Die Programmiersprache Go

Alan A. A. Donovan
Brian W. Kernighan

ins Deutsche übertragen von
Hans-Werner Heinzen

Vorwort

> *„Go is an open source programming language that makes it easy to build simple, reliable, and efficient software“* (aus dem Internetauftritt `golang.org`)

Go wurde erdacht im September 2007 von Robert Griesemer, Rob Pike und Ken Thompson, damals alle bei Google beschäftigt, und im November 2009 wurde es veröffentlicht. Die Sprache und die sie begleitenden Werkzeuge zielten auf Ausdrucksstärke, auf Effizienz sowohl des Kompilierens als auch der Ausführung, sowie auf Effizienz beim Erarbeiten verlässlicher und robuster Programme.

Go ähnelt oberflächlich gesehen C, und wie C ist es ein Werkzeug für professionelle Programmierer, die große Wirkung mit kleinsten Mitteln erreichen. Aber es ist weit mehr als eine aktualisierte Version von C. Es übernimmt gute Ideen von vielen anderen Sprachen, vermeidet dabei aber Merkmale, die woanders zu Komplexität und unzuverlässigem Kode geführt haben. Seine Mechanismen für Nebenläufigkeit sind neu und wirkungsvoll, und seine Herangehensweise an Datenabstraktion und Objektorientiertes Programmieren ist ungewöhnlich flexibel. Und es besitzt eine automatische Speicherbereinigung, auch *Müllabfuhr* (*garbage collection*) genannt.

Go ist besonders gut geeignet zum Erstellen von Infrastruktur wie Netzwerkservern und Werkzeugen für Programmierer, doch im Grunde ist es eine universelle Sprache und wird in den verschiedensten Bereichen angewendet, wie für Grafiken, für Handy-Apps oder auch für maschinelles Lernen. Es ist als Ersatz für typfreie Skriptsprachen populär geworden, weil es eine gute Balance findet zwischen Ausdrucksstärke und Sicherheit: Go-Programme laufen typischerweise schneller als Programme dynamischer Sprachen, und sie leiden weniger unter Abstürzen und unerwarteten Typfehlern.

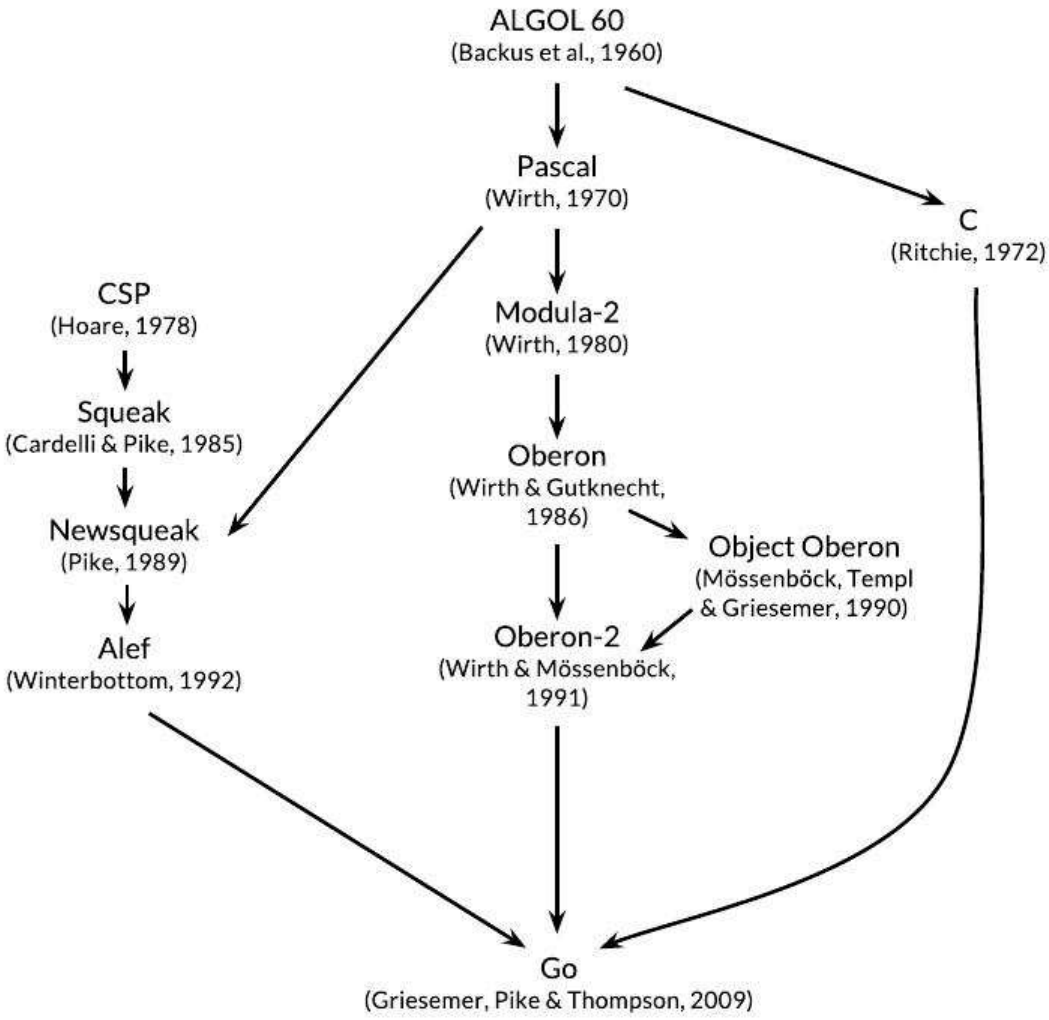
Go ist ein quelloffenes Projekt, also sind der Quellkode für Compiler, Bibliotheken und Werkzeuge für jeden zugänglich. Eine rührige, weltweit verteilte Gemeinde trägt zum Projekt bei. Go läuft auf UNIX-artigen Betriebssystemen – Linux, FreeBSD, OpenBSD, Mac OS X – und auf Plan 9 und Microsoft Windows. Programme, die für eine dieser Umgebungen geschrieben wurden, funktionieren in der Regel ohne Änderung auch in den anderen.

Dieses Buch soll Ihnen dabei helfen, mit Go von Anfang an effektiv zu sein: Nutzen Sie alle Vorteile der Sprache und der Standardbibliothek, und schreiben Sie klare, Go-typische und effiziente Programme.

Die Ursprünge von Go

Wie die Arten in der Biologie zeugen erfolgreiche Sprachen Nachkommen, die die Vorteile ihrer Ahnen beibehalten; Kreuzungen führen manchmal zu erstaunlichen Stärken; und hin und wieder erscheint eine neue Fähigkeit ohne Vorbild. Wir können viel darüber lernen, warum eine Sprache so ist, wie sie ist, und an welche Umgebung sie angepasst ist, wenn wir uns diese Einflüsse anschauen.

Die folgende Abbildung zeigt die wichtigsten Einflüsse früherer Programmiersprachen auf die Gestaltung von Go.



Go wird manchmal als „C-ähnliche Sprache“ bezeichnet, oder als „C für das 21ste Jahrhundert“. Von C hat Go geerbt: die Syntax für Ausdrücke, die Kontrollanweisungen, die

grundlegenden Datentypen, das Rufen mit Wertübergabe (call by value), die Zeiger, und vor allem C's Bemühen um Programme, die zu effizientem Maschinenkode kompiliert werden und die ganz natürlich mit dem jeweiligen Betriebssystem zusammenarbeiten.

Aber es gibt noch andere Vorfahren in Go's Stammbaum. Sehr einflussreich waren die Sprachen einer Linie, die von Niklaus Wirth herrührt und mit Pascal beginnt. Modula-2 regte das Paketierungskonzept an. Oberon schaffte den Unterschied zwischen Schnittstellen- und Implementierungdateien ab. Oberon-2 beeinflusste die Syntax für Paketimport und Paketdeklaration und lieferte die Syntax für die Methodendeklaration.

Eine weitere Linie in Go's Ahnenreihe, diejenige, die Go einzigartig unter den neuesten Programmiersprachen macht, ist eine Reihe wenig bekannter Experimentalsprachen, die bei den Bell-Laboratorien entwickelt wurden; sie wurden alle vom Konzept der *Kommunizierenden Sequentiellen Prozesse* (*communicating sequential processes, CSP*) beeinflusst, aus Tony Hoare's einflussreichem Aufsatz von 1978 über die Grundlagen der Nebenläufigkeit. In CSP ist ein Programm zusammengesetzt aus parallel arbeitenden Prozessen, die keinen gemeinsamen Zustand kennen; die Prozesse kommunizieren und stimmen sich ab mithilfe von Kanälen. Doch das CSP von Hoare war eine formale Sprache zum Beschreiben der grundlegenden Konzepte der Nebenläufigkeit, keine Sprache, um damit ausführbare Programme zu schreiben.

Rob Pike und andere experimentierten dann mit Implementierungen von CSP in echten Programmiersprachen. Die erste hieß Squeak („Eine Sprache zum Kommunizieren mit Mäusen"), die Maus- und Tastaturereignisse handhaben konnte, und zwar über statisch erzeugte Kanäle. Es folgte Newsqueak mit einer C-ähnlichen Syntax für Anweisungen und Ausdrücke und einer Pascal-ähnlichen Notation für Typen. Das war eine rein funktionale Sprache inklusive Müllabfuhr, und zielte erneut auf das Handhaben von Tastatur-, Maus- und Window-Ereignissen. Kanäle wurden vollwertig (first-class values), dynamisch erzeugt und Variablen zuweisbar.

Das Betriebssystem Plan 9 trug diese Ideen weiter zu einer Sprache namens Alef. Alef versuchte, aus Newsqueak eine lebensfähige Sprache für Systemanwendungen zu machen, doch ohne automatische Speicherbereinigung war das nebenläufige Programmieren zu mühsam.

Andere Konstrukte in Go bezeugen den Einfluss fremder Gene hier und dort; zum Beispiel ist `iota` eine lose Verbindung zu APL, und die lexikalische Reichweite in geschachtelten Funktionen stammt aus Scheme (und den meisten Sprachen seither). Wir finden aber auch neue Mutationen. Slices, eine Neuerung in Go, bieten dynamische Arrays mit effizientem Direktzugriff, erlauben aber auch raffiniertere Arrangements gemeinsamen Zugriffs, die an verkettete Listen erinnern. Und ganz neu in Go ist die Anweisung `defer`.

Das Projekt Go

Alle Programmiersprachen reflektieren die Programmierphilosophie ihrer Erfinder, wovon oft ein erheblicher Anteil Reaktion auf wahrgenommene Mängel früherer Sprachen ist. Das Go-Projekt entstand aus Frustration über einige Softwaresysteme bei Google, die

unter der Explosion ihrer Komplexität litten. (Dieses Problem hat nicht nur Google.)

Rob Pike drückte es so aus: „Komplexität ist multiplikativ“: wenn man ein Problem eines Systems dadurch löst, dass man einen Teil davon komplexer macht, dann folgt, langsam aber sicher, zusätzliche Komplexität in anderen Teilen. Mit dem ständigen Zwang, neue Funktionen, Optionen und Einstellmöglichkeiten anzubauen und möglichst schnell auszuliefern, ist es nur zu leicht, Einfachheit zu vernachlässigen; doch auf lange Sicht ist Einfachheit *der* Schlüssel zu guter Software.

Einfachheit verlangt zu Beginn eines Projekts mehr Aufwand, um eine Idee zu ihrer Essenz zu verdichten, und über die Lebensdauer eines Projekts hinweg verlangt sie mehr Disziplin, um gute Änderungen von schlechten oder gar bösartigen zu unterscheiden. Mit genügend großer Anstrengung kann man eine gute Änderung unterbringen, ohne das zu kompromittieren, was Fred Brooks „konzeptionelle Integrität“ des Designs nannte; mit einer schlechten Änderung geht das nicht, und eine bösartige tauscht Einfachheit gegen ihre leichtfertige Verwandte, die Bequemlichkeit, ein. Nur mit einfachem Design kann ein System, während es wächst, stabil, sicher und in sich stimmig bleiben.

Zum Go-Projekt gehören zum Einen die Sprache selbst, dann seine Werkzeuge und seine Standardbibliothek, und nicht zuletzt eine Kultur radikaler Einfachheit. Und als „High-Level“-Sprache, die erst kürzlich erschienen ist, hat sie den Vorteil des erkennenden Blicks zurück. Sein Fundament ist fest und gut gefügt: es gibt eine Müllabfuhr, ein Paketierungssystem, Vollwertfunktionen (first-class functions), lexikalische Reichweiten, eine Systemschnittstelle und unveränderliche Strings, deren Text grundsätzlich UTF-8-kodiert ist. Doch es gibt vergleichsweise wenige Einzelmerkmale (features), und es ist unwahrscheinlich, dass es je mehr werden. Zum Beispiel gibt es keine impliziten numerischen Konversionen, es gibt weder Kon- noch Destruktoren, kein Überladen von Operatoren und keine Vorgabewerte (default values) für Parameter. Es gibt keine Vererbung, keine generischen Konstrukte, keine Ausnahmezustände (exceptions), keine Makros, keine Annotation für Funktionen und keine thread-lokalen Variablen. Die Sprache ist ausgereift und stabil und garantiert rückwärtskompatibel, das heißt, ältere Go-Programme können mit neueren Versionen des Compilers und der Bibliothek kompiliert werden.

Go's Umgang mit Typen ist gut genug, um die meisten der fahrlässigen Fehler zu vermeiden, von denen Programmierer dynamischer Sprachen geplagt werden. Andererseits ist das Typsystem einfacher als das vergleichbarer Sprachen mit Typprüfung. Diese Herangehensweise führt manchmal zu Inseln „typfreien“ Programmierens innerhalb eines größeren Gerüsts von Typen, und Programmierer in Go gehen nicht so weit wie die in C++ oder Haskell, wo Sicherheitseigenschaften in Form von typbasierten Beweisen präsentiert werden. In der Praxis bietet Go mit seinem relativ starken Typsystem viel Sicherheit und Performanz zur Laufzeit, ohne die Last, die ein komplexeres Typsystem mit sich brächte.

Go ermutigt dazu, sich der Architektur zeitgenössischer Rechner bewusst zu werden, insbesondere der Bedeutung von Lokalität. Seine Standarddatentypen und die meisten Strukturen der Standardbibliothek sind so konstruiert, dass sie ganz natürlich ohne explizites Initialisieren oder implizite Konstruktoren arbeiten; also sind relativ wenige Speicherzuteilungen (memory allocations) und Speicherschreibzugriffe im Kode versteckt. Die Sammeltypen in Go (Structs und Arrays) enthalten ihre Elemente direkt, wodurch weni-

ger Speicher, weniger Speicherzuteilungen und weniger Zeigerauflösungen nötig werden als bei Sprachen, die mit indirekten Feldern arbeiten. Und weil der moderne Rechner eine Parallelmaschine ist, besitzt Go nebenläufige Fähigkeiten auf der Basis des schon erwähnten CSP. Die in ihrer Größe variablen Stapelspeicher (*stacks*) von Go's leichtgewichtigen Threads, genannt *Goroutinen*, sind zu Beginn klein genug, dass das Erzeugen einer Goroutine billig und das Erzeugen von millionen praktikabel ist.

Go's Standardbibliothek – oft beschrieben als „Batterien inbegriffen" – liefert sauber definierte Bausteine und Schnittstellen für Datenein- und -ausgabe, Textverarbeitung, Grafik, Kryptografie, Netzwerkprogrammierung und verteilte Anwendungen, mit Unterstützung vieler Standarddateiformate und -protokolle. Die Bibliotheken und Werkzeuge stützen sich stark auf Konventionen und vermeiden damit den Bedarf an Konfiguration und Erläuterungen, vereinfachen so die Programmlogik und machen verschiedene Go-Programme einander ähnlicher und damit leichter zu verstehen. Das Fertigen eines Projekts mit dem Go-Tool benutzt nur Dateinamen und Bezeichner des Kodes (und vereinzelt spezieller Kommentarzeilen) zum Festlegen all der Bibliotheken, ausführbaren Dateien, Tests, Messtests (benchmarks), Beispiele, plattformspezifischen Varianten und der Dokumentation des Projekts: die Spezifikation steckt in den Go-Quellen selbst.

Aufbau des Buches

Wir setzen voraus, dass Sie schon in mindestens einer Sprache programmiert haben, sei es in einer kompilierten Sprache wie C, C++ oder Java, sei es in einer interpretierten Sprache wir Python, Ruby oder JavaScript. Wir werden also nicht alles haarklein auseinandernehmen wie für unbeleckte Anfänger. Die Syntax wird vertraut erscheinen, ebenso Variablen, Konstanten, Ausdrücke, Kontrollkonstrukte und Funktionen.

Kapitel 1 ist eine Einführung in die grundlegenden Konstrukte von Go, die durch ein Dutzend Programme für alltägliche Aufgaben vorgestellt werden, als da wären: Dateien lesen und schreiben, Texte formatieren, Bilder erstellen, und das Kommunizieren von Internet-Klienten und Servern.

Kapitel 2 beschreibt die Strukturelemente eines Go-Programms: Deklarationen, Variablen, Typen, Pakete, Dateien und Reichweiten. Kapitel 3 erläutert Zahlen, Wahrheitswerte, Strings und Konstanten, und erklärt wie man Unicode verarbeitet. Kapitel 4 beschreibt Verbundtypen, also solche, die aus einfacheren zusammengesetzt werden, und zwar als Arrays, Maps, Strukturen und *Slices*, die Go-spezifische Form dynamischer Listen. Kapitel 5 deckt Funktionen ab und diskutiert Fehlerbehandlung sowie `panic`, `recover` und die Anweisung `defer`.

Kapitel 1 bis 5 enthalten also die Basis, die Teil jeder gängigen imperativen Sprache sind. Go-Syntax und Stil unterscheiden sich zuweilen von denen anderer Sprachen, doch das sollten sich Programmierer in kurzer Zeit aneignen können. Die übrigen Kapitel konzentrieren sich auf Themen, für die Go's Herangehensweise weniger konventionell ist: das sind Methoden, Interfaces, Nebenläufigkeit, Pakete, Testen und Reflexion.

An Objektorientiertes Programmieren geht Go auf eher ungewöhnliche Weise heran. Es

gibt weder eine Klassenhierarchie noch überhaupt Klassen; komplexeres Objektverhalten wird nicht über Vererbung sondern über Komposition von einfacherem erreicht. Jedem benutzerdefinierten Typ, also nicht nur den Strukturen, können Methoden zugeordnet werden, und die Beziehung zwischen konkretem und abstraktem Typ (Interface) ist eine implizite; also kann ein konkreter Typ auch Interfaces genügen, von denen sein Designer keine Ahnung hatte. Methoden werden in Kapitel 6 behandelt, Interfaces in Kapitel 7.

Kapitel 8 präsentiert Go's Herangehensweise an Nebenläufigkeit, die auf die Idee der kommunizierenden sequentiellen Prozesse (CSP) aufbaut und die Form von Goroutinen und Kanälen annimmt. Kapitel 9 erläutert dann die eher traditionellen Aspekte der Nebenläufigkeit, die sich auf gemeinsame Variablen stützt.

Kapitel 10 beschreibt Pakete, die dem Strukturieren der Bibliotheken dienen. Außerdem wird gezeigt, wie man das Go-Tool effektiv einsetzt, nämlich fürs Umwandeln, Testen, Untersuchen der Performanz, fürs Kodeformatieren, Dokumentieren, und so weiter, all das mit einem einzigen Kommando.

Kapitel 11 beschäftigt sich mit dem Testen, bei dem Go einen bemerkenswert einfachen Ansatz verfolgt: anstatt eines Frameworks, das mit komplexen Abstraktionen überladen ist, bevorzugt Go einfache Werkzeuge und Bibliotheken. Das `testing`-Paket bildet die Basis, auf die man, falls nötig, komplexere Abstraktionen aufbauen kann.

Kapitel 12 erörtert Reflexion, also die Fähigkeit eines Programms, seine Binnenstruktur zur Laufzeit zu untersuchen. Reflexion ist ein wirkmächtiges Werkzeug, aber eins, das mit Vorsicht einzusetzen ist. Diese Kapitel versucht die richtige Balance zu finden, indem es aufzeigt, wie Reflexion bei der Implementierung einiger wichtiger Go-Bibliotheken eingesetzt wurde. Kapitel 13 erklärt die mörderische Einzelheiten des Low-Level-Programmierens mit dem `unsafe`-Paket, welches das Go-Typsystem umgehen kann ... wenn das angemessen ist.

Zu jedem Kapitel gibt es eine Reihe von Übungen, mit denen Sie Ihr Verständnis prüfen und mit denen Sie außerdem Ergänzungen und Alternativen zu unseren Beispielen erkunden können.

Alle Kodebeispiele dieses Buches, triviale mal ausgenommen, stehen zum Download im öffentlichen Git-Repositorium unter `gopl.io` zur Verfügung. Jedes Beispiel wird durch den Paket-Importpfad identifiziert und kann bequem mit dem Kommando `go get` kopiert, kompiliert und installiert werden. Dazu brauchen Sie einen Ordner, der Ihr Go-Arbeitsbereich sein soll, und eine gesetzte Umgebungsvariable `GOPATH`, die dorthin zeigt. Das Go-Tool übernimmt dann das Anlegen der Ordner, wenn nötig. Zum Beispiel:

```
$ export GOPATH=$HOME/gobook        # Arbeitsbereich wählen
$ go get gopl.io/ch1/helloworld     # kopieren, kompilieren, installieren
$GOPATH/bin/helloworld              # ausführen
Hello, 世界
```

Zum Ausführen der Beispiele brauchen Sie mindestens Go Version 1.5.

```
$ go version
go version go1.5 linux/amd64
```

Ist Ihr installiertes Go-Tool älter, oder fehlt es ganz, so installieren Sie entsprechend den Anweisungen unter `https://golang.org/doc/install`.

Wo gibt's noch mehr Information?

Die beste Informationsquelle zu Go ist der offizielle Netzauftritt `https://golang.org`, mit Zugriff auf die Dokumentation inklusive der *Go-Sprachbeschreibung* (*Go Programming Language Specification*), auf die Standardpakete, und so weiter. Dort gibt es auch Anleitungen, wie man Go-Kode schreibt und wie man das gut macht, sowie eine Menge weiterer Texte und Videos, die eine wertvolle Ergänzung zu diesem Buch sind. Im Go-Blog `blog.golang.org` gibt es ein paar der besten Aufsätze über Go; das sind Texte zum Entwicklungsstand der Sprache, darüber, was für die Zukunft geplant wird, Berichte von Konferenzen sowie ausführliche Behandlung verschiedenster mit Go verknüpfter Themen.

Besonders nützlich beim Online-Zugriff auf Go (und bedauernswerte Beschränkung dieses papierenen Buches) ist die Möglichkeit, Go-Programme von den Web-Seiten aus zu starten, auf denen sie beschrieben werden. Die Funktionalität wird vom Go-Spielplatz (Go Playground) unter `play.golang.org` zur Verfügung gestellt, und kann in andere Seiten eingebettet werden; Beispiele sind die Dokumentationsseiten, die das Werkzeug `godoc` bereitstellt.

Auf dem Go-Spielplatz kann man bequem mit kleinen Programmen experimentieren, um das eigene Verständnis der Syntax, der Semantik oder der Pakete der Standardbibliothek zu prüfen, und er ersetzt in vielerlei Hinsicht die read-eval-print-Loop (REPL) anderer Sprachen. Durch dauerhafte (persistent) URLs eignet er sich hervorragend, um Kode-Schnipsel an andere weiterzugeben, um Fehler zu melden oder um Vorschläge zu machen.

Auf den Spielplatz baut die Go-Tour unter `tour.golang.org` auf, die mit einer Abfolge von kleinen interaktiven Lektionen über grundlegende Konzepte und Konstrukte durch die Sprache führt.

Hauptnachteil sowohl des Spielplatzes als auch der Tour ist, dass nur die Standardpakete importiert werden können und dass viele Bibliotheksfunktionen – zum Beispiel für Netzwerkprogrammierung – aus praktischen oder Sicherheitsgründen nur eingeschränkt benutzt werden können. Außerdem braucht man dafür jedes Mal den Zugriff aufs Internet. Also müssen Sie aufwendigere Experimente mit Go-Programmen auf Ihrem eigenen Rechner durchführen. Zum Glück ist der Download-Prozess unkompliziert, sodass es Sie nur ein paar Minuten kosten sollte, die Go-Distribution von `golang.org` zu kopieren und mit dem Entwickeln am eigenen Rechner zu beginnen.

Weil Go ein quelloffenes Projekt ist, dürfen Sie auch den Quellkode jedes Typs und jeder Funktion der Standardbibliothek online unter `https://golang.org/pkg` einsehen; derselbe Kode ist natürlich auch Teil der kopierten Distribution. Nutzen Sie das, um herauszufinden, wie etwas funktioniert, um Antworten auf Detailfragen zu finden, und um zu sehen wie die Experten richtig gutes Go schreiben.

Danksagung

Rob Pike und Russ Cox aus dem harten Kern des Go-Teams haben das Manuskript mehrere Male sorgfältig gelesen; ihre Kommentare zu allem, von der Wortwahl bis zu Aufbau und Struktur, waren von unschätzbarem Wert. Beim Übersetzen ins Japanische tat Yoshiki Shibata weit mehr als seine Pflicht: sein Adlerauge fand zahlreiche Unstimmigkeiten im englischen Text und mehrere Fehler im Kode. Wir wissen das gründliche Sichten und die Kommentare zum Manuskript von Brian Goetz, Corey Kosak, Arnold Robbins, Josh Bleecher Snyder und Peter Weinberger sehr zu schätzen.

Wir schulden Sameer Ajmani, Ittai Balaban, David Crawshaw, Billy Donohue, Jonathan Feinberg, Andrew Gerrand, Robert Griesemer, John Linderman, Minux Ma, Brian Mills, Bala Natarajan, Cosmos Nicolau, Paul Staniforth, Nigel Tao und Howard Trickey Dank für viele hilfreiche Vorschläge. Wir danken auch David Brailsford und Ralph Levien für Ratschläge zum Schriftsatz.

Unser Lektor Greg Doench bei Addison-Wesley brachte die ganze Sache ins Rollen und stand uns seitdem immer hilfreich zur Seite. Das AW-Produktionsteam – John Fuller, Dayna Isley, Julie Nahil, Chuti Prasertsith und Barbara Wood – war einsame Spitze; mehr Hilfe können Autoren nicht erwarten.

Alan Donovan möchte außerdem danken: Sameer Ajmani, Chris Demetriou, Walt Drummond und Reid Tatge bei Google für die zugestandene Zeit zum Schreiben; Stephen Donovan für Rat und Ermunterung zur rechten Zeit; und vor allen anderen seiner Frau Leila Kazemi für stete Begeisterung und unerschütterlichen Beistand für dieses Projekt, und das trotz langer Stunden Zerstreut- und Abwesenheit vom Familienleben meinerseits.

Brian Kernighan ist Freunden und Kollegen zutiefst dankbar für ihre Geduld und Nachsicht, während er langsam auf dem Pfad der Erkenntnis wandelte, und besonders seiner Frau Meg, die das Bücherschreiben und so vieles andere immer zuverlässig unterstützte.

New York
October 2015

Inhaltsverzeichnis

1 Einführung

Dieses Kapitel ist eine Rundreise durch die wichtigsten Teilbereiche von Go. Wir hoffen, schon hier genügend Information und Beispiele zu geben, damit Sie möglichst bald flügge und produktiv werden. Die Beispiele hier und im Rest des Buches haben Aufgaben im Blick, denen man auch im wirklichen Leben begegnet. Wir wollen Ihnen einen Vorgeschmack auf die große Bandbreite der Programme geben, die man mit Go schreiben kann, angefangen bei simpler Dateiverarbeitung und etwas Grafik bis hin zu nebenläufigen Internet-Klienten und Servern. Natürlich können wir in einem ersten Kapitel nicht alles erklären aber das Studium solcher Programme ist eine wirkungsvolle Lernmethode.

Beim Erlernen einer neuen Programmiersprache neigt man automatisch dazu, Kode so zu schreiben, wie man ihn in einer bereits bekannten Sprache schreiben würde. Seien Sie sich dessen bewusst und versuchen Sie es zu vermeiden. Wir haben uns bemüht zu zeigen, wie man guten Go-Kode schreibt; nutzen Sie das als Orientierung für Ihren eigenen Kode.

1.1 Hello, World

Wir wollen mit dem inzwischen traditionellen Hello-World-Beispiel beginnen, das schon 1978 im Einführungskapitel von „The C Programming Language“ stand. C hatte wohl den direktesten Einfluss auf Go, und „Hello, world“ veranschaulicht ein paar zentrale Konzepte.

gopl.io/ch1/helloworld

```
package main

import "fmt"

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

Go wird kompiliert! Der Go-Werkzeugsatz wandelt das Quellprogramm und all das, wovon es abhängt, in Befehle der Maschinensprache eines Rechners um. Alle Werkzeuge werden über ein einziges Kommando namens `go` erreicht, welches eine Reihe von Unterkommandos kennt. Das einfachste Unterkommando heißt `run`. Es kompiliert den Quellkode aus einer oder aus mehreren Quelldateien, deren Namen auf `.go` enden, bindet es mit Bibliotheksfunktionen zusammen, und startet dann die erzeugte ausführbare Datei. (Das `$` steht hier und im gesamten Buch für die Eingabeaufforderung.)

```
$ go run helloworld.go
```

Es überrascht nicht, dass jetzt erscheint:

```
Hello, 世界
```

Go arbeitet von Haus aus mit Unicode, kann also Text in allen Sprachen der Welt handhaben.

Wenn das Programm mehr als eine Eintagsfliege sein soll, werden Sie es wahrscheinlich nur einmal kompilieren und das Ergebnis für später speichern wollen. Dafür gibt es `go build`:

```
$ go build helloworld.go
```

Das erzeugt eine ausführbare Binärdatei namens `helloworld`, die jederzeit ohne weitere Vorbereitung ausgeführt werden kann:

```
$ ./helloworld
Hello, 世界
```

Wir haben alle wesentlichen Beispiele beschriftet, um daran zu erinnern, dass sie über das Quellkode-Repositorium dieses Buches unter `gopl.io` bezogen werden können:

`gopl.io/ch1/helloworld`

Wenn Sie `go get gopl.io/ch1/helloworld` ausführen, holen Sie sich damit den Quellkode und legen ihn in einem zugehörigen Verzeichnis ab. In den Abschnitten 2.6 und 10.7 steht mehr zu diesem Thema.

Sprechen wir nun über das Programm selbst. Go-Kode wird in Pakete gegliedert, die dem entsprechen, was in anderen Sprachen Bibliotheken oder Module sind. Ein Paket besteht aus einer oder mehreren `.go`-Quelldateien in einem Verzeichnis; dort steht, was das Paket tut. Jede Quelldatei beginnt mit einer Paketdeklaration (hier: `package main`), die festlegt, zu welchem Paket die Datei gehört, darauf folgt eine Liste anderer Pakete, die importiert werden, und dann kommen die zum Programm gehörenden Deklarationen dieser Datei. Die Go-Standardbibliothek enthält über 100 Pakete für die üblichen Aufgaben wie Eingabe, Ausgabe, Sortieren oder Manipulieren von Text. Zum Beispiel enthält das Paket `fmt` Funktionen für formatierte Ausgabe und für das Scannen von Eingabedaten. Eine wichtige Ausgabefunktion in `fmt` heißt `Println`; sie druckt einen oder mehrere Werte getrennt durch Leerzeichen und mit einem Zeilenvorschub am Ende, sodass eine einzelne Ausgabezeile entsteht.

Paket `main` ist speziell. Es definiert ein ausführbares Programm, keine Bibliothek. Innerhalb des `main`-Pakets ist die *Funktion* `main` ebenfalls speziell: hier beginnt die Programmausführung. Was auch immer `main` tut, das ist, was das Programm tut. Aber natürlich wird `main` normalerweise Funktionen anderer Pakete rufen, die dann den Großteil der Arbeit übernehmen, wie in unserem Beispiel die Funktion `fmt.Println`.

Wir müssen dem Compiler mitteilen, welche Pakete eine Quelldatei braucht; das ist die Aufgabe der `import`-Deklarationen, die auf die `package`-Deklaration folgen. Unser Hello-World-Programm benutzt nur eine Funktion aus einem anderen Paket – andere Programme haben mehr zu importieren.

Sie müssen die (und nur die) Pakete importieren, die auch benutzt werden. Eine Umwandlung schlägt fehl, wenn ein `import` fehlt oder wenn ein nicht benötigtes deklariert

wurde. Diese strenge Regel verhindert, dass sich Verweise auf unbenutzte Pakete während der Evolution eines Programms ansammeln.

Die `import`-Deklarationen müssen auf die `package`-Deklaration folgen. Dahinter besteht das Programm aus Deklarationen von Funktionen, Variablen, Konstanten und Typen (jeweils beginnend mit einem der Schlüsselwörter `func`, `var`, `const` oder `type`); die Reihenfolge ist meist ohne Belang. Unser erstes Programm ist in etwa so kurz, wie ein Programm nur sein kann: es deklariert nur eine Funktion, die wiederum nur eine weitere Funktion aufruft. (Um Platz zu sparen, werden wir bei manchen Beispielen hier im Text die `package`- und die `import`-Deklarationen unterschlagen; sie sind aber in der jeweiligen Quelldatei enthalten, und müssen es auch sein, damit kompiliert werden kann.)

Eine Funktionsdeklaration besteht aus dem Schlüsselwort `func`, dem Namen der Funktion, einer Parameterliste (bei `main` ist sie leer), einer Ergebnisliste (hier ebenfalls leer) und einem von geschweiften Klammern eingeschlossen Funktionsrumpf: das sind die Anweisungen, die definieren, was zu tun ist. Wir werden uns Funktionen im Kapitel 5 genauer anschauen.

Go braucht keine Semikolons am Ende von Anweisungen oder Deklarationen, außer wenn zwei davon auf einer Zeile stehen. Beim Kompilieren werden Zeilenvorschübe, denen bestimmte Vokabeln folgen, zu Semikolons konvertiert; das Platzieren der Zeilenvorschübe wirkt sich also auf das Analysieren (parsing) des Go-Kodes aus. Zum Beispiel muss deshalb die öffnende Schweifklammer `{` einer Funktion in derselben Zeile stehen wie das Ende der `func`-Deklaration, und im Ausdruck `x + y` ist ein Zeilenvorschub nach dem `+`-Operator aber nicht davor erlaubt.

In Bezug auf Kodeformatierung nimmt Go eine strenge Haltung ein. Das Werkzeug `gofmt` bringt Kode in das Standardformat, und das Unterkommando `fmt` des `go`-Kommandos wendet `gofmt` auf alle Dateien eines angegebenen Pakets an, bzw. ohne diese Angabe auf alle des aktuellen Verzeichnisses. Wir haben alle Go-Quelldateien dieses Buches von `gofmt` formatieren lassen, und Sie sollten sich angewöhnen, das auch mit Ihrem Kode zu tun. Ein Format zum Standard zu erheben, erspart uns viele unnütze Bagatelldiskussionen, und, was noch wichtiger ist, es ermöglicht eine große Bandbreite automatisierter Kodeumformungen, die nicht machbar wären, wären willkürliche Formate erlaubt.

Viele Texteditoren können so eingerichtet werden, dass sie bei jedem Dateischreiben `gofmt` aufrufen, sodass Ihr Quellkode immer ordentlich formatiert wird. Ein ähnliches Werkzeug namens `goimports` kümmert sich außerdem ums Einfügen oder Löschen von `import`-Deklarationen, je nach Notwendigkeit. Es ist nicht Teil der Standarddistribution, aber Sie können es sich leicht besorgen mit dem Kommando:

```
$ go get golang.org/x/tools/cmd/goimports
```

Für die meisten Nutzer ist es normal, mit dem Werkzeug `go` Pakete fernzukopieren (download), umzuwandeln, die zugehörigen Tests zu fahren, die Dokumentation anzuzeigen, und so weiter. Wir sehen uns das im Abschnitt 10.7 genauer an.

1.2 Kommandozeilenargumente

Die meisten Programme verarbeiten irgendwelche Eingaben und erzeugen irgendwelche Ausgaben; das ist im Wesentlichen die Definition eines Computerprogramms. Aber wie kommen die Programme an die Eingabedaten, auf denen sie dann operieren? Einige Programme erzeugen ihre Daten selbst, aber viel häufiger stammt die Eingabe aus einer externen Quelle: einer Datei, einer Netzwerkverbindung, aus der Ausgabe eines anderen Programms, vom Benutzer über Tastatur, als Kommandozeilenargumente und so weiter.

Das Paket `os` liefert unter anderem Funktionen, die mit dem Betriebssystem in programmunabhängiger Weise zusammenarbeiten. Kommandozeilenargumente sind dem Programm über eine Variable `Args` aus dem Paket `os` zugänglich; der Name außerhalb des `os`-Pakets lautet also `os.Args`.

Die Variable `os.Args` ist ein String-*Slice*[1]. Slices sind ein grundlegendes Konzept in Go, über das wir bald mehr sagen wollen. Fürs Erste stellen Sie sich ein Slice als Folge `s` einer veränderlichen Anzahl von Array-Elementen[2] vor, wobei ein einzelnes Element mit `s[i]` angesprochen werden kann und eine zusammenhängende Teilfolge mit `s[m:n]`. Die Anzahl der Elemente ist durch `len(s)` festgelegt. Wie in den meisten anderen Programmiersprachen wird auch in Go mit *halboffenen* Intervallen indiziert, welche den ersten Index enthalten, den letzten aber ausschließen, weil das die Logik vereinfacht. Zum Beispiel enthält ein Slice `s[m:n]`, wenn `0 ≤ m ≤ n ≤ len(s)` ist, `n-m` Elemente.

Das erste Element von `os.Args`, also `os.Args[0]`, ist der Kommandoname selbst; die anderen Elemente sind dann die Argumente, die dem Programm beim Start mitgegeben wurden. Ein Slice-Ausdruck der Form `s[m:n]` liefert ein Slice, das Bezug nimmt auf die Elemente `m` bis `n-1`; für unser nächstes Beispiel brauchen wir also jene im Slice `os.Args[1:len(os.Args)]`. Wird `m` oder `n` weggelassen, dann ist die Voreinstellung `0` bzw. `len(s)`, also können wir das gewünschte Slice kurz mit `os.Args[1:]` bezeichnen.

Es folgt eine Implementierung des Unix-Kommandos `echo`, das die mitgegebenen Kommandozeilenargumente auf einer Zeile ausgibt. Es importiert zwei Pakete, die hier in einer geklammerten Liste anstatt mit einzelnen `import`-Deklarationen angegeben sind. Beide Formen sind erlaubt – üblicherweise wird die Listenform benutzt. Die Reihenfolge der Importe ist egal; das Werkzeug `gofmt` sortiert die Paketnamen in alphabetischer Reihenfolge. (Wenn es mehrere Versionen für ein Beispielprogramm gibt, dann werden wir sie nummerieren, damit Sie immer wissen, wovon wir reden.)

gopl.io/ch1/echo1

```
// Echo1 zeigt seine Kommandozeilenargumente an.
package main

import (
    "fmt"
    "os"
)
```

[1] Ein String ist eine Zeichenkette. A.d.Ü

[2] Ein Array ist eine Kette gleichartiger Werte. A.d.Ü

```
func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Kommentare fangen mit `//` an. Der Text zwischen `//` und dem Zeilenende ist Erläuterung für Programmierer und wird vom Compiler nicht beachtet. Nach gutem Brauch beschreiben wir jedes Paket mit einem Kommentar unmittelbar vor seiner `package`-Deklaration; bei einem `main`-Paket besteht dieser Kommentar aus einem oder mehreren Sätzen, die das Programm als Ganzes beschreiben.

Die `var`-Deklaration deklariert zwei Variablen `s` und `sep` vom Typ `string`. Eine Variable kann mit seiner Deklaration auch vorbelegt werden. Geschieht das nicht explizit, so wird sie implizit mit dem *Nullwert* (*zero value*) ihres Typs vorbelegt; der ist `0` für numerische Typen und der leere String `""` für Strings. Die Deklaration in unserem Beispiel initialisiert also `s` und `sep` als leere Strings. Mehr zu Variablen und Deklarationen in Kapitel 2.

Für Zahlen bietet Go die üblichen arithmetischen und logischen Operatoren. Wenn man aber den Operator `+` auf Strings anwendet, dann *verkettet* er deren Werte, sodass der Ausdruck

```
sep + os.Args[i]
```

für die Verkettung von `sep` und `os.Args[i]` steht. Die Anweisung, die wir im Programm benutzen:

```
s += sep + os.Args[i]
```

ist eine *Zuweisung*, die den alten Wert von `s` mit `sep` und `os.Args[i]` verkettet und es dann wieder `s` zuweist; sie ist äquivalent zu:

```
s = s + sep + os.Args[i]
```

Der Operator `+=` ist ein *Zuweisungsoperator*. Zu jedem arithmetischen oder logischen Operator wie `+` oder `*` gibt es einen zugehörigen Zuweisungsoperator.

Das `echo`-Programm hätte seine Ausgabe auch Stück für Stück in einer Schleife drucken können, doch in dieser Version baut es einen String zusammen, indem es wiederholt Text am Ende anfügt. Der String `s` beginnt sein Leben als leerer String, das heißt mit dem Wert `""`, und jeder Schleifendurchlauf hängt ein Stück Text an; ab dem zweiten Durchlauf wird auch jedes Mal ein Leerzeichen eingefügt, sodass am Ende der Schleife ein solches zwischen jedem Argument steht. Das ist ein quadratischer Prozess, der bei einer große Anzahl von Argumenten teuer würde – für `echo` sind viele Argumente aber unwahrscheinlich. Wir werden in diesem Kapitel noch mehrere verbesserten `echo`-Versionen zeigen, und ein anderes Beispiel wird sich ernsthaft mit Ineffizienz auseinandersetzen.

Die Schleifenvariable `i` für den Index wird im ersten Teil der `for`-Schleife deklariert. Das Symbol `:=` ist Teil einer *Variablenkurzdeklaration*, einer Anweisung, die eine oder mehrere

Variablen deklariert und ihnen adäquate Typen entsprechend den Vorgabewerten zuweist; mehr dazu im nächsten Kapitel.

Die Inkrement-Anweisung `i++` addiert `1` zu `i`; sie ist äquivalent zu `i += 1`, was wiederum äquivalent ist zu `i = i + 1`. Analog gibt es eine Dekrement-Anweisung `i--`, die `1` subtrahiert. In Go sind das Anweisungen, und nicht Ausdrücke, wie das in den meisten Sprachen der C-Familie der Fall ist; also ist `j = i++` verboten. Außerdem gibt es sie nur in der Postfix-Form, sodass `--i` ebenfalls verboten ist.

In Go gibt es nur eine Schleifenart, die `for`-Schleife. Sie hat aber verschiedene Erscheinungsformen, wovon eine so aussieht:

```
for startschritt; bedingung; zählschritt {
    // keine oder mehr Anweisungen
}
```

Die drei Bestandteile der `for`-Klausel werden nie geklammert. Die Schweifklammern sind aber zwingend und die öffnende muss in derselben Zeile stehen wie der *`zählschritt`*.

Der optionale *`startschritt`* wird durchlaufen, bevor die Schleife ihre Arbeit aufnimmt. Wenn er vorhanden ist, dann muss er eine *einfache Anweisung* sein, das heißt, eine Variablenkurzdeklaration, eine Inkrement-Anweisung, eine Zuweisung oder ein Funktionsaufruf. Die *`bedingung`* ist ein Boole'scher Ausdruck, der zu Beginn jedes Durchlaufs ausgewertet wird; ist das Ergebnis `true`, werden die Anweisungen des Schleifenrumpfs ausgeführt. Der *`zählschritt`* wird nach dem Schleifenrumpf ausgeführt, danach die *`bedingung`* neu ausgewertet. Die Schleife endet, wenn die *`bedingung`* `false` wird.

Jeder dieser drei Teile kann weggelassen werden. Gibt es weder *`startschritt`* noch *`zählschritt`*, dann können auch die Semikolons wegfallen:

```
// eine traditionelle "while"-Schleife
for bedingung {
    // ...
}
```

Wird die `bedingung` weggelassen (in jeglicher Erscheinungsform), zum Beispiel hier:

```
// eine traditionelle Endlosschleife
for {
    // ...
}
```

dann hat man eine Endlosschleife; Schleifen dieser Art werden dann gewöhnlich anders beendet, zum Beispiel durch eine Anweisung `break` oder `return`.

Eine weitere Form der `for`-Schleife iteriert über den Wertebereich von Datentypen wie Strings oder Slices. Dies zeigen wir in einer zweiten Version des `echo`-Programms:

gopl.io/ch1/echo2

```
// Echo2 zeigt seine Kommandozeilenargumente an.
package main

import (
    "fmt"
```

```
    "os"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

Bei jedem Schleifendurchlauf erzeugt `range` ein Wertepaar: den Index und den Wert des Elements zu diesem Index. In unserem Beispiel brauchen wir den Index nicht, aber die Syntax einer `range`-Schleife verlangt, dass, wenn wir das Element haben wollen, wir uns auch mit dem Index abgeben müssen. Man könnte nun auf die Idee kommen, den Index einer offensichtlich temporären Variablen `temp` zuzuweisen und dann zu ignorieren. Doch Go erlaubt keine unbenutzten lokalen Variablen, sodass ein Umwandlungsfehler die Folge wäre.

Die Lösung heißt: Benutze den *Leeren Bezeichner* namens `_` (also: Unterstrich). Den Leeren Bezeichner kann man immer dann benutzen, wenn die Syntax nach einem Variablennamen verlangt, die Programmlogik aber nicht. Also zum Beispiel, wenn wir den unerwünschten Schleifenindex verwerfen wollen, weil wir nur den Elementwert brauchen. Die meisten Go-Programmierer werden diese Methode für das `echo`-Programm bevorzugen, weil das Indizieren von `os.Args` hier implizit und nicht explizit ist, was Fehler vermeidet.

Diese Programmversion benutzt eine Variablenkurzdeklaration, um `s` und `sep` zu initialisieren; wir hätten sie genausogut getrennt deklarieren können. Man kann String-Variablen auf verschiedene Weise deklarieren – gleichbedeutend sind:

```
s := ""
var s string
var s = ""
var s string = ""
```

Warum sollten Sie die eine der anderen Form vorziehen? Am kürzesten ist die erste, eine Variablenkurzdeklaration, aber man kann sie nur innerhalb einer Funktion verwenden, also nicht für Variablen auf Paketebene. Die zweite Form verlässt sich auf das standardmäßige Vorbelegen von Strings mit `""`. Die dritte Form wird kaum genutzt, außer zum gleichzeitigen Deklarieren mehrerer Variablen. Die vierte nennt explizit den Variablentyp, was redundant ist, wenn der Vorgabewert denselben Typ hat, aber notwendig, falls das nicht so ist. Sie sollten die ersten beiden Formen benutzen. Und zwar die mit explizitem Initialisieren, um zu betonen, dass der Vorgabewert wichtig ist, und die mit implizitem Initialisieren, wenn's egal ist.

Wie oben schon bemerkt, bekommt der String `s` bei jedem Schleifendurchlauf einen komplett neuen Inhalt. Die Anweisung `+=` erzeugt einen neuen String, indem sie den alten String, ein Leerzeichen und das jeweils nächste Argument verkettet und dem neuen String

`s` zuweist. Der alte Inhalt von `s` wird nicht länger gebraucht und bei nächster Gelegenheit durch die automatische Speicherbereinigung entsorgt.

Wenn die betroffene Datenmenge groß ist, kann das teuer werden. Einfacher und effizienter ist das mit der Funktion `Join` aus dem `strings`-Paket zu lösen:

gopl.io/ch1/echo3

```
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

Und wenn uns schließlich das Ausgabeformat egal ist, weil wir nur die Werte sehen wollen, vielleicht für eine Fehlersuche, so lassen wir `Println` für uns formatieren:

```
fmt.Println(os.Args[1:])
```

Die Ausgabe dieser Anweisung sieht aus wie die, die wir von `strings.Join` bekommen würden, nur ist sie eingeschlossen in eckige Klammern.

Übung 1.1: Ändern Sie das `echo`-Programm so, dass es außerdem noch `os.Args[0]` druckt, also den Namen mit dem es gestartet wurde.

Übung 1.2: Ändern Sie das `echo`-Programm so, dass es Index und Wert jedes seiner Argumente in eine eigene Zeile druckt.

Übung 1.3: Experimentieren Sie. Messen Sie die unterschiedlichen Laufzeiten unserer potentiell ineffizienten Versionen und der, die `strings.Join` benutzt. (Abschnitt 1.6 zeigt Teile des Pakets `time`, und in Abschnitt 11.4 steht, wie man Messtests (benchmarks) für eine systematische Performanzanalyse schreibt.)

1.3 Aufspüren von Duplikaten

Programme, die Dateien kopieren, drucken, durchsuchen, sortieren, zählen und so weiter, haben alle eine ähnliche Struktur: eine Schleife über die Eingabedaten, irgendeine Verarbeitung für jedes Element, und das Erzeugen einer Ausgabe während der Verarbeitung oder am Ende. Wir zeigen jetzt drei Varianten eines Programms namens `dup`; zum Teil inspiriert vom Unix-Kommando `uniq`, das nach aufeinanderfolgend doppelten Zeilen sucht. Die hier genutzten Datenstrukturen und Pakete sind Muster, die leicht angepasst werden können.

Die erste Version von `dup` druckt jede Zeile, die mehr als einmal von der Standardeingabe kommt mit ihrer Häufigkeit vorneweg.

gopl.io/ch1/dup1

```
// Dup1 druckt den Text jeder Zeile, die mehr als einmal von
// os.Stdin kommt, und die Häufigkeit ihres Auftretens.
package main

import (
    "bufio"
    "fmt"
```

```
    "os"
)

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // ACHTUNG: ignoriert mögliche Fehler in input.Err()
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Wie beim `for` werden in der `if`-Anweisung niemals Klammern um die Bedingung gesetzt, und der Anweisungsrumpf braucht seine Schweifklammern. Optional ist ein `else`-Zweig, der dann durchlaufen wird, wenn die Bedingung nicht erfüllt ist.

Eine *Map* speichert eine Menge von Schlüssel-Wert-Paaren, und für die Map-Operationen Speichern, Abrufen und Existenzprüfung wird eine konstante Laufzeit garantiert. Schlüssel darf von jedem Typ sein, dessen Werte mit `==` verglichen werden können; üblich sind Strings. Wert darf alles mögliche sein. Hier im Beispiel sind die Schlüssel Strings und die Werte sind Ganzzahlen. Die Standardfunktion `make` erzeugt eine neue, leere Map; sie kann aber noch mehr. Maps werden in Abschnitt 4.3 ausführlich behandelt.

Jedesmal, wenn `dup` eine Eingabezeile liest, wird die Zeile als Schlüssel zur Map benutzt, um den entsprechenden Wert hochzuzählen. Die Anweisung `counts[input.Text()]++` entspricht den folgenden beiden Anweisungen:

```
line := input.Text()
counts[line] = counts[line] + 1
```

Kein Problem, wenn die Map den Schlüssel noch nicht kennt. Beim ersten Auftreten einer neuen Zeile wird der rechtsstehende Ausdruck `counts[line]` zum Nullwert seines Typ ausgewertet; für Ganzzahlen ist das `0`.

Um das Ergebnis zu drucken, benutzen wir eine zweite `for`-Schleife mit `range`, diesmal über alle Elemente von `counts`. Wie oben produziert jeder Durchlauf zwei Ergebnisse, einen Schlüssel und den Wert des Map-Elements zu diesem Schlüssel. Die Reihenfolge der Map-Iterationen ist nicht festgelegt; faktisch aber ist sie zufällig und ändert sich ein übers andere Mal. Das ist so gewollt, weil damit verhindert wird, dass sich ein Programm auf eine bestimmte Ordnung verlässt, die ja doch nicht garantiert ist.

Weiter zum Paket `bufio`: dieses macht Ein- und Ausgabe effizient und bequem. Eine besonders wichtige Einrichtung ist ein Typ namens `Scanner`, welcher Eingabedaten liest und sie in Zeilen aufbricht; das ist meist der einfachste Weg, Daten in Zeilenform zu verarbeiten.

Das Programm benutzt eine Variablenkurzdeklaration für eine neue Variable `input` vom Typ `bufio.Scanner`.

```
input := bufio.NewScanner(os.Stdin)
```

Dieser Scanner liest von der Standardeingabe des Programms. Aufruf von `input.Scan()` liest die nächste Zeile und entfernt das Zeichen für Zeilenvorschub am Ende; ans Ergebnis kommt man dann mit `input.Text()`. Die Funktion `Scan` gibt `true` zurück, solange es eine Zeile gibt, und `false`, wenn keine Daten mehr da sind.

Wie `printf` in C und anderen Sprachen produziert `fmt.Printf` eine formatierte Ausgabe aus einer Liste von Ausdrücken. Sein erstes Argument ist ein Format-String, der festlegt, wie die nachfolgenden Argumente zu formatieren sind. Das Format jedes dieser Argumente wird durch ein Konversionszeichen festgelegt, einem Prozentzeichen gefolgt von einem Buchstaben. Zum Beispiel formatiert `%d` eine Ganzzahl in ihre Dezimaldarstellung und `%s` wird zum Wert des String-Operanden expandiert.

`Printf` kennt mehr als ein Dutzend solcher Konversionen, die von Go-Programmierern *Verben* (*verbs*) genannt werden. Die folgende Tabelle ist bei weitem nicht vollständig, zeigt aber schon viele Möglichkeiten:

`%d`	Ganzzahl in Dezimalform
`%x`, `%o`, `%b`	Ganzzahl in Hexadezimal-, Oktal-, Binärform
`%f`, `%g`, `%e`	Gleitkommazahl: `3.141593` `3.141592653589793` `3.141593e+00`
`%t`	Wahrheitswert: `true` oder `false`
`%c`	Rune (Unicode Kodenummer)
`%s`	String
`%q`	zitierter String "abc" oder Rune 'c'
`%v`	beliebiger Wert in natürlicher Form
`%T`	Typ eines beliebigen Werts
`%%`	Prozentzeichen (ohne Operand!)

Der Formatstring in `dup1` enthält außerdem das Tabulatorzeichen `\t` und den Zeilenvorschub `\n`. String-Literale können solche *Fluchtsequenzen* (*escape sequences*) enthalten, die für ansonsten unsichtbare Zeichen stehen. `Printf` druckt von sich aus keine Zeilenvorschübe. Es gibt die Konvention, dass Formatierungsfunktionen, deren Namen auf `f` enden, wie zum Beispiel `log.Printf` oder `fmt.Errorf`, sich nach den Formatierungsregeln von `fmt.Printf` richten. Analog folgen solche mit `ln` am Namensende dem Verhalten von `fmt.Println`, formatieren also ihre Argumente gemäß `%v` und hängen einen Zeilenvorschub an.

Viele Programme lesen wahlweise von der Standardeingabe oder aus den Dateien einer Dateinamenliste. Die nächste Version von `dup` kann sowohl die Standardeingabe lesen, als auch eine Liste von Dateinamen verarbeiten, indem es `os.Open` auf jede dieser Dateien anwendet:

gopl.io/ch1/dup2

```
// Dup2 druckt die Häufigkeit und den Inhalt jeder Zeile, die mehr als
// einmal vorkommt. Es liest von os.Stdin oder aus Dateien einer Liste.
```

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // ACHTUNG: ignoriert mögliche Fehler in input.Err()
}
```

Die Funktion `os.Open` gibt zwei Werte zurück. Der erste ist eine offene Datei (`*os.File`), woraus der `Scanner` im Anschluss liest.

Das zweite Ergebnis von `os.Open` ist ein Wert vom Standardfehlertyp `error`. Wenn `err` gleich dem speziellen Standardwert `nil` ist, dann war das Dateiöffnen erfolgreich. Die Datei wird gelesen, und nachdem das Ende der Eingabe erreicht ist, schließt `Close` die Datei wieder und gibt alle zugehörigen Ressourcen frei. Ist aber `err` nicht gleich `nil`, dann ist etwas schiefgegangen. In diesem Fall beschreibt der `error`-Wert das Problem. Unsere einfach gestrickte Fehlerbehandlung schreibt eine Nachricht in die Standardfehlerausgabe, benutzt dabei `Fprintf` und das Verb `%v`, welches den Wert eines beliebigen Typs

in einem Standardformat druckt. Dann macht `dup` mit der nächsten Datei weiter; die Anweisung `continue` springt zum nächsten Durchlauf der umschließenden `for`-Schleife.

Weil wir den Kode zumutbar kurz halten wollten, verhalten sich unsere ersten Beispiele absichtlich etwas nachlässig in Bezug auf Fehlerbehandlung. Klar, wir müssen den Fehler von `os.Open` abfragen, doch wir ignorieren die weniger wahrscheinlichen Fehler, die während des Lesens mit `input.Scan` auftreten könnten. Wir wollen die Stellen markieren, an denen wir Fehlerabfragen ausgelassen haben, und in Abschnitt 5.4 werden wir dann Fehlerbehandlung im Einzelnen betrachten.

Beachten Sie, dass der Aufruf von `countLines` vor dessen Deklaration erfolgt. Funktionen und andere Objekte auf Paketebene können in beliebiger Reihenfolge deklariert werden.

Eine Map ist eine *Referenz* auf eine mit `make` erzeugte Datenstruktur. Wenn man eine Map einer Funktion übergibt, erhält die Funktion eine Kopie dieser Referenz. Also wird jede Änderung, die die Funktion an der eigentlichen Datenstruktur vornimmt, auch über die Referenz des Rufers auf die Map sichtbar. In unserem Beispiel kann `main` die Werte sehen, die von `countLines` in die `counts`-Map eingefügt wurden.

Die bisherigen Versionen von `dup` arbeiten in einem „Datenstrom-Modus“; je nach Bedarf wird Eingabe gelesen und in Zeilen aufgespalten, sodass diese Programme im Prinzip beliebig viele Eingabedaten verarbeiten können. Alternativ dazu könnte man alle Eingabedaten auf einmal in den Speicher saugen, sie komplett in Zeilen zerlegen, und diese dann verarbeiten. Die folgenden Version, `dup3`, arbeitet so. Sie macht uns vertraut mit der Funktion `ReadFile` aus dem `io/ioutil`-Paket, die den gesamten Inhalt einer Datei anhand ihres Namens liest, und mit `strings.Split`, welche einen String in ein Slice aus Teilstrings aufspaltet. (`Split` ist das Gegenstück zu `strings.Join`, das wir schon kennengelernt haben.)

Wir haben `dup3` etwas vereinfacht. Erstens liest es nur Dateien anhand der Namen und nicht aus der Standardeingabe, weil `ReadFile` als Argument einen Namen verlangt. Zweitens wandert das Zählen wieder zurück nach `main`, weil es nur noch an einer Stelle gebraucht wird.

gopl.io/ch1/dup3

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
```

```
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

`ReadFile` gibt ein Byte-Slice zurück, das zu einem String konvertiert werden muss, bevor es mit `strings.Split` zerhackt werden kann. Wir werden Strings und Byte-Slices im Abschnitt 3.5.4 ausführlich behandeln.

Unter der Haube benutzen `bufio.Scanner`, `ioutil.ReadFile` und `ioutil.WriteFile` die Methoden `Read` und `Write` von `*os.File`, doch solche kleinteiligen Routinen braucht man selten direkt. Höher aggregierte Funktionen, wie eben die aus `bufio` oder `io/ioutil`, sind viel leichter zu benutzen.

Übung 1.4: Ändern Sie `dup2` so, dass es die Namen aller Dateien druckt, in denen jede doppelte Zeile vorkommt.

1.4 GIF-Animation

Das nächste Programm zeigt eine elementare Anwendung von Go's Standardpaket `image`, mit dem wir eine Folge von Bitmap-Bildern erzeugen und diese dann als GIF-Animation kodieren wollen. Die Bilder – man nennt sie *Lissajous-Figuren* – waren der Grundstoff für die visuellen Effekte in den Science-Fiction-Filmen der 60er Jahre. Es sind Kurven die durch Überlagerung zweier harmonischer Kurven in zwei Dimensionen entstehen, wenn zum Beispiel zwei Sinuskurven den x- und den y-Eingang eines Oszilloskops beliefern. Abbildung 1,1 zeigt ein paar Beispiele.

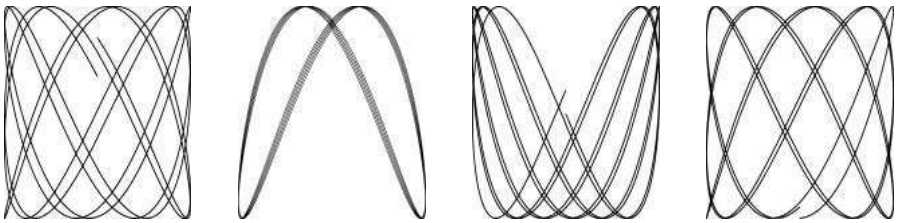


Abbildung 1.1 Vier Lissajous-Figuren.

Der Kode enthält einige neue Konstrukte, als da wären: `const`-Deklarationen, `struct`-Typen und Verbundliterale. Es rechnet außerdem mit Gleitkommazahlen, was in den anderen Beispielen kaum vorkommt. Diese Themen werden hier nur kurz angerissen;

Einzelheiten überlassen wir späteren Kapiteln, weil wir hier erstmal nur zeigen wollen, wie Go aussieht und welche Arten von Aufgaben mit Go und seiner Bibliothek einfach erledigt werden können.

gopl.io/ch1/lissajous

```
// Lissajous generiert GIF-Animationen aus zufälligen L.-Figuren.
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)

var palette = []color.Color{color.White, color.Black}

const (
    whiteIndex = 0 // erste Farbe der Palette
    blackIndex = 1 // nächste Farbe der Palette
)

func main() {
    lissajous(os.Stdout)
}

func lissajous(out io.Writer) {
    const (
        cycles = 5   // Anzahl vollständiger x-Zyklen
        res = 0.001  // Winkelauflösung
        size = 100   // Leinwand geht von -size bis +size
        nframes = 64 // Anzahl der Einzelbilder
        delay = 8    // Verzögerung zwischen Einzelbildern [in 10ms]
    )
    rand.Seed(time.Now().UTC().UnixNano())
    freq := rand.Float64() * 3.0 // Frequenzverschiebung, y-Oszillator
    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0 // Phasendifferenz
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
        img := image.NewPalette(rect, palette)
        for t := 0.0; t < cycles*2*math.Pi; t += res {
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int(x*size+0.5),
                size+int(y*size+0.5), blackIndex)
        }
```

```
            phase += 0.1
            anim.Delay = append(anim.Delay, delay)
            anim.Image = append(anim.Image, img)
        }
        gif.EncodeAll(out, &anim) // ACHTUNG: ignoriert mögliche Fehler
    }
```

Importiert man ein Paket wie `image/color`, dessen Importpfad mehrere Bestandteile enthält, so verweist man auf das Paket mit einem Namen, der von dem letzten Bestandteil abgeleitet ist. Also stammt die Variable `color.White` aus dem Paket `image/color` und `gif.GIF` gehört zu `image/gif`.

Eine `const`-Deklaration (Abschnitt 3.6) vergibt Namen an Konstanten, das heißt, an Werte, die bei der Umwandlung fixiert werden, wie hier die numerischen Parameter für Zyklen, Einzelbilder und Verzögerung. Wie `var`-Deklarationen können `const`-Deklarationen auch auf Paketebene erscheinen (dann sind die Namen im gesamten Paket sichtbar), oder auf Funktionsebene (dann sind die Namen nur innerhalb der Funktion sichtbar). Der Wert einer Konstanten muss eine Zahl, ein String oder ein Wahrheitswert sein.

Die Ausdrücke `[]color.Color{...}` und `gif.GIF{...}` sind *Verbundliterale* (Abschnitte 4.2, 4.4.1); das ist eine kompakte Schreibweise zum Instantiieren der Verbundtypen in Go mithilfe einer Liste von Elementwerten. Hier ist das erste ein Slice und das zweite eine *Struktur*.

Der Typ `gif.GIF` ist ein Strukturtyp (Abschnitt 4.4). Eine Struktur ist eine Gruppe von Werten, die man *Felder* nennt, oft von verschiedenem Typ, zusammengefasst in einem Objekt, das als Einheit behandelt werden kann. Die Variable `anim` ist eine Struktur vom Typ `gif.GIF`. Das Strukturliteral erzeugt einen Strukturwert, dessen `LoopCount`-Feld auf `nframes` gesetzt wird; alle anderen Felder bekommen den Nullwert ihres Typs. Die einzelnen Felder einer Struktur können mit der Punkt-Schreibweise angesprochen werden, so wie in den beiden letzten Zuweisungen, die explizit die Felder `Delay` und `Image` in `anim` verändern.

Die Funktion `lissajous` schachtelt zwei Schleifen. Die äußere Schleife macht 64 Durchläufe, in denen jeweils ein Einzelbild der Animation produziert wird. Dort wird ein neues 201x201-Bild erzeugt, aus einer Palette mit zwei Farben, weiß und schwarz. Alle Pixel werden zunächst mit dem Nullwert der Palette (der nullten Farbe) vorbelegt; die hatten wir auf weiß gesetzt. Jeder Durchlauf durch die innere Schleife erzeugt ein neues Bild, indem einige Pixel auf schwarz gesetzt werden. Das Ergebnis wird mit der Standardfunktion `append` (Abschnitt 4.2.1) an eine Liste von Bildern in `anim` angehängt, jeweils zusammen mit einer Pause von 80ms. Schließlich wird die Folge von Bildern und Pausen ins GIF-Format umgewandelt und in den Ausgabestrom `out` geschrieben. Typ von `out` ist `io.Writer`, wodurch die Ausgabe auf eine große Zahl verschiedenster Ziele möglich wird – wie, das werden wir bald sehen.

Die innere Schleife steuert die beiden Oszillatoren. Der x-Oszillator ist einfach eine Sinus-Funktion. Der y-Oszillator ist auch sinusartig, doch seine Frequenz unterscheidet sich vom x-Oszillator um einen zufälligen Faktor zwischen 0 und 3; die Phasenverschiebung gegenüber dem x-Oszillator ist zu Beginn null, nimmt aber mit jedem Einzelbild der

Animation zu. Die Schleife wird wieder und wieder durchlaufen, bis der x-Oszillator 5 komplette Zyklen vollendet hat. Für jeden Durchlauf ruft sie `SetColorIndex`, um die zu (x, y) gehörenden Pixel auf schwarz zu setzen, was Position 1 der Palette ist.

Die Funktion `main` ruft `lissajous` und weist sie an, in die Standardausgabe zu schreiben, und so produziert dieses Kommando eine GIF-Animation mit Einzelbildern, die aussehen wie die in Abbildung 1.1:

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

Übung 1.5: Ändern Sie die Farbpalette des Lissajous-Programms zu grün auf schwarz – das sieht authentischer aus. Um eine Farbe der RGB-Notation *#RRGGBB* zu erzeugen, benutzen Sie `color.RGBA{0x`*RR*`, 0x`*GG*`, 0x`*BB*`, 0xff}`, wobei jedes hexadezimale Zeichenpaar für die Intensität der Rot-, Grün- oder Blau-Komponente des Pixels steht.

Übung 1.6: Ändern Sie das Lissajous-Programm so, dass es Bilder in mehreren Farben erzeugt. Dazu fügen Sie der Palette weitere Farben hinzu und zeigen sie sie an, indem Sie das dritte Argument für `SetColorIndex` irgendwie interessant ändern.

1.5 Lesen einer URL

Für viele Anwendungen ist der Zugriff aufs Internet so wichtig wie der auf das lokale Dateisystem. Go bietet, zusammengeführt im Verzeichnis `net`, mehrere Pakete, die es leicht machen, Daten übers Internet zu senden und zu empfangen und „Low-Level"-Netzwerkverbindungen und Server einzurichten; dafür sind Go's Fähigkeiten im Bereich Nebenläufigkeit (Einführung in Kapitel 8) besonders nützlich.

Um zu zeigen, was mindestens nötig ist, um über HTTP Daten zu beschaffen, hier nun ein simples Programm namens `fetch`, das sich die Daten einer gewünschten URL besorgt und sie ohne Interpretation als Text ausdruckt; die Idee dazu stammt von dem immens wertvollen Werkzeug `curl`. Normalerweise würde man mehr mit den Daten anstellen, doch das hier zeigt die Grundidee. Wir werden das Programm noch oft benutzen.

gopl.io/ch1/fetch

```
// Fetch druckt die Daten einer URL.
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
```

```
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err) os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

Das Programm führt zwei Funktionen aus den Paketen `net/http` und `io/ioutil` ein. Die Funktion `http.Get` macht eine Anfrage über HTTP und gibt, wenn keine Fehler aufgetreten sind, das Ergebnis in einer Antwortstruktur `resp` zurück. Das `resp`-Feld `Body` enthält die Antwort des Servers als lesbaren Datenstrom. Daraufhin liest `ioutil.ReadAll` die komplette Antwort und sichert das Ergebnis in `b`. Der Datenstrom `Body` wird geschlossen, um Speicherlecks zu vermeiden, und `Printf` schreibt dann die Antwort in die Standardausgabe.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...
```

Wenn die HTTP-Anfrage fehlschlägt, so meldet `fetch` stattdessen den Fehler:

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

Im Fehlerfall, egal welchem, bewirkt `os.Exit(1)`, dass der Prozess mit Statuskode `1` beendet wird.

Übung 1.7: Ein Funktionsaufruf `io.Copy(dst, src)` liest von `src` und schreibt nach `dst`. Benutze diesen anstelle von `ioutil.ReadAll`, um die Antwort nach `os.Stdout` zuschreiben, ohne dass ein großer Puffer für den ganzen Datenstrom nötig wäre.

Übung 1.8: Ändern Sie `fetch` so, dass ein `http://` jeder URL vorangestellt wird, falls es noch fehlt. Sie werden dafür vielleicht `strings.HasPrefix` benutzen wollen.

Übung 1.9: Ändern Sie `fetch` außerdem so, dass der HTTP-Statuskode aus `resp.Status` ausgegeben wird.

1.6 URLs simultan verarbeiten

Ein besonders interessanter neuer Aspekt ist Go's Unterstützung für nebenläufiges Programmieren. Das ist ein großes Thema, dem sich die Kapitel 8 und 9 widmen werden. Hier gibt's deshalb nur einen kleinen Vorgeschmack auf die beiden wichtigsten Mechanismen: Goroutinen und Kanäle.

Das nächste Programm, `fetchall`, liest die Daten der URLs genauso wie das vorherige, nur dass es von vielen URLs simultan liest und der Prozess nicht mehr Zeit braucht, als der langsamste Lesevorgang, also nicht wie für alle Lesevorgänge aufaddiert. Dieses `fetchall` wirft alles Gelesene weg und meldet nur Größe und Laufzeit für jeden Vorgang:

gopl.io/ch1/fetchall

```
// Fetchall liest die Daten von URLs simultan
// und meldet jeweils Laufzeit und Größe.
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)

func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
        go fetch(url, ch) // starten eine Goroutine
    }
    for range os.Args[1:] {
        fmt.Println(<-ch) // empfangen auf Kanal ch
    }
    fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}

func fetch(url string, ch chan<string) {
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt. Sprint(err) // senden auf Kanal ch
        return
    }
    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // Lecks vermeiden
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs %7d %s", secs, nbytes, url)
}
```

Hier ein Beispiel:

```
$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s     6852  https://godoc.org
0.16s     7261  https://golang.org
0.48s     2475  http://gopl.io
0.48s elapsed
```

Eine *Goroutine* ist die nebenläufige Ausführung einer Funktion. Ein *Kanal* ist ein Kommunikationsmechanismus, der einer Goroutine erlaubt, den Wert eines festgelegten Typs an eine anderen Goroutine zu übermitteln. Die Funktion `main` läuft in einer Goroutine, und die Anweisung `go` erzeugt weitere Goroutinen.

Die Funktion `main` erzeugt mit `make` einen String-Kanal. Für jedes Kommandozeilenargument startet die Anweisung `go` in der ersten `range`-Schleife eine neue Goroutine, die asynchron `fetch` aufruft, um von der URL mithilfe von `http.Get` zu lesen. Die Funktion `io.Copy` liest den Antwortrumpf und wirft ihn weg, indem sie in den `ioutil.Discard`-Datenstrom schreibt. `Copy` gibt einen Byte-Zähler und einen möglichen Fehler zurück. Für jedes Resultat sendet `fetch` eine Zusammenfassung als Textzeile in den Kanal `ch`. Die zweite Schleife in `main` empfängt und druckt diese Zeilen.

Sobald eine Goroutine auf einem Kanal zu senden oder zu empfangen versucht, wartet (blockiert) sie solange, bis eine andere Goroutine die korrespondierende Operation auszuführen versucht; dann wird der Wert übertragen und beide Goroutinen arbeiten unabhängig voneinander weiter. In unserem Beispiel sendet jedes `fetch` einen Wert (`ch <-` *Ausdruck*) in den Kanal `ch` und `main` empfängt sie alle (`<-ch`). Dadurch dass `main` das Drucken übernimmt, ist sichergestellt, dass die Ausgabe jeder Goroutine auch als Einheit gedruckt wird, ohne die Gefahr einer Überlappung, wenn zwei Goroutinen gleichzeitig fertig werden.

Übung 1.10: Finden Sie eine Web-Seite, die eine große Menge an Daten liefert. Untersuchen Sie, ob sich das Caching ihres Betriebssystems auswirkt: Starten Sie `fetchall` zweimal kurz hintereinander und prüfen Sie, ob sich die Zeiten stark unterscheiden. Ändern Sie `fetchall` so, dass es seine Ausgabe in eine Datei druckt, wo man sie untersuchen kann.

Übung 1.11: Testen Sie `fetchall` mit einer langen Argumentliste, zum Beispiel einer Zusammenstellung aus der Top-Million Web-Seiten, die man bei `alexa.com` findet. Wie verhält sich das Programm, wenn eine Web-Seite nicht antwortet? (Abschnitt 8.9 beschreibt eine Vorgehensweise, wie man mit solchen Fällen umgehen kann.)

1.7 Ein Web-Server

Mit der Go-Bibliothek ist es einfach, einen Web-Server zu schreiben, der auf Client-Anfragen, wie die von `fetch`, antwortet. In diesem Abschnitt wollen wir einen minimal kleinen Server zeigen, der die Pfadkomponente der URL zurückgibt, mit der er gerufen wurde. Das heißt, wenn die Anfrage lautet `http://localhost:8000/hello`, dann ist die Antwort `URL.Path = "/hello"`.

gopl.io/ch1/server1

```
// Server1 ist ein minimaler "Echo"-Server.
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // alle Anfragen rufen handler
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// handler gibt die Pfadkomponente der URL der Anfrage r zurück.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

Das Programm besteht aus nur wenigen Zeilen, weil Bibliotheksfunktionen die meiste Arbeit übernehmen. Die Funktion `main` verknüpft eine Funktion `handler` mit den URL-Pfaden, die mit `/` beginnen – also mit allen möglichen – und startet einen Server, der auf eingehende Anfragen auf dem Port 8000 lauscht. Eine Anfrage wird durch eine Struktur vom Typ `http.Request` vertreten, welche mehrere Felder enthält, eins davon mit der URL der eingegangenen Anfrage. Wenn eine Anfrage ankommt, wird sie an die Funktion `handler` weitergereicht, die dann die Pfadkomponente (`/hello`) aus der Anfrage-URL extrahiert und mit `fmt.Fprintf` als Antwort zurücksendet. Web-Server werden in Abschnitt 7.7 ausführlich erklärt.

Starten wir nun den Server asynchron. Dazu fügt man auf Mac OS X und Linux am Ende des Kommandos ein Kaufmanns-Und (`&`) an; auf Microsoft Windows müssen sie das Kommando ohne Kaufmanns-Und in einem eigenen Kommandofenster starten.

```
$ go run src/gopl.io/ch1/server1/main.go &
```

Als Klient können wir dann auf der Kommandozeile anfragen:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

Alternativ können wir den Server auch von einem Browser aus, wie in Abbildung 1.2 ansteuern.

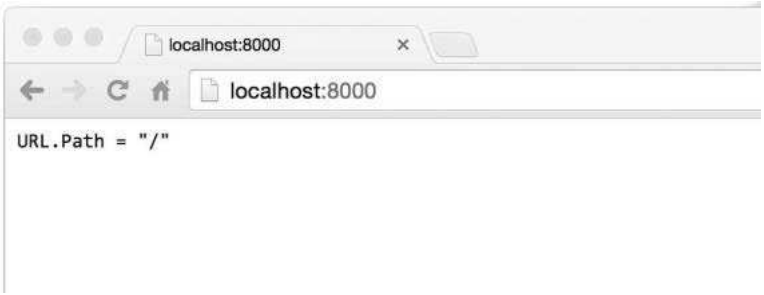


Abbildung 1.2 Eine Antwort vom Echo-Server.

Der Server kann leicht erweitert werden. Eine nützliche Erweiterung wäre zum Beispiel eine eigene URL, die einen Zustand zurückgibt. Die folgende Version macht das Gleiche wie vorher, zählt aber zusätzlich die Anzahl der Anfragen; eine Anfrage an `/count` gibt dann den Zählerstand zurück; Anfragen an `/count` selbst werden nicht gezählt.

gopl.io/ch1/server2

```
// Server2 ist ein minimaler "echo"-Server, der auch zählt.
package main

import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// handler gibt die Pfadkomponente der Anfrage-URL zurück.
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}

// counter gibt die Anzahl der bisherigen Anfragen zurück.
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
```

```
        mu.Unlock()
    }
```

Der Server kennt zwei Bearbeiterfunktionen (handler), und die Anfrage-URL entscheidet darüber, welche der beiden gerufen wird; eine Anfrage an `/count` aktiviert den Bearbeiter `counter`, alle anderen führen zu `handler`. Ein Aufrufmuster, das mit einem Schrägstrich endet, passt zu allen URL-Pfaden, die mit diesem Muster beginnen. Hinter den Kulissen startet der Server für jede Anfrage einen Bearbeiter in einer eigenen Goroutine, sodass mehrere Anfragen simultan bedient werden können. Wenn allerdings zwei nebenläufige Anfragen versuchen, den Zähler zur gleichen Zeit zu ändern, kann der Zähler korrumpiert werden; das Programm hätte dann einen echten Fehler, einen sogenannten „Datenwettlauf“ (race condition) (Abschnitt 9.1). Wir vermeiden das, indem wir sicherstellen, dass immer maximal eine Goroutine Zugriff auf die Variable hat; das ist der Zweck der Aufrufe `mu.Lock()` und `mu.Unlock()`, die jeden Zugriff auf `count` umschließen. Wir werden uns nebenläufige Verarbeitung mit gemeinsamen Variablen in Kapitel 9 genauer ansehen.

Als Beispiel eines Programms, das dabei hilft, Anfragen genauer auf Fehler hin zu untersuchen, kann die Funktion `handler` Kopf- und Formulardaten melden:

gopl.io/ch1/server3

```
// handler gibt die HTTP-Anfrage zurück.
func handler (w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)

    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}
```

Die Felder der Struktur `http.Request` werden damit in folgender Form ausgegeben:

```
GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"]
Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."]
Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```

Beachten Sie, dass der Aufruf der Funktion `ParseForm` in der `if`-Anweisung enthalten ist. Go erlaubt vor der Bedingung eine einfachen Anweisung, wie hier die Deklaration

einer lokalen Variablen; das ist besonders nützlich für die Fehlerbehandlung wie in diesem Beispiel. Wir hätten es natürlich auch so schreiben können:

```
err := r.ParseForm()
if err != nil {
    log.Print(err)
}
```

Aber die kombinierte Anweisung ist kürzer und schränkt außerdem die Reichweite von `err` ein, was gute Praxis ist.

In den bisherigen Programmen haben wir drei verschiedene Arten von Ausgabe-Datenströmen kennengelernt. Das Programm `fetch` hatte HTTP-Antwortdaten in die Datei `os.Stdout` kopiert, das Gleiche tat das `lissajous`-Programm. Das Programm `fetchall` hatte die Antwort verworfen (aber nicht die Länge der Antwort), indem es in die triviale Datensenke `ioutil.Discard` kopiert hatte. Und der Web-Server oben benutzte `fmt.Fprintf`, um in einen `http.ResponseWriter` zu schreiben, der wiederum für den Web-Browser steht.

Und obwohl sich diese drei Typen im Detail unterscheiden, genügen sie alle einem geläufigen *Interface*, dass es erlaubt, jeden dieser Typen zu benutzen, wann immer ein Ausgabestrom benötigt wird. Dieses Interface, namens `io.Writer`, wird im Abschnitt 7.1 erklärt.

Go's Interface-Mechanismus ist Thema in Kapitel 7. Aber um einmal zu zeigen, zu was er fähig ist, schauen wir uns an, wie einfach wir den Web-Server mit der `lissajous`-Funktion verbinden können, sodass die GIF-Animation nicht zur Standardausgabe sondern zum Klienten geschrieben wird. Fügen Sie folgende Zeilen Ihrem Web-Server hinzu:

```
handler := func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/", handler)
```

oder, was äquivalent ist:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
})
```

In der letzteren Form ist das zweite Argument von `HandleFunc` ein *Funktionsliteral*, das heißt, es ist eine namenlose Funktion, die dort definiert wird, wo sie auch benutzt wird. Wir werden das im Abschnitt 5.6 genauer erläutern.

Haben Sie diese Änderung erst einmal gemacht, können Sie in Ihren Browser die Adresse `http://localhost:8000` aufsuchen. Bei jedem neuen Laden der Seite sehen Sie dann eine neue Animation, ähnlich der in Abbildung 1.3.

Übung 1.12: Ändern Sie den Lissajous-Server so, dass er Parameterwerte aus der URL liest. Sie könnten erreichen, dass eine URL wie `http://localhost:8000/?cycles=20` die Anzahl der Zyklen auf `20` statt auf den Standard `5` setzt. Benutzen Sie die Funktion `strconv.Atoi` um den String-Parameter in eine Ganzzahl zu konvertieren. Die Dokumentation dazu erhalten Sie mit dem Kommando `go doc strconv.Atoi`.

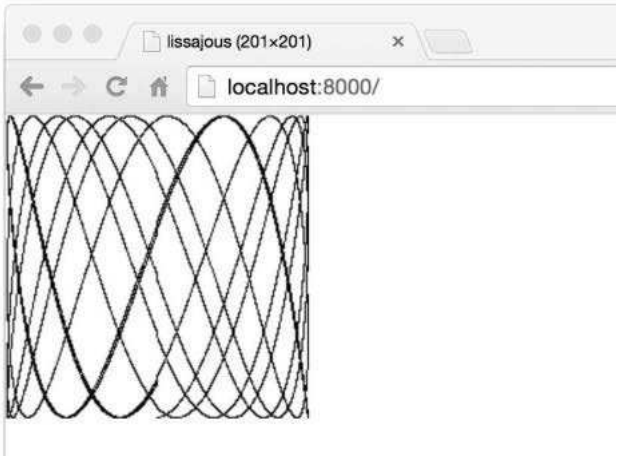


Abbildung 1.3 Animierte Lissajous-Figur in einem Browser.

1.8 Was noch fehlt

Go kann weit mehr, als diese kurze Einführung zeigen konnte. Hier noch ein paar bisher kaum berührte Themen mit nur so vielen begleitenden Worten, dass sie nicht mehr fremd wirken, wenn Sie Ihnen vor ihrer gründlichen Erörterung begegnen sollten.

Kontrollanweisungen: Zwei davon haben wir bereits behandelt, nämlich die Anweisungen `if` und `for`. Aber noch nicht die `switch`-Anweisung, welche eine Mehrfachverzweigung ist. Hier ein kurzes Beispiel:

```
switch coinflip() {
case "heads":
    heads++
case "tails":
    tails++
default:
    fmt.Println("landed on edge")
}
```

Das Ergebnis des Aufrufs von `coinflip` wird nach und nach mit den Werten der `case`-Klauseln verglichen. Diese werden von oben nach unten ausgewertet, sodass der Kode der ersten Übereinstimmung ausgeführt wird. Der optionale `default`-Fall wird ausgeführt, wenn es keine Übereinstimmung gibt; seine Position innerhalb des `switch` ist bedeutungslos. Es gibt kein automatisches Durchfallen von einem zum nächsten Fall wie in anderen C-ähnlichen Sprachen; allerdings gibt es die selten benutzte `fallthrough`-Anweisung, die dieses Standardverhalten ändert.

Die `switch`-Klausel benötigt keinen Operanden; es genügen die der `case`-Klauseln, die dann jeweils Boole'sche Ausdrücke sind:

```
func Signum(x int) int {
```

```
        switch {
        case x > 0:
            return +1
        default:
            return 0
        case x < 0:
            return -1
        }
    }
```

Man nennt diese Form *Nacktes Switch*; es ist gleichbedeutend mit `switch true`.

Wie `if` und `for`, darf auch ein `switch` eine einfache Anweisung enthalten – eine Variablenkurzdeklaration, ein Inkrement, eine Zuweisung oder einen Funktionsaufruf – die man dazu benutzen kann, einen Wert zu ermitteln, bevor man ihn testet.

Die Anweisungen `break` und `continue` ändern den Kontrollfluss. Ein `break` bewirkt, dass die Kontrolle auf die nächste Anweisung nach der innersten `for`-, `switch`- oder `select`-Anweisung übergeht (`select` werden wir später kennenlernen). Und in Abschnitt 1.3 haben wir schon gesehen, dass `continue` den nächsten Durchlauf der innersten `for`-Schleife bewirkt. Anweisungen dürfen eine Sprungmarke (label) besitzen, sodass `break` und `continue` darauf verweisen können, um zum Beispiel aus mehreren geschachtelten Schleifen gleichzeitig auszubrechen oder den nächsten Durchlauf einer äußeren Schleife zu erzwingen. Es gibt sogar eine `goto`-Anweisung, doch die ist eher für maschinengenerierten Kode und weniger für Programmierer gedacht.

Namensbehaftete Typen. Mit einer Typdeklaration kann ein Typ mit einem Namen versehen werden. Strukturtypen sind üblicherweise lang und haben deshalb fast immer auch einen Namen. Ein kurzes Beispiel wäre die Definition des Punkt-Typs für ein 2-D-Grafiksystem:

```
type Point struct {
    X, Y int
}
var p Point
```

Typdeklarationen und namensbehaftete Typen werden in Kapitel 2 behandelt.

Zeiger. Go bietet Zeiger, das heißt Werte, die Adressen von Variablen sind. In einigen Sprachen, insbesondere in C, gibt es kaum Einschränkungen für Zeiger. In anderen Sprachen kommen Zeiger im Gewand von „Referenzen“ daher, und man kann mit ihnen nicht viel mehr tun, als sie herumzureichen. Go positioniert sich irgendwo dazwischen. Zeiger sind explizit sichtbar, der Operator `&` liefert die Adresse einer Variablen und der Operator `*` holt sich die Variable, auf die ein Zeiger zeigt. Es gibt keine Zeigerarithmetik. Wir erklären Zeiger in Abschnitt 2.3.2.

Methoden und Interfaces: Eine Methode ist eine Funktion, die einem namensbehafteten Typ zugeordnet ist; Go ist insofern ungewöhnlich, als Methoden fast allen namensbehafteten Typen zugeordnet werden können. Methoden werden in Kapitel 6 behandelt. Interfaces sind abstrakte Typen, die uns verschiedene konkrete Typen gleich behandeln lassen, und zwar ausgehend von den Methoden, die zum Typ gehören, und nicht von

ihrer Implementierung. Um Interfaces geht es in Kapitel 7.

Pakete: Go bringt eine umfangreiche Standardbibliothek voller nützlicher Pakete mit, und die Go-Gemeinde hat noch vieles darüber hinaus geschaffen und freigegeben. Programmieren ist oft mehr das Benutzen bereits existierender Pakete als das Schreiben eigenen Kodes. Über das ganze Buch verteilt geben wir immer wieder Hinweise auf die wichtigsten Standardpakete. Das werden dann ein paar Dutzend sein, doch es gibt noch viel mehr, die wir aus Platzgründen gar nicht erwähnen können.

Wenn Sie sich an ein neues Programm heranmachen, sehen Sie schlauerweise erst einmal nach, ob es Pakete gibt, die Ihnen Arbeit abnehmen könnten. Das Verzeichnis der Standardbibliothek können Sie unter `https://golang.org/pkg` einsehen, und Pakete, die die Go-Gemeinde beigesteuert hat findet man unter `https://godoc.org`. Mit dem Werkzeug `go doc` erreicht man diese Dokumente leicht von der Kommandozeile aus:

```
$ go doc http.ListenAndServe
package http // import "net/http"

func ListenAndServe(addr string, handler Handler) error

    // ListenAndServe listens on the TCP network address addr
    // and then calls Serve with handler to handle requests
    // on incoming connections.
...
```

Kommentare: Wir haben bereits Dokumentationskommentare am Beginn eines Pakets oder Programms erwähnt. Guter Stil ist auch, vor die Deklaration der Funktionen Kommentare zu schreiben, die deren Verhalten zu beschreiben. Diese Kommentare sind deshalb wichtig, weil sie von Werkzeugen wie `go doc` oder `godoc` gebraucht werden, um Dokumentation finden und anzeigen zu können (Abschnitt 10.7.4).

Für Kommentare über mehrere Zeilen hinweg oder für Kommentare innerhalb eines Ausdrucks oder einer Anweisung gibt es die aus anderen Sprachen bekannte Schreibweise `/* ... */`. Solche Kommentare erscheinen manchmal am Anfang einer Datei als großer erklärender Textblock; man vermeidet damit die häufige Wiederholung von `//` am Zeilenanfang. Innerhalb eines Kommentars haben `//` und `/*` keine besondere Bedeutung, also können Kommentare auch nicht geschachtelt werden.