Prof. Dr. Jörg Desel, Dr. Marc Finthammer

Modul 63811

Einführung in die

Imperative Programmierung

Kapitel 1 – 7

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

3 Praktische Einführung in die Programmierung mit Go

Hinweis 3.1: Go Version 1.22

Dieser Lerntext basiert auf der Go Version 1.22.

3.1 Pakete und die Go Standardbibliothek

Paket (package)

In Go werden *Pakete (packages)* verwendet, um umfangreichere Mengen Code geeignet zu strukturieren und zu organisieren. Programmkomponenten (insbesondere Datenstrukturen und Funktionen), die thematisch zusammengehören oder einem gemeinsamen Zweck dienen (wie z.B. der Ausgabe von Text), können in einem Paket zusammengefasst werden, um sie für andere Programme verfügbar zu machen. Denn Programmcode, der in Form eines Pakets vorliegt, kann in ein anderes Programm importiert werden, um dort wiederverwendet zu werden. Pakete in Go entsprechen also den in anderen Programmiersprachen üblichen Bibliotheken (Libraries) oder Modulen (Modules).

Go Standardbibliothek

Zur Programmiersprache Go gehört die sogenannte *Go Standardbibliothek (Go standard library)*, die über 150 hierarchisch strukturierte Pakete[1] umfasst, welche eine Vielzahl von häufig benötigten Funktionalitäten zur Verfügung stellen, wie z.B.: das Einlesen, Ausgaben, Formatieren und Manipulieren von Text, Zugriff auf das Dateisystem und Netzwerk, Sortierfunktionen, mathematische Funktionen, Zeit- und Datumsfunktionen, Unterstützung für die Verarbeitung von Dateiformaten wie HTML, XML, CSV, JSON, JPEG, PNG, GIF, ZIP, TAR, Kryptographiefunktionen und vieles mehr.

[1] Pakete der Go Standardbibliothek: `https://pkg.go.dev/std`

3.2 Hello-World Programm

Das erste Programm, welches man häufig in einer neuen Programmiersprache kennenlernt, ist ein sehr einfaches Programm, das lediglich den Text „Hello, World!“ ausgibt. Dieser Tradition[2] folgend starten wir mit einem sogenannten *Hello-World Programm*:

Hello-World Programm

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello, World!")
7 }
```

Listing 3.1: Hello-World Programm in Go [hello_world.go]

Nach Eingabe des Befehls

```
go run hello_world.go
```

wird das Programm zunächst kompiliert und anschließend direkt ausgeführt. Das Programm gibt den folgenden Text aus und ist danach beendet:

```
Hello, World!
```

An diesem sehr kurzen, aber nichtsdestotrotz vollständigen Programm lässt sich bereits der prinzipielle Aufbau eines gültigen Go Programms bzw. einer Go Quellcodedatei illustrieren[3].

Paketklausel

package

In der ersten (nicht leeren) Zeile einer jeden Go Quellcodedatei muss die sogenannte *Paketklausel (package clause)*[4] stehen, die festlegt, welchem Paket (siehe Kapitel 3.1) der Code dieser Datei zugeordnet ist. In der ersten Zeile unseres Hello-World Programms steht die spezielle Paketklausel `package main`. Sie bewirkt, dass aus der Quellcodedatei eine ausführbare Programmdatei (anstatt einer Bibliotheksdatei) erzeugt wird. Im Rahmen dieses Lerntextes werden wir nur relativ kurze Go Programme betrachten, die aus jeweils nur einer Quellcodedatei bestehen. Daher werden wir uns mit den Details der Paketstruktur von Go nicht weiter befassen, sondern (der Einfachheit halber) all unsere Programme mit der Paketklausel `package main` beginnen. Dadurch wird der Code dem

[2] Die Tradition der Hello-World Programme geht zurück auf das 1978 von Brian W. Kernighan and Dennis M. Ritchie veröffentlichte Buch „The C Programming Language“[KR78].

[3] Im Folgenden werden wir die Begriffe *Programm* und *Quellcodedatei* gleichwertig verwenden, da wir zunächst ausschließlich Programme betrachten werden, die aus genau einer Quellcodedatei bestehen.

[4] Statt des in der Go Sprachspezifikation eingeführten Begriffs *package clause* wird in der einschlägigen Go Literatur auch häufig der (unpräzisere) Begriff *package declaration* verwendet.

sogenannten *main package* zugeordnet, aus welchem das Go Build-Tool eine ausführbare Datei generiert.

Die leere Zeile nach der Paketklausel dient lediglich der übersichtlicheren Gestaltung des Quellcodes. Zeilenumbrüche (*newlines*) bilden zusammen mit Leerzeichen (*blanks*) und Tabulatoren (*Tabs*) den sogenannten *Leerraum (whitespace)*, der nur der Separierung der Elemente des Quellcodes dient, ansonsten jedoch ignoriert wird[5] [6]. Demzufolge spielt es – zumindest aus syntaktischer Sicht – keine Rolle, ob man an einer Stelle des Quellcodes nur eines oder gleich mehrere Leerraum-Zeichen einfügt. Somit wären z.B. prinzipiell zwischen `package` und `main` anstatt nur eines Leerzeichens auch mehrere Leerzeichen und sogar Zeilenumbrüche zulässig. Aus Sicht des menschlichen Lesers sollte der Leerraum allerdings so verwendet werden, dass sich dadurch eine übersichtliche Formatierung und somit eine gute Lesbarkeit des Quellcodes ergibt.

Leerraum (whitespace)

Auf die obligatorische Paketklausel folgt in der nächsten (nicht leeren) Zeile des Hello-World Programms eine sogenannte *Importdeklaration (import declaration)*. Die Importdeklaration `import "fmt"` bewirkt, dass der Inhalt des Pakets `fmt` unserem Programm verfügbar gemacht wird. Das *Paket* `fmt` (englisch ausgesprochen *fumpt* und Abkürzung für „format“) ist Bestandteil der Go Standardbibliothek. Es stellt grundlegende Funktionen zum Einlesen, Formatieren und Ausgeben von Zeichenketten und Zahlen zur Verfügung und wird daher in der Praxis von fast jedem Go Programm importiert.

Importdeklaration

`import`

Paket `fmt`

Ein Go Programm kann mehrere Importdeklarationen enthalten oder (zumindest theoretisch) gar keine. Sämtliche Importdeklarationen müssen direkt der Paketklausel folgen und jedes importierte Paket muss auch tatsächlich genutzt werden, d.h. es muss mindestens einmal im Code auf eine Funktionalität des Pakets zurückgegriffen werden, da der Compiler ansonsten eine entsprechende Fehlermeldung liefert. Durch diese strikte Regelung wird von vornherein vermieden, dass sich während der Programmentwicklung eine Vielzahl unnötiger Pakete ansammelt.

Die letzten drei Zeilen unseres Hello-World Programm bilden zusammen eine *Funktionsdeklaration (function declaration)*. Mittels `func main()` und dem darauf folgenden, durch geschweifte Klammern `{` und `}` abgegrenzten *Funktionsrumpf (function body)* wird eine Funktion mit Namen *main* deklariert. Die Codezeilen innerhalb des Funktionsrumpfes, also zwischen den geschweiften Klammern, werden beim Aufruf der Funktion ausgeführt. Die Funktion `main` verfügt über keine Funktionsparameter, d.h. der dafür vorgesehene Bereich innerhalb der runden Klammern `(` und `)` ist leer. Funktionen sind ein essentieller Bestandteil eines jeden Go Programms. Daher werden wir uns in Kapitel 3.4 noch genauer mit der Deklaration sowie dem Aufruf und der Ausführung von

Funktionsdeklaration

Funktionsrumpf (function body)

[5] Leerraum wird allerdings *innerhalb* bestimmter Elemente z.T. nicht ignoriert (darauf werden wir an den entsprechenden Stellen noch gesondert hinweisen).

[6] Den Zeilenumbrüchen kommt jedoch noch eine gewisse Bedeutung zu, auf die wir in Kapitel 3.3 eingehen werden.

Funktionen befassen. Zunächst beschränken wir uns auf ein paar kurze Anmerkungen zur speziellen Funktion `main` (im Paket main).

Funktion `main`

Die Funktion `main` ist der Einstiegspunkt der Programmausführung, denn beim Start eines Go Programms wird `main` automatisch aufgerufen. Sobald alle Anweisungen im Funktionsrumpf von `main` ausgeführt wurden, d.h. wenn die Ausführung von `main` beendet ist, ist auch die Ausführung des Programms beendet. Da die Funktion `main` den Startpunkt eines Go Programms darstellt, muss in jedem Go Programm eine Funktion `main` deklariert sein.

Funktionsaufruf (function call)

Argument

String-Literal

Zeichenkette (String)

Im Rumpf der Funktion `main` unseres sehr kurzen Programms steht einzig die Anweisung `fmt.Println("Hello, World!")`. Um dem Leser die Struktur des Programms zu verdeutlichen, ist die Anweisung um vier Leerzeichen eingerückt[7]. Bei dieser Anweisung handelt es sich um einen *Aufruf (call)* der Funktion `Println`, die aus dem importierten Paket `fmt` stammt; der vollständige Funktionsname lautet dementsprechend `fmt.Println`. Der Funktion wird als *Argument*, d.h. innerhalb der runden Klammern `(` und `)`, das String-Literal `"Hello, World!"` übergeben. Ein *String-Literal* besteht in Go aus einer in doppelten Anführungszeichen `"` ... `"` eingeschlossenen Zeichenkette (also einer Folge von Zeichen). Diese Zeichenkette stellt den *Wert* des String-Literals dar. Jedes Leerzeichen innerhalb eines String-Literals ist Bestandteil der Zeichenkette, d.h. Leerzeichen werden hier nicht als Leerraum interpretiert. Anstatt des Begriffs *Zeichenkette* wird auch im Deutschen häufig der englische Begriff *String* verwendet.

Hinweis 3.2: Notation im Lerntext: Zeichenketten

Wenn wir innerhalb des Fließtextes über eine konkrete Zeichenkette sprechen, dann stellen wir diese – wenn der Kontext klar ist – als String-Literal dar (also in Anführungszeichen), um sie deutlich vom umgebenden Text abzugrenzen. Wenn wir also z.B. nachfolgend davon sprechen, dass die Zeichenkette `"Haus"` an eine Funktion übergeben wird, dann meinen wir damit den konkreten *Wert* des String-Literals `"Haus"`, nämlich die Zeichenkette `Haus` (also ohne die Anführungszeichen).

Funktion `fmt.Println`

Die Funktionalität der Funktion `fmt.Println` besteht darin, die ihr übergebenen Argumente auszugeben und abschließend einen Zeilenumbruch (newline) anzufügen[8]. Somit erscheint nach Aufruf der Funktion in der Ausgabe der Text

```
Hello, World!
```

gefolgt von einer neuen Zeile.

[7] Genau genommen ist die Anweisung gemäß der Standardformatierung von Go mit einem Tabulatorzeichen eingerückt worden, dessen Breite hier (wie in den meisten Editoren) vier Leerzeichen entspricht.

[8] Der Funktionsname ist eine Abkürzung für „print line“ und wird auch so ausgesprochen.

3.3 Aufbau und Struktur eines Go Programms

Am einführenden Beispiel des Hello-World Programms haben wir bereits den grundsätzlichen Aufbau eines Go Programms kennengelernt. Nun werden wir anhand weiterer kleiner Beispielprogramme einige essentielle Sprachbestandteile und syntaktische Konstrukte von Go zunächst kurz (und eher informell) einführen, um damit eine Grundlage für alle weiteren Betrachtungen zu schaffen. In den anschließenden Kapiteln werden wir uns dann detaillierter mit den hier vorgestellten Sprachbestandteilen und syntaktischen Konstrukten befassen.

Wir beginnen mit einem kurzen Beispiel, welches sich mit der Umrechnung von Temperaturen beschäftigt (inspiriert von einem Beispielprogramm in [DK15]):

```go
package main

import "fmt"

func main() {
    // Deklaration einer Variablen vom Typ float64
    // und implizite Initialisierung mit Null-Wert
    var tempC float64

    // Deklaration einer Konstanten vom Typ float64
    const nullpunkt float64 = -273.15

    tempC = 10
    fmt.Println("Temperatur zum Start     (°C):", tempC)
    tempC = tempC + 12
    fmt.Println("Temperatur nach Änderung (°C):", tempC)

    var tempF float64
    tempF = 32 + tempC*1.8 // Temperatur in Grad Fahrenheit

    // Deklaration einer Variablen und
    // direkte Initialisierung mit einem Wert
    var tempK float64 = tempC - nullpunkt // Temperatur in Kelvin

    /* Ausgabe der Temperatur in den Einheiten:
       - Celsius
       - Fahrenheit
       - Kelvin
    */
    var tempText string = "Temperatur in "
    fmt.Println()
    fmt.Println(tempText+"°C:", tempC)
    fmt.Println(tempText+"°F:", tempF)
    fmt.Println(tempText+" K:", tempK)
}
```

Listing 3.2: Go Programm zur Ausgabe und Umrechnung von Temperaturen [temp_umrechnung.go]

Ein Aufruf des Programms aus Listing 3.2 liefert folgende Ausgabe:

```
Temperatur zum Start     (°C): 10
Temperatur nach Änderung (°C): 22

Temperatur in °C: 22
Temperatur in °F: 71.6
Temperatur in  K: 295.15
```

Das Programm macht Folgendes: Es gibt eine Start-Temperatur (in Grad Celsius) aus, erhöht anschließend diese Temperatur und gibt dann die geänderte Temperatur aus. Abschließend wird diese Temperatur noch in Grad Fahrenheit und Kelvin umgerechnet und ausgegeben.

Der prinzipielle Aufbau dieses Programms entspricht dem des Hello-World Programms aus Listing 3.1, d.h. auf die obligatorische Paketklausel folgt die Importdeklaration für das Paket `fmt` und schließlich die Deklaration der Funktion `main`.

Hinweis 3.3: Notation im Lerntext: Zeilenumbrüche im Quellcode

Wenn eine Quellcodezeilen zu lang ist, um auf einer Seite des Lerntextes dargestellt zu werden, dann wird die Quellcodezeile automatisch umbrochen. Der Umbruch von Quellcode wird im Lerntext durch die speziellen Zeichen ↩ und ↪ deutlich gemacht, wie z.B. in den folgenden zwei Codezeile:

```
// Diese Kommentarzeile ist so lang, dass sie im Lerntext ↩
   ↪ automatisch umgebrochen wird
fmt.Println("Auch diese Zeile zur Ausgabe von", tempC, "und", ↩
   ↪ tempF, "muss im Lerntext umgebrochen werden.")
```

Das heißt, die speziellen Zeichen ↩ und ↪ sind *nicht* Bestandteil des Quellcodes oder der Go Syntax, sondern werden lediglich innerhalb des Lerntextes bei der Formatierung von Quellcode verwendet.

Wenn wir innerhalb des Textes ein Codefragment wie z.B. `const nullpunkt ↩ ↪ float64 = -273.15` erwähnen, dann kann es auch dabei zu einem entsprechend formatierten Umbruch innerhalb des Codes kommen.

3.3.1 Kommentare

In den Zeilen 6 bzw. 7 bilden jeweils die zwei Schrägstriche `//` zusammen mit dem restlichen Text der Zeile einen *Zeilenkommentar (line comment)*. Kommentare richten

Zeilenkommentar

sich an den menschlichen Leser und dienen der Dokumentation des Quellcodes, um diesen besser lesbar und leichter nachvollziehbar zu machen. Kommentare sind zwar ein Bestandteil der Syntax von Go, haben jedoch keine semantische Bedeutung und werden dementsprechend vom Compiler ignoriert. Ein Zeilenkommentar kann auch erst am Ende einer Zeile stehen, wie z.B. in Zeile 19.

Blockkommentar

Eine zweite Syntax für Kommentare stellt der *Blockkommentar (block comment)* dar, der durch die Zeichen `/*` eingeleitet und durch `*/` beendet wird (siehe Zeilen 25–29). Sämtlicher Text zwischen diesen beiden Zeichenfolgen – auch über Zeilenumbrüche hinweg – wird als Kommentar interpretiert und ebenfalls vom Compiler ignoriert. Die Blockkommentar-Syntax findet dann Verwendung, wenn ein Kommentar (deutlich) mehr als zwei Zeilen umfasst oder wenn ein ganzer Codeabschnitt zu Testzwecken auskommentiert werden soll.

3.3.2 Anweisungen und Abschlusszeichen

Anweisung (statement)

Jede Programmzeile im Rumpf der Funktion `main` (ausgenommen natürlich die Kommentarzeilen) stellt eine *Anweisung (statement)* dar. Anweisungen sind das zentrale syntaktische Konstrukt einer imperativen Programmiersprache wie Go. Jede Anweisung gibt eine Aktion vor, die ausgeführt wird, wenn die Anweisung im Laufe der Programmabarbeitung erreicht wird. Zu den Aktionen, die durch entsprechende Anweisungen vorgegeben werden, zählen z.B. das Aufrufen einer Funktion, die Deklaration einer Variablen, die Zuweisung eines Werts, sowie die bedingte oder wiederholte Ausführung eines Programmabschnitts. Anweisungen werden (im Allgemeinen) sequenziell, also in der Reihenfolge ihres Auftretens, ausgeführt[9]. Da die Funktion `main` den Einstiegspunkt der Programmausführung darstellt, ergibt sich der Ablauf eines Go Programms also aus der Folge von Anweisungen im Rumpf von `main`, wobei diese Anweisungen auch Aufrufe weiterer Funktionen bewirken können.

Abschlusszeichen

Semikolon `;`

In vielen Programmiersprachen, so (im Prinzip) auch in Go, muss das Ende einer Anweisung durch ein vorgegebenes *Abschlusszeichen (terminator)* kenntlich gemacht werden. In der formalen Grammatikdefinition von Go wird – wie z.B. auch in den Sprachen Java und C – das *Semikolon* `;` als Abschlusszeichen für Anweisungen und Deklarationen verwendet. Allerdings fällt bei Betrachtung der Listings 3.1 und 3.2 sofort auf, dass der dort dargestellte Quellcode kein einziges Semikolon enthält. Dies liegt an folgender Besonderheit von Go, die es dem Programmierer ermöglicht, einen Großteil der (syntaktisch eigentlich notwendigen) Semikolons einfach weglassen zu können: Der Go-Compiler fügt

[9] Die Programmiersprache Go ermöglicht u.a. mittels sogenannter *Goroutines* und *Channels* auch nebenläufige Programmierung, d.h. Anweisungen können nicht nur sequentiell, sondern auch unabhängig voneinander ausgeführt werden. Im Rahmen dieses Kurses werden wir jedoch nicht weiter auf diese (sehr mächtigen) Möglichkeiten von Go zur Realisierung von Nebenläufigkeit eingehen, sondern uns auf die rein sequentielle Ausführung von Programmen beschränken.

bei der Analyse des Quellcodes automatisch ein Semikolon am Ende einer Zeile ein, wenn dort das Ende einer Anweisung oder Deklaration stehen könnte (siehe Expertenwissen 3.5 für die exakte Bedingung). Aufgrund dieser Regelung muss in Go nur in wenigen, speziellen Situationen (auf die wir noch gesondert hinweisen werden) ein Semikolon explizit im Quellcode gesetzt werden. Es ist guter Programmier-Stil in Go (und im Sinne einer idiomatischen Verwendung von Go (siehe Kapitel 2.8))Semikolons nur genau dann zu setzen, wenn sie zwingend nötig sind.

implizite Semikolons

Expertenwissen 3.4: Abschlusszeichen vs. Separatorzeichen

Leicht verwechseln kann man das Abschlusszeichen mit einem *Separatorzeichen*. So wird in Pascal das Semikolon verwendet, um auszudrücken, dass eine Anweisung nach einer anderen Anweisung auszuführen ist, also quasi ein Konnektor. Der Unterschied ist im Wesentlichen, dass im Fall eines Separatorzeichens am Ende eines Blocks kein Semikolon steht.

Allerdings hat das automatische Einfügen von Semikolons am Zeilenende zur Folge, dass Zeilenumbrüche (obwohl sie prinzipiell Leerraum sind) nicht völlig beliebig gesetzt werden dürfen. Insbesondere darf kein Zeilenumbruch vor einer öffnenden Klammern `(` oder `{` stehen, d.h. öffnende Klammern dürfen nicht in einer neuen Zeile stehen. So muss z.B. in Zeile 5 die öffnende runde Klammer `(` in derselben Zeile wie der vorausgehende Funktionsbezeichner `main` stehen; ebenso muss die öffnende geschweifte Klammer `{`, die den Funktionsrumpf einleitet, in der selben Zeile wie die vorausgehende schließende Klammer `)` stehen. Würde man hingegen die öffnende Klammer `{` in die nächste Zeile schreiben, dann käme es beim Kompilieren des Programms zu einer entsprechenden Fehlermeldung:

Relevanz von Zeilenumbrüchen

```
syntax error: unexpected semicolon or newline before {
```

Selbst wenn es hier zunächst etwas kompliziert klingen mag, so ist die Positionierung von Zeilenumbrüchen bzw. öffnenden Klammern (ebenso wie das Weglassen von Semikolons) in der Praxis leicht und intuitiv anwendbar, ohne sich jedes Mal die genaue Regelung ins Gedächtnis rufen zu müssen. Neben den hier erwähnten gibt es noch ein paar (wenige) weitere Situationen, in denen die Positionierung von Zeilenumbrüchen beachtet werden muss. Auf diese Situationen werden wir an den entsprechenden Stellen im Lerntext hinweisen.[10]

[10] Die syntaktischen Anforderungen von Go hinsichtlich der Positionierung von öffnenden Klammern mag ein bereits erfahrener Programmierer zunächst als unnötige Einschränkung seiner gewohnten „Lieblings"-Quellcode-Formatierung empfinden. Diese Einschränkung wird in Go jedoch bewusst hingenommen, da sie insgesamt zu einer einheitlicheren Formatierung von Code führt, woraus sich wiederum eine leichtere Lesbarkeit fremden Codes ergibt. Um die Formatierung von Quellcode darüber

Expertenwissen 3.5: Automatisches Einfügen von Semikolons

Bei der Analyse des Quellcodes fügt der Go Compiler automatisch ein Semikolon ein, wenn es sich beim letzten Element einer Zeile (also vor einem Zeilenumbruch) um eines der folgenden handelt:

- einen Bezeichner
- ein Literal (Integer-Literal, String-Literal, etc.)
- eines der Schlüsselwörter `break`, `continue`, `fallthrough`, `return`
- eines der Zeichen `++`, `--`, `)`, `]`, `}`

Zudem dürfen (syntaktisch eigentlich notwendige) Semikolons vor runden und geschweiften schließenden Klammern `)` und `}` weggelassen werden.

Die letzte Regel ermöglicht es, komplexe Ausdrücke ggf. in eine Zeile schreiben zu können, ohne Semikolons einfügen zu müssen, sodass z.B. auch folgende Notation möglich ist[a]:

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

[a] Eine solche Notation ist allerdings bezogen auf die spezielle Funktion `main` nicht üblich, sondern bietet sich im Allgemeinen bei der Deklaration sehr kurzer Funktionen an.

3.3.3 Variablen und Datentypen

Im Folgenden werden wir die Programmzeilen (in Listing 3.2) im Rumpf der Funktion `main` der Reihe nach erläutern.

Variablendeklaration
Variable

In Zeile 8 steht eine *Variablendeklaration (variable declaration)*, welche eine *Variable* vom Typ `float64` erzeugt, ihr den Namen `tempC` gibt und sie implizit mit dem Wert 0 belegt. Eine Variable dient dem Speichern von veränderlichen Werten während des Programmablaufs. Über ihren eindeutigen Namen kann auf eine Variable zugegriffen werden, um ihren *Wert* abzufragen oder ihr einen neuen Wert zuzuweisen (und somit ihren Wert zu verändern).

Wert

Datentyp

Da es sich bei Go um eine typisierte Programmiersprache handelt, muss für jede Variable auch ihr *Datentyp* (kurz *Typ*) festgelegt werden. Der Typ einer Variablen gibt den Wertebereich vor, aus dem die Variable Werte annehmen kann. So umfasst z.B. der Wertebereich des Typs `float64` die 64-Bit-Gleitkommazahlen (eine Approximation der reellen Zahlen mit 64 Bit), der Typ `int` einen großen Bereich der ganzen Zahlen, der

hinaus zu vereinheitlichen, liefert jede Go Installation das Tool `gofmt` zur automatischen Formatierung von Quellcodedateien mit.

Typ `string` (siehe Zeile 30) alle Zeichenketten und der Typ `bool` die Wahrheitswerte `true` und `false`. Darüber hinaus ergibt sich aus dem Typ einer Variablen auch, welche Operationen mit ihrem Wert durchgeführt werden können (z.B. das Addieren von Zahlen, das Aneinanderhängen von Zeichenketten oder die logische Verknüpfung von Wahrheitswerten). Der für eine Variable bei ihrer Deklaration festgelegte Typ ist unveränderlich, da in Go *statische Typisierung (static typing)* verwendet wird. Somit ist z.B. garantiert, dass eine als Typ `int` deklarierte Variable an jeder Stelle im Code immer eine ganze Zahl repräsentiert. Wir werden auf das Typsystem von Go und die bereits in Go vorgegebenen primitiven Datentypen (auf Basis derer auch eigene Typen definiert werden können) detailliert in Kapitel 4 eingehen.

statische Typisierung

Die Variable namens `tempC` vom Typ `float64` speichert also eine Gleitkommazahl; oder anders formuliert: die Variable `tempC` steht für einen Wert vom Typ `float64`. Da bei der Deklaration der Variablen `tempC` kein expliziter Wert mit angegeben ist, wird die Variable implizit mit dem Null-Wert ihres Typs initialisiert, im Falle numerischer Typen wie `float64` also mit dem Wert 0. Da eine Variable also zu jedem Zeitpunkt des Programmablaufs mit einem Wert belegt ist, spricht man in diesem Zusammenhang auch häufig von der (zu einem bestimmten Zeitpunkt geltenden) *Belegung einer Variablen*.

implizite Initialisierung mit Null-Wert

Belegung einer Variablen

In Zeile 11 steht eine *Konstantendeklaration (constant declaration)*, welche eine *Konstante* mit Namen `nullpunkt` vom Typ `float64` und mit dem Wert $-273{,}15$ erzeugt. Ähnlich wie bei einer Variablendeklaration wird auch bei der Deklaration einer Konstanten ein eindeutiger Name mit einem Wert verbunden. Der Wert einer Konstanten ist aber unveränderlich, d.h. eine Konstante kann nicht überschrieben werden. Bereits bei der Deklaration einer Konstanten muss ihr fester Wert mit angegeben werden. Die Konstantendeklaration ermöglicht es uns also, einen festen Wert mittels eines (aussagekräftigen) Namens zu repräsentieren, und ist insb. für Werte sinnvoll, die eine bestimmte Bedeutung haben (wie hier der Temperaturwert des absoluten Nullpunkts auf der Celsius-Skala) und ggf. mehrfach im Programm-Code verwendet werden. Die Zahl `-273.15` wird hier in Form eines *Floating-Point-Literals* dargestellt. Floating-Point-Literale dienen im Quellcode der Repräsentation von reellen Zahlen und verwenden den Punkt `.` als Dezimaltrennzeichen[11].

Konstanten-deklaration

Konstante

Floating-Point-Literal

3.3.4 Bezeichner und Schlüsselwörter

Variablen, Konstanten, Typen und Funktionen stellen wesentliche Bestandteile eines Go Programms dar und werden zusammenfassend als *Programmelemente (program entities)* bezeichnet. Der für ein Programmelement festgelegte, eindeutige Name wird *Bezeich-*

Programm-elemente

[11] Innerhalb unseres Fließtextes verwenden wir für Zahlen – wie im Deutschen üblich – das Komma als Dezimaltrennzeichen. Nur bei Zahlen, die Teil des Quellcodes sind oder im Zusammenhang mit Eingaben bzw. Ausgaben auftreten, verwenden wie den Punkt als Dezimaltrennzeichen, wobei wir dann solche Zahlen in der `Schreibmaschinen-Schriftart` setzen.

Bezeichner (identifier)

Deklaration

ner (identifier) genannt. Bei der *Deklaration (declaration)* eines Programmelements wird dieses immer mit seinem Bezeichner verbunden, um es fortan über diesen Bezeichner ansprechen zu können. Beispielsweise wird bei der Variablendeklaration in Zeile 8 eine neue Variable erzeugt und mit dem Bezeichner `tempC` verbunden, sodass im weiteren Programmablauf auf diese Variable mittels ihres Bezeichners `tempC` zugegriffen werden kann.

zulässige Bezeichner

Als Bezeichner sind in Go alle Zeichenfolgen zulässig (ausgenommen Schlüsselwörter, siehe unten), die folgenden Regeln entsprechen:

- Das erste Zeichen muss ein Buchstabe[12] oder der Unterstrich _ sein.
- Die restlichen Zeichen dürfen eine beliebige Kombination aus Buchstaben, dem Unterstrich _ und Ziffern sein.

Zulässige Bezeichner sind also z.B.:

```
a    B    _    größe    DoIt    xbox360    _intern    mein_Name
```

Hingegen stellt die Zeichenfolge `24stunden` keinen zulässigen Bezeichner dar, weil sie mit einer Ziffer beginnt.

leerer Bezeichner _

Der Unterstrich _ selbst stellt prinzipiell auch einen zulässigen Bezeichner dar. Allerdings kommt dem Bezeichner _, der auch *leerer Bezeichner (blank identifier)* genannt wird, eine besondere Bedeutung zu, da er in bestimmten Situationen quasi als anonymer Platzhalter fungiert. Wir werden später an entsprechenden Stellen darauf eingehen, was genau der leere Bezeichner _ im Sinne eines anonymen Platzhalters bewirkt.

Generell ist zu beachten, dass zwischen *Groß- und Kleinbuchstaben* unterschieden wird, sodass z.B. `tempC` und `tempc` zwei unterschiedliche Bezeichner darstellen würden.

Export eines Bezeichners

An dieser Stelle sei bereits darauf hingewiesen, dass nur auf Bezeichner, die mit einem Großbuchstaben beginnen, auch von außerhalb ihres Pakets zugegriffen werden kann. Man spricht in diesem Falle davon, dass der Bezeichner *exportiert* wird. Dies ist auch der Grund, warum alle in externen Paketen zur Verfügung stehenden Funktion und Typen, wie z.B. die bereits bekannte Funktion `Println` im Paket `fmt`, mit einem Großbuchstaben beginnen. Auch wenn für uns im Rahmen dieses Lerntextes die Aufteilung eines Programms in Pakete keine Rolle spielt, werden wir uns dennoch an folgende Konvention halten:

> Wir beginnen einen Bezeichner genau dann mit einem Großbuchstaben,
> wenn wir ihn bewusst exportieren wollen.

Da wir in unseren kurzen Beispielprogrammen keine Bezeichner exportieren wollen, be-

[12] Da für Go Quellcodedateien das Unicode Zeichenformat benutzt wird, bezieht sich der Begriff „Buchstaben“ hier auf die Menge von Unicode-Buchstaben, die deutlich über die üblichen 26 Groß- und Klein-Buchstaben des ASCII-Zeichensatzes hinausgeht. Somit sind u.a. auch die deutschen Umlaute `Ä`, `ä`, `Ö`, `ö`, und `Ü`, `ü` sowie `ß` prinzipiell zulässige Buchstaben.

ginnen wir dort alle Bezeichner mit Kleinbuchstaben.

Die Variablendeklaration in Zeile 8 wird durch das *Schlüsselwort (keyword)* `var` eingeleitet. Wir haben zuvor bereits – ohne es explizit erwähnt zu haben – drei weitere Schlüsselwörter von Go kennengelernt, nämlich `package`, `import` und `func`.

Schlüsselwort (keyword)

Die Schlüsselwörter einer Programmiersprache sind eine Menge von Wörtern (also Buchstabenfolgen), die von der Syntax der Sprache fest vorgegeben sind und denen in der Semantik der Sprache eine bestimmte Bedeutung zukommt. Schlüsselwörter dürfen nicht als Bezeichner verwendet werden und werden daher auch als *reservierte Wörter* bezeichnet. Aus diesem Grund, und um eine Sprache leichter erlernbar zu machen, wird die Anzahl der Schlüsselwörter in den meisten Programmiersprachen möglichst klein gehalten. In der Go Sprachspezifikation sind die folgenden 25 Schlüsselwörter definiert:

```
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
```

Darüber hinaus gibt es auch noch sogenannte *vorgegebene Bezeichner (predeclared identifiers)*, welche zu vorgegebenen Programmelementen gehören, die einen elementaren Bestandteil der Programmiersprache Go darstellen und daher fest in diese integriert sind. Diese Programmelemente umfassen einige vorgegebene Datentypen (predeclared types), eingebaute Funktionen (built-in functions) und Konstanten. Auf syntaktischer Ebene zählen die vorgegebenen Bezeichner nicht zu den reservierten Wörtern (sondern sind ja gerade *Bezeichner*) und könnten daher theoretisch in Deklarationen als „eigener“ Bezeichner verwendet (sprich redeklariert) werden. In der Praxis sollte diesen Bezeichnern jedoch tunlichst keine neue Bedeutung zugewiesen werden, da dies unvorhersehbare und nur schwer zu entdeckende Probleme hervorrufen könnte, sowie eine Missinterpretation des Codes durch den Leser provozieren würde. Die folgenden vorgegebenen Bezeichner sind in Go für entsprechende Programmelemente definiert[13]:

vorgegebene Bezeichner (predeclared identifiers)

[13] Zwar gibt es auch die eingebauten Funktionen `println` und `print` zur (rudimentären) Ausgabe von Werten, allerdings haben diese Funktionen in der Praxis keine Relevanz, und es sollten stattdessen immer die entsprechenden Funktionen aus dem Paket `fmt` verwendet werden, wie z.B. `fmt.Println`.

vorgegebene Typen:	`any bool byte comparable complex64 complex128 error float32 float64 int int8 int16 int32 int64 rune string uint uint8 uint16 uint32 uint64 uintptr`
eingebaute Funktionen:	`append cap clear close complex copy delete imag len make max min new panic print println real recover`
Konstanten:	`true false iota nil`

3.3.5 Zuweisungen und Ausdrücke

Zuweisung (assignment statement)

In Zeile 13 steht eine *Zuweisung (assignment statement)*, die der Variablen `tempC` den Wert 10 zuweist, d.h. die Zuweisung sorgt dafür, dass in der Variablen `tempC` nun der Wert 10 gespeichert ist. Der vorherige Wert der Variablen (hier war es der Initialwert 0) wird bei einer Zuweisung durch einen neuen Wert (hier 10) ersetzt, d.h. der vorherige Wert ist somit „verloren“. In den drei folgenden Programmzeilen werden die Auswirkungen einer Zuweisung noch genauer illustriert. Die Zahl `10` wird im Quellcode in Form eines *Integer-Literals* (in Dezimalschreibweise[14]) dargestellt. Integer-Literale dienen im Quellcode der Repräsentation natürlicher Zahlen und ermöglichen durch einen ggf. vorangestellten Minus-Operator `-` auch die Darstellung von Ganzzahlen. Da die natürlichen Zahlen eine Untermenge der Gleitkommazahlen darstellen, kann einer Variablen vom Typ `float64` auch der Wert eines Integer-Literals zugewiesen werden.

Integer-Literal

In Zeile 14 wird zunächst die bereits bekannte Funktion `fmt.Println` aufgerufen, wobei ihr diesmal zwei Argumente – getrennt durch ein Komma – übergeben werden: Das erste Argument ist wieder eine Zeichenkette, das zweite Argument ist hier die Variable `tempC`. Genauer gesagt wird der Funktion nicht die Variable selbst übergeben, sondern der *Wert* der Variablen[15]. Wenn der Funktion `fmt.Println` mehrere Argumente übergeben werden, dann gibt sie diese alle nacheinander, getrennt durch ein zusätzliches Leerzeichen, in einer Zeile aus. Dementsprechend wird folgende Zeile ausgeben:

```
Temperatur zum Start     (°C): 10
```

Diese Ausgabe illustriert, dass der aktuelle Wert der Variablen `tempC` (zum Zeitpunkt des Funktionsaufrufs) 10 beträgt.

[14] Alternativ könnte der Dezimalwert 10 auch durch ein Integer-Literal `0b1010` in binärer oder `0xA` in hexadezimaler Schreibweise repräsentiert werden (siehe Kapitel 4.3.1).

[15] Die präzise Unterscheidung, wie Argumente beim Aufruf einer Funktion übergeben werden, mag hier noch relativ unbedeutend erscheinen. Wenn wir uns jedoch später in Kapitel 7.1 intensiver mit Funktionen beschäftigen, werden wir sehen, welche weitreichenden Auswirkungen diese Art der Argumentübergabe hat.

Die Funktion `fmt.Println` gehört zu einer speziellen Klasse von Funktionen, der *beliebig viele* Argumente übergeben werden können, die zudem Werte *beliebigen Typs* sind. Im allgemeinen ist hingegen für eine Funktion die Anzahl ihrer Parameter und deren jeweiliger Typ genau festgelegt, wie wir in Kapitel 3.4 erfahren werden.

In Zeile 15 findet nun erneut eine Zuweisung an die Variable `tempC` statt: Ihr wird hier der Wert des *Ausdrucks* `tempC + 12` zugewiesen. Ein Ausdruck besteht aus einem oder mehreren Operanden (wie z.B. Variablen, Zahlen, Rückgabewerten von Funktionen oder anderen Ausdrücken), die durch *Operatoren* (wie z.B. `+` für Addition oder `*` für Multiplikation) verbunden werden. Um den aktuellen Wert eines Ausdrucks zu ermitteln, muss eine sogenannte *Auswertung (evaluation)* des Ausdrucks erfolgen. Eine solche Auswertung umfasst die Ausführung aller nötigen Berechnungsschritte in einer vorgegebenen Reihenfolge, wie z.B. den Aufruf von Funktionen und die Anwendung von Operatoren auf die ermittelten Werte ihrer Operanden. Der Ausdruck `tempC + 12` bedeutet, dass der Wert, der aktuell unter der Variablen `tempC` gespeichert ist, und die Zahl `12` addiert werden sollen. Dazu wird zunächst die Variable `tempC` (die selbst bereits einen Ausdruck darstellt) ausgewertet, d.h. es wird ihr Wert betrachtet. Da der Wert der Variablen `tempC` aktuell 10 beträgt, muss zur Auswertung des Ausdrucks die Berechnung $10+12$ durchgeführt werden, wodurch sich 22 als Wert des Ausdrucks ergibt. Einen weiteren Ausdruck haben wir übrigens bereits zuvor genutzt, ohne dies explizit erwähnt zu haben: Die Zahl `10` auf der rechten Seite der Zuweisung in Zeile 13 stellt bereits einen Ausdruck dar.

Ausdruck (expression)

Operator

Auswertung (evaluation)

In Zeile 16 wird erneut die Variable `tempC` ausgegeben. Da der aktuelle Wert der Variablen nun also 22 beträgt, erhalten wir die Ausgabe:

```
Temperatur nach Änderung (°C): 22
```

Wir haben nun bereits zwei Beispiele für die einfache Form der Zuweisungen in Go kennengelernt. Zuweisungen in dieser oder ähnlicher Form kommen in allen imperativen Programmiersprachen vor und stellen somit ein typisches Kennzeichen der imperativen Programmierung dar. In Kapitel 5.3 werden wir noch weitere Formen von Zuweisungen vorstellen, die sich aber im Prinzip alle auf diese einfache Form zurückführen lassen. Wie bereits diese Beispiele erkennen lassen, wird eine (einfache) Zuweisung in Go durch den *Zuweisungsoperator* `=` (das Gleichheitszeichen) ausgedrückt. Auf der linken Seite des Zuweisungsoperators muss (vereinfacht ausgedrückt[16]) eine Variable stehen, während sich auf der rechte Seite ein Ausdruck befinden muss. Zudem muss der Typ des ausgewerteten Ausdrucks zum Typ der Variablen passen.

Zuweisungsoperator `=`

[16] Genauer gesagt sind als linker Operand neben Variablen auch noch andere Konstrukte möglich, die wir aber erst im weiteren Verlauf des Kurses kennen lernen werden. Es lässt sich jedoch prinzipiell sagen, dass der linke Operand die Eigenschaft haben muss, etwas zugewiesen bekommen zu können. Dies trifft offensichtlich auf Variablen zu, aber z.B. nicht auf Zahlen oder Zeichenketten.

Hinweis 3.6: Zuweisungsoperator `=`

Auch wenn der Zuweisungsoperator in Go durch das Gleichheitszeichen `=` dargestellt wird, ist dessen Bedeutung keinesfalls mit der des Gleichheitszeichens in der Mathematik zu verwechseln. Die Programmzeile `tempC = 10` ist also zu lesen als „*der Variablen* `tempC` *wird der Wert 10 zugewiesen*“. Der Zuweisungsoperator ist folglich gerichtet, da er den Wert des Ausdrucks auf seiner *rechten* Seite nimmt und ihn der Variablen auf seiner *linken* Seite zuweist. Um die Richtung der Zuweisung zu betonen, wird der Zuweisungsoperator z.B. in der Programmiersprache Pascal dargestellt durch `:=` (einen Doppelpunkt unmittelbar gefolgt von einem Gleichheitszeichen[a]) und in Pseudocode oftmals durch `←` (einen Pfeil nach links). Um die Gleichheit im mathematischen Sinne auszudrücken, kommt in Go hingegen der Gleichheitsoperator `==` zum Einsatz, wie wir in Kapitel 4.5.2 sehen werden.

[a] Der Operator `:=` existiert ebenfalls in Go, hat hier jedoch eine andere Bedeutung, wie wir in Kapitel 5.1.2 im Zusammenhang mit der kurzen Variablendeklarationen sehen werden.

3.3.6 Auswertungsreihenfolge und Bindungsstärken

Auswertungsreihenfolge

Bindungsstärke

In Zeile 18 wird eine weitere Variable vom Typ `float64` mit Namen `tempF` deklariert. Dieser Variablen wird in Zeile 19 der Wert des Ausdrucks `32 + tempC*1.8` zugewiesen. Der obige Ausdruck entspricht der bekannten Berechnungsformel, um eine Temperatur von Grad Celsius nach Grad Fahrenheit umzurechnen, indem man die gegebene Temperatur mit 1,8 multipliziert und dann 32 hinzuaddiert. In diesem Ausdruck kommen sowohl der Additionsoperator `+` als auch der Multiplikationsoperator `*` vor, sodass sich die Frage nach der *Auswertungsreihenfolge* dieser Operatoren stellt. Ausdrücke werden in Go für Operatoren gleicher *Bindungsstärke (binding precedence)* (z.B. bei Additions- und Subtraktionsoperatoren) von links nach rechts ausgewertet. Da der Multiplikationsoperator `*` eine höhere Bindungsstärke als der Additionsoperator `+` hat, wird zunächst der Teilausdruck `tempC*1.8` ausgewertet (auch in Go gilt also Punkt- vor Strichrechnung): Der aktuellen Wert 22 der Variablen `tempC` wird zunächst mit 1,8 multipliziert, wodurch sich 39,6 als Wert des Teilausdrucks ergibt, auf den anschließend 32 addiert wird. Die Auswertung des gesamten Ausdrucks liefert somit den Wert 71,6, der abschließend der Variablen `tempF` zugewiesen wird. In Kapitel 4.1 werden wir uns genauer mit dem Aufbau von Ausdrücken und den dazu in Go zur Verfügung stehenden Operatoren sowie deren Bindungsprioritäten befassen.

In Zeile 23 findet wieder eine Variablendeklaration statt, wobei hier der Variablen `tempK` direkt ein (expliziter) Wert zugewiesen wird (anstatt sie nur implizit mit dem Null-Wert zu initialisieren). Das heißt, hier wird eine Variablendeklaration direkt mit

einer Zuweisung kombiniert, um den Initialwert der Variablen explizit festzulegen. Der Ausdruck `tempC - nullpunkt` entspricht hier der Umrechnung einer Temperatur von Grad Celsius nach Kelvin.[17] Da der aktuelle Wert der Variablen `tempC` weiterhin 22 ist, hat der Ausdruck folglich den Wert 295,15 und die Variable `tempK` wird mit diesem Wert initialisiert.

In Zeile 30 wird der Variablen `tempText` vom Typ `string` bei ihrer Deklaration ebenfalls direkt ein Wert zugewiesen, nämlich die Zeichenkette `"Temperatur in "`. Durch den Datentyp `string` der Variablen `tempText` wird hier also festgelegt, dass unter dieser Variablen nur Werte gespeichert werden können, die einer Zeichenkette entsprechen, d.h. die vom Typ `string` sind.

3.3.7 Ausgabe und Verkettung von Strings

In Zeile 31 wird die Funktion `fmt.Println()` ohne Argumente aufgerufen, wodurch lediglich ein Zeilenumbruch in der Ausgabe durchgeführt wird. Da die aktuelle Ausgabezeile keinen Text enthält, ergibt sich also eine Leerzeile in der Ausgabe.

In Zeile 32 wird die Funktion `fmt.Println` mit zwei Argumenten aufgerufen, wobei das erste Argument diesmal kein String-Literal, sondern der Ausdruck `tempText+"°C:"` ist. In diesem Ausdruck wird der Additionsoperator `+` zusammen mit zwei Operanden vom Typ `string` verwendet. Bei der Auswertung des Ausdrucks passiert folgendes : Es wird ein neuer String erzeugt, welcher der *Verkettung* der Werte des linken und des rechten Operanden entspricht, d.h. der linke und der rechte String werden einfach aneinandergehängt. Da unter der Variablen `tempText` der String `"Temperatur in "` gespeichert ist, ist der Wert des Ausdrucks `tempText+"°C:"` also der String `"Temperatur in °C:"`. Als zweites Argument wird der Funktion `fmt.Println` wiederum der Wert der Variablen `tempC` übergeben, sodass schließlich der Text `Temperatur in °C: 22` ausgegeben wird. Auf gleiche Weise wird in den beiden darauf folgenden Zeilen 33 und 35 jeweils der aktuelle Wert der Variablen `tempF` bzw. `tempK` ausgegeben. Dabei wird jedesmal der unter der Variablen `tempText` gespeicherte String genutzt, um ihn mit `"°F"` bzw. `" K"` zu einem neuen String zu verknüpfen. Somit führen die Zeilen 32–35 insgesamt zu folgender Ausgabe:

Verkettung von Strings

```
Temperatur in °C: 22
Temperatur in °F: 71.6
Temperatur in  K: 295.15
```

[17] Anstatt die Konstante `nullpunkt` zu verwenden, hätten wir natürlich genauso gut `tempC - (-273.15)` oder direkt `tempC + 273.15` schreiben können, ohne dass dies etwas an dem Wert des Ausdrucks geändert hätte. Sogar der Ausdruck `tempC - -273.15` wäre zulässig gewesen.

3.4 Funktionen in einem Go Programm

Funktionen stellen einen essentiellen Bestandteil eines jeden Go Programms dar, da sich ein Go Programm typischerweise aus einer Vielzahl von Funktionsdeklarationen und Funktionsaufrufen zusammensetzt. Daher möchten wir anhand des folgenden Beispielprogramms einige der wichtigsten Aspekte von Funktionen illustrieren, insbesondere wie Funktionen in Go prinzipiell deklariert werden. In Kapitel 7.1 werden wir uns dann ausführlicher mit Funktionen beschäftigen und dabei auch auf die verschiedenen Varianten von Funktionsdeklarationen eingehen.

```
1  package main
2
3  import "fmt"
4
5  // Deklaration einer Funktion zur
6  // Berechnung der Fläche eines Rechtecks aus den Seitenlängen
7  func rechteck(seiteA int, seiteB int) int {
8      var fläche int = seiteA * seiteB
9      return fläche
10 }
11
12 // Deklaration einer Funktion zur
13 // Berechnung des Volumens eines Quaders aus Grundfläche und Höhe
14 func quader(grundfläche int, höhe int) int {
15     var volumen int = grundfläche * höhe
16     return volumen
17 }
18
19 // Deklaration der obligatorischen Funktion main()
20 func main() {
21     var a int = 3
22     var b int = 4
23     var c int = 5
24     fmt.Println("Fläche eines", a, "x", b, "Rechtecks:", ↩
           ↪ rechteck(a, b))
25     fmt.Println("Volumen eines", a, "x", b, "x", c, "Quaders:", ↩
           ↪ quader(rechteck(a, b), c))
26 }
```

Listing 3.3: Deklaration von Funktionen für geometrische Berechnungen [funktionen_geometrie.go]

Das Programm aus Listing 3.3 liefert folgende Ausgabe:

```
Fläche eines 3 x 4 Rechtecks: 12
Volumen eines 3 x 4 x 5 Quaders: 60
```

In dem Programm werden zunächst zwei Funktionen zur Berechnung der Fläche eines Rechtecks bzw. des Volumens eines Quaders deklariert. Anschließend wird durch Aufrufe dieser Funktionen eine Rechteckfläche bzw. ein Quadervolumen berechnet und das Ergebnis ausgegeben.

Der Aufbau des Programms unterscheidet sich von den beiden vorherigen Programmen darin, dass neben der obligatorischen Funktion `main` zwei weitere Funktionen deklariert werden. Wir betrachten zunächst die Deklaration der Funktion `rechteck` in den Zeilen 7 bis 10 : Wesentliche Bestandteile dieser *Funktionsdeklaration* kennen wir bereits vom Aufbau der Funktion `main`: Die Funktionsdeklaration wird durch das Schlüsselwort `func` eingeleitet, darauf folgt der Funktionsname und abschließend wird mittels geschweiften Klammern `{` und `}` der Funktionsrumpf abgegrenzt.

Funktionsdeklaration

`func`

Die Funktion `rechteck` verfügt – im Gegensatz zur parameterlosen Funktion `main` – über zwei *Parameter*. Diese werden innerhalb der runden Klammern deklariert, die unmittelbar auf den Funktionsnamen folgen. Ähnlich wie bei einer Variablendeklaration wird bei der *Parameterdeklaration* der Name und Datentyp des Funktionsparameters festgelegt. Für den ersten Parameter der Funktion `rechteck` ist dies der Name `seiteA` und der Datentyp `int`. Getrennt durch ein Komma `,` folgt darauf die Deklaration des zweiten Parameters `seiteB`, der ebenfalls vom Typ `int` ist.

Parameter

Parameterdeklaration

Da die Funktion `rechteck` nach Beendigung ihres Aufrufs einen *Rückgabewert* liefern soll, muss nach den Parameterdeklarationen und vor Einleitung des Rumpfes (also zwischen der schließenden `)` und der öffnenden `{` Klammer) noch der *Rückgabetyp* der Funktion angegeben werden, d.h. der Datentyp des Rückgabewerts; dies ist hier ebenfalls der Typ `int`. Wenn eine Funktion keinen Rückgabewert liefert, dann wird diese Angabe einfach weggelassen (wie bereits bei der rückgabewertlosen Funktion `main` gesehen), d.h. auf die Parameterdeklarationen folgt dann unmittelbar die Einleitung des Funktionsrumpfes.

Rückgabewert

Rückgabetyp

Die oben beschriebene Deklaration eines Funktionsparameters bewirkt, dass quasi implizit eine entsprechende Variable erzeugt wird, die innerhalb des Funktionsrumpfes zur Verfügung steht. Der initiale Wert dieser *„Parameter-Variable“* entspricht dabei dem Wert des jeweiligen *Arguments*, das der Funktion bei einem Aufruf übergeben wird. Ein Funktionsparameter ist somit im Prinzip nichts anderes als eine Variable, welche allerdings auf eine spezielle Weise deklariert und initialisiert wird. So wird z.B. beim Funktionsaufruf `rechteck(a, b)` in Zeile 24 die Funktion mit den Argumenten `a` und `b` aufgerufen, wodurch der Funktion die Werte(!) übergeben werden, welche in den Variablen `a` und `b` gespeichert sind. Da `a` und `b` zum Zeitpunkt des Funktionsaufrufs mit den Werten 3 und 4

„Parameter-Variable“

Argument

belegt sind, werden bei der Ausführung der Funktion ihre beiden Parameter `seiteA` und `seiteB` mit den Werten 3 und 4 initialisiert. Diese Art der Übergabe von Argumenten wird als *call-by-value* bezeichnet.

Block

Der Rumpf einer jeden Funktion besteht – wie bereits bei der Funktion `main` gesehen – aus einer Folge von Anweisungen. Ganz generell wird eine Folge von Anweisungen, die von geschweiften Klammern `{` und `}` umschlossen ist, als (syntaktischer) *Block* bezeichnet. Somit stellt also der Rumpf einer Funktion einen Block dar.

Gültigkeitsbereich (scope)

Sichtbarkeit

Durch die Deklarationsanweisung in Zeile 8 wird eine Variable vom Typ `int` deklariert und mit dem Bezeichner `fläche` verbunden. Als *Gültigkeitsbereich (scope)* einer Deklaration wird der Codebereich bezeichnet, in dem der deklarierte Name des Programmelements (also der Variablen, Konstanten oder Funktion) bekannt und somit gültig ist. Generell gilt, dass sich der Gültigkeitsbereich einer Deklaration immer nur auf den Block (und ggf. vorhandene Unter-Blöcke[18]) erstreckt, in dem die Deklaration steht. Der Gültigkeitsbereich des Bezeichners `fläche` ist daher auf den Rumpf der Funktion `rechteck` beschränkt, d.h. außerhalb der Funktion ist die Deklaration des Bezeichners nicht bekannt und somit die Variable nicht zugreifbar. Man spricht in diesem Zusammenhang auch häufig von der *Sichtbarkeit* des Bezeichners. Da es sich bei den Parametern einer Funktion um implizit deklarierte Variablen handelt, ist ihr Gültigkeitsbereich ebenfalls auf den Funktionsrumpf beschränkt.

Der Variablen `fläche` wird bei ihrer Deklaration mit dem Wert des Ausdrucks `seiteA * seiteB` initialisiert. Das Ergebnis dieser Multiplikation hängt also von der konkreten Belegung der Parameter `seiteA` und `seiteB` bei Ausführung der Funktion ab. So werden z.B. beim Funktionsaufruf `rechteck(a, b)` in Zeile 24 die aktuellen Werte der Argumente `a` und `b` übergeben, wodurch die beiden Parameter `seiteA` und `seiteB` mit den Werten 3 und 4 initialisiert werden. Unter dieser Belegung liefert die Auswertung des Ausdrucks `seiteA * seiteB` folglich den Wert 12, mit dem dann die Variable `fläche` initialisiert wird.

return-Anweisung `return`

In Zeile 9 sehen wir zum ersten Mal eine sog. *return-Anweisung*, welche durch das Schlüsselwort `return` eingeleitet wird. Das Auftreten einer return-Anweisung bewirkt, dass die Ausführung der Funktion an dieser Stelle beendet wird und zum Aufrufpunkt der Funktion „zurückgekehrt“ wird, wobei ggf. der Wert eines auf `return` folgenden Ausdrucks zurückgegeben wird. Der Typ des zurückzugebenden Werts muss dabei dem in der Funktionsdeklaration festgelegten Rückgabetyp entsprechen. Die Anweisung `return fläche` sorgt hier also dafür, dass die Funktion beendet und der Wert der Variablen `fläche` zurückgegeben wird, welcher (wie in der Funktionsdeklaration festgelegt) vom Typ `int` ist.

[18] Wie wir später noch sehen werden, können innerhalb eines Blocks auch noch weitere (ggf. verschachtelte) Blöcke existieren. Da solche Unter-Blöcke ebenfalls ein Bestandteil des Blocks sind, in dem sie stehen, ist eine Deklaration auch in den Unter-Blöcken ihres Deklarationsblocks gültig.

Hinweis 3.7: Parameter vs. Argument

Wir unterscheiden im Zusammenhang mit Funktionen präzise zwischen den Begriffen „Parameter“ und „Argument“:

Bezogen auf *die Deklaration* einer Funktion sprechen wir von den *Parametern*, welche für die Funktion deklariert werden. Bei der Deklaration eines Parameters wird sein Namen und Typ festgelegt. Somit entspricht ein Parameter einer Variablen, die beim Funktionsaufruf initialisiert wird und deren Gültigkeitsbereich der Funktionsrumpf ist.

Bezogen auf *einen Aufruf* einer Funktion sprechen wir von den *Argumenten*, mit denen die Funktion aufgerufen wird. Die Werte dieser Argumente werden der Funktion bei ihrem Aufruf übergeben.

Bei der Ausführung der Funktion werden dann die *Parameter* der Funktion mit den *Werten der Argumente* des Funktionsaufrufs belegt.

Beispielsweise hat die Funktion `rechteck` in Listing 3.3 zwei Parameter: der erste Parameter ist `seiteA`, der zweite Parameter ist `seiteB`. Dementsprechend müssen bei jedem Aufruf der Funktion `rechteck` immer zwei Argumente angegeben werden. Dies sind beim Funktionsaufruf `rechteck(a, b)` (in Zeile 24) die beiden Variablen `a` und `b`, sodass deren aktuelle Werte 3 und 4 der Funktion übergeben werden. Bei einem Funktionsaufruf `rechteck(7, 9)` sind die Argumente `7` und `9` zwei Integer-Literale, sodass einfach die entsprechenden Zahlenwerte übergeben werden.

Umgangssprachlich wird zuweilen nicht sauber zwischen den Begriffen „Parameter“ und „Argument“ unterschieden. So könnte z.B. beim Aufruf `rechteck(a, b)` unsauber davon gesprochen werden, dass die Funktion mit „den Parametern `a` und `b`“ aufgerufen wird. Oder, etwas irreführend, könnten `a` und `b` auch als „Aufruf-Parameter“ bezeichnet werden. Wir verwenden zwar konsequent die präzisen Bezeichnungen „Parameter“ und „Argument“ im entsprechenden Kontext, möchten aber darauf hinweisen, dass der Gebrauch dieser Begriffe in der Praxis weniger konsequent ausfallen kann.

Die Deklaration der Funktion `quader` in den Zeilen 14–17 erfolgt analog zur Funktion `rechteck`.

In Zeile 24 wird die Funktion `fmt.Println` aufgerufen, wobei eines ihrer Argumente der Ausdruck `rechteck(a, b)` bildet. Da die Funktion `rechteck` einen Wert zurückliefert, stellt der Funktionsaufruf `rechteck(a, b)` einen Ausdruck dar. Die Auswertung des Ausdrucks führt schließlich zum Aufruf der Funktion, welche hier den Wert 12 zurückliefert. Der Wert des Ausdrucks `rechteck(a, b)` – hier also 12 – wird dann wiederum an

Funktionsaufruf als Ausdruck

verschachtelter Funktionsaufruf

die Funktion `fmt.Println` übergeben. Daher spricht man hier von einem *verschachtelten Funktionsaufruf*.

In Zeile 25 sehen wir ein noch deutlicheres Beispiel für einen verschachtelten Funktionsaufruf, denn hier ist eines der Argumente der Funktion `fmt.Println` der Ausdruck `quader(rechteck(a, b), c))`. Das erste Argument dieses verschachtelten Funktionsaufrufs bildet somit wiederum der Funktionsaufruf `rechteck(a, b)`. Dementsprechend wird zuerst die Funktion `rechteck` ausgeführt und ihr Rückgabewert wird anschließend an die Funktion `quader` übergeben. Deren Ausführung liefert dann den Wert 60 zurück, welcher schließlich an die Funktion `fmt.Println` übergeben wird.

3.5 Zusammenfassung

Anhand des kurzen Beispielprogramms in Listing 3.2 haben wir gezeigt, dass sich ein Go Programm aus einer Folge von Anweisungen zusammensetzt, die (im Allgemeinen) der Reihe nach ausgeführt werden. Zudem haben wir illustriert, wie eine Variable deklariert und ihr ein Wert zugewiesen werden kann, wie sich der Wert einer Variablen während des Programmablaufs ändern kann und wie ihr Wert ausgegeben werden kann.

Das Programm in Listing 3.2 enthält Beispiele für alle vier Klassen von symbolischen Elementen, auf denen die Syntax der Programmiersprache Go aufgebaut ist:

- Schlüsselwörter
- Bezeichner
- Literale
- Operatoren (und sonstigen Zeichen wie z.B. Klammern)

In Listing 3.3 haben wir kurz demonstriert, wie Funktionen deklariert und aufgerufen werden. Dabei sind wir auch auf Rückgabewerte von Funktionen eingegangen und haben den verschachtelten Aufruf von Funktionen betrachtet. Zudem haben wir den syntaktische Block vorgestellt, welcher den Gültigkeitsbereich eines deklarierten Bezeichners bestimmt.

Nachdem wir also anhand dreier sehr einfacher Beispiele den prinzipiellen Aufbau eines Go Programms kennengelernt haben, werden wir uns in den folgenden Kapiteln ausführlicher mit einigen der bereits kurz vorgestellten Programmelementen und Konzepten beschäftigen.