11 Prozeduren und Funktionen (Routinen)

Gliederung

11.1 Allgemeines

Selbstdefinierte Prozeduren und Funktionen (Sammelbegriff "Routinen") dienen folgenden Zwecken:

- Gliederung eines größeren Programms in kleinere überschaubare Blöcke, in denen vorrangig lokale Bezeichner verwendet werden. So könnte z.B. ein Programm gegliedert werden in je einen Block für Dateneingabe, -verarbeitung und -ausgabe. Unter Umständen können diese Blöcke in gleicher Weise in verschiedenen Programmen eingesetzt werden, z.B. werden statistische Funktionen für Mittelwert und Standardabweichung in fast allen Programmen für Meßwertverarbeitung gebraucht.
- Vereinfachung des Programms. Wenn bestimmte Programmsequenzen öfters im Programm vorkommen, ist es sinnvoll, diese Sequenz einmal als eine Routine zu deklarieren. Sie kann dann beliebig oft im Programm aufgerufen werden. Die Länge des Blockes ist beliebig.

Pascal enthält bereits viele Standard-Routinen, wie z.B. die Prozedur "*Write()*" oder die Funktion "*Ln()*". Standardroutinen müssen im Gegensatz zu eigenen Routinen nicht extra deklariert werden. Turbo-Pascal setzt aber bei vielen Standardroutinen die Deklaration von Units voraus, die diese Routinen in compilierter Form enthalten. Die Unit "*CRT*" enthält z.B. die Standardprozedur "*ClrScr*" und die Standardfunktion "*ReadKey*".

In Standard-Pascal wird – im Gegensatz zur Sprache C – streng zwischen Prozeduren und Funktionen unterschieden. C kennt nur Funktionen, die aber auch so verwendet werden können, daß sie den Charakter von Pascal-Prozeduren haben, in diesem Fall hat die C-Funktion eben keinen Rückgabewert. Mit der "Erweiterten Syntax" (siehe Kap. 11.16) kann aber auch in neueren Turbo-Pascal-Versionen Funktionen wie Prozeduren verwendet werden. Dies Option ist in der DR-Grundausbildung nicht zugelassen.

Zum Aufbau von Routinen

Routinen haben prinzipiell den gleichen Aufbau wie das Hauptprogramm; die Unterscheidung liegt lediglich in:

- Kein "**program** ..." am Anfang, statt dessen "**procedure ..**." oder "**function ...**"
- Keine Units, also kein "**uses** ..."
- Am Ende der Routine "**end;**" statt "**end.**", also Semikolon statt Punkt.
- Routinen müssen vor dem Ausführungsteil des Hauptprogramm deklariert werden, also vor dessen "**begin**."

Eine Routine kann weitere (lokale) Routinen enthalten, diese wiederum usw. Im Rahmen der (Stack-) Speicherkapazität sind beliebige Routinenschachtelungen möglich, im Gegensatz zur Sprache C.

Somit kann der Aufbau einer Routine schematisch wie folgt angegeben werden:

```
procedure ... bzw. function ...
label ...                              Lokale Labels ("goto" bei DR verpönt)
const ...                              Lokale Konstanten
type ...                               Lokale Datentypen
var ...                                Lokale Variablen
procedure ... bzw. function ...        Lokale Routinen, beliebig viele
...
begin                                  Beginn der Routine
  ...
  ...
end;                                   Ende der Routine mit Semikolon
```

Man kann das Hauptprogramm auch als Spezialfall einer Prozedur auffassen.

Unterschied zwischen Prozeduren und Funktionen in Standard-Pascal (die Unterscheidung gilt sowohl für Standard-Routinen, als auch selbstdefinierte Routinen):

- **Prozeduren** (in Pascal **procedure**) führen Aktionen beliebiger Art aus. Beispiel: Die Standard-Prozedur "*ClrScr*" löscht den Bildschirm.
- **Funktionen** (in Pascal **function**) *können* (d.h. optional) Aktionen beliebiger Art ausführen, liefern aber in jedem Falle bei Standard-Pascal *ein* (und nur *ein*) Ergebnis an die aufrufende Programmstelle zurück. Beispiel: Die Standardfunktion "*Ln(...)*" liefert an die aufrufende Stelle den natürlichen Logarithmus des übergebenen Argumentes zurück. Funktionsergebnisse, egal ob von Standardfunktionen oder von eigenen Funktionen, können beliebig verwertet werden, z.B. Zuweisung an eine typgerechte Variable, Ausgabe in einer Write-Prozedur u.ä. Der Ergebnis-Datentyp der Funktion kann sein: Ordinal, Real, String oder Zeiger; ***nicht*** aber **array**, **record**, **set** oder **file**.

11.2 Einleitende Beispiele

```
program Pas11021;  { Zu Kap. 11: Prozeduren und Funktionen }

uses
  CRT;

var
  x, y, z: Real;  { Globale Variablen "x", "y" und "z" }


procedure Beep;   { Eine Prozedur hat kein Ergebnis }
begin   { Beginn des Rumpfes der Prozedur   }
  Write(#7);  { Ascii-Steuerzeichen #7: Bel }
end;
```

```
procedure WriteXY(Spalte, Zeile: Byte; Meldung: string);
begin                       { Die formalen Parameter einer Prozedur/ }
  GotoXY(Spalte, Zeile);    { Funktion sind  l o k a l e  Variablen, }
  Write(Meldung);           { hier "Spalte", "Zeile" und "Meldung"   }
end;

function Log10(z: Real): Real; { Der formale Parameter "z" ist }
         { eine lokale Variable. Eine Funktion braucht einen   }
begin    { Ergebnisdatentyp, hier "Real" }
  Log10 := Ln(z)/Ln(10);  { Im Rumpf der Funktion Zuweisung eines }
end;                      { Wertes an den Funktionsbezeichner     }

procedure WarteAufTastendruck;
var
  Ch: Char;   { lokale Variable "Ch" }
begin
  while KeyPressed do  { Tastaturpuffer im Allgemeinfall }
    Ch := ReadKey;     { sicherheitshalber leeren        }
  repeat
  until ReadKey <> '';
end;

begin      { ************* Hauptprogramm, main ******************* }
  ClrScr;

  Beep;                     { Aufruf einer Prozedur ohne Parameter }

  WriteXY(5, 2, 'Hallo');   { Übergabe von 3 aktuellen Parametern  }
  WriteXY(6, 3, 'Hallo');   { (Argumenten) an Prozedur, hier zwei  }
  WriteXY(7, 4, 'Hallo');   { Byte- und eine String-Konstante.     }

  WriteXY(1, 7, '');        { Wirkung wie "GotoXY(1, 7);" }

  x := 0.5;
  z := 0.5;            { Globales "z" }
  y := Log10(x);       { Übergabe einer Real-Variablen an Funktion }

  WriteLn('Log10 von ', x:6:4, ' = ', y:6:4);
  WriteLn('Noch ein Log10: ', Log10(47.11));

  WriteLn('x - y + z = ', (x - y + z):6:4);

  WriteXY(5, 12, 'Beenden mit beliebigem Tastendruck ... ');
  WarteAufTastendruck;
end.
```

Die Bildschirmausgabe des Programms "Pas11021.PAS":

```
   Hallo
    Hallo
     Hallo


Log10 von 0.5000 = -0.3010
Noch ein Log10:  1.67311310423793E+0000
```

```
x - y + z = 1.3010

    Beenden mit beliebigem Tastendruck ...
```

Das folgende Programm "Pas11022.PAS" zeigt eine Lösung eines leidigen Pascal-Problems: **Die fehlende Potenzfunktion "x hoch y" durch eine eigene Funktion "Potenz" darstellen**. Die Lösung geht von einer nichtganzzahligen Basis x aus. Der Exponent y kann beliebig sein. Das Ergebnis ist immer ein nichtganzzahliger Typ (Real, Double, Extended usw.). Für ganzzahlige Werte von Basis x und Exponent y gibt es elegantere Lösungen. Das Beispiel zeigt die Probleme, die entstehen können, wenn man die Potenzfunktion nur durch "Exp(y*Ln(x))" darstellt**. Nicht abgesichert ist ein mögliches Überschreiten oder Unterschreiten des darstellbaren Zahlenbereiches**, was bei der Exponentialfunktion leicht passieren kann. Bei Verwendung des Coprozessors tritt bei "Exp(11356.6)" ein Überlauf (Overlow) auf, bei "Exp(11356.5)" noch nicht. Ohne Coprozessor und ohne Emulation, also mit dem Datentyp Real, tritt der Überlauf schon bei "Exp(88.03)" auf, wogegen "Exp(88.02)" noch berechnet werden kann. In ähnlicher Weise kann ein Unterschreiten bei großen negativen Exponenten eintreten; das Ergebnis ist dann zwar praktisch Null, kann aber nicht berechnet werden. Das Programm stürzt sowohl bei Überschreiten als auch bei Unterschreiten mit dem Laufzeitfehler "205: Überlauf bei Gleitkomma-Operation" ab. Das Programm "Pas11022.PAS" enthält (als Vorgriff) eine einfache Absicherung der Numerikeingabe und zwar mit dem Compilerschalter {$I-} bzw.{$I+} und der Standardfunktion "IOResult", die den Wert 0 liefert, wenn die Numerikeingabe kein unzulässiges Zeichen enthält. Weitere Details dazu im Kap. 24.1.

```
program Pas11022; {"Pas11022.PAS", Demo Potenzfunktion "Potenz(x, y)"}
                  { x_hoch_y, 37310398, Dr. K. Haller               }
uses
  CRT;

var
  x, y:   Real;
  Sp, Ze: Byte;

function Potenz(x, y: Real): Real; { ------------------------------ }
begin
  if x = 0.0 then
     if y <> 0.0
        then Potenz := 0.0    { Fall 1a: 0 hoch y = 0          }
        else begin            { Fall 1b: 0 hoch 0 = unbestimmt }
               TextColor(Yellow);
               WriteLn(#7#13#10, '  Fehler bei "Potenz(x, y)":');
               WriteLn('     x und y sind Null. ' +
                       'Unbestimmter Ausdruck.');
               Write(  '  Abbruch nach Tastendruck ... ');
               repeat
               until ReadKey <> '';
               Halt(47); { >>>> Abbruch >>>>>> }
             end;
  ;
```

```
  if x > 0.0
     then Potenz := Exp(y*Ln(x));  { Der einfache Fall }
  ;
  if x < 0.0 then             { "Frac"  liefert Nachkommateil }
     if Frac(y) = 0.0         { "Trunc" liefert Integerteil   }
        then if Odd(Trunc(y)) { "Odd"   liefert "True" bei "ungerade"}
                then Potenz := -Exp(y*Ln(Abs(x)))
                else Potenz :=  Exp(y*Ln(Abs(x)))
        else begin
               TextColor(Yellow);
               WriteLn(#7#13#10, '  Fehler bei "Potenz(x, y)":');
               WriteLn('     x ist negativ bei nichtganzzahligem y');
               Write(  '  Abbruch nach Tastendruck ... ');
               repeat
               until ReadKey <> '';
               Halt(11); { >>>> Abbruch >>>>>> }
             end;
end; { ------------------------------------------------------------- }
begin { =============== Hauptprogramm (main) ====================== }
  TextColor(White);
  TextBackground(Blue);
  ClrScr;
  WriteLn;
  WriteLn('  Demo "x hoch y"  mit Funktion "Potenz(x, y)". ' +
          'Ende mit 0 0');
  WriteLn;
  repeat
    Sp := WhereX;
    Ze := WhereY;
    repeat
      GotoXY(Sp, Ze);
      Write('  Eingabe x y: ');
      ClrEoL;
      {$I-}              { Einfache Numerikabsicherung  ...    }
      ReadLn(x, y);
      {$I+}              { ... mit Compilerschalter "I" ...    }
    until IOResult = 0;  { ... und Standardfunktion "IOResult" }
    GotoXY(40, WhereY - 1);
    Writeln('x hoch y = ', Potenz(x, y));
  until (x = 0.0) and (y = 0.0);
end. { ========================================================== }
```

11.3 Zur Deklaration von Prozeduren und Funktionen

- Eine Prozedur oder eine Funktion kann erst *nach* ihrer Deklaration aufgerufen werden. Eine Ausnahme bildet die Deklaration mit "*forward*", dazu später.
- Prozeduren und Funktionen sind ein Abbild des Hauptprogramms und können somit eigene Deklarationen (eigene Labels, Konstanten, Typen, Variablen und auch eigene Prozeduren und Funktionen) enthalten; lediglich die Unit-Deklaration "**uses**" ist in Routinen nicht zulässig. Die untergeordneten Routinen können wiederum eigene Deklarationen enthalten, diese wiederum ... usw. Eine Routine endet mit "**end;**", das Hauptprogram mit "**end.**"

- Die Deklaration einer Routine wird eingeleitet mit dem reservierten Wort "**procedure**" oder "**function**", es folgt dann ein frei wählbarer Bezeichner und optional eine in runde Klammern gesetzte Liste mit formalen Parametern mit deren Hilfe Daten mit der aufrufenden Programmstelle ausgetauscht werden können. Diese **formalen Parameter** sind Bezeichner von **lokalen Variablen** (ab Turbo-Pascal 5.0 auch Bezeichner von eigenen Routinen), die als **Platzhalter für** die späteren **aktuellen Parameter** dienen. Ab Turbo-Pascal 5.0 können auch Funktionen und Prozeduren als Parameter von anderen Routinen übergeben werden. Es muß immer der Datentyp des Parameters ausgewiesen werden. Die Deklaration der formalen Parameter ist ansonsten ähnlich der üblichen var-Deklaration. Die formale Parameterliste kann auch das reservierte Wort "**var**" enthalten; dazu später. Die Parameterliste kann beliebig lang sein. Die Formatfreiheit von Pascal erlaubt es z.B. auch, die Listenelemente untereinander zu schreiben und mit Kommentaren zu versehen. Die Listenelemente sind mit Semikolon voneinander zu trennen. Variablen mit gleichem Datentyp können aber wie bei der normalen Variablen-Deklaration mit dem Trennungszeichen Komma zusammengefaßt werden. Wenn keine formalen Parameter deklariert werden, dann muß auch das runde Klammernpaar entfallen.

Beispiel für die Deklaration einer Prozedur:

```
procedure IrgendWie(x, y, z: Real; Ch: Char; Test: Boolean);
...
...
begin   { Rumpf der Prozedur }
  ...
  ...
  ...
end;    { Semikolon am Ende einer Prozedur/Funktion }
```

Die Parameterliste dieser Prozedur enthält *fünf* formale Parameter: die Real-Variablen "*x*", "*y*" und "*z*", die Character-Variable "*Ch*" und die boolesche Variable "*Test*". In der Parameterliste von Prozeduren und Funktionen können Variablen gleichen Datentyps wie bei der var-Deklarationen zusammengefaßt werden; als Trennzeichen dient wie üblich das Komma.

Bei Funktionen ist die (optionale) Parameterliste in gleicher Weise aufzubauen; zwingend ist die zusätzliche Angabe des Datentyps des Ergebnisses. Im Rumpf der Funktion muß dem Funktionsbezeichner ein Wert zugewiesen werden. In der Regel wird die Zuweisung nur einmal und zwar meistens am Ende des Funktionsrumpfes erfolgen. Der Turbo-Pascal-Compiler erkennt fehlende Wert-Zuweisung aber nicht; die Folge sind schwer lokalisierbare Fehler, da mit undefinierten Werten weitergerechnet wird.

Beispiel für die Deklaration einer Funktion:

```
function IrgendWas(i: Integer; x: Real): Char;
                                          { |   }
...                                       { +--- Datentyp Funktionsergebnis }
begin   { Rumpf der Funktion }
...
  IrgendWas := ....; { Zuweisung eines Wertes an den Funktionsbezeichner }
  ...
end;    { Semikolon am Ende einer Prozedur oder einer Funktion }
```

In diesem Beispiel besitzt die Funktion eine Parameterliste mit *zwei* Parametern: die Integer-Variable *i* und die Real-Variable *x*. Das Ergebnis der Funktion besitzt den Datentyp Char.

Zur Übergabe von strukturierten Datentypen

Wie bereits erwähnt, muß zu jedem formalen Parameter (oder der Zusammenfassung mehrerer Parameter gleichen Typs) der Datentyp angegeben werden. Bei strukturierten Datentypen ("**array**", "**string**", "**set**", "**record**" und "**file**") muß für solche Zwecke mit "**type**" ein eigener Typbezeichner deklariert werden. Es sei denn, man verwendet mit dem Compilerschalter {$P+} bzw. mit dem Menüpunkt "Option/Compiler../Offene Array-Grenzen" bei Arrayübergabe die "Offenen Arrays" (siehe Kap. 12.8) oder bei Stringübergabe die "Offenen Strings" (siehe Kap. 14).

Somit ist z.B. **nicht zulässig**:

```
procedure Test(s: string[20]; {nur "string" wäre zulässig}
               x: array[1..9] of Real;
               c: set of Char;
               r: record
                    Name:  string[15];
                    Preis: Real;
                  end;
               f: file of Integer);
...
```

Statt dessen z.B:

```
type
  TString = string[20];    { Alle Typbezeichner frei gewählt }
  TArray  = array[1..9] of Real;
  TSet    = set of Char;
  TRecord = record
              Name:  string[15];
              Preis: Real;
            end;
  TFile   = file of Integer;  { Vorgriff }
...
procedure Test(    s: TString;
                   x: TArray;
                   c: TSet;
                   r: TRecord;
               var f: TFile); { File-Typen nur mit "var" !!!     }
                              { "var" bei anderen Typen optional }
...
```

11.4 Zum Aufruf von Prozeduren und Funktionen

Prozeduren und Funktionen werden in Pascal nur mit ihrem Namen und, falls Parameter zu übergeben sind, mit einer Liste der aktuellen Parameter aufgerufen: In anderen Sprachen wird bei Prozeduren häufig ein "CALL" vorangestellt (Fortran, Turbo-Basic, Quick-Basic).

Da Prozeduren keinen Wert zurückliefern, kann ein Prozedurbezeichner nie in einer Zuweisung oder in einem Ausdruck oder in einer Write-Prozedur erscheinen. Beim Aufruf stellt der Prozedurbezeichner zusammen mit der optionalen Parameterliste immer eine alleinstehende Anweisung dar.

Funktionsbezeichner sind in ihrer Anwendung nicht in dieser Weise eingeschränkt; es wird jedoch bei Zuweisungen und bei Verwendung in Ausdrücken passender Datentyp vorausgesetzt. Ein Funktionsbezeichner kann aber nur als Teil einer Anweisung erscheinen; bei Zuweisungen außerhalb des Funktionsrumpfes nur rechts des Zuweisungsoperators ":=".

Beispiel in schematisierter Form:

Es sei:

`Huber`	ein Prozedurbezeichner
`Meier`	ein Funktionsbezeichner mit Real-Ergebnis
`i`	eine Integer-Variable
`x`	eine Real-Variable

dann ist:

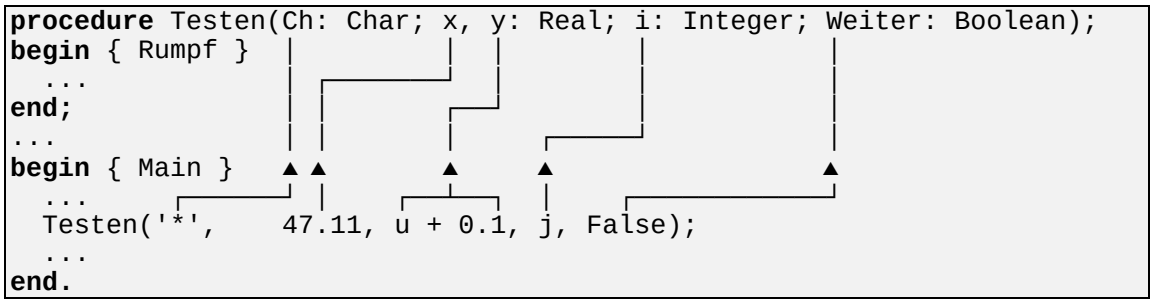
```
x := Huber;               { nicht zulässig }
x := Meier;               { zulässig       }
Huber;                    { zulässig       }
Meier;                    { nicht zulässig }
x := 4711 + Sqr(Meier);   { zulässig       }
Meier := 47.11;           { nicht zulässig }
Write(Huber);             { nicht zulässig }
Write(Meier);             { zulässig       }
if Bedingung
   then Huber             { zulässig       }
   else Meier;            { nicht zulässig }
i := Meier;   { nicht zulässig, da falscher  Datentyp }
```

Mit der "Erweiterten Syntax" ist ab Turbo-Pascal 6.0 dennoch die Verwendung von gewissen Funktionen im Sinne von Prozeduren möglich. Details in Kap. 11.9.

11.5 Zum Datenaustausch zwischen aktuellen und formalen Parametern

Wenn eine Routine eine formale Parameterliste enthält, dann muß an der aufrufenden Stelle ebenfalls eine Liste aufgeführt sein, die aktuelle Parameterliste. Sie muß genau so viele Paramater enthalten wie die formale Liste; zudem müssen die Datentypen der einzelnen Elemente übereinstimmen. Es gilt: Das i-te Element der aktuellen Liste wird an das i-te Element der formalen Liste übergeben.

Das folgende Schema zeigt den Zusammenhang am Beispiel einer Prozedur, wobei vorausgesetzt wird, daß die (globalen) Variablen des Hauptprogramms "u" und "j" deklariert und mit Werten belegt sind.



Aktuelle Parameterliste:			**Formale Parameterliste:**	
Char-Konstante	`'*'`	➧	Char-Variable	`Ch`
Real-Konstante	`47.11`	➧	Real-Variable	`x`
Real-Ausdruck	`u + 0.1`	➧	Real-Variable	`y`
Integer-Variable	`j`	➧	Integer-Variable	`i`
Boolean-Konstante	`False`	➧	Boolean-Variable	`Weiter`

11.6 Gültigkeit der Deklarationen

Die Bezeichner in der Liste der formalen Parameter (üblicherweise Variablenbezeichner oder bei Übergabe von Routinen auch Routinenbezeichner) haben lokale Bedeutung, d.h. sie existieren nur in der betreffenden Routine. Ein namensgleicher Bezeichner außerhalb der Routine steht in der Routine nicht zur Verfügung und kann somit auch nicht in der Routine verändert werden. Wenn die Routine verlassen ist, stehen in umgekehrter Weise die formalen Parameter nicht mehr zur Verfügung.

Eine Sonderstellung nehmen Variablen der formalen Parameterliste ein, die zusätzlich mit "**var**" deklariert sind; dazu später.

Empfehlung:

Routinen sollten möglichst nur **lokale** Deklarationen enthalten bzw. keine globalen Deklarationen benutzen. Auf diese Weise lassen sich sog. Seiteneffekte bei Programm-Änderungen leichter vermeiden. Programme können dann auch leichter modular aus Routinen (oder aus Sammlungen von Routinen in eigenen Units) aufbaut werden. Die formale Parameterliste stellt dann die Software-Schnittstelle zu anderen Programmteilen dar.

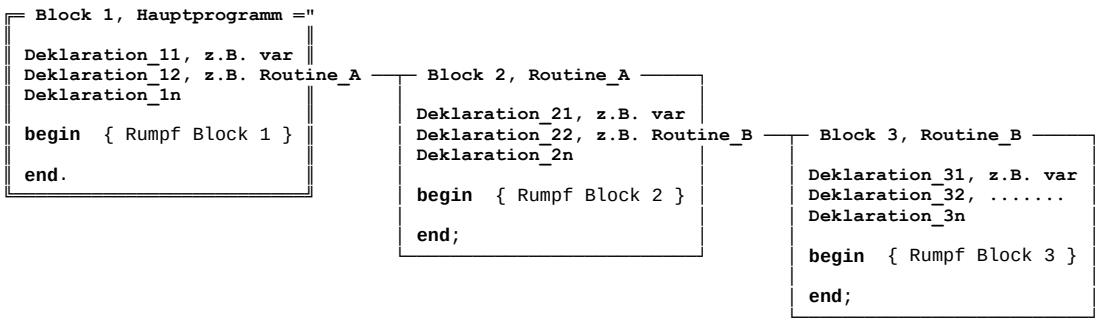
Das folgende Programm demonstriert die lokale Wirkung einer Variablen in der formalen Parameterliste. Das Hauptprogramm enthält die Integer-Variable "*i*". Diese wird an die Prozedur übergeben; hier zur Demo an einen namensgleichen Bezeichner. Innerhalb der Prozedur wird die (lokale) Variable verändert. Anschließend wird die Variable "*i*" im Hauptprogramm wieder ausgedruckt. Es erscheint wieder der alte Wert. Das Programm benutzt weiter die globale String-Variable "*s*", die aber nicht als Parameter an die Prozedur übergeben wird, dennoch aber darin aufgerufen und verändert wird. Da es sich um eine globale Variable handelt, sind die Änderungen auch im Hauptprogramm wirksam.

```
program Pas11061; { Demo: Lokale Bedeutung der Variablen      }
                  {       in der Liste der formalen Parameter }
uses
  CRT;
var
  i: Integer;
  s: string[15];

procedure Test(i: Integer); { Formale Parameter, hier "i", haben  }
                { einer Routine lokale Bedeutung, auch wenn der   }
                { aktuelle Parameter, wie hier demonstriert, eine }
                { namensgleiche Variable ist.                     }
const         { Eine Routine kann darüber hinaus auch noch   }
  c = 4711;   { beliebige lokale Deklarationen enthalten,    }
var           { hier die Konstante "c" und die Variable "j"  }
  j: Integer;

begin
  WriteLn('In Prozedur vor  Veränderung, i:  ', i); {|... 1          }
  WriteLn('In Prozedur vor  Veränderung, s:  ', s); {|... Anton Huber}

  i := i + 99;       { Lokales  "i". Keine Wirkung auf Hauptprogramm }
  s := 'Julia Help'; { Globales "s". Mit   Wirkung auf Hauptprogramm }

  TextColor(LightGray);
  WriteLn('In Prozedur nach Veränderung, i:  ', i); {|... 100        }
  WriteLn('In Prozedur nach Veränderung, s:  ', s); {|... Julia Help }

  j := 9999;
  WriteLn('In Prozedur, lokal c, j:      ', c, ' ', j);
                                                    {|... 4711 9999  }
end;
```

```
begin  { ******** Hauptprogramm ********************************** }
  TextBackGround(Blue);
  ClrScr;

  i := 1;
  s := 'Anton Huber';

  TextColor(Yellow);

  WriteLn('Hauptprogramm, vor  Aufruf Prozedur, i:  ', i);
  { |... 1          }
  WriteLn('Hauptprogramm, vor  Aufruf Prozedur, s:  ', s);
  { |... Anton Huber}

  TextColor(White);

  Test(i);{Prozedur-Aufruf mit aktuellem Parameter, hier Variable "i"}

  TextColor(Yellow);
  WriteLn('Hauptprogramm, nach Aufruf Prozedur, i:  ', i);
  { |... 1          }
  WriteLn('Hauptprogramm, nach Aufruf Prozedur, s:  ', s);
  { |... Julia Help }
  TextColor(White);

  { WriteLn(c, j); }     { Auskommentiert, sonst ...         }
  { Abbruch mit Meldung "Fehler 3: Unbekannter Bezeichner",  }
  { da "c" und "j" nicht im Hauptprogramm deklariert sind.   }
  repeat
  until ReadKey <> '';
end.
```

Allgemeinere Formulierung der Gültigkeit von Deklarationen:

Deklarationen (Label, Konstanten, Typen, Variablen, Prozeduren und Funktionen) haben Gültigkeit in dem Programmblock (Hauptprogramm, Routine A, Routine B, usw.) in dem sie aufgeführt sind und in allen darunter liegenden Blöcken, soweit sie dort nicht durch andere Deklarationen mit gleichen Bezeichnern aufgehoben wird. Der Aufruf eines Bezeichners kann erst nach der Deklaration erfolgen. Ausnahme bei Routinen mit "*forward*", siehe Unterpunkt 11.8



Zur Veranschaulichung wird angenommen:

1. Deklarationen_11, _21, _31 seien Variablendeklarationen. Dann gilt:

- *Variablen_11* sind gültig im Block 1, Block 2 und Block 3
- *Variablen_21* sind gültig im Block 2 und Block 3
- *Variablen_31* sind gültig im Block 3

Die Gültigkeit kann aber durch lokale Deklarationen mit namensgleichen Bezeichnern in den Blöcken 2 und 3 aufgehoben werden.

2. Deklarationen_12, _22, _32 seien Routinendeklarationen. Dann gilt:
   - Routine_12 ist gültig im Block 1, Block 2 und Block 3.
   - Routine_22 ist gültig im Block 2 und Block 3.
   - Routine_32 ist gültig im Block 3.

   Die Gültigkeit kann aber durch lokale Deklarationen in den Blöcken 2 und 3 mit namensgleichen Bezeichnern aufgehoben werden. Innerhalb des gleichen Blockes kann die Deklaration einer *RoutineY* den Aufruf einer *RoutineX* nur dann enthalten, wenn *RoutineX* vor *RoutineY* deklariert ist. Zu Ausnahmen mit "*forward*" siehe Unterpunkt 11.8.

11.7 Parameterübergabe mit Wert oder mit Adresse

Bei den bisherigen Beispielen erfolgte der Datenaustausch zwischen dem aktuellen Parameter und dem formalen Parameter durch **Übergabe mit Wert**. Präziser: Es wurde von den Routinen eine Kopie des Wertes in eigenen Speicherzellen angelegt. Veränderungen des formalen Parameters in der Routine betrafen somit nur diese Speicherzellen und hatten somit keine Rückwirkung auf den aktuellen Parameter im aufrufenden Programmteil.

Wie die früheren Beispiele zeigen, können die aktuellen Parameter bei "*Übergabe mit Wert*" sowohl Variablen, Konstanten als auch beliebige Ausdrücke sein. Die formalen Parameter sind dagegen bekanntlich immer Variablen; bei "*Übergabe mit Wert*" kann man sie auch mit **Werteparameter** bezeichnen.

Viele Situationen erfordern dagegen die Möglichkeit der Veränderung der formalen Parameter mit Rückwirkung auf die aufrufende Programmstelle. Grundsätzlich könnte man das Problem durch Verwendung globaler Variablen umgehen; was aber der Forderung nach modularem Programmaufbau widerspricht.

Die **Übergabe mit Adresse** löst dieses Problem. Hier wird nicht eine Kopie des aktuellen Parameters übergeben, sondern dessen Speicheradresse. Somit haben Veränderungen des formalen Parameters auch eine Veränderung des aktuellen Parameters zur Folge. Da bei dieser Art der Übergabe Speicheradressen übergeben werden, sind für die aktuellen Parameter nur Variablen zulässig, also keine Konstanten und auch keine Ausdrücke! Die Bezeichner der korrespondierenden aktuellen und formalen Variablen können (wie sonst auch) beliebig sein, so z.B. auch gleich.

Der aktuelle Parameter darf bei "*Übergabe mit Adresse*" auch eine nichtinitialisierte Variable (besitzt undefinierten Wert) sein, wobei sinnvollerweise die Initialisierung in der Routine erfolgt und somit ein definierter Wert zurückgeliefert wird.

Die programmtechnische Maßnahme für "*Übergabe mit Adresse*" besteht nur darin, den formalen Parameter mit einem vorausgestelltem "**var**" zu deklarieren. Die Parameter bei "*Übergabe mit Adresse*" bezeichnet man auch mit **Variablenparameter**. File-Typen (Vorgriff) können nur als Variablenparameter übergeben werden.

Beispiele:

```
procedure Demo1(var j: Integer; x: Real);
  "j"          Übergabe mit Adresse,   Typ Integer
  "x"          Übergabe mit Wert,      Typ Real

procedure Demo2(var i, j, k: Integer; x, y: Real);
  "i, j, k"    Übergabe mit Adresse,   Typ Integer
  "x, y"       Übergabe mit Wert,      Typ Real

function Demo3(var i: Integer; x, y: Real; var s: string): Byte;
  "i"          Übergabe mit Adresse,   Typ Integer
  "x, y"       Übergabe mit Wert,      Typ Real
  "s"          Übergabe mit Adresse,   Typ string
```

Bei der aktuellen Parameterliste sind keine Zusatzvereinbarungen notwendig; man hat nur darauf zu achten, daß die korrespondierenden Parameter in der aktuellen Liste bei "*Übergabe mit Adresse*" typgleiche Variablen sind.

```
program Pas11071; {Demo: Parameterübergabe mit Wert oder mit Adresse}

uses
  CRT;

var
  i:   Integer;
  x:   Real;

         {        ┌── j erhält Adresse von i (Variablenparameter) }
         {        │           ┌─ y erhält Wert von x(Werteparameter)}
procedure Test(var j: Integer; y: Real);
begin
  WriteLn('In Prozedur vorher:       j, y: ', j, y:7:2);
                                     { |...  1  47.11 }
  j := 9;
  y := 99.99;
  WriteLn('In Prozedur nachher:      j, y: ', j, y:7:2);
                                     { |...  9  99.99 }
end;

begin
  ClrScr;
```

```
  i := 1;
  x := 47.11;

  WriteLn('Im Hauptprogramm vorher:  i, x: ', i, x:7:2);
                                     { |...  1  47.11 }
  Test(i, x);

  WriteLn('Im Hauptprogramm nachher: i, x: ', i, x:7:2);
                                     { |...  9  47.11 }
  repeat until KeyPressed;
end.
```

11.8 Aufruf einer Routine vor ihrer Deklaration mittels "forward"

Gelegentlich kommt es vor, daß sich zwei Routinen gegenseitig aufrufen. Auch die Änderung der Deklarationsreihenfolge ändert nichts an der Tatsache, daß an einer Stelle eine noch nicht deklarierte Routine aufgerufen wird.

In diesen Fällen wird bei der Routine, die zuerst deklariert wird, zunächst nur die Kopfzeile ("*procedure*" oder "*function*"), der Routinenbezeichner und die optionale Parameterliste angegeben, bei Funktionen zusätzlich noch der Datentyp des Ergebnisses). Diese Kopfzeile erhält den Zusatz "*forward*" (reserviertes Wort).

Es folgt dann die vollständige Deklaration der zweiten Routine in der gewohnten Weise und danach danach der noch ausstehende Teil der Deklaration der ersten Routine. Dieser Teil wird eingeleitet mit "*procedure*" oder "*function*" und dem Routinenbezeichner. Diese "verkürzte Kopfzeile" enthält aber auf keinen Fall eine formale Parameterliste, da diese, falls existent, bereits in der "*forward*"-Deklaration enthalten ist. Nach der verkürzten Kopfzeile folgen in üblicher Weise die lokalen Deklarationen (soweit vorhanden) und der Rumpf der Routine. Im Interface-Teil einer Unit (Vorgriff) ist "*forward*" weder notwendig, noch zulässig.

```
program Pas11081;  {  Demo: "forward" bei gegenseitigem Aufruf
                           zweier Routinen }
uses
  CRT;

var
  i: Integer;

procedure Max(n: Integer); forward;  { Hier kein Rumpf             }
```

```
procedure Moritz(n: Integer);
begin
  WriteLn('Hier ist Prozedur "Moritz", n = ', n);
  if n > 0
     then Max(n - 1);          { Aufruf der Prozedur "Max"    }
end;

procedure Max;                   { Hier keine formale Parameterliste }
begin
  WriteLn('Hier ist Prozedur "Max",    n = ', n);
  if n > 0
     then Moritz(n - 1);       { Aufruf der Prozedur "Moritz" }
end;

function Huber(n: Integer): Integer; forward;  { Hier kein Rumpf }

function Meier(n: Integer): Integer;
begin
  WriteLn('Hier ist Funktion "Meier",  n = ', n);
  if n > 0
     then Meier := Huber(n - 1)
     else Meier := 47;
end;

function Huber;                          { Hier keine Parameterliste }
begin
  WriteLn('Hier ist Funktion "Huber",  n = ', n);
  if n > 0
     then Huber := Meier(n - 1)
     else Huber := 11;
end;

begin  { Hauptprogramm }
  ClrScr;

  WriteLn('Gegenseitiger Aufruf zweier Prozeduren: ');
  WriteLn;

  Max(3);        { Die Bildschirmausgabe:  }
                 { |Hier ist Prozedur "Max",    n = 3 }
                 { |Hier ist Prozedur "Moritz", n = 2 }
                 { |Hier ist Prozedur "Max",    n = 1 }
                 { |Hier ist Prozedur "Moritz", n = 0 }
  WriteLn;

  Moritz(2);     { Die Bildschirmausgabe:  }
                 { |Hier ist Prozedur "Moritz", n = 2 }
                 { |Hier ist Prozedur "Max",    n = 1 }
                 { |Hier ist Prozedur "Moritz", n = 0 }

  WriteLn; WriteLn;
  WriteLn('Gegenseitiger Aufruf zweier Funktionen: ');
  WriteLn;
```

```
  i := Huber(3); { Die Bildschirmausgabe:  }
                 { |Hier ist Funktion "Huber",  n = 3 }
                 { |Hier ist Funktion "Meier",  n = 2 }
                 { |Hier ist Funktion "Huber",  n = 1 }
                 { |Hier ist Funktion "Meier",  n = 0 }
  WriteLn(i);    { |47 }

  repeat
  until KeyPressed;
end.
```

11.9 Übergabe von Funktionen und Prozeduren als Parameter

Ab Turbo-Pascal 5.0 können neben Werteparametern und Variablenparametern auch Routinenparameter (Funktions- und Prozedurvariablen) deklariert und übergeben werden. Ähnlich wie bei strukturierten Typen muß für den Datentyp in der formalen Parameterliste aber vorher mit "**type**" ein eigener Datentyp (hier für eine Funktion oder eine Prozedur) vereinbart werden. Eingeschachtelte Routinen können aber nicht übergeben werden, auch nicht *inline*-Prozeduren.

Das folgende Demo-Programm zeigt zunächst, wie ein Datentyp für eine Funktion "*Funktionstyp*" vereinbart wird. Anschließend werden die zwei Funktionen "*Sinus*" und "*Cosinus*" in nahezu üblicher Weise deklariert. Da diese beiden Funktionen später als Funktionsvariablen an die Prozedur "*Demo_Funktions_Uebergabe*" übergeben werden, müssen sie lediglich mit "*FAR*" vereinbart werden, was mit dem (lokalen) Compilerbefehl "{$F+}" (Vorgriff) in der gezeigten Weise bewirkt wird. Die Parameterliste der Prozedur "*Demo_Funktions_Uebergabe*" enthält den formalen Parameter "*F*" mit dem Datentyp "*Funktionstyp*". An diesen formalen Parameter wird später als aktueller Parameter die Funktion "*Sinus*" und anschließend die Funktion "*Cosinus*" übergeben. Es werden also verschiedene Funktionen an die gleiche Prozedur übergeben.

Das Programm demonstriert weiter die Übergabe verschiedener Prozeduren ("*P1*" und "*P2*") an die Prozedur "*Demo_Prozedur_Uebergabe*". Für die Übergabe wird mit "*type*" der Datentyp "*Prozedurtyp*" vereinbart. Der formale Parameter hat den Bezeichner "*P*".

Es sind alle Kombinationen der Übergabe möglich:

- Funktionen an Prozedur (im Demo-Programm "Pas11091.PAS")
- Prozeduren an Prozedur (im Demo-Programm "Pas11091.PAS")
- Funktionen an Funktion
- Prozeduren an Funktion

```
program Pas11091;  { Demo: Übergabe von Prozeduren und Funktion als  }
                   {       Parameter. Ab Turbo-Pascal 5.0 möglich    }
```

```
                    { Dazu wird der Compilerschalter "$F" benötigt    }
      { Zum Compilerschalter "$F+": Funktionen und Prozeduren, die als
        Parameter an andere Prozeduren oder andere Funktionen über-
        geben werden, müssen mit "FAR" (= fern) deklariert werden, was
        mit der Plus-Option des Compilerbefehls "$F" bewirkt wird.
        Ansonsten Abbruch mit Fehlermeldung "Fehler 143: Üngültiger
        Bezug auf Prozedur oder Funktion". Der Compilerbefehl "$F"
        entspricht dem Menüpunkt "Option/Compiler.../FAR-Aufrufe
        erzeugen". Compilerbefehle im Programm haben höhere Priorität
        als die entsprechenden Menüpunkt-Einstellungen. Es genügt,
        wenn die Kopfzeile der Routine, die später als Parameter über-
        geben wird, mit den beiden Optionen des Compilerbefehls einge-
        schlossen wird; siehe Beispiel. Es ist aber auch möglich, alle
        in Frage stehenden Routinen mit den beiden Optionen des
        Compilerbefehls einzuschließen.
        ------------------------------------------------------------ }
uses
  CRT;
type
  Funktionstyp = function(z: Real): Real;
  Prozedurtyp  = procedure(n: Integer);

{$F+} function Sinus(x: Real): Real; {$F-}
begin
  Sinus   := Sin(x*Pi/180);    { mit Umwandlung Gradmaß in Bogenmaß }
end;

{$F+} function Cosinus(x: Real): Real; {$F-}
begin
  Cosinus := Cos(x*Pi/180);
end;

{$F+} procedure P1(n: Integer); {$F-}
var
  i: Integer;
begin
  for i := 1 to n do WriteLn('1111111111111111');
end;

{$F+} procedure P2(n: Integer); {$F-}
var
  i: Integer;
begin
  for i := 1 to n do WriteLn('2222222222222222');
end;

procedure Demo_Funktions_Uebergabe(F: Funktionstyp;
                                   xMin, xMax, DeltaX: Real);
var
  x: Real;
begin
  x := xMin;
  while x <= xMax do
    begin
      WriteLn(x:9:1, F(x):9:3);
```

```
      x := x + DeltaX;
    end;
end;

procedure Demo_Prozedur_Uebergabe(P: Prozedurtyp; n: Integer);
begin
  P(n);
end;


begin    { --------Hauptprogramm ---------------------------------- }
  TextBackGround(Blue);
  TextColor(White); ClrScr;

  Demo_Funktions_Uebergabe(Sinus,   0, 90, 15);
                         { Sinus 0°, 15°, ... 90° }

  WriteLn;

  Demo_Funktions_Uebergabe(Cosinus, 0, 90, 15); {dto. Cosinus-Tabelle}

  WriteLn;

  Demo_Prozedur_Uebergabe(P1, 2);           { |1111111111111111      }
                                            { |1111111111111111      }
  Demo_Prozedur_Uebergabe(P2, 2);           { |2222222222222222      }
                                            { |2222222222222222      }
  repeat
  until ReadKey <> '';
end.
```

11.10 Rekursionen

Unter Rekursion versteht man den Aufruf einer Routine durch sich selbst. In Pascal sind im Gegensatz zu einigen anderen Programmiersprachen Rekursionen möglich.

Zur Veranschaulichung diene die mathematische Funktion "Fakultät x!"

Die rekursive Definition lautet: `x! = x * (x - 1)!`

Bei jeder Rekursion muß ein Abbruch definiert werden. Im Beispiel Fakultät dient als Abbruch: Wenn `x = 0` dann `x! = 1`.

Die iterative Definition lautet: `x! = 1 * 2 * 3 * ... * x`

Grundsätzlich kann man jede Rekursion auf eine Iteration zurückführen. Die Umformung ist aber nicht immer so einfach wie im Beispiel Fakultät. Grundsätzlich soll man Rekursionen nur dann einsetzen, wenn sie sich aufgrund der Aufgabenstellung anbietet bzw. die iterative Lösung schwierig zu erstellen ist. Rekursionen führen zwar häufig zu überraschend einfachen und kurzen Lösungen; die Nachteile liegen in der häufig längeren Ausführungszeit und in der Belegung des Stack-Speichers, in dem bei jedem

Rekursionsschritt lokale Variablen gespeichert werden. Der Stack-Speicher (Kellerspeicher) ist ein spezieller Speicherbereich, in dem Daten nach dem LIFO-Prinzip (last in, first out) abgelegt werden. In Turbo-Pascal kann man die Stack-Speichergröße mit dem Compilerbefehl "{$M ...}" selber in den Grenzen 1024 Byte und 65520 Byte festlegen; die Standardeinstellung beträgt 16384 Byte. Die Einstellung kann auch über den Menüpunkt "Option/Speicherauslegung" erfolgen. Weitere Details siehe folgendes Demo-Programm und auch Kap. 5.6.

Das Demo-Programm behandelt die einfache Aufgabe "*Summe aller Zahlen von 1 bis n*" einmal iterativ und einmal rekursiv wie folgt:

- iterativ: `Summe = 1 + 2 + 3 + ... + n`
- rekursiv: `Summe = n + Summe(n - 1)`
  Abbruch bei `n = 1`, mit `Summe = 1`

Das Programm ermittelt den Zeitbedarf für beide Methoden. Damit meßbare Zeiten entstehen, werden die Berechnungen in einer for-Schleife wiederholt. Im Mittel dürfte die iterative Methode dreimal schneller sein als die rekursive. Unschlagbar wäre die formelmäßige Berechnung nach Gauß: `Summe = n*(n + 1)/2`

```
program Pas11101;   { Demo:     Rekursion im Vergleich mit Iteration }
                    { Beispiel: Summe der Zahlen von 1 bis n         }
{$M 3411, 0, 655360 }
  { Zum Compilerbefehl "$M" für Memory:
    1. Parameter: Größe des Stack-Speichers in Byte.
                  Für Demo auf   3411 Byte gesetzt.
                  Bei Stackgröße 3411 und n = nMax = 361:
                     "Fehler 202: Stack-Überlauf"
                  Standardgröße: 16384
                  Maximalgröße:  65520
                  Minimalgröße:   1024
    2. Parameter: Minimale Größe des Heap-Speichers: Standard      0
    3. Parameter: Maximale Größe des Heap-Speichers: Standard 655360
                  Zum Heap-Speicher siehe Kap. 19: Dynam. Variablen
  }
uses
  CRT, DOS;
const
  Wiederholungen = 60000; { ggf. ändern }
  nMax           =   361;
    { Summe bis 361: 65341, nach Gauß: S = n*(n + 1)/2 }
    { Summe bis 362: 65703, ---> größer als Typ Word   }
var
  i, n:           Word;
  Summe_Iterat,
  Summe_Rekurs:   Word;
  Zeit,
  Zeit_Iterativ,
  Zeit_Rekursiv:  Real;

function Uhrzeit: Double;
var
  hh, mm, ss, ss100: Word;
```

```
begin
  GetTime(hh, mm, ss, ss100);
  Uhrzeit := 3600.0*hh + mm*60 + ss + ss100/100;
end;      { "3600.0" damit Real-Operation erzwingen, !!!!!!!!!!!!! }
          { sonst temp. Word-Überlauf ab 19.00 Uhr: 19*3600 = 68400 }

function Summe_Rekursiv(n: Word): Word;
begin
  if n = 1
     then Summe_Rekursiv := 1
     else Summe_Rekursiv := n + Summe_Rekursiv(n - 1);
end;

function Summe_Iterativ(n: Word): Word;
var
  i:    Word;
  Temp: Word;
begin
  Temp := 0;
  for i := 1 to n do
    Temp := Temp + i;
  Summe_Iterativ := Temp;
end;


begin
  TextBackground(Blue);
  TextColor(Yellow);
  ClrScr;

  WriteLn('Demo Rekursion <---> Iteration, Beispiel Summe n. ');
  WriteLn('Bei n >= ', nMax, ' Stacküberlauf. Ende mit n = 0');
  WriteLn;
  WriteLn('Bei kleinen n und schnellen Rechnern ist die Angabe');
  WriteLn('über das Zeitverhältnis von Rekursion zu Iteration');
  WriteLn('trotz ', Wiederholungen, ' Wiederholungen sehr ' +
          'ungenau oder auch ganz unsinnig.', #13#10);

  TextColor(White);
  repeat
    repeat
      Write('n = ');
      ReadLn(n); { Numerikeingabe nicht abgesichert! }
    until (n >= 0) and (n <= nMax);

    if n = 0 then Exit;  { >>>>>>>>>>>>>>>>>>>>>>>>>> }

    GotoXY(10, WhereY - 1);

    Zeit := Uhrzeit;
    for i := 1 to Wiederholungen do
      Summe_Iterat := Summe_Iterativ(n);
    Zeit_Iterativ := (Uhrzeit - Zeit)/Wiederholungen;

    Zeit := Uhrzeit;
    for i := 1 to Wiederholungen do
      Summe_Rekurs := Summe_Rekursiv(n);
```

```
    Zeit_Rekursiv := (Uhrzeit - Zeit)/Wiederholungen;

    WriteLn('Rekursiv: ', Summe_Rekursiv(n):5, ', ',
            'Iterativ: ', Summe_Iterativ(n):5, ', ',
            'Zeit_Rekursiv/Zeit_Iterativ: ',
             Zeit_Rekursiv/Zeit_Iterativ: 5:2);
  until n = 0;
end.
```

Die Ackermann-Funktion

Die Ackermann-Funktion hier kurz mit "F(m, n)", im späteren Programm aber mit "Pas11102.PAS" aber mit "AckermannFkt(m, n)" bezeichnet, ist ein Extrembeispiel für Rekursion. Sie wird häufig für Geschwindigkeitsvergleiche von Rechner-Systemen (Benchmark-Tests) benutzt.

Die Ackermann-Funktion ist für ganzzahlige und positive Werte von "m" und "n" wie folgt definiert:

```
F(0, n) = n + 1
F(m, 0) = F(m - 1, 1)
F(m, n) = F(m - 1, F(m, n - 1))
```

Durch die Beschränkung des Stack-Speichers auf max. 65520 Bytes (Turbo-Pascal unter MS-DOS) kann die Ackermann-Funktion nur für relativ kleine Werte von "n" und insbesondere von "m" berechnet werden. "F(4, 1)" und "F(5, 0)" können beispielsweise nicht mehr berechnet werden.

Die Rechenzeiten sind anfangs sehr klein; steigen dann aber schnell an. Bei Pentium-Rechnern mit Taktfrequenzen 166 MHz sind sie aber dennoch zu klein für eine präzise Messung, deshalb wird in dieser Programmversion darauf verzichtet. Die Begrenzung ist durch den Stack-Überlauf gegeben. Dennoch: Bei einem PC mit dem Prozessor i80386 und einer Taktfrequenz von 16 MHz ergaben sich folgende Rechenzeiten (in Klammern ist zusätzlich die Anzahl der Rekursionsaufrufe i angegeben):

Ackermann(m, n)	Rechenzeit	Anzahl Rekursionsaufrufe
F(3, 0)	< 0.5 s	15
F(3, 1)	< 0.5 s	106
F(3, 2)	< 0.5 s	541
F(3, 3)	< 0.5 s	2 432
F(3, 4)	< 0.5 s	10 307
F(3, 5)	<. 0.5 s	42 438
F(3, 6)	ca. 2 s	172 233
F(3, 7)	ca. 8 s	693 964
F(3, 8)	ca. 33 s	2 785 999
F(3, 9)	ca. 133 s	11 643 370

Auszug aus der Wertetabelle der Ackermann-Funktion. Bei den mit "?" markierten Feldern konnte die Ackermann-Funktion wegen "Fehler 202: Stack-Überlauf" nicht mehr berechnet werden.

	n = 0	n = 1	n = 2	n = 3	n = 4	n = 5	n = 6	n = 7	n = 8	n = 9
m = 0	1	2	3	4	5	6	7	8	9	10
m = 1	2	3	4	5	6	7	8	9	10	11
m = 2	3	5	7	9	11	13	15	17	19	21
m = 3	5	13	29	61	125	253	509	1021	2045	4093
m = 4	13	?	?	?	?	?	?	?	?	?
m = 5	?	?	?	?	?		?	?	?	?

```
program Pas11102;   { Ackermann-Funktion, Dr. K. Haller, 77120495 }
{$M 65520, 0, 655360 }

{ Zu: $M 65520, 0, 655360 }
{ Stack-Speicher auf Größtwert 65520 setzen              }
{ Dennoch bei AckermannFkt(4, 1) und AckermannFkt(5, 0)  }
{ "Fehler 202: Stack-Überlauf"                           }

uses
  DOS, CRT;

var
  m, n:      Word;
  Ackermann: Word;
  i:         LongInt;  { Für Anzahl der Rekursionsaufrufe }
  Ch:        Char;
  Anzeige_der_Rekursionsschritte: Boolean;

function AckermannFkt(m, n: Word): Word; { ----------------------+}
begin                                                            {¦}
  if Anzeige_der_Rekursionsschritte then  { Nur für Demo, }      {¦}
     begin                                { kostet viel Zeit } {¦}
       Inc(i);                                                   {¦}
       Write(i:6, ': Ack(', m:2, ', ', n:2, ')');                {¦}
     end;                                                        {¦}
  ;                                                              {¦}
  if m = 0                                                       {¦}
     then AckermannFkt := n + 1                                  {¦}
     else if n = 0                                               {¦}
             then AckermannFkt := AckermannFkt(m - 1, 1)         {¦}
             else AckermannFkt :=                                {¦}
                  AckermannFkt(m - 1, AckermannFkt(m, n - 1)); {¦}
end; { ----------------------------------------------------------+}


begin
  repeat
    Textbackground(Blue);
    TextColor(Yellow);
    ClrScr;

    repeat
      GotoXY(20, 2);
      Write('********** Ackermann-Funktion **********  ');
      GotoXY(20, 5);
      Write('Eingabe m, n  (max. 9, Ende nach "0 0"):  ');
      ReadLn(m, n);  { Keine Numerikabsicherung! }
```

```
    until (m in [0..9]) and (n in [0..9]);
    repeat
      GotoXY(20, 8);
      Write('Unter Umständen sehr lange Zeiten! ');
      GotoXY(20, 7);
      Write('Mit Anzeige der Rekursionsschritte (j/n): n');
      GotoXY(WhereX - 1, WhereY);
      Ch := ReadKey;
      if Ch = #13
         then Ch := 'n';
    until UpCase(Ch) in ['J', 'N'];
    WriteLn(Ch); WriteLn;
    if UpCase(Ch) = 'J'
       then Anzeige_der_Rekursionsschritte := True
       else Anzeige_der_Rekursionsschritte := False;
    if Anzeige_der_Rekursionsschritte then
       begin
         Window(3, 9, 78, 21);       { Kleineres Fenster }
         TextBackground(Cyan);       { und andere Farbe  }
         ClrScr;
       end;
    i := 0;  { Zähler für Rekursionsschritte }
   {+-------------------------------------+}
   {¦} Ackermann := AckermannFkt(m, n);  {¦} { Aufruf der         }
   {+-------------------------------------+} { Ackermann-Funktion }
    if Anzeige_der_Rekursionsschritte then
       begin
         Window(1, 1, 80, 25);  { Wieder volles Fenster }
         TextBackground(Blue);
       end;
    GotoXY(10, 23);
    Write('Ackermann(', m, ', ', n, ') = ' +
          'Ackermann(0, ', Ackermann - 1, ') = ', Ackermann);
    Write(#7);
    repeat
    until ReadKey <> '';
  until (m = 0) and (n = 0);
end.
```

11.11 Abbruch einer Routine mit Exit

Mit der Standardprozedur "*Exit*" kann jede Routine (oder auch das Hauptprogramm) voreitig beendet werden. Beim Abbruch einer Routine wird das Programm an der aufrufenden Stelle fortgesetzt; ein Hauptprogramm logischerweise aber beendet. Der Abbruch kann ohne (nur bei Tests sinnvoll) oder mit Bedingung erfolgen. Es empfiehlt sich, die Ausstiegsstelle mit Kommentar hervorzuheben. Bei Funktionen achte man darauf, daß dem Funktionsbezeichner vor "*Exit*" ein Wert zugewiesen wird. Weiter achte

aber darauf, daß vor "*Exit*" evtl. geöffnete Dateien geschlossen werden; Datenverlust könnte die Folge sein!

Formate:

```
...
Exit;  { >>>>>>>>>>> }
...
if Bedingung
   then Exit;  { >>>>>>>>> }
...
```

Grundsätzlich kann man mit if/then-Konstruktionen "*Exit" v*ermeiden; in manchen Fällen kann die maßvolle Verwendung von "*Exit*" zur Programmklarheit beitragen.

11.12 inline-Prozeduren

Kurze Folgen von Maschinencodes, wie z.B. Interrupt-Aufrufe, können mit "*inline*" (reserviertes Wort) direkt in den Pascal-Quelltext eingefügt werden. Üblicherweise geschieht dieses in Hex-Schreibweise, dann aber mit Pascal-Hex-Vorsatzzeichen `"$".` Dezimale Eingabe ist möglich, aber nicht üblich. Möglich ist auch die Eingabe über Konstantenbezeichner. Die Maschinencodefolge ist in runde Klammern zu setzen. Als Trennzeichen zwischen den Bytes ist das Zeichen "`/`" zu verwenden. Inline-Prozeduren lassen sich nicht als Parameter an andere Routinen übergeben!

Die Aufnahme von Maschinencodes in den Quelltext schränkt dessen Portierbarkeit ein. Der Quelltext kann ohne Anpassung nur auf Rechnern mit Prozessoren der gleichen oder kompatiblen Prozessorfamilie, gleichem Betriebssystem und gleichem oder kompatiblen Compiler verwendet werden. Durch die jetzige Vorherrschaft der Intel-Prozessoren, des Betriebssystems MS-DOS und des Turbo-Pascal-Compilers ist die Einschränkung nicht mehr so von Bedeutung wie früher.

Das folgende Demo-Programm enthält die inline-Prozedur "Hardcopy". Darin wird lediglich der MS-DOS-Interrupt h05 (hex 05) aufgerufen, der eine "Hardcopy" des (Text-) Bildschirms auf dem Drucker erzeugt. Der Interrupt h05 wird auch durch Betätigen der Taste "PrtScr" (Print Screen, bei eingedeutschten Tastaturen Taste "Druck") ausgelöst. Die inline-Prozedur enthält lediglich zwei Bytes: $CD und $05. Das erste Byte steht für Interrupt, das zweite Byte für die Interrupt-Nummer. Eine weitere Behandlung der Maschinencodes würde in die fast unendlichen Abgründe der (prozessor-spezifischen) Assemblerprogrammierung führen. Die späteren Kapitel "Systemnahe Programmierung" und "Betriebssystem" enthalten weitere Beispiele.

```
program Pas11121;    { Demo: inline-Prozeduren }
      { Kurze Folgen von Maschinencodes, wie z.B.Interrupt Aufrufe)
        können mit "inline" direkt in den Pascal-Quelltext eingefügt
        werden; üblicherweise in Hex-Schreibweise, dann aber mit dem
        Pascal-Hex-Vorsatzzeichen "$". Dezimale Eingabe ist möglich,
        aber nicht üblich. Die Maschinencodefolge ist in runde
```

```
        Klammern zu setzen. Als Trennzeichen zwischen den Bytes ist
        das Zeichen "/" zu verwenden. Inline-Prozeduren lassen sich
        nicht als Parameter an andere Routinen übergeben!
      }

uses
  CRT;
var
  Ch: Char;

procedure Hardcopy;
begin
  inline ($CD/$05);     { Hier nur zwei Byte inline-Code }
  {  MS-DOS-Interrupt h05: Bildschirminhalt an Drucker   }
  {  In Assembler:  int 5   (2 Byte Maschinencode: CD 05 }
  {  In Hex-Maschinencode:  CD (dez 205) für "int"       }
  {                         05 (dez 005) für Nr "h05"    }
end;

begin
  ClrScr;

  WriteLn;
  Writeln('          ╔═══════════════════════════╗');
  WriteLn('          ║                           ║');
  WriteLn('          ║   Hardcopy (j/n):         ║');
  WriteLn('          ║                           ║');
  Writeln('          ╚═══════════════════════════╝');

  GotoXY(30, WhereY - 3);

  repeat
    Ch := ReadKey;
  until UpCase(Ch) in ['J', 'N'];

  Write(Ch);

  if UpCase(Ch) = 'J'
     then Hardcopy;
end.
```

11.13 external-Prozeduren einfügen

Mit "**inline**" (reserviertes Wort) kann man kurze Befehlsfolgen in Maschinensprache direkt in den Pascal-Quelltext einfügen. Für längere Folgen ist das Verfahren zu umständlich. Größere Prozeduren (auch Funktionen) erstellt man zweckmäßigerweise mit einem Assembler (MASM von Microsoft oder TASM von Borland). Nach dem Assemblieren und Linken kann man das Maschinenprogramm mit "*external*" (ebenfalls reserviertes Wort) durch Aufruf seines Namens in das Pascal-Programm einbinden. Datenaustausch ist möglich. Details siehe Turbo-Pascal-Handbuch.

11.14 Assembler-Anweisungen einfügen

Mit "**asm**" (reserviertes Wort) kann man direkt Assembler-Anweiszngen in den Pascal-Quelltext einfügen. Turbo-Pascal assembliert diese Anweisungen. Die Assembler-Anweisungen werden mit "**asm**" eingeleitet und mit "**end;**" beendet. Die Syntax der Assembler-Anweisungen richtet sich nach Turbo-Assembler von Borland. Details siehe Turbo-Pascal-Handbuch.

11.15 Anwendungsbeispiel: Binomialverteilung

Das spätere Programm "Pas11151.PAS" berechnet die Binomialverteilung.

Die Binomialverteilung kann man mit dem "Urnenmodell" veranschaulichen:

In einer Urne befinden sich schwarze und rote Kugeln. Der Anteil der roten Kugeln ist p, z.B. $p = 0.3 = 30\%$. Nun werden zufällig n Kugeln gezogen, wobei die gezogenen Kugeln anschließend wieder in die Urne zurückgelegt werden, damit p konstant bleibt.

Die Wahrscheinlichkeit bei n-Ziehungen k-rote Kugeln zu ziehen, ist binomialverteilt. Die Wahrscheinlichkeit wird mit $B(n, k, p)$ bezeichnet. Die Berechnung der Binomialverteilung kann nach zwei Methoden erfolgen:

1. **Methode**: Berechnung der Wahrscheinlichkeit B(n, k, p) mit Hilfe der Binomial-Koeffizienten über die Newton-Formel:

   (1.1) $$B(n,k,p) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

   Die Binomial-Koeffizienten "n über k" werden nach folgender Formel berechnet:

   (1.2) $$\binom{n}{k} = \frac{(n-k+1)\cdot(n-k+2)\cdot(n-k+3)\cdot\ldots\cdot n}{1\cdot 2\cdot 3\cdot\ldots\cdot k} = \frac{Zaehler}{Nenner}$$

   Die Berechnung der Binomialkoeffizienten mit dem Datentyp LongInt führt bei der vorliegenden Programmierung bei $n > 29$ zum temporären Überlauf des LongInt-Bereiches und damit zu einem Total-Fehler. Wenn man die (genaue) LongInt-Berechnung der Binomialkoeffizienten durch eine Real-Berechnung ersetzt, tritt ein Real-Overflow erst bei $n > 49$ ein. Bei der Berechnung der Terme p^k und $(1 - p)^{(n - k)}$ können zudem noch Unterlauf-Probleme auftreten, die ebenfalls einen Total-Fehler zur Folge haben.

2. **Methode**: Berechnung der Wahrscheinlichkeit $B(n, k, p)$ mit Hilfe folgender Rekursionsformel, die aber auch iterativ ausgewertet werden kann. Die Binomialkoeffizienten werden bei der Methode 2 nicht benötigt. Somit treten die bei der Methode 1 beschriebenen Überlauf-Probleme nicht auf, dafür aber Unterlauf-Probleme bei der Berechnung des Anfangswertes, siehe unten.

(2.1) $$B(n,k,p) = B(n,k-1,p) \cdot \frac{n-k+1}{k} \cdot \frac{p}{1-p}$$

und dem Anfangswert für $k = 0$:

(2.2) $$B(n,0,p) = (1-p)^n$$

Bei großen Werten von **n** (und ggf. auch von **p**) wird der Anfangswert nach dieser Formel sehr klein und kann zum Unterlauf führen,der ebenfalls einen Total-Fehler zur Folge hat. Man teste bei $p = 0.9$ mit $n = 38$ (o.k) und $n = 39$ (Total-Fehler).

Zur Vermeidung der numerischen Probleme benutzt man in der Praxis in den Grenzfällen Näherungsformeln, wie die Laplace-Formel für große n oder die Poisson-Formel für kleine Werte von p und k/n. Siehe Statistik-Literatur.

In diesem Programm wird davon kein Gebrauch gemacht. Es wird lediglich eine Fehlermeldung ausgegeben, wenn "SummeB" am Ende der Summation von $k = 0$ bis n um einen vorgegebenen Fehlerbetrag von den theoretischen Endwert "1.0" abweicht.

Bei Verwendung der Coprozessor-Datentypen "Double" oder "Extended" statt "Real" und "Comp" statt "LongInt" treten Total-Fehler erst wesentlich später auf. Wenn der Coprozessor nicht existiert, muß er auf Kosten der Rechenzeit softwaremäßig emuliert werden. Zur Übung: Compilerschalter " {$N+,E+}" für Tests verwenden und Datentypen variieren.

```
{$N+,E+   Mathematischen Coprozessor benutzen, ggf. Emulation }
program Pas11151; { Turbo-Pascal, Dr. K. Haller, 77060597 ************

uses
  CRT;

type
  TReal         = Double; { Man teste mit "Single", "Real", }
                          {        "Double" und "Extended"  }
  Verfahrenstyp = (Formel1_Integer,  Formel1_TReal,
                   Formel2_Iterativ, Formel2_Rekursiv);

var               { ---- Globale Variablen: ------------------------- }
  p:      TReal; { Anteil der roten Kugeln in der Urne. p = 0...1 }
  n,             { Anzahl der gezogenen Kugeln }
  k:      Byte;  { Anzahl der roten Kugeln unter den gezogenen Kugeln}
  B:      TReal; { Wahrscheinlichkeit, daß sich unter den n-gezogenen
                   Kugeln k-rote Kugel befinden = f(n, k, p)         }
  SummeB: TReal; { Summe der Wahrscheinlichkeiten }
  Ch:     Char;
  Verfahren:  Verfahrenstyp;

procedure WriteXY(Spalte, Zeile: Byte; Meldung: string);
begin
  GotoXY(Spalte, Zeile);
  Write(Meldung);
end;
```

```
function TReal_Hoch_Integer(x: TReal; n: Byte): TReal;  { ---------- }
  { Eigene Funktion für die in Pascal fehlende Potenzfunktion x^n,
    wobei "x" ein Realtyp und "n" ein Bytetyp ist }
var
  Temp: TReal;
begin
  Temp := 1.0;
  while n > 0 do
    begin
      while not Odd(n) do  { ... solange geradzahlig True }
        begin
          n := n div 2;
          x := Sqr(x)
        end;
      Dec(n);
      Temp := x * Temp;
    end;
  TReal_Hoch_Integer := Temp;
end; { ------------------------------------------------------------- }


function Binomialkoeffizient_Int(n, k: Byte): LongInt; { ----------- }
var
  i:      Byte;
  Zaehler,         { Mit "Integer" u. "Word" statt "LongInt" }
  Nenner: LongInt; { noch früher temporärer Overflow         }
begin
  if k > (n div 2)
     then k := n - k;   { Symmetrie-Eigenschaft }
  Zaehler := 1;
  Nenner  := 1;
  for i := 1 to k do
    begin
      Zaehler := Zaehler * (n - k + i);
      Nenner  := Nenner  * i;
      if Zaehler mod Nenner = 0 then
         begin                              { Mit diesem "Zwischen- }
           Zaehler := Zaehler div Nenner; { kürzen" kommt man bei }
           Nenner  := 1;                  { "LongInt" bis n = 29, }
         end;                             { sonst nur bis n = 17  }
     end;
  Binomialkoeffizient_Int := Zaehler div Nenner;
end; { ------------------------------------------------------------- }


function Binomialkoeffizient_TReal(n, k: Byte): TReal; { ----------- }
var
  Temp:  TReal;
  i:     Byte;
begin
  if k > (n div 2)
     then k := n - k;   { Symmetrie-Eigenschaft }
  Temp  := 1;
  for i := n - k + 1  to  n do Temp := Temp * i;  { Zähler }
  for i := 1          to  k do Temp := Temp / i;
```

```
  Binomialkoeffizient_TReal := Temp;
end; { ------------------------------------------------------------ }

function Binom_Rekursiv(n, k: Byte; p: TReal): TReal; { ------------ }
begin
  if k = 0
     then Binom_Rekursiv := TReal_Hoch_Integer((1 - p), n)
     else Binom_Rekursiv := Binom_Rekursiv(n, k - 1, p) *
                                (n - k + 1) / k * p / (1 - p);
end; { ------------------------------------------------------------ }

procedure Binom_Tabelle(n: Byte; p: TReal; { ---------------------- }
                        Verfahren: Verfahrenstyp);
var
  BiKoeff_Int:   LongInt;
  BiKoeff_TReal: TReal;

begin
  ClrScr;
  WriteXY(4, 1,
    '-------------- Tabelle der Binomialverteilung ------------');
  GotoXY(4, 2);

  case Verfahren of
    Formel1_Integer:
      Write('Newton-Formel. Binomialkoeffizient "LongInt".  ');
    Formel1_TReal:
      Write('Newton-Formel. Binomialkoeffizient "TReal".  ');
    Formel2_Iterativ:
      Write('Rekursionsformel. Iterative Berechnung.  ');
    Formel2_Rekursiv:
      Write('Rekursionsformel. Rekursive Berechnung.  ');
  end;

  WriteLn('n = ', n, '  p = ', p:6:4);

  WriteLn('   ',
    '+-----------------------------------------------------------+');
  WriteLn('   ',
    '¦      ¦    Binomial-     ¦  Wahrscheinl. ¦  Summe von    ¦');
  WriteLn('   ',
    '¦   k  ¦    Koeffizient   ¦  B(n, k, p)   ¦  B(n, k, p)   ¦');
  WriteLn('   ',
    '¦------+-----------------+---------------+---------------¦');

  SummeB := 0.0;

  for k := 0 to n do
    begin
      case Verfahren of   { ------------------------ }
        Formel1_Integer:
          begin
            BiKoeff_Int := Binomialkoeffizient_Int( n, k);
            B := BiKoeff_Int * TReal_Hoch_Integer(p, k) *
                 TReal_Hoch_Integer((1 - p), (n - k));
          end;
        Formel1_TReal:
          begin
            BiKoeff_TReal := Binomialkoeffizient_TReal(n, k);
            B := BiKoeff_TReal * TReal_Hoch_Integer(p, k) *
```

```
                          TReal_Hoch_Integer((1 - p), (n - k));
          end;
        Formel2_Iterativ:
          if k = 0
             then B := TReal_Hoch_Integer((1 - p), n)
             else B := B * (n - k + 1) / k * p / (1 - p);
        Formel2_Rekursiv:  B := Binom_Rekursiv(n, k, p);
      end; { ---- von "case Verfahren of" ------------ }

      SummeB := SummeB + B;

      case Verfahren of
        Formel1_Integer:  WriteLn('   ¦', k:4, '  ¦', BiKoeff_Int:16,
                          '  ¦',  B:13:9, '  ¦', SummeB:13:9, '  ¦');
        Formel1_TReal:    WriteLn('   ¦', k:4, '  ¦',
                          BiKoeff_TReal:16:0,'  ¦',  B:13:9,
                          '  ¦', SummeB:13:9, '  ¦');
        Formel2_Iterativ,
        Formel2_Rekursiv: WriteLn('   ¦', k:4,
                          '¦  -------------  ¦':22, B:13:9,
                          '  ¦', SummeB:13:9, '  ¦');
      end;
      if k = n
         then WriteXY(61, WhereY - 1, '<--- Soll: 1.0' + #13#10);

      if WhereY = 25 then
         begin
           WriteXY(25, 25, 'Weiter mit Tastendruck ... ');
           repeat
           until ReadKey <> '';
           ClrScr;
         end;
    end;   { Ende der Schleife "for k := ...."  }

  WriteLn('   ',
    '+-----------------------------------------------------------+');

  if Abs(SummeB - 1.0) > 1e-4 { Fehlermeldung bei     }
     then WriteLn(#7,         { numerischen Problemen }
          '   ***************** Fehler. SummeB <> 1.0 **************');
end; { ------------ Ende "Binom_Tabelle" -------------------------- }


begin { =========== Hauptprogramm ================================== }
  TextBackGround(Blue);

  repeat
    ClrScr;
    TextColor(Yellow);
    WriteXY(10, 2, 'Berechnung der Binomialverteilung');
    WriteXY(10, 4, '--------------------------------' +
                   '--------------------------------');
    WriteXY(10, 5, '1    Newton-Formel, '    +
                   'Bionomialkoeffizienten mit LongInt-Berechnung');
    WriteXY(10, 6, '2    Newton-Formel, '    +
                   'Bionomialkoeffizienten mit TReal-Berechnung  ');
    WriteXY(10, 7, '3    Rekursionsformel, ' +
```

```
                    'Berechnung rekursiv');
    WriteXY(10, 8, '4    Rekursionsformel, ' +
                    'Berechnung iterativ');
    WriteXY(10, 9, 'Esc  Programm beenden');
    WriteXY(10,10, '--------------------------------' +
                    '--------------------------------');
    GotoXY( 10, 11);
    repeat
      Ch := ReadKey;
    until Ch in ['1'..'4', #27];
    Write(Ch);

    TextColor(White);

    case Ch of
      #27: Halt; { >>>>>>>>>>> Programm beenden >>>>>>>>> }
      '1': Verfahren := Formel1_Integer;
      '2': Verfahren := Formel1_TReal;
      '3': Verfahren := Formel2_Iterativ;
      '4': Verfahren := Formel2_Rekursiv;
    end;

    WriteXY(10, 12, 'Binomialverteilung. n = 0 ... (50),  ' +
                    'p >= 0.0 ...< 1.0 ');
    WriteXY(10, 13, 'Eingabe n, p: ');
    repeat
      GotoXY(24, 13); ClrEoL;
      GotoXY(24, 13);
      {$I- Automatische I/O-Prüfung aus }
      ReadLn(n, p);
      {$I+ I/O-Prüfung ein }
    until (IOResult = 0) and (n >= 0) and
          (p >= 0.0)     and (p < 1.0);

    if (Verfahren = Formel1_Integer) and (n > 29)
       then Write(#7);   { Warnung: LongInt-Überlauf bei  }
                         { Berechnung Binomialkoeffizient }

    Binom_Tabelle(n, p, Verfahren);

    WriteXY(25, 25, 'Zum Menü mit Tastendruck ... ');
    repeat
    until ReadKey <> '';

  until False; { Pseudo-Endlosschleife, Ende weiter oben mit "Halt" }
end. { ========================================================== }
```

11.16 Anwendungsbeispiel: Numerische Integration nach Simpson

```
{$N+   Compilerschalter "$N+": Mathematischen Coprozessor benutzen }
program Pas11161; { Integral nach Simpson. Dr. K. Haller, 87060490 }
                  { Mit Übergabe einer Funktion an eine Funktion.  }
```

```
uses
  CRT;

type
  TReal        = Double;   { Für Tests: "Real", "Single", "Double" }
  Funktionstyp = function(x: TReal): TReal;
  { Funktionstyp-Deklaration für Übergabe an Routine notwendig     }

{$F+   FAR-Aufruf notwendig, wenn Funktion als Parameter    }
function f1_x(x: TReal): TReal; { an Routine übergeben wird }
begin
  f1_x := Sin(x);
     { Für Test: Bei f1_x := Sin(x) und Intervall [0..Pi]:  }
     { Integral mathematisch exakt = 2.0                    }
end; {$F-  Zurücksetzen auf NEAR-Aufrufe (Standard)         }


{$F+   FAR-Aufruf notwendig, wenn Funktion als Parameter    }
function f2_x(x: TReal): TReal; { an Routine übergeben wird }
begin
  f2_x := Sqr(x);
     { Für Test: Bei f2_x := Sqr(x) und Intervall [0..1]:   }
     { Integral mathematisch exakt = 1/3 = 0.3333...        }
end; {$F-  Zurücksetzen auf NEAR-Aufrufe (Standard)         }


{$F+   FAR-Aufruf notwendig, wenn Funktion als Parameter    }
function f3_x(x: TReal): TReal; { an Routine übergeben wird }
begin
  f3_x := Exp(-Sqr(x)/2)/Sqrt(2*Pi);
     { f3_x = Gauß'sche Fehlerfunktion (nicht integrierbar) }
     { Bei Intervall [0..1] nach Tabelle: 0.3413...         }
     { Hinweis: Oft ist das Gauß-Integral von minus-unend-  }
     { lich bis x gemeint. In diesem Fall ist zum Integral- }
     { wert noch der Wert 0.5 zu addieren. Im Beispiel wird }
     { dann das Ergebnis zu:  0.5 + 0.3413... = 0.8413...   }
end; {$F-  Zurücksetzen auf NEAR-Aufrufe (Standard)         }


{ ================================================================ }
function IntegralSimpson(f_x:                Funktionstyp;
                         Untergrenze,
                         Obergrenze,
                         RelativeGenauigkeit: TReal;
                         Testanzeige:        Boolean): TReal;
  { Zu "RelativeGenauigkeit": Zu strenge Forderungen }
  { können zu langen Rechenzeiten und numerischen    }
  { Instabilitäten führen.     }
const
  StreifenMax = 4096;       { Wenn "Streifen" größer, dann Abbruch.  }
                            { Man teste mit 1024, 2048, 4096, 8192,  }
                            { 16384 und 32768 Streifen               }
var
  i, Streifen:    Word;
  x, DeltaX,
  SummeGrenzen,
```

```
  SummeGerade,
  SummeUngerade,
  IntegralAlt,
  IntegralNeu:    TReal;

begin
  Streifen      := 2;
  DeltaX        := (Obergrenze - Untergrenze) / Streifen;

  SummeGrenzen  := f_x(Untergrenze) + f_x(Obergrenze);
  SummeGerade   := f_x(Untergrenze  + DeltaX);
  SummeUngerade := 0.0;

  IntegralNeu   := (SummeGrenzen + 4.0 * SummeGerade) * DeltaX / 3.0;

  if Testanzeige then
     begin
       WriteLn;
       WriteLn('+-- Testausgabe in "function ' +
               'IntegralSimpson()" --+');
       WriteLn(Streifen:8, ':   ', IntegralNeu);
     end;

  repeat
    Streifen := Streifen * 2;

    if Streifen > StreifenMax then
       begin
         WriteLn(#7, 'Anzahl der Streifen: ', Streifen);
         WriteLn('Der Wert ist größer als der vorgegebene ' +
                 'Grenzwert: ', StreifenMax);
         WriteLn('Deshalb eventuell keine genügend genaue Lösung ' +
                 'zu erwarten.');
         WriteLn('Wenn möglich, Genauigkeit reduzieren und/oder '  +
                 'Streifenzahl erhöhen.');
         Write(  'Abbruch nach Taste "Esc": ');
         repeat                { Noch besser: Vorher Tastatur- }
         until ReadKey = #27; {               puffer leeren    }
         Halt;     { >>>>> Abbruch der Sitzung >>>> }
       end;

    DeltaX        := (Obergrenze - Untergrenze) / Streifen;

    SummeUngerade := SummeUngerade + SummeGerade;
    SummeGerade   := 0.0;

    for i := 1 to (Streifen div 2) do
      begin
        x           := Untergrenze + DeltaX * (2*i - 1);
        SummeGerade := SummeGerade + f_x(x);
      end;

    IntegralAlt := IntegralNeu;
    IntegralNeu := (SummeGrenzen + 4*SummeGerade + 2*SummeUngerade)
                   *DeltaX/3;

    if Testanzeige
       then WriteLn(Streifen:8, ':   ', IntegralNeu);

  until Abs(IntegralNeu - IntegralAlt)  <=
```

```
        Abs(IntegralNeu * RelativeGenauigkeit);

  IntegralSimpson := IntegralNeu;

  if Testanzeige
     then WriteLn('+------------- Ende der Test' +
                  'ausgabe -------------+');

end;     { von "function IntegralSimpson(...)" }
{ ================================================================ }

begin    { ---------------- Hauptprogramm für Test ---------------- }
  ClrScr;

  TextColor(LightGray);
  WriteLn('Beispiel1: Der Integralwert: ',
          IntegralSimpson(f1_x, 0.0, Pi,  1.0E-11, True));
  WriteLn;

  TextColor(Yellow);
  WriteLn('Beispiel2: Der Integralwert: ',
          IntegralSimpson(f2_x, 0.0, 1.0, 1.0E-11, False));
  WriteLn;

  TextColor(LightCyan);
  WriteLn('Beispiel3: Der Integralwert: ',
          IntegralSimpson(f3_x, 0.0, 1.0, 1.0E-11, True));

  repeat
  until ReadKey <> '';
end.
```

11.17 Erweiterte Syntax: Funktionen wie Prozeduren-verwenden

Ab Turbo-Pascal 6.0 steht über den Menüpunkt "Optionen/Compiler.../Erweiterte Syntax" oder über den äquivalenten Compilerschalter **{$X+}** die sogenannte erweiterte Syntax zur Verfügung, die es ohne besondere Vorkehrungen erlaubt, selbstdefinierte Funktionen und Standardfunktionen, soweit diese nicht aus der Unit SYSTEM (darin z.B. *Sin*(), *Ln*(), *Length*() usw.) stammen, optional auch wie Prozeduren zu verwenden, wobei ganz einfach der Rückgabewert der Funktion ignoriert wird. Diese Möglichkeit ist von gewissem praktischen Interesse, widerspricht aber ganz der Pascal-Strenge; der arme Nikolaus Wirth! Das folgende Programm demonstriert die erweiterte Syntax und enthält weitere Informationen dazu. Bezüglich der Übergabe von Parametern an Funktionen mit der erweiterter Syntax (ohne oder mit Parametern, Parameter mit Wert oder Adresse besteht kein Unterschied zu "normalen" Funktionen.

Für die DV-Grundausbildung im Studiengang Druckereitechnik wird die **erweiterte Syntax** mit Ausnahme der ausdrücklich erklärten Anwendungen bei nullterminierten

Strings (siehe Kap. 14) **nicht zugelassen**. Entsprechende Programmkonstruktionen werden als Fehler gewertet.

```
{$X+}
program Pas11171;  { Demo "Erweiterte Syntax". kha, 77060497          }
 { Mit globalem Compilerschalter "$X+" oder über die Einstellung mit }
 { dem Menüpunkt "Option/Compiler.../Erweiterte Syntax" können ab     }
 { Turbo-Pascal 6.0 selbstdefinierte Funktionen und Standard-         }
 { funktionen (soweit sie nicht in der Unit Standard-SYSTEM dekla-    }
 { riert sind) auch als Prozeduren verwendet werden, ohne daß es      }
 { besonderer Vorkehrungen bedarf. Es wird in diesem Fall einfach     }
 { der Rückwert der Funktion ignoriert. Der arme Nikolaus Wirth!      }
uses
  CRT;

function Test(j: Byte): Integer;
begin
  Write(#7, j, 'Huber ');
  Test := 4711;
end;

begin
  ClrScr;
  Test(1);        { Die Funktion "Test" als Prozedur,       }
  Test(2);        { nur mit "Erweiterter Syntax" möglich.   }
  Write(Test(3), 'Meier');  { "Test" als "normale" Funktion }
  { Die Bildschirmausgabe: "1Huber 2Huber 3Huber 4711Meier" }
  ReadKey; { Die Standardfunktion "ReadKey" kann mit "Erweiterter    }
           { Syntax" so wie hier auch als Prozedur verwendet werden, }
           { da diese Funktion in der Unit "CRT" deklariert ist.     }
end.
```