Einführung in die Programmierung mit *Go*

Prof. Dr. Christoph Reichenbach

13. März 2015

1 Konstanten

Go erlaubt die Definition von Konstanten durch das Schlüsselwort **const**:

```
const Pi = 3.14159265357989
const SekundenInEinemTag int64 = 24 * 60 * 60
```

Solche Konstanten erlauben uns, die Bedeutung einer Zahl expliziter zu machen; ein Ausdruck der Form `7 * SekundenInEinemTag` ist leicher zu verstehen als ein Ausdruck der Form `604800` und macht den Programmcode daher verständlicher und somit leichter wartbar.

Ähnlich wie bei **import**-Deklarationen kann auch **const** eine vereinfachte Notation mit Klammern verwenden:

```
const (
  Montag     int = 0
  Dienstag   int = 1
  Mittwoch   int = 2
  Donnerstag int = 3
  Freitag    int = 4
  Samstag    int = 5
  Sonntag    int = 6
)
```

Um solches Durchzählen 'von Hand' zu vermeiden, kann das Schlüsselwort **iota** verwendet werden. **iota** ist eine `int`-Zahl, die in einer solchen geklammerten sequentiellen **const**-Deklaration in der ersten deklarierten Konstanten den Wert 0 hat, in der zweiten den Wert 1 usw.

Das obige Beispiel ist also gleichwertig zu:

```
const (
  Montag     int = iota
  Dienstag   int = iota
  Mittwoch   int = iota
  Donnerstag int = iota
  Freitag    int = iota
  Samstag    int = iota
```

```
  Sonntag    int = iota
)
```

Wenn der zu setzende Wert für jede Konstante aus der gleichen Berechnung stammt, kann die Konstante auch ausgelassen werden:

```
const (
  Montag    int = iota
  Dienstag
  Mittwoch
  Donnerstag
  Freitag
  Samstag
  Sonntag
)
```

Die Konstante kann zudem in Berechnungen eingesetzt werden, sofern die Berechnung keine Funktionsaufrufe o.ä. beinhaltet[1]:

```
const (
  B          int = 1 << (10 * iota)
  KiB            // 1024
  MiB            // 1024 * 1024 ...
  GiB
  TiB
)
```

2 Grundlegende Kontrollstrukturen

2.1 Bedingte Ausführung

Die Ausführung eines Code-Stücks kann so beschränkt werden, daß sie nur stattfindet, wenn eine bestimmte Bedingung erfüllt ist:

```
if a > 0 {
    fmt.Println("a ist groesser als null")
}
```

Die Bedingung `a > 0` muß erfüllt sein, damit `fmt.Println(ä ist groesser als null")` ausgeführt wird. Um eine Alternative anzubieten, kann eine **else**-Klausel angehängt werden, die genau dann ausgeführt wird, wenn die per **if** bedingte Klausel nicht ausgeführt wird:

```
if a > 0 {
    fmt.Println("a ist groesser als null")
} else {
```

[1] Die Sprachspezifikation erlaubt 'constant expressions' und erlaubt Konstanten, arithmetische, logische und Bit-Operatoren, und Typkonvertierungen.

```
    fmt.Println("a ist nicht groesser als null, sondern ", a)
}
```

Einem **else** darf *nur* ein Block (geschweifte Klammern mit Anweisungen) oder ein weiteres **if** folgen:

```
if a > 0 {
    ...
} else if a < 0 {
    ...
} else {
    ...
}
```

Go erlaubt eine sogenannte 'kleine Anweisung' vor der eigentlichen Bedingung, durch ein Semikolon abgetrennt. Eine solche Anweisung wird normalerweise verwendet, um eine oder mehrere Variablen zu initialisieren, die nur im Körper des bedingten Codes sichtbar sein sollen:

```
var x = 10
if x := eingabe(); if x != "" {

    fmt.println("Benutzer hat eingegeben: ", x)
} // das 'x' aus 'x := eingabe()' ist hier nicht mehr sichtbar

fmt.Println(x) // gibt 10 aus
```

2.2 Die **switch**-Anweisung

Wenn der Kontrollfluß sich zwischen vielen verschiedenen Fällen unterscheiden muß, wird es umständlich, die nötigen **if** und **else**-Konstrukte zu schreiben. *Go* unterstützt ein alternatives Konstrukt zur Kontrollflußwahl namens **switch**, das eine Verallgemeinerung des gleichen Konstrukts aus C und Java ist. Beispielsweise können wir zwischen den verschiedenen Wochentagkonstanten wie folgt unterscheiden:

```
switch day {
  case Montag     : fmt.Println("Montag")
  case Dienstag   : fmt.Println("Dienstag")
  case Mittwoch   : fmt.Println("Mittwoch")
  case Donnerstag : fmt.Println("Donnerstag")
  case Freitag    : fmt.Println("Freitag")
  case Samstag    : fmt.Println("Samstag")
  case Sonntag    : fmt.Println("Sonntag")
}
```

Diese Anweisung prüft, zu welcher der **case**-Klauseln der Inhalt des Switch-Wertes `day` paßt, und führt diese Klausel aus. Wenn mehrere Klauseln passen, wird die erste passende Klausel ausgeführt.

Anders als in C und java ist es nicht nötig, ein **break** hinter jede **case**-Klausel zu setzen.

Der Switch-Wert muß keine Variable sein; jeder denkbare Ausdruck ist erlaubt. Ähnlich wie **if** erlaubt auch **case** eine einfache Anweisung vor der Variablen:

```
switch x := readInput(); x.message { // Switch-Wert ist 'x.
    message'
    ...
```

Eine **case**-Klausel kann mehrere Fälle abdecken:

```
var wochentag bool
switch day {
  case Samstag, Sonntag : wochentag = false
  default               : wochentag = true
}
```

Die optionale **default**-Klausel wird ausgeführt, wenn keine der **case**-Klauseln paßt.

switch hat alternative Formen. Wenn Sie den Switch-Wert auslassen, wird er implizit zu `true`. In der folgenden **switch**-Anweisung wird daher die erste Regel genommen, deren Bedingung wahr ist, da diese Bedingung dann ebenfalls zu dem Wert `true` auswertet:

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

Typ-Switch: **switch** erlaubt auch die Unterscheidung zwischen verschiedenen Typen mit einem sogenannten Typ-**switch**:

```
switch v.(type) {
  case nil     : return 0
  case int     : return x
  case string  : return strconv.atoi(x)
  case float64 : return int(x)
}
```

2.3 Schleifen

Schleifen erlauben die Wiederholung ihres Schleifenkörpers. Im folgenden Programm wird dieser 100 mal wiederholt, wobei die Variable x bei jedem Durchlauf

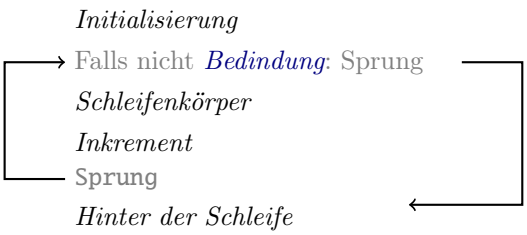
von initial 0 bis schlußendlich 99 wächst:

```
for x := 0; x < 100; x = x + 1 {
  /* Schleifenkoerper */
}
```

Allgemein haben **for**-Schleifen folgende Form:

for *Initialisierung*; *Bedingung*; *Inkrement* {
  Schleifenkörper
}
Hinter der Schleife

Schleifen werden wie folgt ausgeführt:



Die Pfeile zeigen den Kontrollfluß an. Wenn kein Pfeil genommen wird, fließt die Ausführung weiter nach unten.

Schleifen haben zwei Kurzformen. Die Schleife

```
for x > 0 { ... }
```

entspricht der Schleife

```
for /*leer*/; x > 0; /*leer*/ { ... }
```

und somit den 'while'-Schleifen anderer Programmiersprachen. **for**-Schleifen komplett ohne Bedingung wiederholen den Schleifenkörper unbegrenzt oft:

```
for {
    // hoere niemals auf
}
```

Zu **for**-Schleifen existieren drei weitere Sonderformen (Abschnitte 5.4, 8.3, und 9.2).

2.4 Sprungmarken und erweiterter Schleifen-Kontrollfluß

Ähnlich anderen imperativen Sprachen unterstützt auch *Go* Anweisungen, die die aktuelle Schleife sofort beenden oder den Rest der Schleife überspringen:

- **break** bricht die innerste Schleife mit sofortiger Wirkung ab, und
- **continue** überspringt den Rest der Schleife.

Das folgende Programm gibt beispielsweise die Zahlenfolge `1, 3, 4` aus:

```
i := 0
for {
  i++
  if i == 5 {
    break
  }
  if i == 2 {
    continue
  }
  fmt.Println(i)
}
```

break und **continue** können auch auf äußere Schleifen angewendet werden. Dazu müssen die Schleifen mit einer Markierung (label) versehen werden, die aus einem Namen gefolgt von einem Doppelpunkt besteht:

```
Outer:
for {
  for {
    break Outer
  }
}
```

Go unterstützt auch die **goto**-Anweisung, die jedoch nur in Ausnahmefällen zur Programmierung angemessen ist.

3 Funktionen

Funktionen werden mit dem **func**-Schlüsselwort, gefolgt vom Funktionsnamen, gefolgt von einer (möglicherweise leeren) Parameterliste in runden Klammern deklariert; der folgende Funktionskörper wird als Block (in geschweiften Klammern) angegeben. Die folgende Funktion nimmt z.B. drei Parameter und gibt diese an die Ausgabefunktion `fmt.Println` weiter:

```
func funcname(a, b int, c string) {
  fmt.Println(a, b, c)
}
```

Funktionen können auch einen Rückgabewert haben. Der Typ dieses Wertes wird zwischen schließender Parameterklammer und öffnender geschweifter Klammer des Funktionskörpers angegeben. Funktionen mit Rückgabewert müssen mindestens eine **return**-Anweisung haben. Diese Anweisung spezifiziert den zurückzugebenden Wert. Im folgenden Beispiel wird z.B. `true` zurückgegeben gdw der erste Parameter arithmetisch größer als der zweite Parameter ist; sonst wird `false` zurückgegeben:

```
func funcname(a, b int, c string) bool {
  fmt.Println(c)
  return a > b
}
```

Solce Funktionen können nun von Ausdrücken aus aufgerufen werden, z.B. `funcname(x, 3, "foo") || x == 0`.

return (ohne dahinterstehenden Ausdruck) kann auch bei Funktionen ohne Rückgabetyp eingesetzt werden, um die Ausführung der Funktion frühzeitig zu beenden.

Rückgabewerte können auch Namen erhalten; sie entsprechen damit den 'out-mode'-Parametern anderer Programmiersprachen. Diese Rückgabewerte werden auf den Nullwert ihres Typs initialisiert; wenn Ihnen ein Wert zugewiesen wird, müssen sie beim **return** nicht mehr explizit angegeben werden (das **return** ist aber weiterhin nötig).

```
func funcname(c string) (s string) {
  s = "[" + s + "]"
  return
}
```

Funktionen können mehrere Rückgabewerte haben, die entweder per Zuweisung oder durch ein Mehrfach-**return** gesetzt werden können:

```
func funcname(c string) (s string, error bool) {
  return "[" + s + "]", false
}
```

Die Rückgabewerte können beim Aufruf entpackt werden:

```
s, err := funcname("foo")
```

Diese Strategie wird oft verwendet, um einen 'normalen' Rückgabewert zusammen mit einem Fehlercode zurückzuliefern.

Funktionen können mit einer variablen Anzahl von Parametern definiert werden, indem der Typ des letzten (beliebig oft wiederholbaren) Funktionsparameters den Präfix `...` vor der Typangabe erhält:

```
func count(a ...int) int {
    return len(a)
}
```

Diese Funktion nimmt beliebig viele Parameter (z.B. `count(1, 2, 3, 4, 5)` und liefert die Anzahl der Parameter. Dabei hat `a` den tatsächlichen Typen `[]int`. Umgekehrt können auch Slices in eine Funktion mit variablen Parametern übergeben werden:

```
  a := []int{0, 1, 2, 3, 4, 5, 6}
  fmt.Println(count(a...))
```

Hier wird `...` als Suffix hinter der Slice-Variablen eingesetzt, um diese als Parameterfolge einzusetzen.

3.0.1 Funktionen als Werte erster Stufe

Go erlaubt es, Funktionen als Werte zu behandeln, also Funktionen in Variablen zu speichern, Funktionen als Parameter zu übergeben, usw. Der Typ solcher Funktionswerte ist **`func`**, gefolgt von der *Signatur* der Funktion, z.B.:

```
var f func(int, int) int
```

Dies deklariert eine Funktionsvariable `f` für eine Funktion mit zwei `int`-Parametern, die einen `int`-Parameter zurückliefert. Funktionsvariablen verhalten sich ähnlich Zeigervariablen; ihr Nullwert ist ebenfalls **`nil`**. Funktionsvariablen werden wie Funktionen aufgerufen.

3.0.2 Anonyme Funktionen

Go erlaubt auch anonyme Funktionen, hier zum Beispiel als Wert, der der Variablen `f` zugewiesen wird:

```
f := func(a, b int) int { return a + b }
```

Anonyme Funktionen können sich auf Variablen aush ihrer Umgebung beziehen und auch als Rückgabewerte zurückgeliefert werden:

```
func g(z int) (func(int) int) {
  f := func(a int) int { return a + z }
  return f
}
```

4 Operatoren

Go bietet folgende binäre Operatoren:

Präzedenz	Operatoren
5	* / % << >> & &^
4	+ - | ^
3	== != < <= > >=
2	&&
1	||

Höhere Präzedenz bindet stärker; bei gleicher Präzedenz sind die Operatoren links-assoziativ:

```
i * 2 - 6 / 3 * 2  =  (i * 2) - ((6 / 3) * 2)
```

Die Bedeutung der Operatoren ist wie folgt:

- `*`: Multiplikation (numerische Typen 8)
- `+`: Addition (numerische Typen 8) bzw. Zeichenketten-Konkatenierung (`string`-Typen)
- `-`: Subtraktion
- `/`: Division
- `%`: Divisionsrest
- `&`: Bits der Zahl einzeln verUNDen (integrale Typen)
- `|`: Bits der Zahl einzeln verODERn (integrale Typen)
- `^`: Bits der Zahl mit Exklusiv-Oder verknüpfen (integrale Typen)
- `&^`: Bitweise Nicht-Und (NAND) (integrale Typen)
- `<<`: Bitweise nach links schieben (integrale Typen)
- `>>`: Bitweise nach rechts schieben (integrale Typen)
- `==`: Gleichheitstest: `true` gdw beide Parameter strukturell gleich sind, sonst `false`
- `!=`: Negierter Gleichheitstest
- `<`, `>`, `<=`, `>=`: Arithmetischer Vergleich (nur für numerische Typen)
- `&&`: Logisches UND (`bool`) mit Kurzschlußauswertung
- `||`: Logisches ODER (`bool`) mit Kurzschlußauswertung

Go bietet zudem einige unäre Operatoren. Diese Operatoren haben immer Präzedenz über die binären Operatoren.

- `+`x: $0 + x$
- `-`x: $0 - x$
- `^`x: Bitweise Komplement
- `!`x: Logisches Komplement
- `&`x: Adreßoperator (Abschnitt 6.1)
- `*`x: Dereferenzierungsoperator (Abschnitt 6.1)
- `<-`x: Kanal-Leseoperator (Abschnitt 9.2)

Diese Operatoren können in beliebigen Ausdrücken verwendet werden. Für die binären Operatoren $\otimes$ existiert zudem eine Anweisungs-Sonderform:

$$x \otimes =y \text{ entspricht } x=x \otimes y$$

wobei allerdings x nur ein Mal ausgewertet wird[2].

Betrachten Sie das häufige Beispiel

Sei $\otimes$ ein Operator:

```
v ⊗= e
```

ist (fast) äquivalent zu

```
v = v ⊗ e
```

Damit können wir z.B. Schleifen wie

```
for x := 0; i < 10; i = i + 1 { ...
```

vereinfachen zu

```
for x := 0; i < 10; i = i += 1 { ...
```

Zwei weitere Operatoren decken den häufigen Fall ab, daß genau um eins erhöht oder vermindert wird:

- `v++` entspricht `v += 1`
- `v--` entspricht `v -= 1`

Die Schleife läßt sich also weiter vereinfachen zu:

```
for x := 0; i < 10; i = i++ {
```

5 Eingebaute Datenstrukturen

Neben den primitiven Datentypen verfügt *Go* auch noch über einige eingebaute Datenstrukturen, die wir in diesem Abschnitt betrachten:

- *Arrays*, die Sequenzen von Daten gleichen Typs im Speicher repräsentieren
- *Slices*, die einen Ausschnitt eines Arrays repräsentieren, und
- *Maps*, die einen (fast) beliebigen Datentyp auf einen anderen Datentyp abbilden.

[2] Dies ist relevant, wenn x z.B. `a[f()]` ist und `f()` Seiteneffekte hat

5.1 Arrays

Jedes Array `a` hat zwei Attribute:

- Größe: Die Anzahl der Elemente im Array, Notation `len(a)`
- Elementtyp: Der Typ der einzelnen Elemente im Array.

Beide Eigenschaften sind im Arraytyp fest notiert. So beschreibt der Typ `[7]int` ein Array der Größe 7 mit Elementtyp `int`.

Arrays können mit geschweiften Klammern initialisiert werden:

```
a := [3]double32{1.0, 2.0, 3.0}
```

Bei einer solchen Initialisierung ist die Anzahl der Elemente ggf. aus der Deklaration selbst ersichtlich. Man kann daher auch kurz schreiben:

```
b := [...]double32{1.0, 2.0, 3.0}
```

Dabei ist '`...`' kein eigener Typ, sondern nur eine vereinfachte Schreibweise; Array `b` hat ebenfalls den Typ `[3]double32`.

Wenn `a` ein Array ist, kann man auf den Eintrag an Stelle `i` per `a[i]` zugreifen, lesend oder schreibend. Dabei muß `i` ein integraler Typ (z.B. `int`) sein und zudem gößer oder gleich 0 und kleiner als `len(a)` sein.

```
package main
import "fmt"

func main() {
    a := [...]int{1, 2, 3}
    b := [...]int{-1, -1, -2}
    vp := vmult(a, b)
    fmt.Printf("%v * %v = %v\n", a, b, vp)
}

func vmult(va, vb [3]int) [3]int {
    for i := 0; i < len(va); i++ {
        va[i] *= vb[i]
    }
    return va
}
```

Dieses Beispiel demonstriert die Verwendung von Integer-Vektoren. Das Programm erzeugt folgende Ausgabe:

```
[1 2 3] * [-1 -1 -2] = [-1 -2 -6]
```

5.2 Slices

Ein *Slice* (Scheibe, im Sinne von 'Brotscheibe') ist ein Ausschnitt eines Arrays. Ein Slice `s` besteht dabei aus vier Komponenten:

- Eine Referenz auf das unterliegende Array
- Einen Index in das betreffende Array
- Eine Länge, `len(s)`
- Eine Kapazität, `cap(s)`

Slices werden in *Go* gegenüber Arrays bevorzugt verwendet, da sie flexibler sind. Wir betrachten daher zunächst ein Beispiel mit Slices statt Arrays:

```
package main
import "fmt"

func main() {
    a := []int{1, 2, 3}
    b := []int{-1, -1, -2}
    vp := vmult(a, b)
    fmt.Printf("%v * %v = %v\n", a, b, vp)
}

func vmult(va, vb []int) []int {
    for i := 0; i < len(va); i++ {
        va[i] *= vb[i]
    }
    return va
}
```

Dieses Beispiel ist fast identisch zu dem vorherigen; wir haben lediglich den Typ von Array auf Slice geändert, indem wir ihn als `[]int` geschrieben haben. `[]int` ist also ein Slice, während `[3]int` ein Array ist. Die Operation `len(s)` ist sowohl für Arrays als auch für Slices verfügbar.

Wenn wir das Programm nun ausführen, erhalten wir folgende Ausgabe:

```
[-1 -2 -6] * [-1 -1 -2] = [-1 -2 -6]
```

Dies ist wesentlich anders als im vorherigen Beispiel: Der Inhalt des Arrays `a` hat sich geändert! Diese Änderung ergibt sich durch einen subtilen semantischen Unterschied zwischen Arrays und Slices: Erstere verwenden Wertsemantik, letztere Referenzsemantik (Abschnitt 5.2.1). Das heißt u.a., bei der Parameterübergabe Arrayinhalte kopiert werden, während Slice-Inhalte nicht kopiert werden—stattdessen wird der gleiche Inhalt geteilt, und eine Änderung an einer Stelle (z.B. in `va` in `vmult`) ist an anderer Stelle (z.B. in `a` in `main`) ebenfalls sichtbar.

Um dieses Problem zu beseitigen, müssen wir einen neuen Slice zur Rückgabe allozieren. Slice-Allozierung erfolgt über die Operation **`make`**, z.B.:

```
make([]int, 100)
```

alloziert einen neuen Slice mit 100 Einträgen (gültige Indizes 0 bis 99). Das dem Slice zugrunde liegende Array ist dabei 'unsichtbar'. Wir können damit `vmult` wie folgt korrigieren:

```
func vmult(a, b []int) (result []int) {
    result = make(int[], len(a))
    for i := 0; i < len(a); i++ {
        result[i] = a[i] * b[i]
    }
    return
}
```

Slices ausschneiden: Um ein Slice aus einem existierenden Array zu erstellen, können wir den Slice-Operator `[:]` verwenden:

```
array := [...]int{0, 1, 2, 3, 4, 5}
a := [3:5]
```

Hierbei ist `a` nun eine Sicht auf die Elemente 3 und 4 des Arrays `array`; `len(a)` ist also 2, und Schreiboperationen auf `array[3]` und `a[0]` sind gleichwertig:

```
a[0] = 1000
fmt.Println(array[3])
```

gibt `1000` aus.

Slices können auch aus existierenden anderen Slices erstellt werden, per `a`[*min*:*max*]. Sowohl beim Erzeugen aus Arrays heraus als auch beim Erzeugen aus anderen Slices heraus muß folgendes eingehalten werden:

$$0 \leq min \leq max \leq \texttt{len(a)} \leq \texttt{cap(a)}$$

Wenn *min* ausgelassen wird, wird stattdessen 0 substituiert; wenn *max* ausgelassen wird, wird `len(a)` substituiert. Die folgende Operation erzeugt also ein Slice mit der exakt gleichen Größe wie `a`:

```
array := [...]int{0, 1, 2, 3, 4, 5}
a := array[:]
```

Kapazität von Slices: `cap(a)` ist die *Kapazität* des Slices, also die maximale Größe, auf die es wachsen kann. Diese Größe ist durch die Größe des unterliegenden Arrays beschränkt. Im Falle von

```
array := [6]int{0, 1, 2, 3, 4, 5}
a := array[3:4]
```

ist also `len(a)` = 1 und `cap(a)` = 3; das heißt, wir können mit `a[:3]` einen gültigen Slice erzeugen, der auf die Elemente `{3, 4, 5}` von Array `array` zeigt.

Um ein Slice mit größerer Kapazität direkt zu allozieren, kann ein dritter Parameter zu **`make`** übergeben werden:

```
make([]int, 100, 1000)
```

Das unterliegende 'unsichtbare' Array hat nun die Größe 1000, der Slice selbst weiterhin die Größe 100 aber die Kapazität 1000.

Kopieren und Anhängen in Slices: Gelegentlich ist es nötig, Daten von einem Array oder Slice in einen anderen zu kopieren oder neue Elemente an einen Slice anzuhängen. Dazu bietet *Go* die eingebauten Funktionen `copy` und `append`.

- `copy(a, b)` kopiert alle Elemente von `b` nach `a`, sofern `a` groß genug ist—ansonsten werden nur soviele Elemente kopiert, wie möglich ist.
- `append(a,` x_1`, ...,` x_n`)` hängt alle zusätzlich angegebenen Elemente $x_1, \ldots, x_n$ an `a` an. Falls die Kapazität von `a` nicht groß genug dafür ist, wird ein neuer Slice alloziert und zurückgegeben.

```
s := []int{0, 1, 2, 3, 4, 5}
s = append(s, 6, 7)
copy(s[2:4], []int{0, 0}) // 2, 3 auf Null setzen
```

5.2.1 Wertsemantik und Referenzsemantik

Das Zuweisen von Werten, z.B. beim Zuweisen eines Variableninhaltes in eine andere Variable, kann in Programmiersprachen auf zwei Weisen interpretiert werden: als *Wertsemantik* oder als *Referenzsemantik*. Wertsemantik bedeutet, daß der Wert selbst kopiert wird; Referenzsemantik bedeutet, daß beide Variablen nun die gleiche Bedeutung haben. Beachten Sie folgendes Beispiel:

```
a := b
a[1] = 2
b[1] = 3
```

- *Wertsemantik*: Nach diesem Code gilt `a[1]` $= 2$ und `b[1]` $= 3$. Der Rest von `b` stimmt mit `a` überein.
- *Referenzsemantik*: Nach diesem Code gilt `a[1]` $= 3$ und `b[1]` $= 3$. Die Variablen `a` und `b` bezeichnen den gleichen Speicherinhalt, so daß Änderungen in einem auch im anderen sichtbar sind.

Eine Variable mit Wertsemantik merkt sich also den Inhalt eines Objektes. Eine Variable mit Referenzsemantik hingegen merkt sich nur die Position des Objektes.

In *Go* haben Arrays Wertsemantik und Slices Referenzsemantik; Sie können also beide der obigen Ergebnisse beobachten, je nachdem, welchen Typ `b` hat.

5.3 Maps

Arrays und Slices repräsentieren Sequenzen. Sie können auch verwendet werden, um kleine Zahlen auf Werte abzubilden, z.B.:

```
a := []string{"null", "eins", "zwei", "drei"}
fmt.Println(a[2])
```

Sie eignen sich aber nicht dazu, andere Arten von endlichen Abbildungen (z.B. `string` auf `int`) zu realisieren, und sie sind sehr Speicher-ineffizient, wenn die Indizes bzw. Schlüssel sehr groß werden können, aber Teile des Indexbereiches nicht genutzt werden. Daher stellt *Go* einen allgemeinen Typen für partielle (endliche) Abbildungen zur Verfügung, sogenannte Maps (Abbildungen):

```
m := map[string]int{"null":0, "eins":1, "zwei":2, "drei":3,
                    "tausend":1000}
fmt.Println(m["drei"])
```

Der Typ einer Map ist **`map`**`[`*`Schlüsseltyp`*`]`*`Werttyp`*, wobei *Werttyp* beliebig ist und *Schlüsseltyp* jeder Typ sein kann, der mit == vergleichbar ist— dies sind fast alle Typen, bis auf Maps, Slices, und Funktionen. Eine solche Map bildet Werte des Schlüsseltyps auf Werte des Werttyps ab.

Lese- und Schreibzugriff auf Maps ist gleich wie für Arrays und Slices. Wenn der gesuchte Schlüssel nicht in der Map ist, wird allerdings der Nullwert des Wert-Typs zurückgeliefert. Um herauszufinden, ob der Wert in der Map ist, bietet *Go* eine alternative Leseoperation mit zwei Rückgabewerten an:

```
v, ok := m["dreiundzwanzig"]
```

Dabei ist `ok` vom Typ `bool` und `true` genau dann, wenn der Schlüssel gefunden wurde.

Es existiert eine zusätzliche Operation, um Einträge der Map zu löschen, nämlich **`delete`**`(map, schlüssel)`.

Als Beispiel:

```
delete(m, "eins")
```

Allozierung: Eine Map kann per **`make`** explizit alloziert werden:

```
make(map[string]int)
```

Mengen: *Go* hat keine explizite Unterstützung für Mengen, aber es ist leicht, eine Menge durch eine Map mit Wert-Typ `bool` zu simulieren.

5.4 Schleifen über Datentypen

Für Arrays, Slices und Maps existiert eine Sonderform der **for**-Schleife:

```
for key, value := range a {
    ...
}
```

Diese iteriert über alle Schlüssel bzw. Indizes und dazugehörigen Inhalte. Für Slices und Arrays erfolgt die Iteration aufsteigend; bei Maps ist die Reihenfolge undefiniert.

Wenn die Iteration nur die Schlüssel/Indizes benötigt, kann die Schleife verkürzt wie folgt geschrieben werden:

```
for key := range a {
    ...
}
```

Um nur über Werte zu iterieren, kann der Schlüssel ignoriert werden:

```
for _, value := range a {
    ...
}
```

Dabei steht '_' für den 'leeren Bezeichner', eine Pseudovariable, die nur beschrieben und nie gelesen werden kann. '_' kann überall dort verwendet werden, wo ein Ergebnis auftaucht, das irrelevant ist.

6 Eigene Datenstrukturen

Go erlaubt die Konstruktion eigener Datenstrukturen durch **struct**-Typen. Ein solcher Typ besteht aus beliebig vielen Feldern, die voneinander unabhängige Typen haben können. Folgender Typ beispielsweise beschreibt eine Struktur mit drei Feldern:

```
struct {
  name string
  semester uint8
  matnr int
}
```

Eine Variable dieses Typs kann z.B. wie folgt definiert werden:

```
var x struct { name string; semester uint8; matNr int; }
```

Der Nullwert einer solchen Struktur ist der punktweise Nullwerte der Felder; in diesem Fall wird also das Feld `name` auf `""` gesetzt, und `semester` und `matNr` auf 0.

Zugriff auf Felder erfolgt durch Punkt-Notation:

```
  var x struct { name string; semester uint8; matNr int; }
  x.semester = 5
  if x.matnr == 0 {
    ...
  }
```

Um die Struktur nicht bei jeder Verwendung ausschreiben zu können, kann ein neuer Typname erzeugt werden:

```
  type student struct { name string; semester uint8; matNr int;
      }
...
  var x student
  x.semester = 5
```

Diese Typnamen sind nicht ganz gleichwertig zu einem buchstäblich hingeschriebenen Strukturtypen; dieser Punkt wird in Abschnitt 8.2 diskutiert.

Strukturtypen dienen auch als Konstruktoren zur Initialisierung:

```
  x := student{ name:"Vorname Nachname", matNr:10 }
```

Strukturen können Substrukturen und komplexe Datentypen beinhalten:

```
type name struct {
  vorname, nachname string
}
type student struct {
  name name
  semester uint8
  matNr int
  kurseUndNoten map[string]float32
}
```

Substrukturen wie `name` sind in diesem Fall im Speicher direkt Teil der Struktur `student`.

Einbetten von Strukturen Eine Struktur kann Felder direkt von einer anderen Struktur *erben*, indem sie diese einbettet, ohne dem Feld einen Namen zu geben:

```
type name struct {
  vorname, nachname string
}
type student struct {
  name              // kein expliziter Variablenname
  semester uint8
  matNr int
  kurseUndNoten map[string]float32
}
```

In diesem Fall erbt `student` die Feldnamen `vorname` und `nachname` von `name`. Die beiden `fmt.Println`-Ausgaben im Folgenden sind also gleichwertig:

```
var s student
fmt.Println(s.vorname)
fmt.Println(s.name.vorname)
```

6.1 Zeiger

Da Substrukturen direkt eingebettet werden, ist es nicht möglich, eine Struktur sich selbst beinhalten zu lassen:

```
type list {
  value int
  next list  // Nicht moeglich!
}
```

Diese Datenstruktur wäre im Speicher unendlich groß, da jedes `list`-Objekt ein `next`-Element haben *muß*. Um dieses Problem zu umgehen, verwendet *Go* Zeiger. Diese entsprechen den Referenzen aus Sprachen wie Java. Wenn `T` ein Typ ist, dann ist `*T` der Typ eines Zeigers auf `T`. Zeiger verwenden explizit Referenzsemantik.

Die folgende Deklaration erzeugt einen Zeiger auf `int`-Werte:

```
var a *int
```

Der Nullwert von Zeigern ist ein spezieller Wert **`nil`**. Um mit Zeigern zu arbeiten, können wir explizit Adressen von verschiedenen Objekten nehmen und aus den Zeigern wieder die zugrundeliegenden Objekte extrahieren:

- `&v` ist ein Zeiger auf das Objekt `v`. Wenn `v` den Typen `T` hat, hat `&v` den Typen `*T`.

  `&` ist der *Referenzierungsoperator*.

- `*r` liest einen Zeiger aus. Wenn `r` den Typen `*T` hat, dann hat `*r` den Typen `T`. Wenn `r` gleich **`nil`** ist, ist dieser Zugriff ein Laufzeitfehler.

  `*` ist der *Dereferenzierungsoperator*.

  `*(&(v))` ist gleichwertig zu `v`.

Als Beispiel:

```
a := 1
ra := &a  // Zeiger auf a
rb := &a  // ra und rb zeigen beide auf a
*ra = 2   // Dereferenziere ra; schreibt indirekt auf a
fmt.Println(a, *ra, *rb)     // gibt "2 2 2" aus
```

Mit Zeigern läßt sich nun ein Typ für Listen von `int`-Zahlen definieren:

```
type list {
  value int
  next *list
}

func Length(ll *list) int {
    if ll == nil {
        return 0
    } else {
        return 1 + Length(ll.next)
    }
}
```

Beachten Sie, daß es nicht nötig ist, `(*ll).next` zu schreiben. Wenn ein Typ `T` ein Feld `f` hat, dann erlaubt *Go* den Zugriff auf `f` auch für Objekte vom Typ `*T`; die Dereferenzierungsoperation ist in diesem Fall implizit.

Meist ist eine Speicherallozierung über den Referenzierungsoperator `&` in der Praxis ausreichend. Alternativ stellt *Go* auch einen expliziten Allozierungsoperator zur Verfügung: Das Programm

```
  n := new(student)
```

erzeugt ein Objekt vom Typ `*student`. Alle Felder in `*student` sind auf ihre Nullwerte gesetzt.

7 Das Typsystem

Go ist statisch getypt. Abbildung 1 führt einige der zugrundeliegenden Typen auf; weitere Typen sind in Abschnit 5 (Array, Slice, Map), Abschnitt 6.1 (Zeiger) und Abschnitt 6 erläutert.

Es existieren einige weitere vordefinierte Typen:

- `byte` = `uint8`
- `rune` = `int32`
- `int` ist ein benannter Typ (Abschnitt 8.2), der `int32` oder `int64` entspricht.
- `uintptr` ist ein integraler Typ zur (typ-unsicheren) Konvertierung zwischen Zeigern und Zahlen; wir betrachten ihn hier nicht weiter.

Die numerischen `int` und `uint`-Typen (`int`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `byte`, `rune` und alle benannten Typen, die diesen entsprechen) werden auch als *integrale Typen* bezeichnet. Die integralen Typen zusammen mit den `float`-Typen (`float32`, `float64`) und den komplexen Zahlen (Abschnitt 8.1)

Typ	Reichweite	Beispiel	Nullwert
int8	$-2^7 \ldots 2^7 - 1$	-1, 0x2a	0
uint8	$0 \ldots 2^8 - 1$	1, 0xa0	0
int16	$-2^15 \ldots 2^{15} - 1$	-1000, 0x3e8	0
uint16	$0 \ldots 2^{16} - 1$	1, 0xa000	0
int32	$-2^{31} \ldots 2^{31} - 1$	-10000000, 0x1234567	0
uint32	$0 \ldots 2^{32} - 1$	1, 0xa0000000	0
int64	$-2^{63} \ldots 2^{63} - 1$	-1234567890	0
uint64	$0 \ldots 2^{64} - 1$	0x102030405060cafe	0
float32	32-Bit Fließkomma	1e+7, 13.7	0.0
float64	64-Bit Fließkomma	1.2e+200, 13.7	0.0
string	Zeichenketten	"Unĩcøδe", ‘xy‘	""
bool	Wahrheitswerte	true, false	false
*T	Zeiger auf T	&x, wenn x den Typ T hat	nil

Abbildung 1: Einige der eingebauten Typen

Dabei ist rune der Typ von Zeichen in einem string. Solche Zeichen können zwischen einzelnen Anführungszeichen angegeben werden; hier ist z.B. ein Slice von rune-Werten:

```
[]rune{'a', 'ä', '0'}
```

Zusätzlich zu den benannten Typen verfügt *Go* über folgende Typen:

- Funktionstypen (Abschnitt 3)
- Interface-Typen (Abschnitt 8.5)
- Kanaltypen

7.1 Komplexe Zahlen

Go hat eingebaute Unterstützung für komplexe Zahlen, durch die Typen complex64 und complex128. Komplexe Zahlen bestehen aus einem Realteil und einem Imaginärteil, die in *Go* mit dem gleichen Fließkommatyp kodiert werden müssen: float32 für complex64, oder float64 für complex128. Arithmetische Operationen werden wie auf normalen Zahlen unterstützt.

Imaginäre Zahlen können entweder mit der eingebauten Funktion complex, oder mit spezieller i-Notation erzeugt werden:

```
x1 := complex(3, 2)
x2 := 3+2i          // Aequivalent zu x1
r1, i1 := real(x1), imag(x1)
```

Die Funktionen real und imag extrahieren den Realteil bzw. den Imaginärteil der komplexen Zahl.

7.2 Benannte Typen

Go erlaubt die Einführung neuer, benannter Typen über **type**-Deklarationen. Jede solche Deklaration führt einen neuen Typnamen ein:

```
type id int32

var x id = 17
```

Mit diesen Typnamen kann man das Typsystem nutzen, um Fehler vor der Ausführung zu finden:

```
type id int
type age int

agemap := map[id]age{10001:22, 10002:37, 10003:49}
age := agemap[10002]
agemap[age] = 10004 // Typfehler: 'age' ist nicht vom Typ 'id'
```

Dies ist insbesondere in Strukturen und Funktionen nützlich, die gleich strukturierte Typen für unterschiedliche Zwecke nutzen (so können Identifikationsnummern, Kosten, Alter, Anzahl und viele andere Konzepte als `int` ausgedrückt werden).

Dieser Typname muß zur expliziten Konvertierung verwendet werden, wenn man strukturgleiche Typen verwenden will:

7.3 Zeichenketten und Runen

Zeichenketten in *Go* speichern Strings im UTF-8-Format. Dieses Format ist kompakt, wenn man Zeichen verwendet, die dem ASCII-Standard entsprechen– also ohne Umlaute, CJK-Zeichen etc. Wenn man allerdings Umlaute verwendet, werden Zeichen gegebenenfalls durch zwei Bytes dargestellt. Die **range**-Iteration über Zeichenketten berücksichtigt dies, so sehen wir beispielsweise für das Programm

```
for i, rune := range "blåbärsoppa"{
    fmt.Println(i, rune)
}
```

die Ausgabe:

```
0 98
1 108
2 229
4 98
5 228
7 114
8 115
9 111
10 112
```

```
11 112
12 97
```

Dabei ist jeweils die erste Ausgabe der Index in die Zeichenkette, und die zweite Ausgabe die `rune` an dieser Stelle in der Zeichenkette.

Wir sehen hier deutlich den Sprung in der Index-Variable von 2 auf 4, und von 5 auf 7. Die Sprünge können noch größer ausfallen; im Wort "お好み焼" ist beispielsweise jedes Zeichen mit drei Bytes kodiert.

`len(s)` für eine Zeichenkette liefert die Anzahl der Bytes in der Zeichenkette; um die Anzahl der Runen zu bestimmen, muß entweder eine Schleife oder eine Bibliotheksfunktion verwendet werden.

7.4 Methoden

In einigen Fällen kann es nützlich sein, Strukturen mit Helferfunktionen zu versehen. Wenn wir beispielsweise Studenteninformationen mit getrennten Vor- und Nachnamen speichern, aber oft den vollständigen Namen aus diesen konstruieren müssen, können wir eine Helferfunktion in der Struktur anbringen:

```
type student struct {
  firstname, lastname string
  fullname func(student) string
}

var s student
// Initialisiere s.fullname mit anonymer Funktion:
s.fullname = func(self student) string {
  return self.firstname + self.lastname
}
// Jetzt koennen wir s.fullname() aufrufen:
fmt.Println(s.fullname(s))
```

Diese Konstruktion ist allerdings umständlich: Zum Einen müssen bei jedem Bau von `student` die Zuweisung auf `s.fullname` durchführen, zum Anderen müssen wir beim Aufruf immer `s` selbst als Parameter explizit übergeben.

Um dies zu vereinfachen, erlaubt *Go* neben der Definition von Funktionen die Definition von *Methoden*, um auf Typen zu arbeiten. Methoden sind eine Sonderform von Funktionen, die immer einen zusätzlichen Parameter nehmen, z.B. `self` in diesem Beispiel:

```
func (self student) fullnameMethod() string { ... }
```

Sie werden ähnlich wie in Feldern gespeicherte Funktionen mit der Punkt-Notation aufgerufen, z.B. `s.fullnameMethod()`. Im Gegensatz zum Aufruf von `fullname` wird beim Aufruf von `fullnameMethod` allerdings der zugrundeliegende Wert implizit mit übergeben, so daß wir keinen zusätzlichen Parameter übergeben müssen.

Solche Methoden können auch verwendet werden, um Informationen zu verbergen. Im folgenden Beispiel ist das Feld `semester` für externe Module unsichtbar, aber die Methode `Semester` ist sichtbar:

```
type student struct {
    firstname, lastname string
    semester int
}

func (s student) Semester() int {
  return s.semester
}

...
var st student
fmt.Prinltn(st.Semester())
```

Um Methoden zum Ändern der zugrundeliegenden Daten zu verwenden, müssen wir allerdings vorsichtig sein. Wenn der implizite Parameter mit Wertsemantik übergeben wird, wird er kopiert, so daß jegliche Änderungen im Wert sich nur auf den formalen Parameter der Methode beziehen:

```
func (s student) IncSemester() {
  s.semester++ // Aktualisierung wird verworfen
}

var st student
st.IncSemester() // Wirkungslos!
```

Um dies zu vermeiden, können wir stattdessen die Methode nur auf einem *Zeiger* auf den Strukturtyp definieren. Dies erlaubt uns die Aktualisierung:

```
func (s *student) IncSemester() {
  (*s).semester++
}

var st student
st.IncSemester() // Funktioniert!
```

Beachten Sie, daß *Go* an dieser Stelle implizit die Adresse von `st` nimmt, um die Operation durchzuführen.

Der Typ `*T` erbt alle Methoden von `T`, analog zu den Feldzugriffen. In unserem obigen Beispiel können wir auf eine Variable `sp` vom Typ `*student` also sowohl `sp.IncSemester()` als auch `sp.Semester()` ausführen.

Wenn eine Struktur eine andere Struktur einbettet, erbt sie auch deren Methoden, analog zu Feldern. Diese Methoden können aber nicht überschrieben werden.

```
type masterStudent {
  student
```

```
  bachelorNote float32
}

var st masterStudent
st.IncSemester() // geerbt!
```

Methoden können nicht nur auf Strukturen und Zeigern definiert werden, sondern auch auf den meistern anderen Typen:

```
type id int

func (id id) String() string {
  // strconv.Itoa aus dem Paket "strconv" wandelt
  // 'int' nach 'string' um:
  return "id#" + strconv.Itoa(int(id))
}

func (num int) String() string {
  return "nr:" + strconv.Itoa(int(id))
}
```

Diese Methode `String()` ist nun für den Typen `id` definiert; wenn wir sie auch für `int` definieren, beschreibt der Typ des Wertes, auf dem wir sie aufrufen, welche der beiden Methoden ausgeführt wird:

```
  var a id
  var b int
  fmt.Println(a.String(), b.String())
```

Die Methode `String()` wird im Übrigen implizit von `fmt` verwendet, um Werte bei der Ausgabe in Zeichenketten zu verwandeln, wenn definiert[3]; `fmt.Println(a.String())` ist also gleichwertig zu `fmt.Println(a)`.

7.5 Interfaces

In vielen Fällen müssen Algorithmen verschiedene Arten von Objekte betrachten, die sich unterschiedlich verhalten können. Betrachten Sie z.B. eine physikalische Gravitationssimulation. In einer solchen Simulation können verschiedene Objekte auftauchen (Planeten, Sonnen, Meteoriten, Kometen) zu denen wir gegebenenfalls jeweils andere Information speichern wollen. Um an der Simulation der Gravitationskraft teilzunehmen, müssen allerdings alle diese Objekte erklären, wie groß ihre Masse ist und wo sie sind, und sie müssen eine Änderung ihres Impulses erlauben. Beispielsweise würden wir gerne folgenden Code schreiben:

```
type planet ...
type sun ...
type comet ...
```

[3] Die genauen Regeln zur Ausgabe sind etwas komplizierter und in der Moduldokumentation (`https://golang.org/pkg/fmt/`) genauer beschrieben.

```
var universe = []universeObject{sun(...), planet(...), comet
    (...) }

func computeAndApplyGravity(universe []universeObject) {
  ...
}
```

Die Funktion `computeAndApplyGravity` könnte dann durch `universe` iterieren und den Impulsvektor jedes Elementes im Universum aktualisieren. Allerdings benötigen wir dazu einen Typ `universeObject`, der uns erlaubt, sowohl über `sun`, als auch über `planet` und `comet` zu sprechen.

In *Go* können solche Typen über gemeinsame *Methodenmengen* der verschiedenen Typen definiert werden. In unserem Beispiel benötigen wir Methoden zur Bestimmung der Masse und der Position, und zur Änderung des Impulses. Wir fassen diese Methoden in einem sogenannten **`interface`** zusammen:

```
type position []float64
type impulse []float64

type universeObject interface {
  Mass() float64
  Position() position
  AddToImpulse(impulse)
}
```

Jeder Typ, der die gegebenen Methoden definiert, kann nun als `universeObject` verwendet werden:

```
type sun *struct { ... }

func (s sun) Mass() float64 {
  ...
}

func (s sun) Position() position {
  ...
}

func (s sun) AddToImpulse(impulse) {
  ...
}

var universeObject obj = sun(...)
```

Der Typ `sun` muß nichts über `universeObject` wissen. Wenn später in einem unabhängigen Modul ein anderes Interface mit der Methode `Mass() float64` definiert wird, kann `sun` dort problemlos eingesetzt werden. Allgemein ist ein

Typ `T` ein Subtyp eines Interface-Typs `I` genau dann, wenn `T` alle von `I` geforderten Methoden implementiert.

Diese Form der Subtypisierung nennt sich *strukturelle Subtypisierung*.

7.6 Dynamische Typprüfung

Da *Go* zu jedem Interface-Typen dynamische Typinformationen speichert, können wir jedes Objekt eines solchen Typen auf die Typinformationen prüfen. Die einfachste Art, dies zu tun, ist die Typversicherung:

```
var z interface{} = ...
var s string
s = z.(string)
```

Diese Operation macht `z` als Variable vom `string`-Typ verfügbar, sofern der dynamische Typ von `z` der Typ `string` ist. Ansonsten wird eine Laufzeitpanik ausgelöst, und das programm bricht ab.

Allgemeiner ist eine solche Konvertierung erfolgreich, wenn der Typ kompatibel ist. Beispielsweise können wir jeden Typen, der die Methode `func String() string` implementiert, erkennen:

```
var z interface{} = ...
s := z.(interface{ String()string })
```

Um nur zu prüfen, ob der betreffende Typ vorliegt, und ansonsten keine Panik auszulösen, können wir eine Form der Zuweisung mit zwei Rückgabewerten verwenden:

```
s, ok := z.(interface{ String()string })
```

verwenden. `ok` wird auf `true` gesetzt, wenn `z` den gewünschten Typ hat, sonst auf `false`. In letzterem fall wird `s` auf `nil` gesetzt.

8 Nebenläufige Ausführung

Go vereinfacht die Verwendung von nebenläufigen Prozessen durch zwei Konzepte:

- *Goroutinen*, die Threads in anderen Programmiersprachen entsprechen, und
- *Kanälen*, die serialisierte Kommunikation zwischen Threads ermöglichen.

8.1 Goroutinen

Jedes `main`-Programm startet in einer einzelnen Goroutine. Um eine zusätzliche Goroutine zu starten, wird das Schlüsselwort `go` verwendet. Folgende Zeile startet die Funktion `f` mit dem Parameter `1` in einer nebenläufigen Goroutine:

```
go f(1)
```

Oft wollen wir nur eine anonyme Goroutine starten. In diesem Fall reicht es, eine anonyme Funktion zu definieren und zu starten, und ein **go** voranzustellen:

```
go func() {
  ...
}()
```

Gefahren bei Nebenläufigkeit: Eine neu gestartete Goroutine läuft im gleichen Adreßraum wie die alte Goroutine, von der aus sie gestartet wurde; Zustandsänderungen in einer Goroutine sind also auch in der anderen Goroutine sichtbar. Allerdings ist nicht garantiert, daß die Zustandsänderungen in einer bestimmten Reihenfolge sichtbar werden oder konsistent sind; so ist folgendes möglich:

```
a := 0
b := 0

go func() { // A
  a = 1
  b = 1
}()

go func() { // B
  a = 2
  b = 2
}()

a0 = a
b0 = b

fmt.Printf("[%d %d]", a0, b0)
```

An dieser Stelle erlaubt die Sprachsemantik die Ausgabe von `[0 0]`, `[0 1]`, `[0 2]`, `[1 0]`, `[1 1]`, `[1 2]`, `[2 1]`, `[2 1]`, oder `[2 2]`. Beachten Sie insbesondere, daß die Haupt-Goroutine die Variablenänderungen nicht notwendigerweise in der gleichen Reihenfolge sieht, in der sie die einzelnen Threads sehen; Goroutinen `A` und `B` schreiben `b` nach `a`, aber eine andere Goroutine kann die Änderungen auf `b` unter Umständen vor den Änderungen auf `a` sehen. Grund dafür sind verschiedene Optimierungen im Übersetzer und der Hardware-Architektur des Prozessorsystems.

Aus den gleichen Grüdnen ist im Programm

```
type t struct {
  a int
  b string
```

```
}

func main() {
  t0 := t{1, "X"}
  t1 := t{2, "Y"}

  go func() {
    t0 = t1
  }
  fmt.Println(t0)
}
```

die Ausgabe `{1 "Y"}` genauso möglich wie `{2 "X"}`.

Parallele Ausführung von Goroutinen: Mit den Standardeinstellungen wird maximal eine Goroutine gleichzeitig ausgeführt, selbst auf Mehrkernprozessoren. Um dies zu ändern, muß die Umgebungsvariable `GOMAXPROCS` auf einen Wert größer als 1 geändert werden; diese Variable gibt an, wieviele Goroutinen gleichzeitig laufen können.

Nebenläufigkeit und Parallelismus: Die Begriffe *parallel* und *nebenläufig* (engl. *concurrent*) bedeuten verschiedene Dinge.

Zum Verständnis dieser Konzepte betrachten wir Threads (wie z.B. Goroutinen) als Folge von Zustandsänderungen. Wenn beide Threads zusammen ausgeführt werden können, stellt sich die Frage, in welcher Reihenfolge die Zustandsänderungen eines Threads relativ zu den Zustandsänderungen des anderen Threads stattfinden. Zwei Threads sind *nebenläufig*, wenn es mehr als eine Reihenfolge gibt, in der diese Zustandsänderungen stattfinden können.

Nebenläufigkeit ist also eine *semantische* Eigenschaft, die Aussagen über das beobachtbare Verhalten von Programmen trifft.

Parallelität hingegen ist eine *operationale* Eigenschaft, die ausdrückt, daß zwei Threads *gleichzeitig* ausgeführt werden können. Motivation für Parallelität ist dabei meist die effizientere Nutzung von Rechenressourcen.

In einem parallelen System ist Nebenläufigkeit notwendig gegeben. Ein System kann allerdings nebenläufig sein, ohne parallel zu sein.

8.2 Kommunikationskanäle

Da Goroutinen aufgrund ihrer Nebenläufigkeit keine Variablen zur Kommunikation verwenden können, benötigen wir einen anderen Kommunikationsmechanismus. *Go* stellt dafür *Kanäle* (channels) zur Verfügung. Der Typ eines Kanals ist `chan T`, wobei `T` ein weiterer Typ ist, der über den Kanal übertragen werden kann. Beispielsweise ist `chan int` ein Kanal, der `int`s übertragen kann. Um eine Variable `x` in einen Kanal `c` zu schreiben bzw. von diesem Kanal zu lesen, verwendet *Go* die folgende Notation:

- `c <- x` (schreiben)

- `x := <- c` (lesen)

Beachten Sie, daß diese Operationen *blockieren*: wenn Sie in einen Kanal schreiben, wird das Programm nicht weiter ausgeführt, bis der Wert am anderen Ende gelesen wurde. Ebenso wird beim Erreichen einer Leseoperation nicht weiter ausgeführt, bis ein zu lesender Wert geschrieben wurde.

Um einen Kanal zu erzeugen, verwenden Sie **`make(chan T)`**. Der Nullwert des Kanaltyps ist **`nil`**.

```
var c chan int = make(chan int)

go func() {
  fmt.Println(<-c)
}()

fmt.Println("A")
c <- 17
```

In diesem Beispiel sehen Sie zwei Goroutinen– die implizite Goroutine im Hintergrund, und eine explizit mit **`go`** gestartete Routine. Erstere schreibt in `c` und letztere liest aus `c`. Obwohl die zweite Goroutine vor dem `fmt.Println("A")` gestartet wird, ist garantiert, daß zuerst `A` und dann `17` geschrieben wird, da die zweite Goroutine darauf warten muß, daß 17 in textttc geschrieben wurde.

Kanäle können auch unidirektional sein. Ein Kanal vom Typ `<-chan T` kann nur gelesen werden, während ein Kanal vom Typ `chan<- T` nur zum Schreiben verwendet werden kann:

```
func countUp(max int) <-chan int {
c := make(chan int)
go func() {
  for i := 1; i < max; i++ {
    c <- i
  }
  close(c)
}
return c
}
```

Diese Funktionen erzeugt einen Kanal, in den sie alle Zahlen bis zum gegebenen Limit schreibt. Auf diese Weise können wir eine Art der Auswertung die der Bedarfsauswertung ähnelt realisieren: die Schleife in der Goroutine in `countUp` blockiert jedes mal, solange ihr nächster Wert noch nicht benötigt/ausgelesen wurde.

Beachten Sie die Verwendung der Funktion `close`: Diese Operation schließt einen Kanal. Das Lesen von einem geschlossenen Kanal ist ein Laufzeitfehler.

Wenn Sie nicht wissen, ob ein Kanal geschlossen wurde, können Sie dies durch eine erweiterte Leseoperation herausfinden, und zwar `x, ok := <- c`:

```
  c := countUp(1000)
  x := <- c     // liest 1
  x, ok := <- c // liest '2, true'
  if !ok {
    fmt.Println("Kanal geschlossen")
  }
```

Die beiden möglichen Fälle können wir aber anhand von `ok` unterscheiden, wobei `ok = false` besagt, daß der Kanal geschlossen wurde.

for-Schleifen: Analog zu Strings und Slices bietet Go auch die Möglichkeit, über Kanäle zu iterieren:

```
  for i := range countUp(1000) {
    fmt.Println(i)
  }
```

Kanäle mit Puffern: Normalerweise blockieren Kanäle, sobald sie beschrieben wurden. Manchmal ist dies nicht die gewünschte Semantik. Wir können daher einen zusätzlichen Parameter zu **make** angeben, der einen Puffer alloziert:

- **make(chan *T*, s)**: Alloziert Kanal mit Buffergröße `s`

```
  c := make(chan string, 3)

  c <- "eins"
  c <- "zwei"
  c <- "drei"

  fmt.Println(<-c)

  c <- "vier"
  c <- "fuenf" // Puffer voll: blockiert!
```

Wir können dies z.B. nutzen, um Arbeitsqueues zu bauen:

```
var workqueue chan workitem = make(chan workitem, 1000)

func enqueueWork(w workitem) {
  workqueue <- w
}

func runWorker(process func(workitem)) {
  for w <- workqueue {
    process(w)
  }
}
```

```
func main() {
  ...
  for i := 0; i < cap(workqueue); i++ {
    go runWorker(doStuff)
  }
  // genau vier Goroutinen
}
```

Kanäle als Semaphoren: Wie wir gesehen haben, können Kanäle nicht nur zur Kommunikation verwendet werden, sondern auch, um den Kontrollfluß zu steuern. Wir können daher Kanäle gezielt als *Locks* oder *Semaphoren* einsetzen:

```
lock := make(chan bool, 1)

func() setLocked(v value) {
  lock <- true // der Wert ist irrelevant
  globalVar = value
  <-lock
}
```

Hier kann immer nur eine Goroutine gleichzeitig auf `globalVar` zugreifen[4]. Da der Puffer für `lock` die Größe 1 hat, wird jede zusätzliche Goroutine beim Versuch, `lock <- true` auszuführen, blockieren, bis die zuerst ausführende Goroutine `<-lock` ausgeführt und damit den Puffer wieder geleert hat.

Wir können durch solche Semaphoren natürlich auch die Anzahl der erlaubten Threads auf 2 oder 3 setzen, indem wir die Puffergröße erhöhen. Auch auf diese Weise

```
var semaphore chan bool = make(chan bool, 4)

func main() {
  ...
  for {
    semaphore <- true // Wartet, wenn mehr als 4 Goroutinen
        aktiv sind
    go func() {
      doWork()
      <- semaphore    // Gibt einen Platz wieder frei
    }
  }
  ...
}
```

[4] Alternativ bietet *Go* auch 'Atomics' an, die einen atomischen Schreibzugriff ermöglichen. In bestimmten parallelen Anwendungen kann dies effizienter sein; wir behandeln dieses Konzept aber nicht weiter.

8.3 select

Ein Stück Code kann als *Multiplexer* fungieren, also von mehreren Kanälen gleichzeitig lesen. Dies ist z.B. für ein Visualisierungsprogramm wichtig, das aus mehreren Quellen relevante Informationen erhalten könnte. Für diesen Fall bietet *Go* die Kontrollstruktur **select**:

```
select {
  case x := <-c1:
       fmt.Println("c1: ", x)
  case x := <-c2:
       fmt.Println("c2: ", x)
}
```

Jedes **select** besteht aus beliebig vielen **case**-Klauseln, die die gegebene Form haben Dieser Code wartet darauf, ob ein Kanal bereit zu lesen ist, und aktiviert an dieser Stelle die betreffende **case**-Klausel. Wenn beide gleichzeitig bereit sind, wird zufällig eine der Klauseln ausgewählt.

Wenn wir nicht wollen, daß **select** blockiert, können wir eine **default**-Klausel einsetzen:

```
select {
  case x := <-c1:
       fmt.Println("c1: ", x)
  case x := <-c2:
       fmt.Println("c2: ", x)
  default:
       fmt.Println("Niemand hat etwas zu lesen")
}
```

case-Klauseln können auch auf Geschlossenheit des Kanals prüfen:

```
select {
  case x, ok := <-c1:
  if !ok {
    c1 = nil
  } else {
    result <- x
  }
  case x, ok := <-c2:
  if !ok {
    c2 = nil
  } else {
    result <- x
  }
}
```

Hier werden Kanäle, die geschlossen wurden, auf **nil** gesetzt. **select** ignoriert Kanäle, die **nil** sind (betrachtet sie als *blockierend*).

Wenn die Anzahl der Kanäle, auf die Sie warten wollen, erst zur Laufzeit bekannt ist, können Sie stattdessen `reflect.Select` aus dem `"reflect"`-Paket verwenden.

9 Verzögerte Ausführung mit `defer`

In komplexeren Funktionen ist es oft nötig, Aufräumarbeiten durchzuführen– Dateien oder Netzwerkverbindungen zu schließen, den Abschluß einer Berechnung zu melden etc. Wenn die Funktion auf mehrere verschiedene Weisen beendet werden kann, kann es aufwändig werden, all diese Anforderungen korrekt umzusetzen. Um dies zu vereinfachen, erlaubt *Go* das Konzept eines *verzögerten Aufrufs*. Ein solcher Aufruf, mit dem Schlüsselwort **`defer`** markiert, wird durchgeführt, sobald die Funktion beendet wird– unabhängig davon, welches Ende sie nimmt.

In dem folgenden Beispiel wird eine Datei geöffnet. Wenn diese Operation erfolgreich war, wird sofort mit **`defer`** `file.Close()` eine Operation registriert, um diese Datei wieder zu schließen. `file.Close()` wird also am Ende der Funktion in jedem Fall ausgeführt, auch, wenn ein Fehler weiter hinten in der Funktion stattfinden sollte.

```
func readFile(filename string) string {
  file, err := os.Open(filename)
  if err != nil {
    return ""
  }
  defer file.Close()
  ...
  // Datei laden und Inhalt zurueckgeben
  if ... {
    // Fehler!
    return ""
  }
  ...
}
```

Ein Vorteil dieser verzögerten Ausführung ist auch, daß das `file.Close()` strukturell in der Nähe des Datei-Öffnens gehalten werden kann, was die Wartbarkeit des Codes erhöhen kann.

Wenn mehrere **`defer`**-Anweisungen ausgeführt werden, werden sie in der umgekehrten Reihenfolge ausgeführt, in der sie registriert wurden:

```
func f() {
  defer func() {fmt.Println("zuletzt")}
  defer func() {fmt.Println("zuerst")}
  return
}
```

10 Fehlerbehandlung

Go unterstützt keine Ausnahmen. Stattdessen werden Fehler explizit zurückgegeben oder lokal behandelt. In einigen Fällen ist aber keine von beiden Optionen angemessen– die Funktion kann lokal mit dem Fehler nicht umgehen, und der Fehler ist so kritisch, daß nicht zu erwarten ist, daß das Programm sich davon erholen kann.

In diesem Fall kann die Funktion `panic` verwendet werden, um die Programmausführung mit einer Fehlermeldung abbrechen:

```
func divide(divident, divisor int) int {
  if divisor == 0 {
    panic("Division durch 0!")
  }
  return divident/divisor
}
```

Panic nimmt einen beliebigen Parameter und wickelt den Ausführungsstapel ab, bevor es den Parameter als Teil einer Fehlermeldung ausgibt. Beispielsweise können wir so das vorherige Beispiel des Lesens aus einer Datei mit einer einfachen Fehlerbehandlung versehen:

```
func readFile(filename string) string {
  file, err := os.Open(filename)
  if err != nil {
    panic(err)
  }
  defer file.Close()
  ...
}
```

Mit **`defer`** verzögerte Operationen werden weiterhin ausgeführt– **`panic`** wickelt diese nacheinander ab, und sie können z.B. zum Sichern von kritischen Daten verwendet werden.

Solche verzögerte Methoden können das Abwickeln der Fehlerbehandlung optional auch unterbrechen, indem Sie die Operation **`recover`** ausführen. Diese Operation liefert **`nil`** zurück, wenn kein Fehler vorliegt, und ansonsten das an `panic` übergebene Fehlerobjekt. So können Sie sich z.B. von dem `panic` in `readFile` wie folgt 'erholen':

```
func loadAllFiles(filenames []string) (result string) {
  defer func() {
    if err := recover(); err != nil {
      fmt.Println("Fehler beim Lesen!");
      result = ""
    }
  }
  for fn := range filenames {
    result += readFile(fn)
```

```
  }
  return
}
```

11 Das Paketsystem

Go verwendet ein Paketsystem, das die Übersetzung von Programmen und die Installation von externen Paketen stark vereinfacht.

Dazu setzt das Programm `go` auf der Umgebungsvariable `GOPATH` auf, die auf ein Verzeichnis zeigen muß, das als Depot für alle *Go*-Programme dient. Beispielsweise können Sie dazu ein Verzeichnis `goprojects` in Ihrem Heimatverzeichnis verwenden.

Unter UNIX (Bourne shell):

```
export GOPATH=${HOME}/goprojects
mkdir ${GOPATH}   # stellt sicher, daß das Verzeichnis existiert
```

Unter Windows:

```
SET GOPATH=C:\Go
mkdir C:\Go   # stellt sicher, daß das Verzeichnis existiert
```

(Sie können natürlich auch ein anderes Verzeichnis verwenden.)

Innerhalb dieses Verzeichnisses legt das `go`-Programm nach und nach folgende Unterverzeichnisse an:

```
${GOPATH} --+-- bin
            +-- pkg
            `-- src
```

Die Bedeutungen sind wie folgt:

- `bin` beinhaltet ausführbare (übersetzte, kompilierte) Programme (s.u.).
- `pkg` beinhaltet Pakete, die zu Bibliotheken in Maschinensprache übersetzt wurden.
- `src` beinhaltet Quellcode sowohl von externen als auch von selbstgeschriebenen Paketen.

11.1 Installation externer Pakete

Sie können per `go get` externe Pakete installieren, die in bestimmten Revisionskontrollsystemen untergebracht sind. Zum Beispiel gibt es ein offizielles Paket `github.com/golang/example/hello`, das Sie wie folgt installieren können:

```
go get github.com/golang/example/hello
```

Dieses Paket hängt von einem weiteren Paket ab, das implizit mit installiert wird. Sie müssen sich also nicht selbst um das Auflösen von Paketabhängigkeiten kümmern.

Zur Installation ist es nötig, daß Sie das Programm `git`[5] installiert haben.

11.2 Paket-Übersetzung

Um ein Paket in ein direkt ausführbares Programm zu übersetzen, verwenden Sie `go install`. Zum Beispiel:

```
go install github.com/golang/example/hello
```

Dieser Befehl übersetzt das zuvor installierte Paket `github.com/golang/example/hello` und erzeugt ein Maschinenprogramm `${GOPATH}/bin/hello`, das Sie direkt ausführen (kopieren, weitergeben) können.

11.3 Eigene Pakete

Um ein eigenes Paket anzulegen (z.B. `paket`) suchen Sie sich zunächst einen eindeutigen Namen aus, beispielsweise

```
cs.uni-frankfurt.de/${USER} /paket
```

wobei `$USER` Ihr eigener Nutzername ist.

Erzeugen Sie im `src`-Verzeichnis die notwendigen Unterverzeichnisse `cs.uni-frankfurt.de/${USER}/paket` und legen Sie dann im `paket`-Unterverzeichnis Ihre `.go`-Dateien an. Wenn diese Dateien ausführbar sein sollten, müssen sie mit **`package main`** beginnen, ansonsten können Sie den Paketnamen nach Bedarf wählen.

Angenommen, `${USER}` ist `user`, dann können Sie von einem anderen Modul aus nun auf Ihr Paket wie folgt zugreifen:

```
import "cs.uni-frankfurt.de/user/paket"

func main() {
  paket.MeineFunktion()
}
```

11.4 Importierung und Sichtbarkeit zwischen Paketen

Zur Sichtbarkeit von Namen zwischen Paketen und Dateien gelten folgende Regeln:

- Alle Go-Dateien können alle globalen Definition (z.B. `len` oder `nil`) sehen.
- Alle Go-Dateien im gleichen Paket können ihre Definitionen gegenseitig sehen.

[5] `http://git-scm.com/`

- Alle Go-Dateien können Namen aus importierten Modulen sehen, wenn diese Namen mit einem Großbuchstaben beginnen.

Beim Importieren von Paketen wird der im `package` des Moduls angegebene Name zum Zugriff auf Modulinhalte verfügbar gemacht. Dies kann (z.B. bei Namenskonflikten) übergangen werden, indem das Programm einen gewünschten Namen direkt vor dem Importierpfad angibt:

```
package main

import f "fmt"

func main() {
  f.Println("Test")
}
```

Um mehrere Pakete gleichzeitig zu importieren, können Sie auch folgende Kurzschreibweise verwenden:

```
import (
          f "fmt"
          "math/rand"
          "cs.uni-frankfurt.de/user/paket"
       )
```

11.5 Paket-Initialisieurng

Einige Pakete müssen Initialisierung durchführen, bevor sie verwendet werden können. Beispiele dafür sind Plug-ins, die sich in einem zentralen System registrieren müssen, oder konfigurierbare Komponenten, die externe Konfigurationsdateien lesen. Um dies zu ermöglichen, erlauben Pakete sogenannte `init`-Funktionen:

```
func init() {  // keine Parameter, keine Rueckgabewerte
    fmt.Println("Initialisiert");
}
```

Jedes Paket kann beliebig viele `init`-Funktionen haben. Diese werden garantiert vor der `main`-Funktion ausgeführt. Die Funktionen werden zudem sequentiell (also nicht nebenläufig) gestartet.

11.6 Test-Mechanismen

`go` hat direkte Unterstützung für Unit Testing. Unit Tests werden in Dateien mit der Endung `_test.go` abgelegt, z.B. `paket_test.go`. Um auf Interna eines Paketes zugreifen zu können, ist es oft sinnvoll, Unit Tests zu Teilen des Pakets zu machen; sie werden nicht in den gebauten Bibliothekspaketen integriert und benötigen somit keinen Platz in 'normalen' ausführbaren Programmen.

Solche Unit Tests müssen Funktionen definieren, die mit `Test` beginnen, keine Rückgabe haben, und einen Parameter vom Typ `*testing.T` nehmen (aus dem Paket `"testing"`). Der Parameter stellt mehrere nützliche Testoperationen zur Verfügung[6], insbesondere `Fatal()`, eine Funktion mit beliebig vielen Parametern, die eine Fehlermeldung ausgibt und den Test als fehlgeschlagen abbricht:

```
package paket

import "testing"

func TestF(t *testing.T) {
    t.Fatal("Fehler: Test ist noch nicht implementiert:", -1)
}
```

Um diesen Test und alle andere Tests im gleichen Paket auszuführen, verwenden Sie `go test`:

```
go test cs.uni-frankfurt.de/${USER}/paket
```

11.7 Benchmarking-Mechanismen

Go verfügt ebenfalls über Unterstützung zum automatischen Benchmarking. Das Benchmarking-System setzt auf dem Unit Test-System auf und erwartet Funktionen mit folgenden Eigenschaften:

- Der Funktionsname muß mit `Benchmark` beginnen
- Die Funktion muß genau einen Parameter `b *testing.B` nehmen (der Parametername ist natürlich beliebig)
- Die Funktion muß die zu messende Berechnung `b.N` mal durchführen.

Der betreffende Code sollte wie folgt aussehen:

```
func BenchmarkF(b *testing.B) {
    for i := 0; i < b.N; i++ {
            // Berechnung
    }
}
```

Die Wiederholung ist nötig, um bei Berechnungen, die sehr schnell gehen, genug Laufzeiten zu erzeugen, um hinreichend genaue Aussagen über die Laufzeit treffen zu können.

Um die Benchmarks durchzuführen, muß beim Testen zusätzlich der Parameter `-bench=`***x*** angegeben werden. Dabei ist ***x*** ein regulärer Ausdruck, der die auszuführenden Benchmark-Namen beschreibt; wir können hier '`.`' einsetzen, um alle Benchmarks auszuführen:

```
go test -bench=.
```

[6] `http://golang.org/pkg/testing/`

11.8 Formatierung und Code-Stil

Go definiert ein normalisiertes Formatierungsformat; Leerzeilen dürfen z.B. keine Leerzeichen behinhalten, und nutzlose Kommas müssen entfernt werden. Um diese Normalisierung vereinfacht umsetzen zu können, kann das Werkzeug `go fmt` eingesetzt werden. `go fmt f.go` formatiert `f.go` gemäß normalisierter Formatierung automatisch um.

In EMACS unterstützt `go-mode` diese Operation direkt (`M-x gofmt`).

Dieses Werkzeug erzwingt allerdings nicht alle Go-Stilrichtlinien. Insbesondere die folgenden beiden Stil

```
var meinLangerName int // CamelCase, um Namen (Variablen,
                       // Typen) aus mehreren Worten zu bauen
if error {
  return
}  // kein 'else', wenn 'if' mit 'return' endet
fmt.Println("Kein Fehler")
```