Einführung in die Programmierung

Funktionen und Prozeduren

1. Zweck

Funktion oder Prozedur stellt einen Code dar, der innerhalb des Programms mehrmals aufgerufen (eingesetzt) wird. Dadurch wird auch Lesbarkeit und Verständnis des Programms erhöht.

Man versieht diesen Code mit einem Namen, unter dem dieser Code im Programm bekannt ist. Damit dieselbe Funktion oder Prozedur bei unterschiedlichen Aufrufen auch unterschiedliche Ergebnisse liefen kann, übergibt man an sie die Parameter. Parameter sind Platzhalter für Variablen oder Konstanten, die beim Aufruf konkrete Werte bekommen. Funktion oder Prozedur kann die übergebenen Parameter analysieren und entsprechende Aktionen ausführen. Die Ergebnisse können dem aufrufenden Programm auf zwei Arten zurück geliefert werden:

- Als Rückgabewert in einem Register des Prozessors — schnellere Variante, aber nur ein Wert kann damit zurückgegeben werden. Solche Codes heißen Funktionen.

- Als Call-by-Reference-Parameter durch Stack im Arbeitsspeicher — langsamere Variante, aber mehrere Werte können damit zurückgegeben werden. Solche Codes heißen Prozeduren.

Die klassische Sprache Pascal behält immer noch diese Unterschiede, die moderne Sprache Go nicht mehr, da deren Funktionen mehrere Werte zurückgeben können. Moderne Hardware ist aber so schnell geworden, dass der zeitliche Unterschied zwischen Register- und Arbeitsspeicher-Übergabe kaum zu erkennen ist.

Es geht hier um die Funktonen und Prozeduren, die in der selben Datei geschrieben sind, wie das Hauptprogramm.

2. Klassische Funktionen und Prozeduren

```
PROGRAM Mein1Programm;

FUNCTION Zahl_PI : REAL;  // Typ vom Rückgabewert ist obligatorisch
BEGIN
  Zahl_PI := 3.14159265359;  // Rückgabewert, es gibt kein RETURN
END;  // Ende der Funktion

BEGIN             // Die Reihenfolge ist wichtig: Zuerst Codes von allen Funktionen,
  Writeln ('PI = ', Zahl_PI);  // dann Code vom Hauptprogramm
END.
```

```
func Zahl_PI () float64 { // () und Typ vom Rückgabewert sind obligatorisch
  return 3.14159265359  // Rückgabewert
}  // Ende der Funktion

// Die Reihenfolge ist nicht wichtig: Ob zuerst Codes von allen Funktionen,
// und dann Code vom Hauptprogramm, oder umgekehrt
func main() {
  fmt.Println("PI = ", Zahl_PI())  // () nach der Funktion sind obligatorisch
}  // Genauigkeit dieser Ausgabe ist von float64 oder float32 abhängig – testen Sie!
```

```
PROGRAM Mein1Programm;

FUNCTION Zahl_PI (n:INTEGER; v:INTEGER) : REAL;  // Parameter mit ; getrennt
BEGIN
  Zahl_PI := 3.14159265359;
  Writeln ('PI = ', Zahl_PI:n:v);
END;

VAR Z : REAL;
BEGIN
  Z := Zahl_PI(6, 3);  // Zuweisung bei Funktionen ist nicht obligatorisch
  Writeln ('PI = ', Z);
END.
```

```
func Zahl_PI (n int, v int) float64 { // Parameter mit , getrennt. Rückgabewert
  PI := 3.14159265359                 // Paket "strconv" selbständig inkludieren
  fmt.Printf ("\nPI = %" + strconv.Itoa(n) + "." + strconv.Itoa(v) + "f", PI) // %6.3f
  return PI    // Funktion Itoa() konvertiert ganze Zahl in String, z.B. 12 -> "12"
}

func main() {
  Z := Zahl_PI(6, 3)  // Zuweisung bei Funktionen ist nicht obligatorisch
  fmt.Println ("\nPI = ", Z)
}
```

```
PROGRAM Mein1Programm;

PROCEDURE Summe (p1:INTEGER; p2:INTEGER);
BEGIN
  Writeln ('Summe von ', p1, ' und ', p2, ' = ', p1+p2);
END;

BEGIN
  Summe(3,4); // Zuweisung, z.B. S:=Summe(3,4); oder
END.          // Verwendung in Writeln() ist nicht zulässig
```

```
func Summe (p1 int, p2 int) { // Procedure-Analog in Pascal, da kein return steht
  fmt.Println ("\nSumme von ", p1, " und ", p2, " = ", p1+p2)
}

func main() { // Zuweisung, z.B. S:=Summe(3,4) oder
  Summe(3,4)  // Verwendung in fmt.Println() ist nicht zulässig
}
```

Rekursive Funktionen sind in beiden Sprachen möglich.

```
PROGRAM Mein1Programm;

FUNCTION faktorial(zahl: integer) : integer;  // Berechnung von Fakultät
BEGIN
  IF zahl = 1 THEN
    faktorial := 1  // end of recursion
  ELSE
    faktorial := faktorial(zahl-1) * zahl;
END;

VAR z : integer = 7;
BEGIN
  Writeln(z, '! = ', faktorial(z))  // Aufruf
END.
```

```
func faktorial(zahl int) int {  // Berechnung von Fakultät
  if zahl == 1 {
    return 1  // end of recursion
  } else {
    return faktorial(zahl-1) * zahl
  }
}

func main() {
  z := 7; fmt.Println(z, "\b! =", faktorial(z))  // Aufruf
}
```

Aufgabe 1. Schreiben Sie ein Programm in Pascal und in Go, in dem Sie den größten gemeinsamen Teiler (ggT) von zwei ganzen Zahlen berechnen. Die Berechnung passiert mit Hilfe einer Funktion, die zwei Parameter erwartet und einen Wert zurückgibt. Im Hauptprogramm wird die Funktion getestet. Zum Testen werden erstmal nur die positiven ganzen Zahlen eingesetzt. Nehmen Sie den Algorithmus aus Vorlesungen.

Aufgabe 2. Erweitern Sie die vorige Aufgabe. Ist mindestens eine von beiden Zahle nicht positiv, arbeitet der Algorithmus unendlich lange. Ihre Funktion muss zuerst überprüfen, ob beide Zahlen positiv sind, und wenn nicht, gibt sie eine Null zurück. Im Hauptprogramm wird die Funktion getestet.

Aufgabe 3. Schreiben Sie ein Programm in Pascal und in Go, in dem Sie den Zeller-Algorithmus für Berechnung des Wochentages als Funktion implementieren. Die Zeller-Funktion wird im Hauptprogramm getestet. Eingabe (Parameter) für Funktion sind die ganzen Zahlen: Tag, Monat, Jahr. Rückgabe der Funktion ist die Nummer des Wochentages (1 — Montag, 2 — Dienstag, ..., 7 — Sonntag).

Aufgabe 4. Implementieren Sie Plausibilität in der vorigen Aufgabe.

3. Moderne Funktionen und Prozeduren

Moderne Programmiersprachen bringen neue Konzepte von Funktionen und Prozeduren. Die Sprache Go kennt Funktionen, die mehrere Werte zurückgeben können, Funktionen ohne Namen (anonyme Funktionen), go-Routinen, die sehr effektiv parallel ausgeführt werden. Diese Sprache verzichtet gänzlich auf Prozeduren (wie auch C/C++/Java).

```
func swap(x, y string) (string, string) {
  return y, x  // Die Funktion tauscht die Werte in zwei Variablen aus
}              // und gibt ZWEI Werte zurück

func main() {
  a, b := "Harry", "Sally"     // Neue Zuweisungsmöglichkeit
  fmt.Println(a, " und ", b)
  a, b = swap(a, b)            // Für diese Funktion muss die Zuweisung so aussehen!
  fmt.Println(a, " und ", b)   // Versuchen Sie mit a = swap(a, b)
  a, _ = swap(a, b)            // Bedeutung von _ :
  fmt.Println(a, " und ", b)   // zweiter Rückgabewert wird weiter nicht benötigt
}

// Dasselbe kann man erreichen, ohne Rückgabewerte in return aufzulisten:
func QuotRestBerechnen(divident, divisor int) (quot, rest int) {
  quot = divident / divisor
  rest = divident % divisor
  return  // bedeutet Rückgabe von quot und rest, die im Kopf beschrieben sind
}

func main() {
  fmt.Println(QuotRestBechnen(42,9))
}
```

Aufgabe 5. Verbessern Sie die vorige Aufgabe, indem die Funktion drei Werte zurück gibt: Nummer, kurze und lange Bezeichnung des Wochentages, z.B.: 4,"Do.", "Donnerstag". Die Funktion wird wie immer im Hauptprogramm getestet.

Funktionen in Go können als Variablen behandelt werden.

```
func main() {
  var op func(int, int) int  // Eine neue Variable op vom Typ func(int, int) int
  s := ""
  fmt.Println("... Operation eingeben: +,-,*,/ "); fmt.Scan(&s)
  switch s {
    case "+":
      op = add
    case "-":
      op = sub
  }
  fmt.Println(op(10,7))
}

func add(a, b int) int { return a+b }

func sub(a, b int) int { return a-b}
```

Funktionen in Go können zeitlich bis zum Ende der aufrufenden Funktion versetzt werden.

```
func main() {
  defer fmt.Println("... und Tschüsssss")
  fmt.Println("Aller Anfang ist leicht...")
}
```

Dasselbe in traditioneller Schreibweise.

```
func main() {
  s := ""
  fmt.Println("... Operation eingeben: +,-,*,/ "); fmt.Scan(&s)
  switch s {
    case "+":
      fmt.Println(add(10,7))
    case "-":
      fmt.Println(sub(10,7))
  }
}

func add(a, b int) int { return a+b }

func sub(a, b int) int { return a-b}
```

Funktionen als Werte der Variablen sind dann nützlich, wenn man sie an andere Funktionen als Parameter übergibt.

```
func main() {
  var op func(int, int) int
  s := ""
  fmt.Println("... Operation eingeben: +,-,*,/ "); fmt.Scan(&s)
  switch s {
    case "+":
      op = add
    case "-":
      op = sub
  }
  aus(op,10,7)  // Übergabe op als Parameter
}

func aus(op func(int, int) int, a int, b int) { fmt.Println(op(a,b)) }

func add(a, b int) int { return a+b }

func sub(a, b int) int { return a-b}
```

Anonyme Funktionen ist eine einzige Möglichkeit in Go, Funktionen innerhalb der Funktionen zu schreiben.

```
func main() {
  a := 1            // a ist überall in main() bekannt, wenn nicht überdeckt wird
  fmt.Println(a)    // 1
  func() {          // Kein Parameter wird erwartet             1. Anonyme Funktion
    fmt.Println(a)  // 1
    a = 7           // Das ist die vorige a, a ist hier bekannt
    fmt.Println(a)  // 7
  }()               // Kein Parameter wird übergeben
  fmt.Println(a)    // 7, Wert von a wurde zu 7 geändert
  func(a int) {     // Parameter wird erwartet, a ist formaler Parameter – besser x
    fmt.Println(a)  // 7 – besser x                             2. Anonyme Funktion
    a = 42          // a ist neu und überdeckt die globale a aus main() – besser x
    fmt.Println(a)  // 42 – besser x
  }(a)  // Parameter wird übergeben, a kommt aus main(), globale a, Parameter konkret
  fmt.Println(a)    // 7
}
```

In Pascal darf man Funktionen/Prozeduren innerhalb von anderen Funktionen/Prozeduren unbegrenzt platzieren (schreiben), aber es ist nicht empfehlenswert, da jeglicher Überblick schnell verloren geht.

```
PROGRAM Mein1Programm;

PROCEDURE aaus(s_extern: STRING);   // Prozedur ist für Hauptprogramm bekannt
  PROCEDURE aus(s_intern: STRING);  // Prozedur ist für Hauptprogramm nicht bekannt
  BEGIN  // Start von aus()
    Writeln('aus: ', s_intern);
  END;
BEGIN    // Start von aaus()
  aus('aaus: ' + s_extern);
END;

BEGIN
  aaus('xyz');  // Aber: Aufruf aus('xyz'); funktioniert nicht
END.

Ablauf:
  aus: aaus: xyz
```

4. Call-by-Value- und Call-by-Reference-Parameter

Es gibt zwei Arten für Parameterübergabe in eine Funktion/Prozedur:

- Call-by-Value — Parameter, die an eine Funktion/Prozedur übergeben wurde, können innerhalb der Funktion/Prozedur geändert werden, aber die geänderten Werte werden im aufrufenden Modul nicht bekannt. Parameter sind geschützt.

- Call-by-Reference-Parameter — Parameter, die an eine Funktion/Prozedur übergeben wurde, können innerhalb der Funktion/Prozedur geändert werden, und die geänderten Werte werden im aufrufenden Modul bekannt. Parameter sind nicht geschützt.

Standardmäßig (wenn man nichts besonderes unternimmt) werden die Call-by-Value-Parameter verwendet. Die Call-by-Reference-Parameter braucht man nur dann, wenn das aufrufende Programm die geänderten Werte weiter in eigenem Code verwenden muss.

In Go ist es kein Problem, da eine Funktion in Go mehrere Parameter zurückgeben kann. Außerdem steht der Zeiger-Mechanismus in Go zur Verfügung, der für diese Zwecke verwendet werden kann.

In Pascal ist es anders. Ist es nur ein Parameter, der zurückgegeben werden soll, dann stehen die Funktionen, wie üblich, zur Verfügung. In Prozeduren kommen ein spezieller Modifikator VAR in der Parameterliste zum Einsatz. Dieser Modifikator wird im Hintergrund selbstverständlich durch die Zeiger implementiert.

```
PROGRAM Mein1Programm;

PROCEDURE swap(VAR x, y: integer);  // Modifikator VAR in der Parameterliste
VAR temp: integer;
BEGIN
   temp := x;
   x:= y;
   y := temp;
END;

VAR a : integer = 7;
    b : integer = 42;
BEGIN
  Writeln('Vorher:  ', a, '  ', b);
  swap(a,b);
  Writeln('Nachher: ', a, '  ', b);
END.
```

Die Sprache Go beherrscht einen sehr ähnlichen Zeiger-Mechanismus wie C/C++.

```
func swap(x *int, y *int) {  //  x und y sind Adressen von int (Zeiger auf int)
  var temp int               //  * ist Referenz-Operator
  temp = *x                  //  *x und *y sind int-Zahlen,
  *x = *y                    //  die an den Adressen x und y stehen
  *y = temp
}

func main() {
  var a, b int = 7, 42
  fmt.Println("Vorher:  ", a, b)
  swap(&a, &b)      //  & ist Adress-Operator, er ermittelt Adresse der Variable a
  fmt.Println("Nachher: ", a, b)                            // im Arbeitsspeicher
}
```