Einführung in die Programmierung

Weiterführende Themen

1. Slices

Arrays in Go sind nicht als Zeiger implementiert, sondern als Variablen, die mehrere Elemente enthalten. Deswegen arbeitet man mit dem ganzen Array, wenn man auch Operationen nur mit einzelnen Elementen durchführt. Daher ist die Arbeit mit Arrays in Go nicht effektiv und zeitlich aufwendig.

In Go werden öfter Slices statt Arrays eingesetzt, weil sie diese Nachteile nicht haben, und liefern außerdem sehr flexible Funktionalitäten mit.

```
func main() {
    // Array zur Erinnerung
    var a [2] string
    a[0] = "Arnold"
    a[1] = "Schwarzenecker"
    fmt.Println(a[0], a[1]); fmt.Println(a)
    // Slices
    var tcpports [] int                          // Deklaration
    tcpports = []int { 21, 22, 42, 53, 80, 135 }  // Zuweisung
    fmt.Println(tcpports)
    ausschnitt := [] int {}     // Leeres Slice
    ausschnitt = tcpports [2:5] // Immer: links inklusiv, rechts exklusiv
    fmt.Println(ausschnitt)     // Elemente 2,3,4
    fmt.Println("tcpports", len(tcpports), cap(tcpports) )       // Länge, Kapazität
    fmt.Println("ausschnitt", len(ausschnitt), cap(ausschnitt) )
    ausschnitt = tcpports [:6]  // Ab 0 inklusiv bis 6 exklusiv
    fmt.Println(ausschnitt)     // Elemente 0,1,2,3,4,5
    nix := make( [] byte, 7)    // Standard-Funktion generiert alles (oder fast alles)
    fmt.Println("nix", nix)
}
```

<u>Aufgabe 1.</u> Experimentieren Sie wie im oberen Beispiel mit anderen Slice-Operationen, einzelne Elementen bearbeiten, ein Slice vergrößern.

<u>Aufgabe 2.</u> Recherchieren Sie, welche Operationen mit Slices noch möglich sind (append, copy).

```
func ShowSlice(x []int) {  // Parameterübergabe
    fmt.Printf("\nSlice %v, len=%d, cap=%d.", x, len(x), cap(x))
}

func main() {
    tcpports := []int { 21, 22, 42, 53, 80, 135 } // Zuweisung
    ShowSlice(tcpports)
    ausschnitt := tcpports [2:5]
    ShowSlice(ausschnitt)
}
```

2. Kanäle

Kanäle stellen zusätzliche Kommunikationsmittel zwischen den Funktionen dar.

```
func main() {
    k := make(chan int, 12)  // Kanal mit Buffer 12 Elemente
    for i := 0; i <= 7; i++ {
        k <- i               // Zahlen in den Kanal platzieren
    }
    for i := 0; i <= 7; i++ {
        j := <- k            // Zahlen aus dem Kanal auslesen
        fmt.Println ("empfangen:", j)
    }
    fmt.Println("....")
}
```

Dieser Ansatz bringt aber keine besonderen Vorteile im Vergleich zu Parametern. Viel mehr werden die Kanäle in parallel laufenden Funktionen eingesetzt, wo Parameterübergabe versagt (nicht immer möglich ist).

Aufgabe 3. Recherchieren Sie nach den Arten von Kanälen — gepuffert und nicht gepuffert und schreiben Sie Test-programme.

3. Parallelität

Die Sprache Go definiert einen sehr einfachen und effektiven Mechanismus, um Funktionen parallel (gleichzeitig) auszuführen. Solche Funktionen, die parallel arbeiten, heißen Go-Routinen (Go-Funktionen), und sie werden mit dem Schlüsselwort *go* gestartet.

```
func main() {
    for i := 0; i <= 10; i++ {
        go paral(i)  // paral() wird der Reihe nach gestartet
    }
    time.Sleep(time.Second*1)
    fmt.Println("Ende")
}

func paral (i int) {  // Die Ergebnisse kommen nicht unbedingt der Reihe nach
        fmt.Println ("paral: ", i)
}
```

Die Go-Routinen verlieren Verbindung zu dem aufrufenden Programm und werden als einzelne Tasks von dem Betriebssystem betrachtet. Die von Go-Routinen berechneten Ergebnisse müssen aber im aufrufenden Programm irgendwie landen. Dafür passen die Kanäle hervorragend.

```
func main() {
    k := make(chan int, 7)  // Definition des Kanals mit Puffer für 7 Zahlen
    // Parallele Ausführung der Funktionen empfangen1() und empfangen2()
    go empfangen2(k); go empfangen1(k); go empfangen2(k); go empfangen1(k)
    go empfangen1(k); go empfangen2(k); go empfangen1(k); go empfangen2(k)
    go empfangen2(k)  // Reihenfolge der Funktionen ist egal
    go senden(k)      // Zahlen 0,1,2,3 werden in den Kanal gesendet
    fmt.Println("Ende")
    time.Sleep(time.Second*2)
}

func senden (c chan int) {      // Zahlen 0,1,2,3 werden in den Kanal gesendet
    for i := 0; i < 4; i++ { c <- i }
    close(c)
}

func empfangen1 (c chan int) {
    j := <-c                        // Nur eine Zahl wird aus dem Kanal empfangen
    fmt.Println ("empfangen1: ", j)  // und auf dem Bildschirm gezeigt
}

func empfangen2 (c chan int) {
    j := <-c                        // Nur eine Zahl wird aus dem Kanal empfangen
    fmt.Println ("empfangen2: ", j)  // und auf dem Bildschirm gezeigt
}
```

Dieses Programm kann unterschiedliche Ergebnisse bringen, wenn man es mehrmals startet (und das ist richtig!).

Aufgabe 4. Ändern Sie die Reihenfolge der Funktionen suchen(), empfangen1() und empfangen2() im obigen Beispiel. Hat es Einfluss auf Ergebnisse? Was passiert, wenn die Funktion senden() mehr Zahlen in den Kanal sendet, als Puffer? Ändern Sie die Funktionen empfangen1() und empfangen2() so, dass sie mehr als eine Zahl aus dem Kanal lesen können.

```
// Das Programm sucht in einer Zeichenkette nach zwei Mustern gleichzeitig

func main() {
    c1 := make(chan int)    // Nicht gepufferte Kanäle
    c2 := make(chan int)
    z := "abc xyz fgh jkl"  // Zu untersuchende Zeichenkette
    go er(c1, z, "abd")     // Muster 1
    go er(c2, z, "jkl")     // Muster 2
    <- c1               // Etwas wird aus dem Kanal empfangen, es ist unwichtig, was.
    <- c2               // Eigentlich ist es Prüfung, ob die Go-Funktion zu Ende ist.
}

func er(c chan int, s string, p string) {  // p ist in s zu suchen
    if strings.Contains(s,p) {             // Ist p in s ?
        fmt.Println("      gefunden...", p, " in ", s)
    } else {
        fmt.Println("nicht gefunden...", p, " in ", s)
    }
    c <- 0  // Markierung des Endes der Funktion – irgendwas in den Kanal senden
}
```

```
// Das Programm berechnet die Kreiszahl nach der Leibniz-Reihe

func main() {
    fmt.Println(math.Pi)     //  Richtige Annäherung der Zahl Pi aus dem Paket math.
    fmt.Println(pi(600000))  //  Anzahl der Terme. 5 richtige Nachkommastellen.
}

func pi(n int) float64 {
    ch := make(chan float64, n)
    for k := 0; k < n; k++ {  // Alle n Werte parallel berechnen
        go term(ch, k)        // Aufruf der Go-Routine
    }
    f := float64(0.0)     // Die Ergebnis-Variable mit Null initialisieren
    for k := 0; k < n; k++ {    // alle n Werte addieren
        f += <-ch
    }
    return f
}

func term(ch chan<- float64, k int) {  // Go-Routine berechnet einen Term der Reihe
    p := float64(k)        // Typkonvertierung für math.Pow()
    ch <- 4 * math.Pow(-1, p) / (2*p + 1)  // Ein Term wird berechnet
}
```

Aufgabe 5. Testen Sie das obere Programm für die Anzahl der Terme 900000000 (neun hundert Millionen?) und merken Sie, wie lange es dauert. Sie bekommen dabei nur 8 richtige Nachkommastellen der Kreiszahl. Ändern Sie das Programm folgenderweise. Die Funktion pi() muss die Funktion term100() statt term() aufrufen, wobei die Funktion term100() für die Summe aller 100 Terme zuständig sein soll. Die Funktion term100() ruft auf ähnliche Weise die Go-Routine term(), die eigentlich nicht geändert wird. Dadurch entsteht im Arbeitsspeicher eine baumartige Struktur aus Go-Funktionen. Kommt es schneller zum Ergebnis?

Aufgabe 6. Schreiben Sie ein Programm, das die eulersche Zahl e berechnet. Finden Sie entsprechende mathematische Formel, schreiben Sie die Go-Routine, die einen Term berechnet, und testen Sie sie im Hauptprogramm, ähnlich wie im vorigen Beispiel.

Aufgabe 7. Recherchieren Sie, welche Probleme beinhalten solche parallele Zweige.

Die Sprache Pascal bietet ebenfalls eine Möglichkeit für parallele Ausführung der Modulen. Dafür gibt es extra Prozedur *BeginThread()*, die mehrere Funktionen parallel startet. Sie nimmt als Parameter u.a. die Adresse der zu startenden Funktion (erster Parameter) und die Adresse der Parameter dieser Funktion (zweiter Parameter). Die Funktion, die von *BeginThread()* gestartet wird, kann mit dem aufrufenden Programm und mit anderen parallelen Funktionen durch globale Variablen kommunizieren. Das aufrufende Programm darf nicht enden, solange alle parallelen Funktionen laufen, sonst gehen die Ergebnisse der parallelen Funktionen verloren. Dieser Warteprozess kann sehr einfach mit einer WHILE-Schleife organisiert werden.

Pascal stellt atomare Funktionen zur Verfügung, damit die parallelen Funktionen thread-sicher an globalen Variablen arbeiten können. Wenn eine atomare Funktion/Operation an einer globalen Variable arbeitet, dann wird es gewährleistet, dass keine andere Operation diese Variable ändert, bis die erste Operation sie nicht frei gibt. Das sind z.B. die Funktionen *InterLockedIncrement()*, *InterlockedDecrement()*, *InterlockedExchange()*.

Die Sprache Go stellt nicht nur das Paket *sync/atomic* mit vielen atomaren Funktionen zur Verfügung, sondern liefert den Mutex-Mechanismus, der solche Funktionalitäten gewährleistet.

Betrachten Sie das folgende Pascal-Beispiel und machen Sie sich klar, wie es funktioniert.

```
uses crt, sysutils;

VAR CounterThreads: longint = 0;  // Anzahl von Threads
           // Variable zur Kommunikation zwischen Funktion und Hauptprogramm

FUNCTION Paral(i: pointer): ptrint;  // Parameter müssen Zeiger sein
BEGIN
    WRITELN('Paral ', integer(i^));
    InterLockedIncrement(CounterThreads);  //  Atomare Increment-Funktion
    Paral := 0;  // RETURN
END;

(**)
VAR j: INTEGER;
BEGIN
    clrscr;
    FOR j:=1 TO 5 DO
    BEGIN
        BeginThread(@Paral, @j);
        sleep(500);
    END;
    while CounterThreads<5 do;  //  Warten bis alle Threads beenden
    WRITELN('Ende');
END.
(**)

(*
Wenn Sleep() ausgeschaltet ist, dann erklären Sie die Ausgabe:
Paral 5
Paral 5
Paral 5
Paral 5
Paral 5
Ende

Wenn Sleep() eingeschaltet ist, dann hat der Code wenig Sinn wegen Sleep(),
obwohl funktioniert richtig:
Paral 1
Paral 2
Paral 3
Paral 4
Paral 5
Ende
*)









// Deswegen testen Sie den unteren Code, indem Sie ihn MEHRMALS starten:
```

```
(*
VAR j, k, l, m, n: INTEGER;
BEGIN
    clrscr;  // Ohne Schleife !
    j:=1; BeginThread(@Paral, @j);
    k:=2; BeginThread(@Paral, @k);
    l:=3; BeginThread(@Paral, @l);
    m:=4; BeginThread(@Paral, @m);
    n:=5; BeginThread(@Paral, @n);
    while CounterThreads<5 do;  //  Warten bis alle Threads beenden
    WRITELN('Ende');
END.
*)

(*
Die Ergebnisse sehen mal so aus:
Paral 1
Paral 2
Paral 3
Paral 4
Paral 5
Ende
... mal so:
Paral 2
Paral 3
Paral 4
Paral 1
Paral 5
Ende
... mal so:
Paral 1
Paral 3
Paral 2
Paral 5
Paral 4
Ende
*)
```

4. Rekursivität

Rekursivität ist eine sehr nützliche Eigenschaft, die von vielen Programmiersprachen unterstützt wird. Sie stellt die Möglichkeit dar, eine Funktion im Körper der selben Funktion aufzurufen:

```
func recurs (i int) {
    . . .
    recurs (42)     //  Aufruf von sich selbst
    . . .
}
```

Das kann nur dann funktionieren, wenn entsprechender Compiler die Befehls- und Daten-Segmente unterscheidet, und nicht alles (Befehle und Daten) in einem Segment platziert. Außerdem muss das Betriebssystem solche rekursive Aufrufe behandeln können.

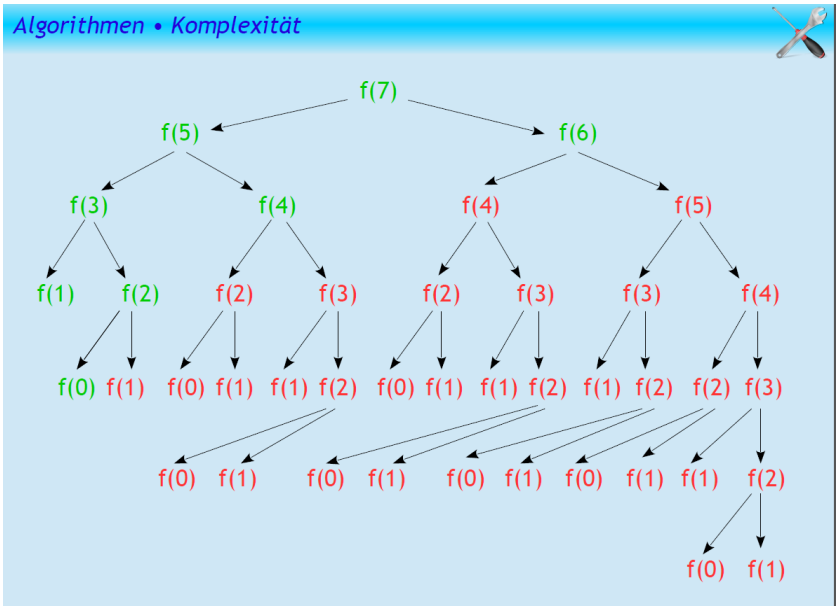
Fast alle modernen Betriebssysteme und Compiler besitzen diese Fähigkeiten.

Rekursivität darf man nicht ohne gute Überlegung einsetzen, da Rekursivität einen bestimmten Zeit- und Speicher-aufwand mitbringt. Muss man z.B. das Hauptprogramm mehrmals wiederholen, dann ist dies ist sehr bedenklich:

```
func main() {
    . . .
    . . .
    main()     //  Wiederholung
}
```

Für solche Zwecke gibt es eigentlich die Schleifen.

Man soll die Rekursivität nur dann verwenden, wenn sie in der Natur des Algorithmus liegt, und trotzdem muss man auch gut überlegen, weil die Realisierung des rekursiven Algorithmus für Berechnung der Fibonacci-Zahlen zur unnötigen Speicher- und Prozessor-Belegung führt:



<u>Aufgabe 8.</u> Die folgenden Beispiele realisieren die Berechnung der Fibonacci-Zahlen und sind in der Sprache C geschrieben. Schreiben Sie diese Programme in Go um.

```
// Definition der Fibonacci-Funktion für n≥0
//   Fib(0) = 1
//   Fib(1) = 1
//   Fib(n) = Fib(n-1) + Fib(n-2) fur n>1

// Programmiersprache C
// Definitionen/Deklarationen sind nicht vollständig
```

```
int Fib_1 (int n)         // Die n-te Fibonacci-Zahl
{
    if ( n==0 || n==1 )   // Fib(0) oder Fib(1)
        return 1;
    else       // Zwei rekursive Aufrufe
        return Fib_1(n-1) + Fib_1(n-2);  //  Als Summe von zwei vorherigen Zahlen
}

void main(void)
{
    printf ("\n%d", Fib_1(8));  // n=8
}
```

```
// Schon bei n=40 ist die Verzögerung spürbar.
```

```
fibo_zahl Fib_2 (int n)             // Globale Variable ret muss als eine Struktur
{                                   // definiert werden,
    unsigned long int x;            // die zwei int-Komponenten act und prev hat,
    if ( n==0 )                     // fibo_zahl ist Name dieser Struktur.
    {
        ret.act = ret.prev = 0;
        return ret;
    }
    if ( n==1 )
    {
        ret.act = 1;
        ret.prev = 0;
        return ret;
    }     // Nur ein rekursiver Aufruf
    ret = Fib_2(n-1);               // Die Zahl Fib_2(n-2) wird nicht berechnet,
    x = ret.act;                    // sondern aus ret.prev genommen.
    ret.act += ret.prev;
    ret.prev = x;
    return ret;
}

// Bei n=40 gibt es keine Verzögerung.
// Deutliche Verbesserung der Rechenzeit gegenüber von Fib_1(n).
```

```
unsigned long int Fib_3 (int n)
{
    if ( n==0 || n==1 )
        return n;
    if ( n==2 )
        return 1;
    x2 = 0;
    x1 = 1;
    x = 1;
    for (j=3; j<=n; j++)    // Hier gibt es gar keine Rekursivität,
    {                       // sie ist durch diese Schleife ersetzt
        x2 = x1;
        x1 = x;
        x = x1 + x2;
    }
    return x;
}

// Bei n=40 gibt es keine Verzögerung.
// Zusätzliche Verbesserung der Laufzeit durch Entfallen von Stack-Verarbeitung.
```

5. Aufschieben der Funktionsaufrufe

Funktionsaufrufe können am Ende der aufrufenden Funktion ausgeführt werden.

```
func main() {
    defer fmt.Println("Das war es")  // Schlüsselwort defer steht für Aufschieben
    fmt.Println("Start")
}
```

Aufgabe 9. Platzieren Sie mehrere aufgeschobene Aufrufe in dieser Funktion und klären Sie, in welcher Reihenfolge werden Sie ausgeführt.

Die aufgeschobenen Funktionsaufrufe sind dann nützlich, wenn einige Aktionen (z.B. Öffnen der Dateien am Anfang des Programms) wegen Fehler im normalen konventionellen Verlauf nicht beendet werden können (z.B. Schließen der Dateien am Ende des Programms).

Aufgabe 10. Im unteren Beispiel ist eine ewige Schleife einprogrammiert und die Ausführung dieses Programms kann nur durch Strg+C unterbrochen werden. Arbeitet in diesem Fall der aufgeschobene Aufruf?

```
func main() {
    defer fmt.Println("Aufgeschobene Funktion - das war es")
    fmt.Println("Start")
    for {
        fmt.Println("Ewige Schleife...")
    }
    fmt.Println("Die Zeile wird nie angezeigt")
}
```

6. Ordnerstruktur in Go

Die Sprache Go impliziert das Vorhandensein einer fest definierten Ordnerstruktur (Go Workspace), wo Go den Quellcode, ausführbaren Code und Pakete erwartet. Die Umgebungsvariable GOPATH verweist auf diese Struktur.

Das Go Workspace ist in einer produktiven Umgebung (in einem Unternehmen) sehr wichtig, weil der Befehl *go* eigentlich nur ein Verwaltungstool für Go-Programme ist und viele untergeordnete Befehle hat, wie z.B.

go run go build go test go doc

Diese Befehle suchen oder platzieren ihre Dateien in bestimmtem Unterverzeichnissen von Go Workspace. Die komplizierten Go-Projekte basieren auf jeden Fall auf solchen Verzeichnisstrukturen, wo Standard-Pakete sehr breiten Einsatz finden. Dazu kommen noch Anwendungen für Versionsverwaltung, was die Verzeichnisstruktur auch verkompliziert.

Die Struktur des Go Workspace ist nicht unbedingt gerade einfach, deswegen im Unterricht auf sie verzichtet wurde.

Aufgabe 11. Machen Sie sich mit dem empfohlenen Go Workspace vertraut, z.B. schauen Sie hier:
https://golang.org/doc/gopath_code.html
Welche Umgebungsvariablen außer GOPATH unterstützt GO?

Aufgabe 12. Machen Sie sich mit Programmen für Versionsverwaltung ansatzweise vertraut, z.B. github.