Programmieren 1

ptlwedel

PRIVATE BERUFSFACHSCHULE

Wintersemester 2016/2017

Marcus Riemer, B.Sc.

Basierend auf den Unterlagen „Programmstrukturen 1“
von Prof. Dr. Andreas Häuslein

16 Aufgabe "Mengenoperationen"

Dieses Programm soll für zwei vom Benutzer einzugebende Mengen verschiedene Mengenoperationen durchführen.

- Legen Sie einen Typ `TZahlenmenge` an, dieser soll Werte zwischen 0 und 255 speichern können. Deklarieren Sie von diesem Typ zwei Variablen namens `mengeLinks` und `mengeRechts`.
- Fragen Sie vom Benutzer zwei Strings ab, in denen jeweils durch ein Leerzeichen getrennt die Elemente der rechten und linken Menge auftauchen.
- Geben Sie die linke und die rechte Menge aus.
- Geben Sie die Ergebnisse der folgenden Mengen-Operationen aus:

$a)\ links \cup rechts$

$b)\ links \cap rechts$

$c)\ links \setminus rechts$

$d)\ rechts \setminus links$

Beispiel:

```
M1? 5 2 7 22
M2? 5 22
Links = 2 5 7 22
Rechts = 5 22
a) 2 5 7 22
b) 5 22
c) 2 7
d)
```

16 Aufgabe "Mengenoperationen"

```
program set_operations;

type
   TZahlenMenge = set of byte;

var
   mengeLinks, mengeRechts : TZahlenMenge;
   userInput : string;

begin
   write('M1? ');
   readln(userInput);
   mengeLinks := readSet(userInput);

   write('M2? ');
   readln(userInput);
   mengeRechts := readSet(userInput);

   printSet('links', mengeLinks);
   printSet('rechts', mengeRechts);

   printSet('links  +  rechts', mengeLinks + mengeRechts);
   printSet('links  *  rechts', mengeLinks * mengeRechts);
   printSet('links  -  rechts', mengeLinks - mengeRechts);
   printSet('rechts -  links ', mengeRechts - mengeLinks);
   printSet('links sym rechts', (mengeLinks + mengeRechts) -
                                (mengeLinks * mengeRechts));
end.
```

14 Strukturierung von Programmen

- Wege aus der "Textwüste":
  - Logische (Grob-)Struktur im Programmtext deutlich werden lassen
    - Inhaltlich zusammenhängende Programmteile voneinander abgrenzen ("Funktionsblöcke")
    - Benennung der Programmteile, so dass vom Namen auf die Bedeutung/Wirkung der Programmteile geschlossen werden kann
    - Prinzip der Schrittweisen Verfeinerung auch im Programmtext anwendbar/erkennbar machen
  - Redundanzen im Programmtext weitgehend vermeiden

16 Block-Konzept

- Prinzip: Einteilung des Programms in getrennte Abschnitte (Blöcke)
- Jeder Block fasst Deklarationen und Anweisungen zusammen
- Jeder Block kann wiederum Blöcke enthalten (Hierarchisierung)
- Deklarationen eines Blockes (z.B. Variablendeklarationen) gelten nur innerhalb dieses Blockes (auch in den enthaltenen Blöcken)

Blockkonzept
in Pascal

(Block = Hauptprogramm oder Prozedur/Funktion)

16 Prinzip der Prozeduren und Funktionen

- Abgrenzung/Zusammenfassung von inhaltlich/logisch zusammengehörenden Programmteilen unter einem Namen (Prozedur-/ Funktionsbezeichner)
- Zusammengefasste Programmteile bestehen aus:
  - Deklarationen von Programmelementen, die nur in zusammengefassten Programmteilen benötigt werden (z.B. Variablen für Zwischenergebnisse) ⟶ *lokale* Deklarationen
  - Anweisungsfolgen
- Zusammengefasste Programmteile bilden somit einen Block

- Zuordnung zwischen dem Namen und den Programmteilen, die durch den Namen bezeichnet werden sollen, erfolgt in Form einer

Deklaration

16 Prinzip der Prozeduren und Funktionen

- Verwendung der Programmteile durch einen

Aufruf

- Ein Aufruf erfolgt durch Nennung des Namens des verwendenden Programmteils in der Anweisungsfolge des Programms
- Bei Auftreten des Namens im Programmtext während der Programmabarbeitung werden
  - gemäß den lokalen Deklarationen zusätzliche Programmelemente geschaffen, z.B. lokale Variablen
  - zusammengefasste Anweisungsfolgen ausgeführt
- Beliebig häufige Verwendung der Programmteile allein durch entsprechend häufige Aufrufe, d.h. Nennung des Namens im Programmtext

16 Prinzip der Prozeduren und Funktionen

- Abhängig von der Art der zusammengefassten Programmteile Unterscheidung zwischen Prozeduren und Funktionen:
  - Prozedur:
    - Anweisungsfolge, der mit einer ***Prozedur*deklaration** ein Name zugeordnet wird und die bei Nennung des Namens als Anweisung (***Prozedur*aufruf**) ausgeführt wird
    - Jede aufgerufene Prozedur fügt sich als Baustein in die ausgeführte Anweisungsfolge eines Programms ein
  - Funktion:
    - Anweisungsfolge, der mit einer ***Funktions*deklaration** ein Name zugeordnet wird und die bei Nennung des Namens als Bestandteil eines Ausdrucks (***Funktions*aufruf**) ausgeführt wird
    - Ausführung der Anweisungsfolge liefert einen Ergebniswert (**Funktionswert** oder auch **Funktionsergebnis** genannt)
    - Jede aufgerufene Funktion fügt sich als Baustein in einen Ausdruck ein, bei dessen weiterer Auswertung das Funktionsergebnis genutzt wird

16 Prinzip der Prozeduren und Funktionen

- Visualisierung des Prinzips

```
program xxz

var ...

begin  {Hauptprogramm}
    ...
    writeln ('Eingabe Geburtstag:');
    write ('Bitte Tag eingeben:');
    readln (Tag);
    write ('Bitte Monat eingeben:');
    readln (Monat);
    write ('Bitte Jahr eingeben:');
    readln (Jahr);
    ...
    ...
    writeln ('Eingabe Studienbeginn');
    write ('Bitte Tag eingeben:');
    readln (Tag);
    write ('Bitte Monat eingeben:');
    readln (Monat);
    write ('Bitte Jahr eingeben:');
    readln (Jahr);
    ...
end.
```

Gleiche Anweisungsfolgen, Zusammenfassen in Prozedur `DatumEingeben`

16 Prinzip der Prozeduren und Funktionen

- Visualisierung des Prinzips

```
program xxz

var ...
procedure DatumEingeben;
begin
begin  {Hauptprogramm}
        ...
        writeln ('Eingabe Geburtstag:');
        write ('Bitte Tag eingeben:');
        readln (Tag);
        write ('Bitte Monat eingeben:')
        readln (Monat);
end;    write ('Bitte Jahr eingeben:');
        readln (Jahr);
        ...
        ...
        writeln ('Eingabe Studienbeginn');
        write ('Bitte Tag eingeben:');
DatumEingeben;
        readln (Tag);
        write ('Bitte Monat eingeben:');
        readln (Monat);
        write ('Bitte Jahr eingeben:');
DatumEingeben;
        readln (Jahr);
  ...
 end.
```

Prozedur**deklaration**

Prozedur**aufrufe**

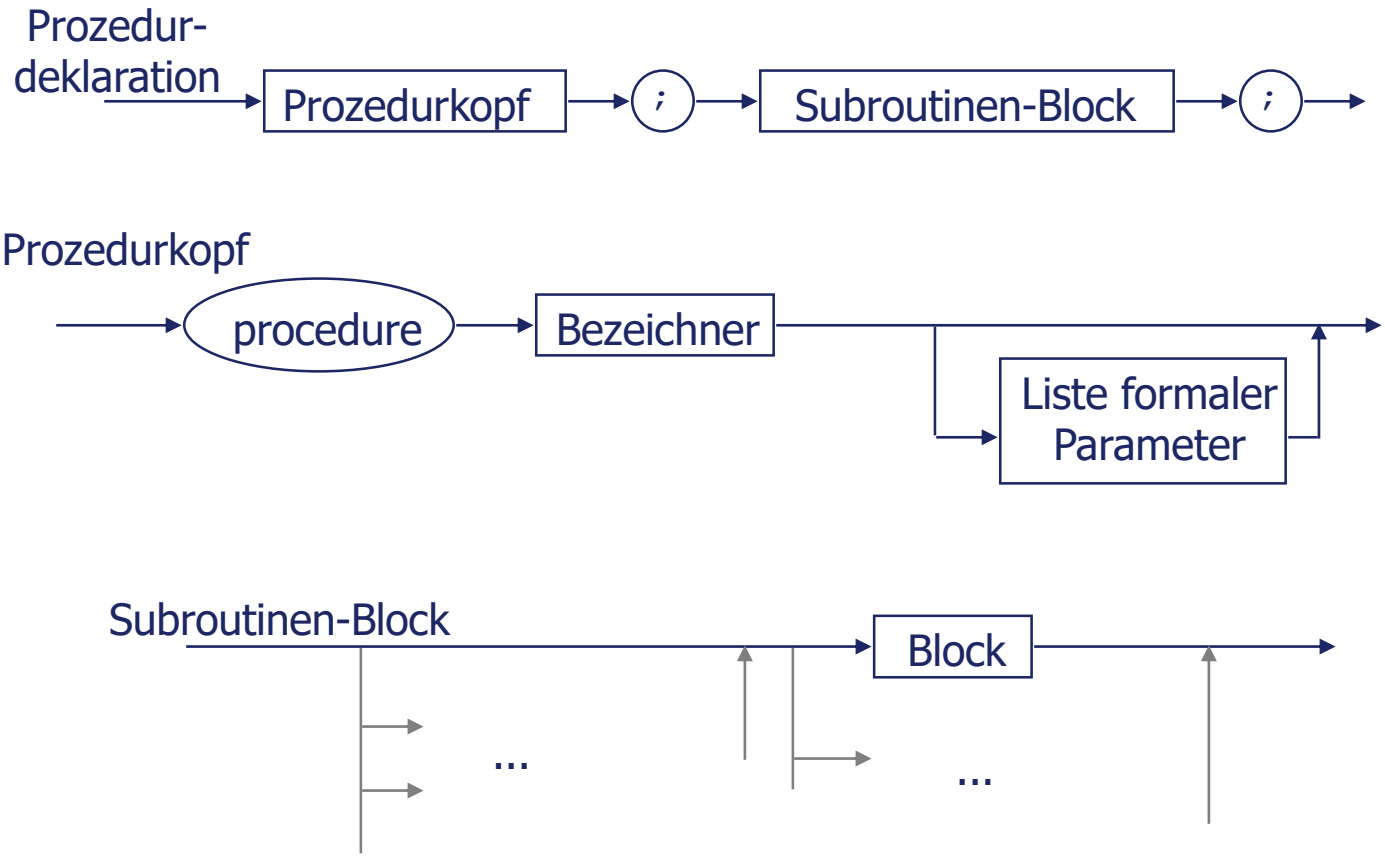
16 Prozeduren

- Eine Prozedur fasst Anweisungen unter einem Namen zusammen
- Eine Prozedur entsteht durch eine **Prozedurdeklaration**
- Eine Prozedur muss *vor* ihrer Verwendung deklariert werden
- Wenn benötigt, in der Prozedurdeklaration zusätzliche *lokale Deklarationen* (z.B. lokale Variablen)
- Bei der Deklaration evtl. explizite Festlegung des Datenaustausches zwischen Prozedur und ihrer Umgebung durch *Parameter*

- Der **Prozeduraufruf** erfolgt als *Anweisung* durch Nennung des Prozedurnamens (sogen. "Prozeduranweisung"), evtl. mit Parameterliste für Datenaustausch
- Bei Aufruf wird der Prozedurrumpf abgearbeitet (lokale Deklarationen und Anweisungen)

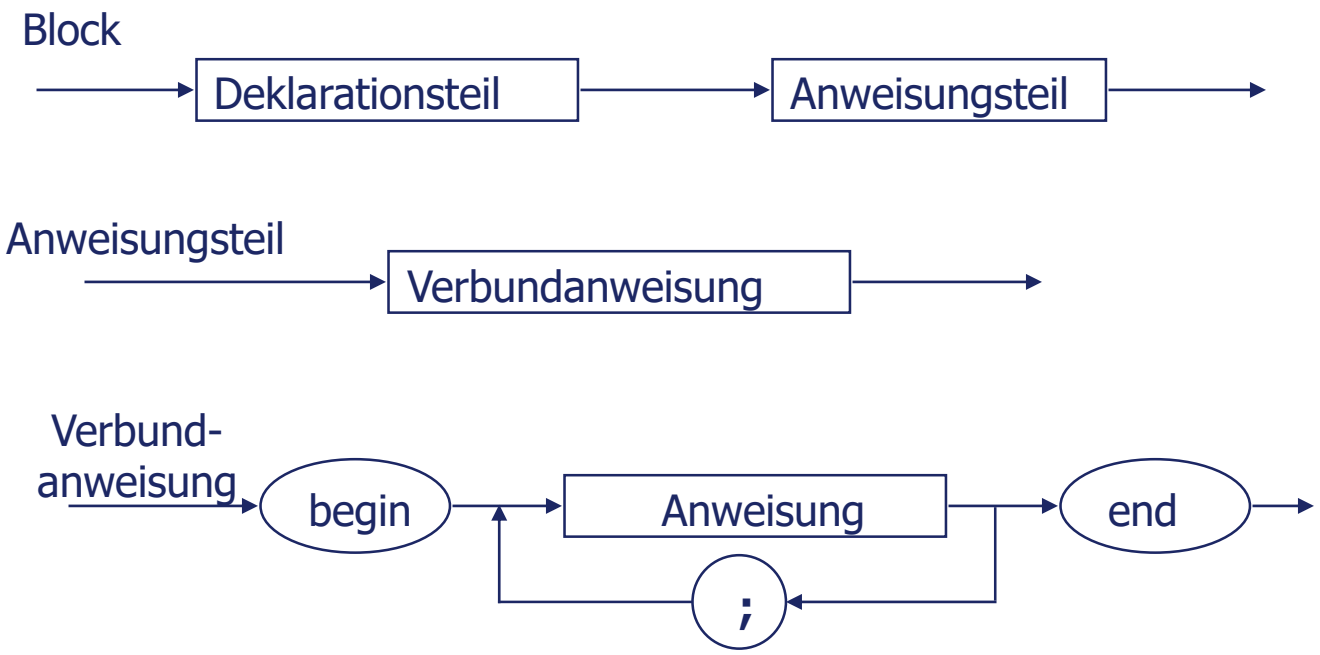
16 Prozeduren

- Syntaktischer Aufbau der Deklaration:



16 Prozeduren

- Syntaktischer Aufbau der Deklaration:



16 Prozeduren

- Beispiel für Prozedur**deklaration**:

```
program Bruch;
var Zaehler_Erg, Nenner_Erg : integer;
...
procedure Kuerzen;

 var M, N, R : integer;

begin
  M := Zaehler_Erg;
  N := Nenner_Erg;
  while N <> 0 do
  begin
    R := M mod N;
    M := N;
    N := R;
  end;
  Zaehler_Erg := Zaehler_Erg div M;
  Nenner_Erg  := Nenner_Erg  div M;
end;
```

Prozedurkopf

Lokale Deklarationen

Anweisungsteil

Zugriffe auf globale Variablen

Im Anweisungsteil einer Prozedur kann auf Größen zugegriffen werden, die außerhalb (vor) der Prozedur deklariert worden sind (*globale* Variable)

Durch Nutzung von Parametern können/sollen solche Zugriffe vermieden werden

16 Prozeduren

- Beispiel für Prozedur**aufruf**:

```
program Bruch;
var Zaehler_Erg, Nenner_Erg : integer;
...
procedure Kuerzen;
  ...
end;
...
   if Aktion = 5 then
     begin
       Zaehler_Erg := Zaehler1 * Zaehler2;
       Nenner_Erg := Nenner1 * Nenner2;


       Kuerzen;


       writeln ('Ergebnis Zähler: ', Zaehler_Erg);
       writeln ('Ergebnis Nenner: ', Nenner_Erg);
     end
     else
       if Aktion = 6 then
       begin
         ...
```

Prozedurdeklaration (siehe vorherige Folie)

Prozedur-aufruf

```
var M, N, R : integer;

begin
  M := Zaehler_Erg;
  N := Nenner_Erg;
  while N <> 0 do
  begin
    R := M mod N;
    M := N;
    N := R;
  end;
  Zaehler_Erg := Zaehler_Erg div M;
  Nenner_Erg  := Nenner_Erg  div M;
end;
```

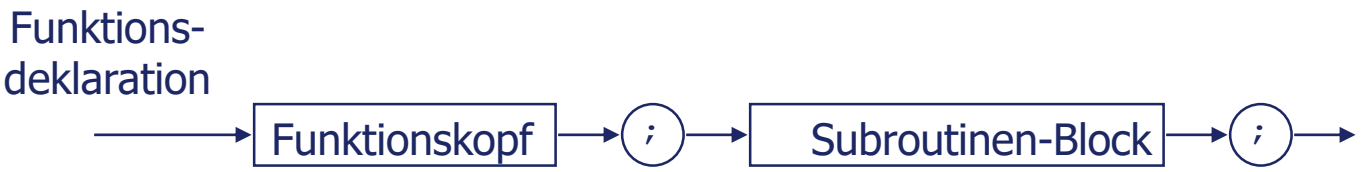
16 Funktionen

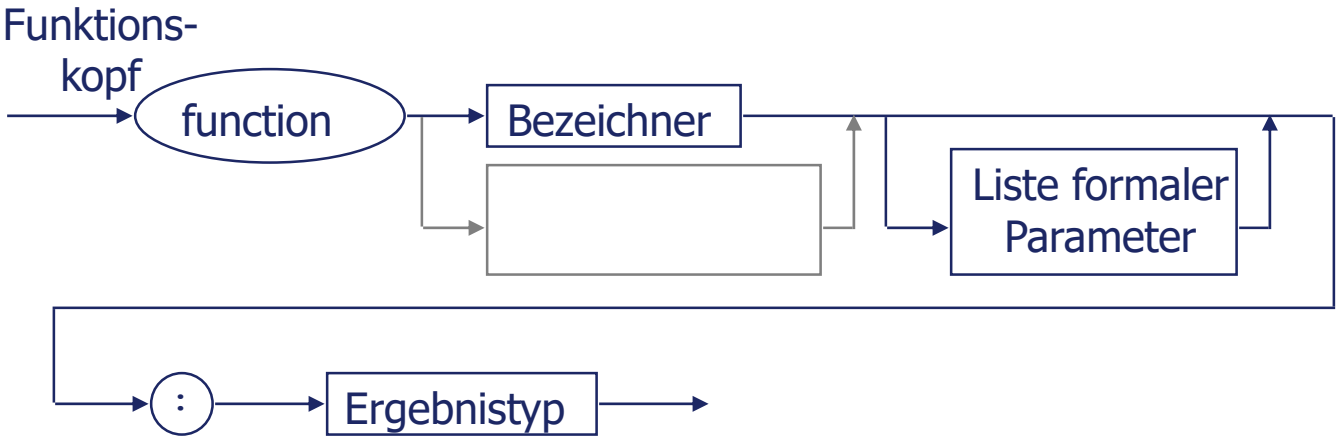
- Funktionen fassen Anweisungen unter einem Namen zusammen *und liefern einen Wert als Ergebnis (Funktionswert)*
- Funktionen entstehen durch eine **Funktionsdeklaration**
- Eine Funktion muss vor ihrer Verwendung deklariert werden
- Innerhalb der Funktion sind lokale Deklarationen möglich, wenn benötigt
- Bei der Deklaration evtl. explizite Festlegung des Datenaustausches zwischen Funktion und Umgebung durch Parameter
- Funktionen enthalten *mindestens eine Wertzuweisung an ihren Bezeichner* im Rumpf, der zugewiesene Wert legt das Funktionsergebnis fest

- Der **Funktionsaufruf** erfolgt durch Nennung des Namens (meist mit Parameterliste) *innerhalb eines Ausdrucks*
- Bei Aufruf wird der Funktionsrumpf (lokale Deklarationen, Anweisungen) abgearbeitet, der Funktionswert geht an der Stelle des Aufrufs in die weitere Verarbeitung des Ausdrucks ein

16 Funktionen

- Syntaktischer Aufbau





Syntax sonst wie bei Prozeduren

16 Funktionen

- Beispiel für Funktions**deklaration**:

```
programm Bruch;
var Zaehler1, Nenner1 : integer;
...

function GGT : integer;

 var M, N, R : integer;

begin
  M := Zaehler1;
  N := Nenner 1;
  while N <> 0 do
  begin
    R := M mod N;
    M := N;
    N := R;
  end;
  GGT := M;
end;
```

Funktionskopf

Lokale Deklarationen

Anweisungsteil

Zugriffe auf globale Variablen

Zuweisung an Funktionsbezeichner

Im Anweisungsteil einer Funktion kann auf Größen zugegriffen werden, die außerhalb (vor) der Funktion deklariert worden sind (globale Variable)

Durch Nutzung von Parametern können/sollen solche Zugriffe vermieden werden.

16 Funktionen

- Beispiel für Funktions**aufruf**:

```
programm Bruch;
var Zaehler1, Nenner1, Teiler : integer;
...

function GGT : integer;
...
end;
...
else
  if Aktion = 2 then
  begin

    Teiler := GGT;

    Zaehler_Erg := Zaehler1 div Teiler;
    Nenner_Erg := Nenner1 div Teiler;
    writeln ('Ergebnis Zähler: ', Zaehler_Erg);
    ...
```

Funktionsdeklaration (siehe vorherige Folie)

Funktions-aufruf

```
var M, N, R : integer;

begin
  M := Zaehler1;
  N := Nenner 1;
  while N <> 0 do
  begin
    R := M mod N;
    M := N;
    N := R;
  end;
  GGT := M;
end;
```

16 Gültigkeitsbereiche

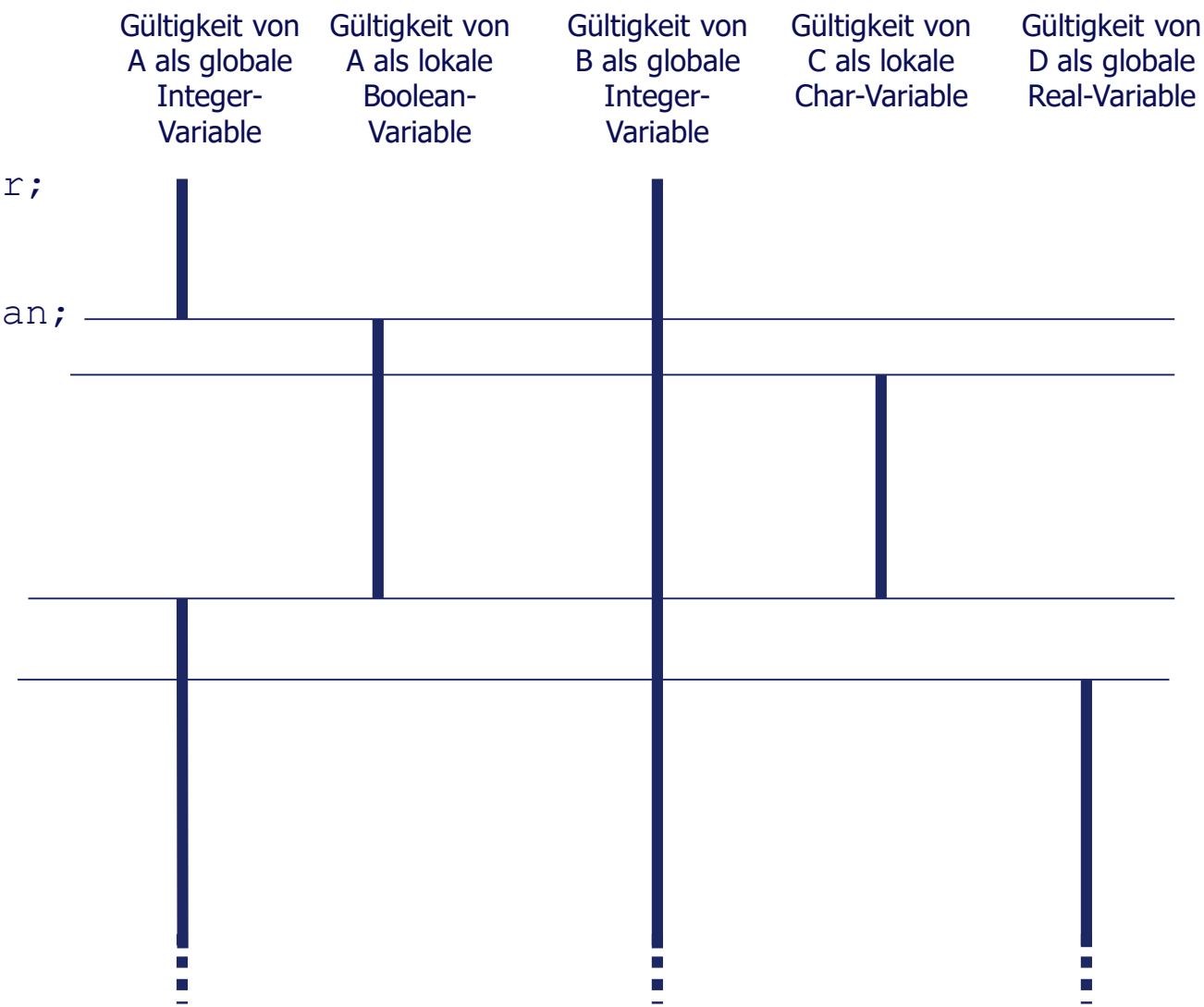
- Beispielkonstellation:

```
program Gueltig;
var A, B : Integer;

  procedure X;
    var A : Boolean;
        C : Char;
  begin
    A := true;
    B := 42;
    C := 'X';
  end;

var D : Real;

begin
  A := 13;
  B := 14;
  D := 2.25;
...
```



16 Parameter

- Unterscheidung von
  - innerem Prozedur-/Funktionsbereich (Rumpf)
  - Umgebung der Prozedur/Funktion am Ort ihres Aufrufs (Aufrufumgebung)
- Fragestellungen:
  - Auf welche Werte der Aufrufumgebung kann innerhalb einer Prozedur/Funktion zugegriffen werden und wie?
  - Wie können Resultatwerte der Prozedur/Funktion in die Aufrufumgebung geliefert werden

↓

Problem des **Datenaustauschs** zwischen Prozedur/Funktion und ihrer Umgebung

16 Parameter

- 2 Möglichkeiten den Datenaustausch zu gestalten:
  - *Impliziter* Datenaustausch:

    Schlecht!!
    - In der Prozedur/Funktion werden Variablen der Umgebung (globale Variablen) direkt verwendet
    - Keine Angabe zum Datenaustausch, keine Parameter, keine sichtbare Schnittstelle
  - *Expliziter* Datenaustausch:
    - Der Datenaustausch erfolgt mit Hilfe von explizit angegebenen **Parametern**
    - Parameter bilden sichtbare Schnittstelle zwischen Innerem der Prozedur/Funktion und ihrer Umgebung

      Gut!!
    - Parameter werden in der Prozedur/Funktion als lokale Größen verwendet

16 Nachteile des impliziten Datenaustauschs

- **Impliziter** Datenaustausch erfolgt mit Hilfe globaler Variablen:

```
program XYZ;
var
  Zahl1, Zahl2 : integer;
...

function GGT : integer;

 var M, N, R : integer;

 begin
  M := Zahl1;
  N := Zahl2;
  while N <> 0 do
  begin
    R := M mod N;
    M := N;
    N := R;
  end;
  GGT := M;
 end;
...

begin {Hauptprogramm}
  Zahl1 := 9;
  Zahl2 := 6;
...
  writeln (GGT);
...
end.
```

Deklaration globaler Variablen

Verwendung globaler Variablen in der Funktion

Aufruf der Funktion ohne erkennbaren Datenaustausch

16 Nachteile des impliziten Datenaustauschs

- **Impliziter** Datenaustausch ohne Parameter mit gravierenden Nachteilen:
  - In Prozedur-/Funktionsdeklaration ist nicht erkennbar, von welchem Typ die verwendeten Werte sind
  - Keine Steuerung/Einschränkung der Außenwirkung möglich
  - Beim Aufruf ist nicht erkennbar, welche Werte in der Prozedur verwendet/verändert werden.
  - Aufruf der Prozedur ist auf einen bestimmten Kontext von globalen Variablen festgelegt

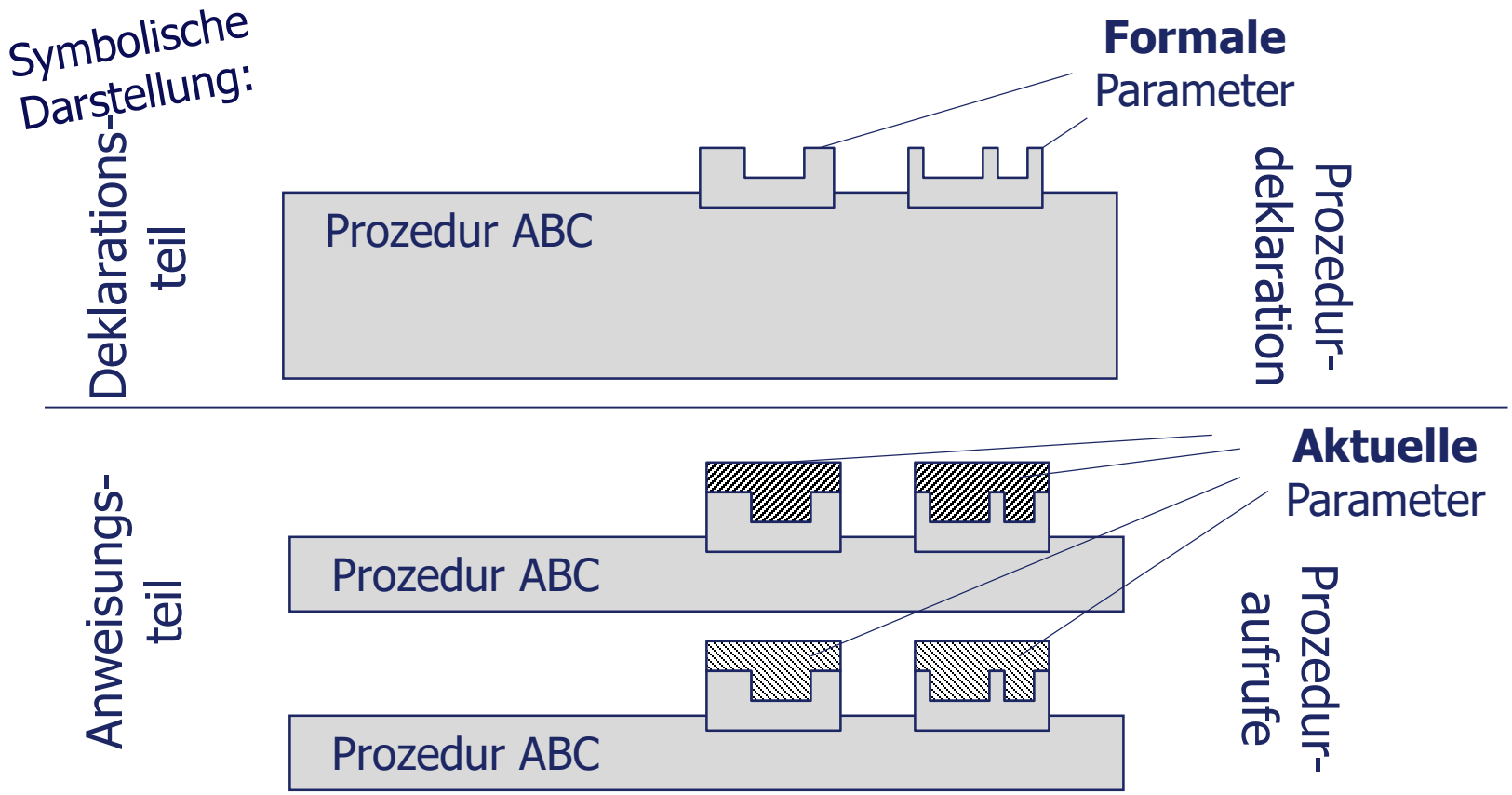
↓

Um diese Nachteile zu vermeiden:
Expliziter Datenaustausch mit **Parametern**

16 Formale und aktuelle Parameter

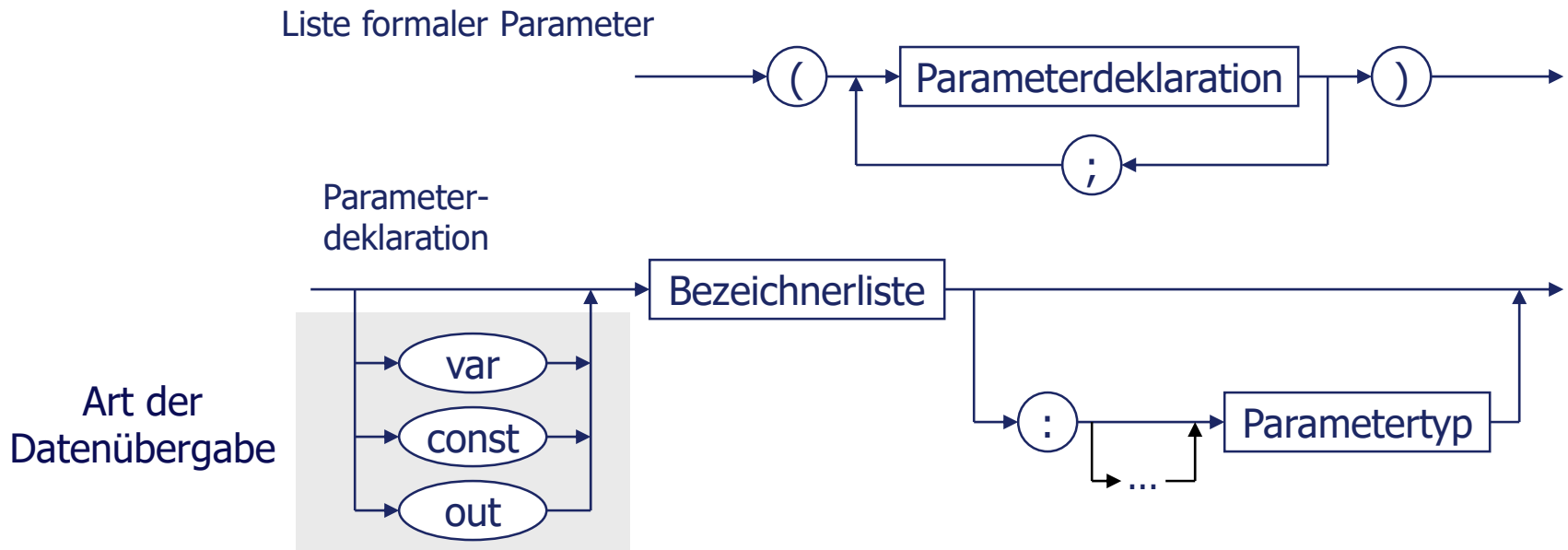
Formale Parameter: Angabe der Anzahl und Art der Parameter **bei der Deklaration** (Festlegung des Datenaustausches/der Schnittstelle)

Aktuelle Parameter: Belegung der Parameterpositionen mit Größen **beim Aufruf** (Durchführung des Datenaustausches/Belegung der Schnittstelle)



16 Formale und aktuelle Parameter

- **Formale** Parameter legen für jeden einzelnen Parameter fest:
  - Bezeichner
  - Datentyp
  - Art der Datenübergabe zwischen Prozedur/Funktion und ihrer Umgebung
- Formale Parameter sind innerhalb der Prozedur/Funktion gültig
- Syntax:



16 Formale und aktuelle Parameter

- **Aktuelle** Parameter versorgen die Schnittstelle von Prozeduren/Funktionen beim Aufruf mit Werten/Variablen
- Zuordnung zwischen formalem und entsprechendem aktuellen Parameter erfolgt gemäß der Reihenfolge der Angaben
- Formale und aktuelle Parameter müssen hinsichtlich Anzahl und Datentypen übereinstimmen
- Aktuelle Parameter können von Aufruf zu Aufruf wechseln

16 Formale und aktuelle Parameter

Beispiel 1 für expliziten Datenaustausch:

```
function GGT (Zahl1 : integer; Zahl2 : integer) : integer;

 var M, N, R : integer;

begin
  M := Zahl1;
  N := Zahl2;
  while N <> 0 do
  begin
    R := M mod N;
    M := N;
    N := R;
  end;
  GGT := M;
end;
…
var A, B, X, Y, Z, Zahl1 : integer;

begin {Hauptprogramm}
  …
  A := GGT (X, Y);
  …
  Z := GGT (A, B) + GGT (X, Y);
  …
  writeln (GGT (Zahl1, Z));
```

Formale Parameter

Aktuelle Parameter

16 Aufgabe “Hangman”

In dieser Aufgabe bauen Sie eine Konsolenvariante des bekannten Spiels “Galgenmännchen”. Dabei muss ein dem Benutzer nur teilweise präsentiertes Wort durch schrittweise Eingabe einzelner Buchstaben erraten werden.

Für die Lösung dieser Aufgabe ist ein Rahmenprogramm aus Prozedur- und Funktionsdeklarationen vorgegeben. Sie müssen nicht das Hauptprogramm schreiben, sondern die geforderten Blöcke implementieren.

```
var
   currentChars : TCharacterSet;
   wordToGuess : string;
begin
   currentChars := [];
   wordToGuess := 'AcaB';
   repeat
      printUsedCharacters(currentChars);
      printPartialWord(wordToGuess, currentChars);
      currentChars := readNewChar(currentChars);
   until isSolved(wordToGuess, currentChars);
   writeln('Glückwunsch, das zu lösende Wort war "', wordToGuess, '"');
end.
```

16 Aufgabe “Hangman”

```
// Gibt alle Buchstaben der übergebenen Menge aus.
//
// @param current Die auszugebende Menge.
procedure printUsedCharacters(current : TCharacterSet);

{ ... }
writeln('Tests für printUsedCharacters');
writeln('Erwartet: Ausgabe der leeren Menge');
printUsedCharacters([]);
writeln('Erwartet: A B Z');
printUsedCharacters(['Z', 'B', 'A']);
writeln('Erwartet: A');
printUsedCharacters(['A'] + ['A']);
```

16 Aufgabe “Hangman”

```
// Gibt nur die erratenen Buchstaben des zu lösenden
// Wortes aus. Für die Menge ['A', 'B'] und das zu
// lösende Wort 'AcaB' soll die Ausgabe 'A _ a B'
// lauten. Die Groß- und Kleinschreibung des zu lösenden
// Wortes wird in der Ausgabe also beibehalten, unbekannte
// Buchstaben werden durch einen Unterstrich ersetzt.
//
// @param solvedWord Das vollständige zu erratende Wort
// @param available Die verfügbaren Buchstaben
procedure printPartialWord(solvedWord : string;
                           available : TCharacterSet);
{ ... }
writeln('Erwartet: _ _ _ _');
printPartialWord('abcd', []);
writeln('Erwartet: _ _ _ _');
printPartialWord('abcd', ['E', 'F']);
writeln('Erwartet: a _ _ _');
printPartialWord('abcd', ['A']);
writeln('Erwartet: a _ _ d');
printPartialWord('abcd', ['A', 'D']);
writeln('Erwartet: a b _ d');
printPartialWord('abcd', ['A', 'D', 'B']);
writeln('Erwartet: a b c d');
printPartialWord('abcd', ['A', 'D', 'B', 'C']);
```

16 Aufgabe “Hangman”

```
// Berechnet ob das zu erratende Wort mit den verfügbaren
// Buchstaben schon gelöst ist.
//
// @param solvedWord Das vollständige zu erratende Wort
// @param available Die verfügbaren Buchstaben
// @return True, wenn alle Buchstaben des zu erratenden Worts
//         in der übergebenen Menge enthalten sind.
function isSolved(solvedWord : string;
                  available : TCharacterSet) : boolean;
{ ... }
writeln('Erwartet: FALSE');
writeln(isSolved('abcd', []));
writeln('Erwartet: FALSE');
writeln(isSolved('abcd', ['E', 'F']));
writeln('Erwartet: FALSE');
writeln(isSolved('abcd', ['A']));
writeln('Erwartet: FALSE');
writeln(isSolved('abcd', ['A', 'D']));
writeln('Erwartet: FALSE');
writeln(isSolved('abcd', ['A', 'D', 'B']));
writeln('Erwartet: TRUE');
writeln(isSolved('abcd', ['A', 'D', 'B', 'C']));
```

16 Aufgabe “Hangman”

```
// Liest einen Buchstaben ein, der bisher nicht verwendet
// worden ist. Wenn der Benutzer mehr als einen Buchstaben
// oder einen schon vorhandenen Buchstaben eingibt, soll
// er erneut zur Eingabe aufgefordert werden.
//
// Die schon genutzten Buchstaben werden dabei stets als
// Großbuchstaben gespeichert, die Eingabe eines Kleinbuchstaben
// soll aber ebenfalls möglich sein.
//
// @param current Bereits verwendete Buchstaben
// @return Current + ein neuer Buchstabe
function readNewChar(current :  TCharacterSet) : TCharacterSet;
```

16 Aufgabe “Hangman”

Für Fortgeschrittene:

- Benutzt Randomize und Random um den Benutzer nicht immer nach dem gleichen Wort zu fragen. Zur Erinnerung:
  - Die Prozedur `Randomize` initialisiert den Zufallszahlengenerator mit einer Serie von zufälligen Zahlen.
  - Die Funktion `Random(maximal : Integer)` liefert eine ganze Zahl zwischen `0` und `maximal - 1`.
- Erweitert euer Spiel um eine maximale Anzahl von Versuchen, die vom Benutzer vorgenommen werden kann. Wenn der Benutzer mehr Buchstaben eingibt als Versuche erlaubt sind, hat er das Spiel verloren.
- Gebt die Anzahl der verbleibenden Versuche nicht einfach als Zahl aus, sondern malt ein ASCII-Art-Galgenmännchen zur Visualisierung der verbleibenden Versuche. Implementiert dafür eine Prozedur `printNumberOfTries(tries : integer)`

```
_______
|      |
|      0
|     /|\
|     / \
|
|
```