskytované možnosti objevili vývojáři, kteří psali své síťové a speciálně pak webové aplikace v Pythonu, Ruby či JavaScriptu/TypeScriptu. Přepis takových aplikací do jazyka Go mnohdy znamenal řádový a někdy i dvouřádový (tedy 100× a více) nárůst výkonu těchto aplikací. Do jisté míry je nárůst výkonu způsobem překladem do nativního kódu, ovšem nesmíme zapomenout na gorutiny, které nejsou (na rozdíl od klasických vláken) příliš náročné na paměť, takže se můžeme setkat s aplikacemi, v nichž bez větších problémů běží stovky či dokonce tisíce gorutin (což si ostatně ještě ukážeme v prakticky zaměřené části této knihy).

3.2 Go z pohledu vývojáře

Go je staticky typovaný programovací jazyk a programy, které jsou v něm napsány, jsou kompilovány (překládány) do nativního kódu (někdy se setkáme s přesnějším označením *strojový kód* neboli *machine code*). Použití statických datových typů vede k tvorbě bezpečnějšího kódu a společně s výše uvedeným konceptem překladu do strojového kódu jsou výsledné aplikace velmi rychlé, zejména v porovnání s aplikacemi naprogramovanými například v Pythonu či v dalších interpretovaných jazycích.

Programovací jazyk Go navíc programátorům nabízí i koncept výše zmíněných *gorutin* a kanálů používaných při komunikaci mezi gorutinami. Díky tomu lze velmi snadno a především bezpečně psát aplikace, jejichž části běží souběžně (*concurrently*) či dokonce paralelně.

3.3 Go není „pouze" vylepšený programovací jazyk C

Mezi autory návrhu i první implementace programovacího jazyka Go patří mj. i Ken Thompson. Když si uvědomíme, že Ken vytvořil i (dnes prehistorický) programovací jazyk B, který je předchůdcem slavného jazyka C, mohlo by se zdát, že Go bude pouhým vylepšením programovacího jazyka C (například nějaká obdoba jazyka C++). Je sice pravda, že Go alespoň částečně z programovacího jazyka C skutečně vychází, ovšem z hlediska syntaxe jsou si mnohem bližší například jazyky C a Java, než C a Go.

Tvůrci programovacího jazyka Go se navíc snažili poučit se z některých problematických rysů jazyků C a taktéž C++. Z tohoto důvodu navrhli Go takovým způsobem, aby byla jeho syntaxe jednodušší a snáze pochopitelná (sami se po prohlédnutí zdrojových kódů můžete rozhodnout, do jaké míry se to splnilo). Samotný jazyk je velmi konzervativní a někteří kritici dokonce (s nadsázkou) tvrdí, že zcela ignoruje všechny novinky, které se na poli programovacích jazyků za posledních dvacet let objevily.

3.4 Automatická správa paměti

Navíc se tvůrci jazyka Go zaměřili i na další typické problémy, s nimiž se musí vypořádat prakticky všichni programátoři používající jazyky C, C++ a vlastně i uživatelé Rustu (ti nepřímo): jedná se o problematiku správy paměti a uvolňování zdrojů (*resources*) a v neposlední řadě i o podporu pro paralelní běh částí programů, například částí komunikujících přes síť, pracujících se souborovým systémem atd. Z tohoto důvodu byly přímo do jazyka Go přidány dvě důležité technologie: automatický správce paměti a již výše zmíněné *gorutiny* (*goroutines*). S oběma těmito technologiemi se podrobněji seznámíme v navazujících kapitolách.

Poznámka: na tomto místě je vhodné si uvědomit, že použití automatického správce paměti přenáší část problémů (které by jinak musel řešit programátor) do takzvaného runtime*, což se může negativně projevit na výkonu aplikace (zpomalení aplikace, větší nároky na kapacitu haldy, musí se počítat i s okamžiky, kdy je aplikace pozastavena apod.). Vlivem správce paměti na výkon se budeme zabývat v samostatné kapitole, takže zde jen bez dalších důkazů uveďme, že automatický správce paměti implementovaný v jazyku Go nemá větší negativní vliv na dobu startu aplikací (ostatně, nemusí se například spouštět ani virtuální stroj ani inicializovat interpret), spotřeba paměti je však průměrně větší, než v případě C/C++, ale na druhou stranu menší, než je tomu při porovnání s obdobnými aplikacemi psanými v Javě.*

3.5 Minimalisticky pojatý programovací jazyk

Některé technologie naopak do jazyka Go přidány *nebyly*, a to z toho důvodu, aby byl jazyk snadno použitelný a navíc i snadno implementovatelný. Právě fakt, že se v Go *zpočátku* neobjevily dnes již široce akceptované technologie, jako je obsluha výjimek či práce s generickými datovými typy, byl poměrně často kritizován; ovšem Go se postupně mění a přidávají se do něj další vlastnosti. Ovšem tyto nové vlastnosti jsou přidávány relativně pomalu a poměrně konzervativním způsobem, což je podle názoru autora této knihy v této oblasti jen dobře. Poslední velkou změnou je přidání podpory pro generické datové typy.

4 Instalace překladače jazyka Go i dalších podpůrných nástrojů

4 Instalace překladače jazyka Go i dalších podpůrných nástrojů

I když existují různá webová „pískoviště“, ve kterých je možné si vyzkoušet základy programovacího jazyka Go, je pro výuku tohoto jazyka výhodnější přímá instalace všech základních nástrojů Go, tj. jak vlastního překladače, tak i nástrojů pro naformátování zdrojových kódů, instalaci modulů, prohlížení nápovědy atd. Instalace Go se pochopitelně liší podle toho, jaký operační systém používáte. Z tohoto důvodu je popis instalace rozdělen do několika částí – instalace Go na systému Microsoft Windows a instalace na vybraných distribucích Linuxu.

4.1 Instalace v operačním systému Microsoft Windows

Instalace nástrojů jazyka Go pro operační systém Microsoft Windows je snadná a poměrně rychlá. Postačuje přejít na stránku https://go.dev/. Kromě dalších informací zde nalezneme tlačítka *Get Started* a *Download*. Stiskneme tlačítko *Download* a po přechodu na další stránku v sekci nazvané *Featured downloads* (zhruba v polovině obrazovky) nalezneme i instalátor nástrojů Go pro systém Microsoft Windows. V době psaní této knihy se jedná o soubor `go1.24.2.windows-amd64.msi`, ovšem číslo verze se pochopitelně bude postupně zvyšovat. Tento soubor je nutné stáhnout a spustit. Další kroky instalace jsou snadné – instalátor se spustí, ponechají se všechna výchozí nastavení a tlačítkem `Next` se postupně instalace dokončí. Následně si spusťte konzoli (postačuje `cmd.exe`) a v ní napište příkaz:

```
go
```

V případě, že instalace proběhla v pořádku, měla by se vypsat nápověda k nástrojům jazyka Go.

4.2 Instalace v operačním systému Linux

Instalace jazyka Go v operačním systému Linux se liší v závislosti na tom, jakou konkrétní distribuci používáte a jaký má tato distribuce systém pro správu balíčků. V současnosti se používá několik správců balíčků, ovšem nejčastěji se pravděpodobně setkáte se systémem nazvaným *yum* (*Yellowdog Updater Modified*) resp. jeho následovníkem *dnf*, nebo s nástrojem *apt* (*Advanced Package Tool*). Z tohoto důvodu byly pro účely popisu instalace Go vybrány dva typy distribucí Linuxu. Do první skupiny patří Fedora, do skupiny druhé pak Ubuntu a Linux Mint.

4.2.1 Instalace do distribuce Fedora

Popišme si nyní instalaci základních nástrojů programovacího jazyka Go do distribuce Fedora. Instalace bude na všech třech posledních vydáních Fedory prakticky totožná, takže se zaměřme

na Fedoru 40, pro niž byl vytvořen balíček s Go verze 1.23.8 (což je v současnosti stabilní verze tohoto jazyka). Instalaci provedeme buď přímo z terminálu s přihlášeným superuživatelem (rootem) příkazem:

```
dnf install golang
```

Alternativně samozřejmě můžeme použít instalaci s využitím nástroje `sudo`, pokud má ovšem přihlášený uživatel příslušná práva:

```
sudo dnf install golang
```

V obou případech by měla instalace proběhnout prakticky totožným způsobem:

```
================================================================================================
 Package                         Architecture    Version              Repository        Size
================================================================================================
Installing:
 golang                          x86_64          1.23.8-1.fc40        updates          667 k
Installing dependencies:
 go-filesystem                  x86_64          3.5.0-1.fc40         fedora           8.8 k
 golang-bin                      x86_64          1.23.8-1.fc40        updates           27 M
 golang-src                      noarch          1.23.8-1.fc40        updates           13 M
 libserf                         x86_64          1.3.10-5.fc40        fedora            59 k
 subversion-libs                 x86_64          1.14.4-1.fc40        updates          1.5 M
Installing weak dependencies:
 mercurial                       x86_64          6.7.4-3.fc40         updates          6.4 M
 python3-fb-re2                  x86_64          1.0.7-15.fc40        fedora            23 k
 subversion                      x86_64          1.14.4-1.fc40        updates          1.0 M

Transaction Summary
================================================================================================
Install  9 Packages

Total download size: 50 M
Installed size: 239 M
Is this ok [y/N]:
```

Po odpovědi „Y“ se instalace rozběhne a pochopitelně se nijak zásadně neliší od instalace jakéhokoli jiného balíčku:

```
Downloading Packages:
(1/9): go-filesystem-3.5.0-1.fc40.x86_64.rpm                                  113 kB/s | 8.8 kB
00:00
...
...
...
Running transaction
  Running scriptlet: golang-1.23.8-1.fc40.x86_64                               1/1
...
...
...
Installed:
  go-filesystem-3.5.0-1.fc40.x86_64     golang-1.23.8-1.fc40.x86_64        golang-
bin-1.23.8-1.fc40.x86_64
  golang-src-1.23.8-1.fc40.noarch       libserf-1.3.10-5.fc40.x86_64
mercurial-6.7.4-3.fc40.x86_64
  python3-fb-re2-1.0.7-15.fc40.x86_64   subversion-1.14.4-1.fc40.x86_64    subversion-
libs-1.14.4-1.fc40.x86_64

Complete!
```

Povšimněte si, že se po nainstalování rozbalí přibližně 239 MB dat, což je sice poměrně velká hodnota, ovšem je vhodné si uvědomit, že se kromě samotného překladače jazyka Go nainstalují i další podpůrné nástroje (formátovač zdrojového kódu atd.), celá základní knihovna Go, zdrojové kódy základní knihovny (ty slouží pro čtení dokumentace) atd. Samotný ekosystém jazyka Go se všemi svými nástroji vyžaduje necelých 300 MB diskového prostoru.

4.2.2 Ubuntu a Mint

V distribucích Ubuntu a Mint se jako správce balíčků používá nástroj *apt* (*Advanced Package Tool*). Instalaci Go můžeme provést buď jako administrátor (root) příkazem:

```
apt install golang
```

nebo lze, jako na Fedoře, použít příkaz `sudo`, ovšem za předpokladu, že jste součástí skupiny uživatelů, kteří mohou získat administrátorská práva:

```
sudo apt install golang
```

Průběh instalace probíhá zhruba následovně:

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
golang-1.23 golang-1.23-doc golang-1.23-go golang-1.23-src golang-doc
golang-go golang-src
Suggested packages:
bzr | brz mercurial subversion
The following NEW packages will be installed:
golang golang-1.23 golang-1.23-doc golang-1.23-go golang-1.23-src
golang-doc
golang-go golang-src
0 upgraded, 8 newly installed, 0 to remove and 623 not upgraded.
Need to get 82,5 MB of archives.
After this operation, 438 MB of additional disk space will be used.
```

4.2.3 Lokální instalace

V případě, že nemáte práva administrátora (a ani možnost použít příkaz `sudo`) lze instalaci nástrojů jazyka Go provést i lokálně – pouze pro přihlášeného uživatele. Opět přejděte na stránku https://go.dev/, na níž nalezneme tlačítka *Get Started* a *Download*. Stiskneme tlačítko *Download* a po přechodu na další stránku v sekci nazvané *Featured downloads* nalezneme i balíček s Go pro Linux (takzvaný *tarball*). V době psaní této knihy se jedná o soubor se jménem `go1.24.2.linux-amd64.tar.gz`. Ten stáhneme a přesuneme do vhodného adresáře, například do domovského adresáře uživatele. V tomto adresáři stačí otevřít terminál (konzoli) a napsat příkaz pro rozbalení staženého *tarballu*:

```
tar xvfz go1.24.2.linux-amd64.tar.gz
```

Po rozbalení vznikne adresář nazvaný `go`. Nastavte si cestu do podadresáře `go/bin`:

```
PATH=go/bin:$PATH
```

Nyní by měl být dostupný příkaz `go` i další příkazy ekosystému jazyka Go.

4.3 Kontrola, zda je Go nainstalován korektně

Po instalaci základních nástrojů programovacího jazyka Go je vhodné si otestovat, jestli jsou tyto nástroje skutečně použitelné. Většina nabízených funkcí je dostupná přes „univerzální“ příkaz `go`. V případě, že na terminál (konzoli) zapíšeme pouze tento příkaz bez dalších doplňujících informací, měla by se vypsat krátká nápověda:

```
Go is a tool for managing Go source code.

Usage:

    go <command> [arguments]

The commands are:

    bug         start a bug report
    build       compile packages and dependencies
    clean       remove object files and cached files
    doc         show documentation for package or symbol
    env         print Go environment information
    fix         update packages to use new APIs
    fmt         gofmt (reformat) package sources
    generate    generate Go files by processing source
    get         add dependencies to current module and install them
    install     compile and install packages and dependencies
    list        list packages or modules
    mod         module maintenance
    work        workspace maintenance
    run         compile and run Go program
    test        test packages
    tool        run specified go tool
    version     print Go version
    vet         report likely mistakes in packages

Use "go help <command>" for more information about a command.

Additional help topics:

    buildconstraint build constraints
    buildmode       build modes
    c               calling between Go and C
```

```
    cache           build and test caching
    environment     environment variables
    filetype        file types
    go.mod          the go.mod file
    gopath          GOPATH environment variable
    goproxy         module proxy protocol
    importpath      import path syntax
    modules         modules, module versions, and more
    module-auth     module authentication using go.sum
    packages        package lists and patterns
    private         configuration for downloading non-public code
    testflag        testing flags
    testfunc        testing functions
    vcs             controlling version control with GOVCS

Use "go help <topic>" for more information about that topic.
```

4.3.1 Ověření verze jazyka Go

Pro začátek si vypíšeme, která verze jazyka Go je vlastně aktuálně dostupná. K tomuto účelu slouží následující varianta již výše zmíněného příkazu `go`:

```
go version
```

V případě, že jste si nainstalovali Go verze 1.23.3 na 64bitové architektuře x86-64 (známé též pod označením AMD64), měla by se na plochu terminálu vypsat následující zpráva:

```
go version go1.23.3 linux/amd64
```

Při instalaci starší verze Go se pochopitelně vypíše odlišná verze jazyka. Opět si uveďme příklad:

```
go version go1.22.1 linux/amd64
```

V této kapitole jsme si ukázali instalaci jazyka Go ve verzi 1.23 popř. 1.24. Ve skutečnosti budou všechny demonstrační příklady pracovat i ve starší verzi 1.22 (a mnohé i ve starších verzích, teoreticky i verze 1.9 atd.). Go je v tomto ohledu velmi konzervativní a i když se jazyk postupně vyvíjí, zachovává si zpětnou kompatibilitu.

4.3.2 Proměnné prostředí používané jazykem Go

V některých situacích je nutné si ověřit konfiguraci Go. Pro zobrazení všech proměnných, které ovlivňují chování Go, se používá příkaz:

```
go env
GO111MODULE=''
GOARCH='amd64'
GOBIN=''
GOCACHE='/home/ptisnovs/.cache/go-build'
GOENV='/home/ptisnovs/.config/go/env'
GOEXE=''
GOEXPERIMENT=''
GOFLAGS=''
GOHOSTARCH='amd64'
GOHOSTOS='linux'
GOINSECURE=''
GOMODCACHE='/home/ptisnovs/go/pkg/mod'
GONOPROXY=''
GONOSUMDB=''
GOOS='linux'
GOPATH='/home/ptisnovs/go'
```

Samozřejmě je možné si nechat vypsat jen jednu konkrétní hodnotu, například:

```
go env GOPATH
/home/ptisnovs/go
```

5 První kroky s Go

5 První kroky s Go

V této kapitole se seznámíme se základními vlastnostmi programovacího jazyka Go. Ukážeme si také, jakým způsobem se zdrojové kódy překládají a spouští. Předpokládá se, že již máte nainstalovány základní nástroje jazyka Go (viz předchozí kapitolu). Pro tvorbu a úpravy zdrojových kódů je možné použít jakýkoli (programátorský) textový editor popř. integrované vývojové prostředí (VSCode apod.). Můžete taktéž použít webové „pískoviště" dostupné na adrese https://go.dev/play/. Pro první seznamování s Go může být toto pískoviště dostačující (navíc nevyžaduje žádné instalace), ale pro vážně pojatý vývoj se nehodí.

5.1 Způsob zobrazení zdrojových kódů programů v této knize

Ve zdrojových kódech programů, které jsou napsány v jazyce Go, se pro odsazení programových řádků používají ASCII znaky s kódem 9. Tyto znaky se do zdrojových kódů vkládají klávesou *Tab* neboli *tabulátorem*. V programátorských textových editorech popř. v integrovaných vývojových prostředích je většinou možné si nastavit, jakým způsobem má být znak s ASCII kódem 9 zobrazen. Typicky se používají čtyři mezery nebo osm mezer (méně často například jen dvě mezery, to ovšem nedoporučuji). V této knize z důvodu úspory místa používáme čtyři mezery namísto obvyklejších osmi mezer. Ovšem v repositáři se všemi ukázkovými příklady (https://github.com/tisnik/programovaci-jazyk-Go) je korektně použit již zmíněný ASCII znak 9.

5.2 Struktura zdrojových kódů

Zdrojové kódy vytvářené v jazyce Go mají do určité míry pevnou strukturu, se kterou se nyní seznámíme.

5.2.1 Jméno balíčku

Každý soubor obsahující zdrojový kód musí začínat klíčovým slovem `package`, za nímž následuje jméno balíčku, kterého je tento kód součástí. Naše první programy budou velmi jednoduché a budou se skládat z jediného balíčku, který se jmenuje `main`, což znamená, že první řádek našich zdrojových kódu bude vypadat následovně:

```
package main
```

Poznámka: ve skutečnosti se mohou ještě před tímto řádkem nacházet textové řádky s komentáři nebo prázdné řádky. Ovšem první skutečný *programový řádek začíná právě slovem* `package`.

5.2.2 Deklarace funkcí

Ve zdrojových kódech dále nalezneme deklarace dalších „součástek", ze kterých se program skládá. V první řadě se jedná o deklaraci funkcí. Deklarace funkce se skládá z hlavičky funkce (*header*) následovaného tělem funkce (*body*). Každá deklarace nové funkce přitom začíná klíčovým slovem *func*, za nímž následuje jméno funkce. V kulatých závorkách jsou zapsána jména a typy parametrů funkce a za závorkami pak typ návratové hodnoty popř. typy (a dokonce i volitelné názvy) návratových hodnot. Samotné *tělo* funkce je vždy zapsáno mezi složené závorky `{}`, přičemž otevírací levá složená závorka musí být vždy zapsána na stejném textovém řádku, jako hlavička funkce (pozor – jiné programovací jazyky nejsou až takto striktní).

Jméno funkce musí začínat podtržítkem nebo písmenem a může obsahovat jak znaky (prakticky jakékoli) abecedy, tak i číselné cifry či podtržítka. Pozor ovšem na to, že jméno tvořené jediným podtržítkem _ má v Go speciální význam.

Nejjednodušší funkce nemají žádné parametry, takže namísto nich uvedeme pouze prázdné kulaté závorky. Na rozdíl od některých jiných programovacích jazyků tedy nepoužíváme „prázdný" typ `void`. V té nejjednodušší podobě funkce ani nevrací žádnou hodnotu a její tělo je prázdné. Deklarace takové funkce by měla mohla vypadat následovně:

```
func jmenoFunkce() {
}
```

5.2.3 Komentáře ve zdrojových kódech

Do zdrojových kódů je pochopitelně možné přidávat i komentáře. V tomto ohledu programovací jazyk Go používá podobné zásady zápisu komentářů, jaké jsou použité v programovacích jazycích syntakticky vycházejících z jazyka C (sem spadá C++, ale vlastně i Java či JavaScript). Jednořádkové komentáře začínají dvojicí znaků `//`, ovšem zapisovat je možné i víceřádkové komentáře. V takovém případě komentář začíná znaky `/*` a končí znaky `*/`. Ukažme si několik jednoduchých příkladů.

Jednořádkové komentáře, které končí na tom samém řádku, na němž jsou zapsány:

```
// jméno balíčku
package main // i toto je komentář

// funkce main představuje v jazyce Go
// vstupní bod do aplikace
func main() { // začátek těla funkce
}
```

Komentář pokračující přes větší množství řádků a komentář na jediném řádku (který však může být později rozšířen):

```
/*
    Jméno balíčku, do kterého zdrojový kód patří.

Balíček main má specifický význam pro běžné programy, ovšem
v knihovnách se nepoužívá.
*/
package main

/* Funkce main, která slouží jako vstupní bod do aplikace. */
func main() {
}
```

Zakomentovat můžeme i část zdrojového kódu, což může být využito při ladění atd.:

```
package main

/* Funkce main, která slouží jako vstupní bod do aplikace. */
func main() {
}

// Zakomentovaná část zdrojového kódu

/*
func main() {
}
*/
```

Ovšem pozor na to, že víceřádkové komentáře není možné „zanořit“ do jiného komentáře. Následující příklad tedy nebude přeložen:

```
package main

/* Funkce main, která slouží jako vstupní bod do aplikace. */
func main() {
}

// Zakomentovaná část zdrojového kódu

/*
func main() {
    /* vložený komentář */
}
*/
```

5.2.4 Funkce `main`

Již v předchozích příkladech jsme specifikovali jméno balíčku `main`. Dále ve zdrojovém kódu programu s využitím klíčového slova `func` nadeklarujeme funkci nazvanou taktéž `main` (což může být zpočátku matoucí, ale rychle si zvyknete). Takto pojmenovaná funkce, která je současně vstupním bodem do aplikace, nemá žádné parametry ani návratovou hodnotu (více se o návratových hodnotách dozvíme v navazujících kapitolách). Minimalisticky pojatý, ovšem zcela funkční program napsaný v jazyku Go tedy bude vypadat následovně:

```
package main

func main() {
}
```

5.3 Program typu „Hello, world!“

Podívejme se nyní na nepatrně složitější program. Samozřejmě začneme klasickým programem typu „Hello, world!“, což je tradice, která již před několika desetiletími byla zavedena v učebnici programovacího jazyka C. Konkrétně si na standardní výstup necháme vypsat zprávu „Hello, world!“ a poté program ukončíme. Nejprve si ukažme, jak bude zdrojový kód takového programu vypadat a posléze si popíšeme jeho jednotlivé části:

```
// První kroky v jazyku Go
//
// - program typu "Hello, world!" ve variantě naprogramované
```

```
//   v jazyku Go.

// specifikace jména balíčku (zde musíme použít "main")
package main

// import konstant, typů a funkcí z jiného balíčku
import "fmt"

// deklarace funkce nazvané main (vstupní bod do programu)
func main() {
    // volání funkce náležející do jiného balíčku
    // s využitím takzvané tečkové notace
    fmt.Println("Hello, world!")
}
```

5.3.1 Anatomie programu typu „Hello, world!"

Zdrojový kód našeho programu si můžeme rozdělit do několika logických částí, a to následujícím způsobem:

1. Jméno balíčku
2. Jméno jiného balíčku, který importujeme
3. Definice funkce

Povšimněte si jedné důležité a důsledně dodržované vlastnosti programovacího jazyka Go – každá uvedená logická část vždy začíná nějakým *klíčovým slovem*, což je slovo, které má v Go speciální a pevně daný význam. Díky této vlastnosti je usnadněno čtení zápisů zdrojových kódů, a to jak pro programátory, tak i pro počítač (konkrétně pro překladač a další nástroje zpracovávající zdrojové kódy). V mnoha dalších programovacích jazycích není tento koncept použit, popř. je použit jen částečně.

Vraťme se však k oněm třem logickým částem zdrojového kódu. Programový řádek s názvem balíčku již známe; jedná se o klíčové slovo `package` následované jménem, které se může skládat z písmen, znaku podtržítka a číslic, ovšem nesmí začínat číslicí. Řádek obsahující klíčové slovo `import` obsahuje jméno jiného balíčku, jehož konstanty, datové typy a funkce budeme chtít volat. V tomto případě je skutečně nutné takový balíček nejdříve „naimportovat". V praxi se setkáme s tím, že je nutné importovat větší množství balíčků. To lze provést buď pomocí více příkazů importu:

```
// První kroky v jazyku Go
//
// - import symbolů (funkcí atd.) ze dvou balíčků
// - každý balíček je naimportován přes samostatný příkaz "import"

package main

// import jediného balíčku
import "fmt"

// import jediného balíčku
import "os"

func main() {
    // zde můžeme používat funkce z obou naimportovaných balíčků
    fmt.Println("Hello, world!")
    os.Exit(0)
}
```

Doporučený zápis je však odlišný: více názvů balíčků se v tomto případě zapíše za klíčové slovo `import` do kulatých závorek (skutečně se jedná o kulaté závorky – zde se poměrně často dělá chyba):

```
// První kroky v jazyku Go
//
// - import symbolů (funkcí atd.) ze dvou balíčků
// - oba balíčky jsou naimportovány jediným příkazem "import"

package main

// import dvou balíčků v jediném bloku "import"
import (
    "fmt"
    "os"
)

func main() {
    // zde můžeme používat funkce z obou naimportovaných balíčků
    fmt.Println("Hello, world!")
    os.Exit(0)
}
```

Poznámka: převod skupiny příkazů import na blok většinou dokážou provést programátorské editory zcela automaticky. Moderní editory taktéž seznam balíčků abecedně seřadí. Pokud vám toto řazení nevyhovuje, vložte mezi jména balíčků prázdné řádky.

Nejsložitější je třetí logická část zdrojového kódu naší aplikace, která obsahuje deklaraci funkce. Z hlavičky funkce je zřejmé, že se jedná o funkci bez parametrů a bez návratových hodnot. A tělo funkce obsahuje volání jediné další funkce, konkrétně funkce se jménem `Println` z balíčku `fmt`. Podrobnosti si řekneme v dalších kapitolách, ovšem již nyní je vhodné poznamenat, že z jiných balíčků lze volat jen ty funkce, jejichž název začíná velkým písmenem. A tento požadavek funkce `Println` nepochybně splňuje.

5.4 Překlad a spuštění programu

Náš základní demonstrační příklad, jehož zdrojový kód je uložený v souboru `hello_world.go`, si můžeme spustit a to konkrétně příkazem `go run hello_world.go`. Před samotným spuštěním se sice (většinou zcela na pozadí) provede překlad, přičemž spustitelná verze programu je uložena do dočasného souboru, ovšem po ukončení programu se tento dočasný soubor smaže, takže se nám může zdát, jako bychom zdrojový kód přímo spouštěli (tj. Go se zdánlivě tváří jako interpret, i když jsou zdrojové kódy *vždy* překládány):

```
go run hello_world.go
Hello, world!
```

5.4.1 Překlad do nativního spustitelného souboru

V případě, že si budeme přát program explicitně přeložit a zachovat výsledný binární spustitelný soubor, použijeme namísto příkazu `go run` příkaz `go build`, a to konkrétně tímto způsobem:

```
go build hello_world.go
```

Výsledkem překladu bude soubor, který bude pojmenovaný stejně jako soubor zdrojový, ovšem s odlišnou koncovkou. Na Linuxu nebude koncovka použita vůbec:

```
ls -1

total 1880
-rwxr-xr-x. 1 ptisnovs ptisnovs 1894201 Dec  8 17:16 hello_world
-rw-r--r--. 1 ptisnovs ptisnovs     330 Dec  7 09:48 hello_world.go
```

V operačním systému Microsoft Windows je výsledkem překladu soubor s koncovkou `.exe`, tedy konkrétně `hello_world.exe`.

5.4.2 Vlastnosti souboru získaného překladem

V obou případech obsahuje spustitelný soubor, který jsme získali překladem, jak nativní kód naší přeložené aplikace, tak i všechny potřebné funkce nutné pro spuštění této aplikace. Tyto funkce tvoří takzvaný *runtime* a mj. zde nalezneme inicializační rutiny, ale například i celý subsystém pro automatickou správu paměti (včetně *garbage collectoru*). Právě kvůli tomu, že se v tomto binárním souboru nachází i všechny potřebné funkce pro *runtime*, je soubor relativně velký – jeho velikost se blíží ke dvěma megabajtům, zatímco aplikace se stejnou funkcionalitou naprogramovaná v jazyku C bude mít po překladu velikost 6 až 14 kB v závislosti na použité architektuře i na použitém překladači a linkeru. Na druhou stranu bude náš `hello_world` prakticky bez problémů přenositelný i na další počítače se stejnou architekturou a stejným operačním systémem (k této problematice se ještě vrátíme).

Právě přeloženou aplikaci si můžeme velmi snadno spustit následujícími příkazy. Na Linuxu:

```
./hello_world

Hello, world!
```

Na systému Microsoft Windows (z příkazové řádky):

```
hello_world.exe
```

V případě, že budeme chtít prozkoumat, jaké symboly se vlastně v souboru `01_hello_world` nachází (je jich zhruba 2000, protože je zde skutečně celý runtime), můžeme na Lixuxu použít příkaz `objdump`:

```
hello_world:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 go.go
0000000000401000 l     F .text  0000000000000000 runtime.text
0000000000402d20 l     F .text  000000000000022d cmpbody
0000000000402f70 l     F .text  0000000000000153 memeqbody
0000000000403120 l     F .text  0000000000000129 indexbytebody
000000000045f600 l     F .text  0000000000000040 gogo
000000000045f640 l     F .text  000000000000002b callRet
```

```
000000000045f680 l     F .text  000000000000002f gosave_systemstack_switch
000000000045f6c0 l     F .text  000000000000000d setg_gcc
000000000045f6d0 l     F .text  0000000000000564 aeshashbody
...
...
...
```

5.5 Důležité nástroje, které jsou součástí instalace Go

Kromě samotného překladače jazyka Go, který jsme použili v předchozím textu, je součástí instalace Go i celá řada dalších užitečných nástrojů. O těch nejdůležitějších se nyní ve stručnosti zmíníme.

5.5.1 Zobrazení dokumentace

Dokumentace k jazyku Go i ke standardním knihovnám je umístěna na adrese https://godoc.org/. V předchozích ukázkových příkladech jsme mj. používali i balíček nazvaný `fmt`, který je součástí standardní knihovny jazyka Go a jehož dokumentaci samozřejmě můžeme taktéž nalézt na výše zmíněných stránkách popsán. Konkrétně je dokumentace tohoto balíčku dostupná na adrese https://godoc.org/fmt.

Kromě toho je však možné si nechat dokumentaci zobrazit přímo v terminálu (konzoli) při vývoji. K tomuto účelu slouží příkaz `go doc`, kterému se jako další parametr předá jméno balíčku popř. přímo jméno funkce, jejíž popis potřebujeme získat:

```
go doc fmt
```

Výsledek by měl vypadat následovně:

```
package fmt // import "fmt"

Package fmt implements formatted I/O with functions analogous to C's printf and
scanf. The format 'verbs' are derived from C's but are simpler.

# Printing

The verbs:

General:
```

```
    %v  the value in a default format
        when printing structs, the plus flag (%+v) adds field names
...
...
...
```

Poznámka: celkem se vypíše přibližně 380 řádků nápovědy.

Zobrazit si ovšem můžeme i nápovědu ke konkrétní funkci z balíčku `fmt`, a to následujícím způsobem:

```
go doc fmt.Println
```

V tomto případě je výsledek poměrně stručný:

```
package fmt // import "fmt"

func Println(a ...any) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output. Spaces are always added between operands and a newline
    is appended. It returns the number of bytes written and any write error
    encountered.
```

5.5.2 Doporučovaný styl zápisu programů

Tvůrci programovacího jazyka Go navrhli, jakým způsobem se mají formátovat zdrojové kódy. Nezůstalo ovšem pouze u doporučení, protože přímo v základní sadě nástrojů jazyka Go nalezneme i utilitu nazvanou příznačně `gofmt`. Tento nástroj slouží k přeformátování zdrojových kódů takovým způsobem, aby jejich výsledné naformátování odpovídalo specifikaci. Díky tomu mají (nebo by alespoň měly mít) všechny zdrojové kódy napsané v jazyku Go jednotný standardní formát a například diskuze o použití tabulátorů nebo mezer, popř. o kolik znaků (mezer) se má provést odsazení bloků, přestávají mít smysl.

Ve zkratce autoři Go vlastně říkají: namísto dlouhých popisů standardů formátování zdrojových kódů, které známe například z *PEP-8* (Python), *GNU Coding Standards* či *Linux kernel coding style* se v případě jazyka Go formátováním programátoři příliš zabývat nemusí, protože za ně většinu práce odvede nástroj `gofmt`, který má navíc (z definice) vždycky pravdu.

Poznámka: naprosto stejné formátování nabízí i běžná integrovaná vývojová prostředí a programátorské editory.

5.5.3 Nástroj `gofmt` určený pro přeformátování zdrojových kódů

Pokud jste provedli instalaci Go a jeho nástrojů podle pokynů ze čtvrté kapitoly, bude nástroj `gofmt` jednoduše použitelný:

```
gofmt --help
```

Tento nástroj podporuje různé volby, ovšem v praxi využijete pravděpodobně především volbu `-w` pro zápis přeformátovaného zdrojového kódu do jiného souboru:

```
usage: gofmt [flags] [path ...]
  -cpuprofile string
        write cpu profile to this file
  -d    display diffs instead of rewriting files
  -e    report all errors (not just the first 10 on different lines)
  -l    list files whose formatting differs from gofmt's
  -r string
        rewrite rule (e.g., 'a[b:len(a)] -> a[b:]')
  -s    simplify code
  -w    write result to (source) file instead of stdout
```

Zkusme například formátovači předat následující zdrojový kód se středníky, špatným odsazením, prázdnými řádky atd.:

```
package main




import "fmt";
func main() {
  fmt.Println("Hello, playground");
}
```

Po naformátování získáme tento obsah, který plně odpovídá specifikaci:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, playground")
}
```

6 Základy programovacího jazyka Go

6 Základy programovacího jazyka Go

V této kapitole se seznámíme se základními rysy i některými specifickými vlastnostmi programovacího jazyka Go, takže po jejím přečtení již budete schopni tvořit jednodušší aplikace běžící v jediné *gorutině*. Ovšem pro plné využití možností, které jsou jazykem Go nabízeny, bude nutné prostudovat i další rysy tohoto jazyka – zejména se jedná o velmi důležitý koncept datových typů, použití rozhraní (*interface*), využití výše zmíněných *gorutin* a kanálů atd.

Všechny dále popisované vlastnosti programovacího jazyka Go pochopitelně budou prakticky předvedeny v celé řadě demonstračních příkladů. Tyto příklady byly otestovány v Go verze 1.23, ovšem naprostá většina příkladů je použitelná i ve starších verzích tohoto jazyka. Na ty ukázkové příklady, kde tomu tak není (například při popisu některé nejnovější vlastnosti Go), budete pochopitelně upozorněni.

6.1 Klíčová slova

Ve specifikaci programovacího jazyka Go je definováno celkem dvacet pět rezervovaných *klíčových slov* (v angličtině *keywords*). Jedná se o slova, která mají ve zdrojových kódech speciální význam a nelze je tedy použít pro pojmenování balíčků, konstant, funkcí, metod ani proměnných – v takovém případě překladač ohlásí syntaktickou chybu. Všechna rezervovaná klíčová slova jazyka Go jsou vypsána pod tímto odstavcem a postupně se s jejich významem pochopitelně seznámíme:

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Poznámka: zajímavé je, že i když se samotný jazyk Go postupně rozšiřuje, například o generické datové typy nebo o nové varianty programových smyček, samotný seznam klíčových slov se prozatím nerozšířil. Z tohoto pohledu je tedy zachována přísná zpětná kompatibilita.

Jen pro zajímavost si vyzkoušejme, co se stane v případě, že se pokusíme použít klíčová slova v jiném významu, konkrétně jako jméno proměnné. Takto použijeme klíčová slova `type` a `interface`:

```
// Základy programovacího jazyka Go
//
// - pokus o použití klíčových slov ve funkci identifikátorů
```

```
//
// POZOR: tento příklad nebude možné přeložit

package main

func main() {
    // pokus o použití klíčového slova "interface"
    // jako identifikátoru při deklaraci konstanty
    const interface int = 10

    // pokus o použití klíčového slova "type"
    // jako identifikátoru při deklaraci proměnné
    var type string = "Škoda 100"
}
```

Pokus o překlad tohoto zdrojového kódu nebude (podle očekávání) úspěšný:

```
./01_01_keywords_as_identifiers.go:10:8: syntax error: unexpected interface, expected name
./01_01_keywords_as_identifiers.go:11:6: syntax error: unexpected type, expected name
```

6.1.1 Jména standardních datových typů nejsou klíčovými slovy

Za povšimnutí stojí, že mezi rezervovanými klíčovými slovy nenajdeme jména standardních datových typů, mezi než patří `bool`, `int`, `float32` či `string`. A skutečně: jména standardních datových typů nejsou ve standardu Go rezervována, což znamená, že můžeme například napsat následující program, ve kterém je změněn význam typu `int` (resp. přesněji řečeno se změní význam identifikátoru `int`):

```
// Základy programovacího jazyka Go
//
// - deklarace datového typu nazvaného "int", který přepíše původní typ "int"
//
// POZOR: tento způsob není doporučený a je dobré vědět, kdy ho použít
//        a především, kdy ho naopak nepoužít.

package main

import "fmt"

// nový datový typ "int", který v tomto balíčku přepíše původní typ
```

```
// se stejným jménem
type int = float32

func main() {
    // proměnná nazvaná "pi" je sice typu "int", ovšem na rozdíl od jména
    // typu se jedná o numerickou hodnotu s plovoucí řádovou čárkou
    var pi int

    // nyní můžeme do proměnné přiřadit hodnotu s plovoucí řádovou čárkou
    // i když se (zdánlivě) jedná o proměnnou celočíselného typu
    pi = 3.14
    fmt.Println(pi)
}
```

! *Pozor: v tomto případě se jedná pouze o ukázku, jak by se programy v praxi psát* neměly. *Tímto způsobem se totiž celý program znepřehlední a navíc bude mít jméno datového typu odlišný význam v různých balíčcích.*

6.1.2 Pravdivostní konstanty nejsou klíčovými slovy

Mezi rezervovanými klíčovými slovy jazyka Go nenajdeme dokonce ani pravdivostní konstanty `true` a `false`, což opět znamená, že můžeme (ale nikdy bychom to dělat neměli) změnit jejich význam. Opět si uveďme příklad, jak bychom *neměli* psát programy:

```
// Základy programovacího jazyka Go
//
// - deklarace proměnné "true", která přepíše původní hodnotu "true"
//
// POZOR: tento způsob není doporučený a je dobré vědět, kdy ho použít
//        a především, kdy ho naopak nepoužít.

package main

import "fmt"

func main() {
    // deklarace proměnné nazvané "true", tím v tomto balíčku ztratíme
    // přístup k původní hodnotě true (což většinou nechceme!)
    var true string = "pravda"
```

```
    // deklarace proměnné, která je ve skutečnosti typu "string"
    // (překladač nyní použije hodnotu proměnné "true")
    var answer string = true
    fmt.Println(answer)
}
```

Poznámka: podobně je tomu s hodnotou `nil`*, s jejímž významem se seznámíme při popisu ukazatelů a rozhraní.*

6.2 Operátory a znaky se speciálním významem

Ve specifikaci programovacího jazyka Go taktéž můžeme najít tabulku se všemi operátory (*operators*) a dalšími znaky se speciálním významem. Podívejme se, jaké jednotlivé znaky nebo jejich kombinace překladač programovacího jazyka Go rozpoznává:

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=          ~
```

Opět platí, že tyto kombinace znaků mají svůj pevně daný význam, který není možné žádným způsobem měnit. Navíc jazyk Go nepodporuje ani přetěžování operátorů (*operator overloading*).

6.2.1 Tabulka všech operátorů

V případě, že se zaměříme pouze na operátory (a nikoli na speciální znaky), tabulka z předchozího odstavce se zmenší na:

```
+    &     +=    &=     &&    ==    !=
-    |     -=    |=     ||    <     <=
*    ^     *=    ^=     <-    >     >=
/    <<    /=    <<=    ++    =
%    >>    %=    >>=    --    !
     &^          &^=
```

Pro čtenáře této knihy pravděpodobně nebude žádnou novinkou, že operátory určené pro manipulaci s číselnými, pravdivostními, řetězcovými aj. hodnotami, tvoří podstatnou část syntaxe v prakticky všech mainstreamových programovacích jazycích. V programovacím jazyku Go nalezneme celkem 26 základních operátorů, k nimž navíc ještě musíme připočíst operátory spojené s přiřazením (tedy například přičtení hodnoty namísto obecnějšího součtu atd.):

Skupina operátorů	**Operátory**
aritmetické	`+ - * / %`
aritmetické s přiřazením	`+= -= *= /= %=`
logické	`&& || !`
posuny a bitové operace	`<< >> & | ^ &^`
posuny a bitové operace s přiřazením	`<<= >>= &= |= ^= &^=`
relační	`== != < <= > >=`
operace s adresami (unární)	`* &`
další unární operátory	`+ - ^`
speciální operátory	`<- := ++ --`

Poznámka: ve skutečnosti má operátor + ještě další význam. Kromě součtu dvou numerických hodnot totiž realizuje i spojení (konkatenaci) dvou řetězců; není tedy svázán jen s numerickými datovými typy.

V navazujících podkapitolách si ukážeme význam většiny operátorů z předchozí tabulky.

6.2.2 Priority operátorů

Zajímavá je tabulka s prioritami operátorů. Ta určuje, jakým způsobem se budou vyhodnocovat výrazy s větším množstvím operátorů, například výraz `a+b*c`. V jazyce Go je tabulka s prioritami operátorů poměrně jednoduchá, zejména v porovnání s tabulkami, které někteří čtenáři pravděpodobně znají z jazyků C, C++ či Javy. Nejvyšší prioritu mají prakticky všechny unární operátory (s jediným operandem zapisovaným za operátor) a následně existuje pouze pět priorit, které si můžete zapamatovat s využitím mnemotechnické pomůcky MACAO:

Úroveň	Mnemotechnická pomůcka	Operátory
1	*M*ultiplicative	`* / % << >> & &^`
2	*A*dditive	`+ - | ^`
3	*C*omparison	`== != < <= > >=`
4	*A*nd	`&&`
5	*O*r	`||`

Poznámka: některé operátory, zejména ++ a --, stojí mimo tuto tabulku. Je tomu tak i z toho důvodu, že tyto operátory vystupují vždy v roli příkazu a nikoli jako součást výrazu (příkaz je vykonán, kdežto výraz vrací hodnotu).

6.2.3 Unární operátory

Nejprve se podrobněji podívejme na použití unárních operátorů. Ty mají, jak již víme, nejvyšší prioritu a proto se téměř nikdy nesetkáme s nutností uzavírat tyto operátory do kulatých závorek, které (podobně jako v mnoha dalších programovacích jazycích a ostatně i v matematice) slouží ke změně priority. V jazyce Go nalezneme tyto unární operátory:

Operátor	Význam operátoru
`+`	nemění znaménko výsledku (opak dalšího operátoru)
`-`	mění znaménko výsledku
`!`	logická negace
`^`	negace bit po bitu (podobně jako ~ v C)
`*`	přístup na adresu přes ukazatel (referenci)
`&`	získání adresy operandu
`<-`	přečtení hodnoty z kanálu

Ukažme si nyní, jakým způsobem se jednotlivé unární operátory používají. Povšimněte si, že se všechny zmíněné operátory vždy zapisují před operand, tj. na levou stranu od operandu:

```
// Základy programovacího jazyka Go
//
// - unární operátory definované v jazyku Go

package main

import "fmt"

func main() {
    // unární operátory + a -
    var i int = 42
    fmt.Println(+i)
    fmt.Println(-i)

    // unární operátor ^
    i = 0
    fmt.Println(^i)
    i++
    fmt.Println(^i)

    // unární operátor !
    var b bool = false
    fmt.Println(!b)

    // unární operátor &
    fmt.Println(&i)

    // unární operátory & a *
    var ptrI *int = &i
    fmt.Println(*ptrI)

    // unární operátor <-
    var channel chan int
    channel = make(chan int)
    <-channel
}
```

Tento příklad je pouze ilustrační, protože po spuštění skončí s chybou při pokusu o čtení hodnoty z kanálu `channel` s čekáním (viz další kapitoly):

```
42
-42
-1
-2
true
0xc000124010
1
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
    /home/ptisnovs/Documents/Go/priklady/06_zaklady/02_01_unary_operators.go:33 +0x24f
exit status 2
```

6.2.4 Základní aritmetické operátory

V programovacím jazyce Go nalezneme všech pět základních binárních aritmetických operátorů (binárních proto, že akceptují dva operandy – levý a pravý). Jedná se o operátory určené pro realizaci operace součtu, rozdílu, součinu, podílu a zbytku po dělení. Těchto pět operátorů můžeme použít s dvojicí celočíselných operandů tak, jak je to naznačeno v dalším ukázkovém příkladu:

```
// Základy programovacího jazyka Go
//
// - základní aritmetické operátory použité s proměnnými
//   typu int

package main

import "fmt"

func main() {
    // dvojice proměnných typu int
    var x int = 10
    var y int = 3

    // výpis původních hodnot proměnných
    fmt.Printf("x = %d\n", x)
    fmt.Printf("y = %d\n", y)

    fmt.Println()
```

```
    // výpis výsledků aritmetických operací s proměnnými
    fmt.Printf("x+y = %d\n", x+y)
    fmt.Printf("x-y = %d\n", x-y)
    fmt.Printf("x*y = %d\n", x*y)
    fmt.Printf("x/y = %d\n", x/y)
    fmt.Printf("x%%y = %d\n", x%y)
}
```

Výsledky:

```
x = 10
y = 3

x+y = 13
x-y = 7
x*y = 30
x/y = 3
x%y = 1
```

U hodnot s plovoucí řádovou čárkou (*floating point numbers*) není k dispozici operátor pro výčet zbytku po dělení, takže je možné provádět jen součet, rozdíl, součin a podíl:

```
// Základy programovacího jazyka Go
//
// - základní aritmetické operátory s proměnnými
//   typu float32

package main

import "fmt"

func main() {
    // dvojice proměnných typu float32
    var x float32 = 1.1
    var y float32 = 2.2

    // výpis původních hodnot proměnných
    fmt.Printf("x = %f\n", x)
    fmt.Printf("y = %f\n", y)

    fmt.Println()
```

```
    // výpis výsledků aritmetických operací s proměnnými
    fmt.Printf("x+y = %f\n", x+y)
    fmt.Printf("x-y = %f\n", x-y)
    fmt.Printf("x*y = %f\n", x*y)
    fmt.Printf("x/y = %f\n", x/y)
}
```

Výsledky:

```
x = 1.100000
y = 2.200000

x+y = 3.300000
x-y = -1.100000
x*y = 2.420000
x/y = 0.500000
```

Stejná čtveřice operátorů, tedy součet, rozdíl, součin a podíl, je použita i pro komplexní čísla, i když součin a podíl se interně počítá poněkud komplikovanějším způsobem:

```
// Základy programovacího jazyka Go
//
// - základní aritmetické operátory s proměnnými
//   typu complex64

package main

import "fmt"

func main() {
    // dvojice proměnných typu complex64
    var x complex64 = 1 + 1i
    var y complex64 = 2 + 2i

    // výpis původních hodnot proměnných
    fmt.Printf("x = %v\n", x)
    fmt.Printf("y = %v\n", y)

    fmt.Println()
```

```
    // výpis výsledků aritmetických operací s proměnnými
    fmt.Printf("x+y = %v\n", x+y)
    fmt.Printf("x-y = %v\n", x-y)
    fmt.Printf("x*y = %v\n", x*y)
    fmt.Printf("x/y = %v\n", x/y)
}
```

Opět si ukažme výsledky:

```
x = (1+1i)
y = (2+2i)

x+y = (3+3i)
x-y = (-1-1i)
x*y = (0+4i)
x/y = (0.5+0i)
```

Jak jsme již mohli vidět, v programovacím jazyce Go se pro zápis podílu používá operátor `/` a pro výpočet zbytku po dělení operátor `%`. Ostatně podobně je tomu i v prakticky všech programovacích jazycích více či méně odvozených od jazyka C. Vyzkoušejme si nyní, jak vlastně dělení a výpočet zbytku probíhá pro kladné i záporné dělence a dělitele, protože chování pro záporné hodnoty nemusí být příliš intuitivní. Pomůže nám tento demonstrační příklad:

```
// Základy programovacího jazyka Go
//
// - operátory pro celočíselný podíl a výpočet zbytku po dělení
// - použití pro různé kombinace kladných i záporných operandů

package main

import "fmt"

// Výpočet podílu a zbytku po dělení
func computeDivMod(x, y int) {
    fmt.Printf("%3d / %2d = %3d   %3d %% %2d = %3d\n",
        x, y, x/y, x, y, x%y)
}

func main() {
    // otestování výpočtu podílu a zbytku po dělení pro všechny
    // kombinace kladných a záporných hodnot dělenců a dělitelů
```

```
    computeDivMod(10, 3)
    computeDivMod(-10, 3)
    computeDivMod(10, -3)
    computeDivMod(-10, -3)

    fmt.Println()

    // tabulka s výsledky zbytku po dělení 100/i
    // pro hodnoty i v rozsahu od 1 do 10
    for i := 1; i <= 10; i++ {
        computeDivMod(100, i)
    }
}
```

V programovacím jazyce Go je znaménko zbytku odvozeno od *znaménka dělence* (tedy prvního operandu). Toto chování je zachováno na všech platformách, zatímco v jazycích C a C++ je implementačně závislé.

V Go tedy dostaneme tyto výsledky:

```
10 /  3 =   3    10 %  3 =   1
-10 /  3 =  -3   -10 %  3 =  -1
 10 / -3 =  -3    10 % -3 =   1
-10 / -3 =   3   -10 % -3 =  -1
```

Z toho ovšem vyplývají i další vlastnosti, například to, že se při celočíselném dělení kladných čísel zaokrouhluje výsledek směrem dolů. Je tomu tak z toho důvodu, že zbytek musí být v tomto případě buď nulový nebo kladný. Opět si to ověřme:

```
100 /  1 = 100   100 %  1 =   0
100 /  2 =  50   100 %  2 =   0
100 /  3 =  33   100 %  3 =   1
100 /  4 =  25   100 %  4 =   0
100 /  5 =  20   100 %  5 =   0
100 /  6 =  16   100 %  6 =   4
100 /  7 =  14   100 %  7 =   2
100 /  8 =  12   100 %  8 =   4
100 /  9 =  11   100 %  9 =   1
100 / 10 =  10   100 % 10 =   0
```

6.2.5 Spojení řetězců

Připomeňme si, že operátorem + je kromě aritmetického součtu možné provést spojení dvou řetězců. Jak si řekneme později, jsou řetězce neměnitelné (*immutable*), takže výsledkem spojení dvou řetězců je řetězec nový:

```
// Základy programovacího jazyka Go
//
// - spojení řetězců operátorem +

package main

import "fmt"

func main() {
    // dvojice proměnných typu string
    var x string = "Hello "
    var y string = "world!"

    // výpis původních hodnot proměnných
    fmt.Printf("x = '%s'\n", x)
    fmt.Printf("y = '%s'\n", y)

    fmt.Println()

    // spojení řetězců operátorem +
    fmt.Printf("x+y = %s\n", x+y)
}
```

Výsledek vypsaný tímto příkladem:

```
x = 'Hello '
y = 'world!'

x+y = Hello world!
```

6.2.6 Relační operátory

Nabídka relačních operátorů v programovacím jazyce Go by nás neměla ničím nepřekvapit: k dispozici je totiž všech šest základních operátorů pro test na rovnost, nerovnost, větší než, menší než

a kombinací větší nebo rovno a menší nebo rovno. Těchto šest operátorů je možné použít pro všechny celočíselné datové typy i pro datové typy s plovoucí řádovou čárkou. Kromě toho ovšem můžeme porovnávat i řetězce. Navíc je možné test na rovnost a nerovnost použít i pro porovnání struktur atd. – s tímto velmi užitečným konceptem se seznámíme v navazujících kapitolách.

Nejprve si ukažme porovnání dvou proměnných typu celé číslo:

```
// Základy programovacího jazyka Go
//
// - základní relační operátory s proměnnými
//   typu int

package main

import "fmt"

func main() {
    // dvojice proměnných typu int
    var x int = 1
    var y int = 2

    // výpis původních hodnot proměnných
    fmt.Printf("x = %d\n", x)
    fmt.Printf("y = %d\n", y)

    fmt.Println()

    // výpis výsledků relačních operací s proměnnými
    fmt.Printf("x == y: %t\n", x == y)
    fmt.Printf("x != y: %t\n", x != y)
    fmt.Printf("x > y: %t\n", x > y)
    fmt.Printf("x >= y: %t\n", x >= y)
    fmt.Printf("x < y: %t\n", x < y)
    fmt.Printf("x <= y: %t\n", x <= y)
}
```

Výsledky:

```
x = 1
y = 2
```

```
x == y: false
x != y: true
x > y: false
x >= y: false
x < y: true
x <= y: true
```

Naprosto stejným způsobem lze porovnávat i dvě hodnoty s plovoucí řádovou čárkou:

```
// Základy programovacího jazyka Go
//
// - základní relační operátory s proměnnými
//   typu float32

package main

import "fmt"

func main() {
    // dvojice proměnných typu float32
    var x float32 = 1.1
    var y float32 = 2.2

    // výpis původních hodnot proměnných
    fmt.Printf("x = %f\n", x)
    fmt.Printf("y = %f\n", y)

    fmt.Println()

    // výpis výsledků relačních operací s proměnnými
    fmt.Printf("x == y: %t\n", x == y)
    fmt.Printf("x != y: %t\n", x != y)
    fmt.Printf("x > y: %t\n", x > y)
    fmt.Printf("x >= y: %t\n", x >= y)
    fmt.Printf("x < y: %t\n", x < y)
    fmt.Printf("x <= y: %t\n", x <= y)
}
```

Nyní budou výsledky vypadat stejně, jako v předchozím příkladu:

```
x = 1.100000
y = 2.200000

x == y: false
x != y: true
x > y: false
x >= y: false
x < y: true
x <= y: true
```

U komplexních čísel, tj. u datových typů `complex64` a `complex128` je ovšem situace odlišná, protože tyto hodnoty můžeme porovnávat jen na rovnost a nerovnost (tvůrci programovacího jazyka Go se tak vyhnuli problémům, jak vlastně definovat relaci mezi dvojicí komplexních čísel):

```
// Základy programovacího jazyka Go
//
// - základní relační operátory s proměnnými
//   typu complex64

package main

import "fmt"

func main() {
    // dvojice proměnných typu complex64
    var x complex64 = 1 + 0i
    var y complex64 = 1 + 1i

    // výpis původních hodnot proměnných
    fmt.Printf("x = %v\n", x)
    fmt.Printf("y = %v\n", y)

    fmt.Println()

    // výpis výsledků relačních operací s proměnnými
    fmt.Printf("x == y: %t\n", x == y)
    fmt.Printf("x != y: %t\n", x != y)
}
```

Výsledky:

```
x = (1+0i)
y = (1+1i)

x == y: false
x != y: true
```

Zajímavé je, že hodnoty typu `boolean`, tj. hodnoty `true` a `false`, je možné v programovacím jazyce Go porovnávat pouze na rovnost a nerovnost. Zbylé čtyři relační operátory (tj. například „menší než“) nelze v tomto případě použít, tj. v jazyce Go není možné rozhodnout, zda je hodnota `true` větší či menší než `false`. Naproti tomu například v Pascalu nebo Pythonu platí, že `false < true`:

```
// Základy programovacího jazyka Go
//
// - základní relační operátory s proměnnými
//   typu boolean

package main

import "fmt"

func main() {
    // dvojice proměnných typu bool(ean)
    var x bool = true
    var y bool = false

    // výpis původních hodnot proměnných
    fmt.Printf("x = %t\n", x)
    fmt.Printf("y = %t\n", y)

    fmt.Println()

    // výpis výsledků relačních operací s proměnnými
    fmt.Printf("x == y: %t\n", x == y)
    fmt.Printf("x != y: %t\n", x != y)
    fmt.Printf("x > y: %t\n", x > y)
    fmt.Printf("x >= y: %t\n", x >= y)
    fmt.Printf("x < y: %t\n", x < y)
    fmt.Printf("x <= y: %t\n", x <= y)
}
```

Tento příklad nebude možné přeložit:

```
./02_09_relational_operators_boolean.go:16:28: invalid operation: x > y (operator > not
defined on bool)
./02_09_relational_operators_boolean.go:17:29: invalid operation: x >= y (operator >= not
defined on bool)
./02_09_relational_operators_boolean.go:18:28: invalid operation: x < y (operator < not
defined on bool)
./02_09_relational_operators_boolean.go:19:29: invalid operation: x <= y (operator <= not
defined on bool)
```

Nakonec si ukažme lexikografické porovnání dvou řetězců, což je v Go povolená operace (nikoli však například v jazyku C):

```
// Základy programovacího jazyka Go
//
// - základní relační operátory s proměnnými
//   typu řetězec

package main

import "fmt"

func main() {
    // dvojice proměnných typu string
    var x string = "foo"
    var y string = "bar"

    // výpis původních hodnot proměnných
    fmt.Printf("x = '%s'\n", x)
    fmt.Printf("y = '%s'\n", y)

    fmt.Println()

    // výpis výsledků relačních operací s proměnnými
    fmt.Printf("x == y: %t\n", x == y)
    fmt.Printf("x != y: %t\n", x != y)
    fmt.Printf("x > y: %t\n", x > y)
    fmt.Printf("x >= y: %t\n", x >= y)
    fmt.Printf("x < y: %t\n", x < y)
    fmt.Printf("x <= y: %t\n", x <= y)
}
```

Výsledky vypsané tímto ukázkovým příkladem budou vypadat následovně:

```
x = 'foo'
y = 'bar'

x == y: false
x != y: true
x > y: true
x >= y: true
x < y: false
x <= y: false
```

6.2.7 Logické operátory

V programovacím jazyku Go dále můžeme použít trojici operátorů, které jsou aplikovatelné na logické (pravdivostní) hodnoty `true` a `false` (tedy hodnot typu `bool`). Jedná se o logickou negaci, logický součin a logický součet. Tyto tři operátory se zapisují následujícím způsobem:

Operátor	Význam operátoru
!	negace pravdivostní hodnoty
&&	logický součin
||	logický součet

Základní funkci těchto tří operátorů si můžeme velmi snadno otestovat:

```
// Základy programovacího jazyka Go
//
// - logické operátory (pro hodnoty typu bool)

package main

import "fmt"

func main() {
    // dvojice proměnných typu bool(ean)
    var x bool = true
    var y bool = false
```

```
    // výpis původních hodnot
    fmt.Printf("x = %t\n", x)
    fmt.Printf("y = %t\n", y)

    fmt.Println()

    // výpis výsledků operací
    fmt.Printf("not x = %t\n", !x)
    fmt.Printf("not y = %t\n", !y)
    fmt.Printf("x && y = %t\n", x && y)
    fmt.Printf("x || y = %t\n", x || y)
}
```

S výsledky:

```
x = true
y = false

not x = false
not y = true
x && y = false
x || y = true
```

V jazyce Go není možné použít varianty `&` ani `|` s nezkráceným vyhodnocením výrazu. Operátory `&&` a `||` vždy používají zkrácené vyhodnocování (*short circuit*), což konkrétně znamená, že pokud je po vyhodnocení prvního operandu zřejmé, jaký bude výsledek operace, druhý operand se nevyhodnocuje. To má význam v případě, že se při vyhodnocování volají funkce popř. pokud se například dělí nulou atd.:

```
// Základy programovacího jazyka Go
//
// - zkrácené vyhodnocení operátorů && a ||

package main

import "fmt"

func f1() bool {
    println("f1")
    return true
}
```

```
func f2() bool {
    println("f2")
    return false
}

func f3() bool {
    println("f2")
    return false
}

func main() {
    // zkrácené vyhodnocení operátorů && a || má vliv na to,
    // které funkce f1() a f2() se budou volat
    fmt.Printf("short circuit &&: %v\n", f1() && f2())
    fmt.Printf("short circuit ||: %v\n", f1() || f2())
    fmt.Printf("short circuit &&: %v\n", f2() && f3())
    fmt.Printf("short circuit ||: %v\n", f2() || f3())
}
```

Při pohledu na výsledky vypsané tímto příkladem se skutečně můžeme přesvědčit, že se druhá funkce ve výrazu v některých případech vůbec nevolá:

```
f1
f2
short circuit &&: false
f1
short circuit ||: true
f2
short circuit &&: false
f2
f2
short circuit ||: false
```

6.2.8 Bitové operátory

Nyní se budeme zabývat bitovými operacemi, které jsou v programovacím jazyce Go realizovány s využitím pětice operátorů. Tyto operátory jsou vypsány v následující tabulce:

Operátor	Význam operátoru
^	negace bit po bitu (podobně jako operátor ~ v C)
&	logický součin prováděný bit po bitu
|	logický součet prováděný bit po bitu
^	logická nonekvivalence prováděná bit po bitu
&^	maskování bitů vybraných zadanou maskou (operace AND NOT)

První operátor je unární, tj. aplikuje se pouze na jediný operand, který je zapsán za operátor, podobně jako u dalších unárních operátorů (operace ++ a - jsou výjimkou, ale v tomto případě se nejedná o plnohodnotné operátory). Tento operátor slouží pro negaci všech bitů v celočíselné hodnotě a byl již popsán v podkapitole se všemi unárními operátory. Tento operátor je možné aplikovat jak na hodnoty bez znaménka (*unsigned*), tak i na hodnoty se znaménkem (*signed*).

Poznámka: v programovacích jazycích odvozených od céčka se tento operátor většinou nezapisuje znakem ^, ale s využitím znaku ~, což může být matoucí.

Další trojice operátorů zapisovaných znaky `&`, `|` a `^` patří mezi binární operátory, které jsou aplikovány na dvojici operandů. Opět se může jednat jak o hodnoty se znaménkem, tak i o hodnoty bez znaménka. Význam těchto operátorů odpovídá podobně zapisovaným operátorům v C, C++ či Javě. Zajímavější a ve vyšších programovacích jazycích pravděpodobně i unikátní je však poslední z bitových operátorů, který se zapisuje dvojicí znaků `&^`. Tento operátor provádí logický součin bit po bitu, ovšem všechny bity druhého operandu jsou nejdříve znegovány. Operátor `&^` se tedy používá pro takzvané maskování, kdy například budeme potřebovat vynulovat bit 1 pomocí operace `x &^ 1` atd..

Všechny bitové operátory jsou použity v tomto demonstračním příkladu:

```
// Základy programovacího jazyka Go
//
// - bitové operátory pro osmibitové hodnoty

package main

import "fmt"

func main() {
```

```
    // dvojice proměnných typu byte
    // (osmibitové celé číslo bez znaménka)
    var x byte = 42
    var y byte = 100

    // výpis původních hodnot
    fmt.Printf("x =      %08b\n", x)
    fmt.Printf("y =      %08b\n", y)

    // výpis výsledků bitových operací
    fmt.Printf("neg x =  %08b\n", ^x)
    fmt.Printf("neg y =  %08b\n", ^y)
    fmt.Printf("x && y = %08b\n", x&y)
    fmt.Printf("x || y = %08b\n", x|y)
    fmt.Printf("x ^ y =  %08b\n", x^y)
    fmt.Printf("x &^ y = %08b\n", x&^y)
}
```

Výsledky:

```
x =      00101010
y =      01100100
neg x =  11010101
neg y =  10011011
x && y = 00100000
x || y = 01101110
x ^ y =  01001110
x &^ y = 00001010
```

Využít lze i 32bitové operace:

```
// Základy programovacího jazyka Go
//
// - bitové operátory pro 32bitové hodnoty

package main

import "fmt"
```

```
func main() {
    // dvojice proměnných typu uint32
    // (32bitové celé číslo bez znaménka)
    var x uint32 = 42
    var y uint32 = 100

    // výpis původních hodnot
    fmt.Printf("x =      %032b\n", x)
    fmt.Printf("y =      %032b\n", y)

    // výpis výsledků bitových operací
    fmt.Printf("neg x =  %032b\n", ^x)
    fmt.Printf("neg y =  %032b\n", ^y)
    fmt.Printf("x && y = %032b\n", x&y)
    fmt.Printf("x || y = %032b\n", x|y)
    fmt.Printf("x ^ y =  %032b\n", x^y)
    fmt.Printf("x &^ y = %032b\n", x&^y)
}
```

Opět se podívejme na výsledky:

```
x =      00000000000000000000000000101010
y =      00000000000000000000000001100100
neg x =  11111111111111111111111111010101
neg y =  11111111111111111111111110011011
x && y = 00000000000000000000000000100000
x || y = 00000000000000000000000001101110
x ^ y =  00000000000000000000000001001110
x &^ y = 00000000000000000000000000001010
```

Poznámka: povšimněte si, že pokud se bitové operátory použijí ve své základní formě (tj. bez přiřazení), zapisují se „nalepené" na operandy – nepoužívají se zde oddělovací mezery (viz příkaz `go fmt`*).*

6.2.9 Bitové posuny

Dále si ukažme použití bitových posunů (*bit shifts*). U těchto operátorů je nutné dodržet jedno pravidlo – samotný posun (tedy pravý operand) musí být reprezentován kladným číslem (kon-

stantou) popř. proměnnou typu celé kladné číslo. Nejprve si ukažme, jak proběhne bitový posun doleva a doprava u hodnoty typu `byte` (tj. budeme posunovat osmibitové celé číslo bez znaménka):

```
// Základy programovacího jazyka Go
//
// - bitové posuny pro hodnoty typu byte

package main

import "fmt"

func main() {
    // celočíselný typ bez znaménka
    var x byte = 42

    // zobrazení původní hodnoty x
    fmt.Printf("x =     %08b\n", x)

    // zobrazení hodnoty x po jejím posunu doprava a doleva
    // vždy o jeden bit
    fmt.Printf("x>>1 =  %08b\n", x>>1)
    fmt.Printf("x<<1 =  %08b\n", x<<1)
}
```

Pro lepší názornost je jak vstupní hodnota, tak i výsledek posunu doprava resp. doleva, zobrazen ve dvojkové soustavě:

```
x =     00101010
x>>1 =  00010101
x<<1 =  01010100
```

U celočíselných hodnot se znaménkem se provedou aritmetické posuny, tj. takové operace, které zachovají znaménko původní (posouvané) hodnoty:

```
// Základy programovacího jazyka Go
//
// - bitové posuny pro hodnoty typu int

package main

import "fmt"
```