Teil V: Prozeduren – Unterprogramme

1. **Unterprogramme und Parametrisierung**
2. **Prozeduren und Funktionen in Modula-2**
3. **Blockstruktur und Schachtelung**
4. **Unterprogramm-Techniken**

Unterprogramme und Parametrisierung

Allgemeines zu Prozeduren

Einordnung

1. Prozeduren dienen der Strukturierung / Vereinfachung
   - Gliederung (i.S. der **schrittweisen Verfeinerung**)
   - Reduktion des Schreibaufwands (bei **Mehrfachbenutzung**)

   → die jeweils veränderbaren Elemente („slots") müssen bei der Verwendung des Programms durch aktuelle Werte belegt werden

   aktuelle Parameter → formale Parameter

   Wichtig : Festlegung der Ersetzungsvorschriften !

2. Konzept zur **Abstraktion** in Algorithmen :

   Benutzersicht dessen, **WAS** zum Zeitpunkt des Aufrufs ausgeführt wird, **nicht** jedoch **WIE** !

Grundstruktur und Elemente

1. **Name**
2. **Liste von Parametern (Parameter-Namen)**
3. **Kopf und Rumpf**
   - **Kopf :** **Vereinbarungen, Deklarationen**
   - **Rumpf :** **Anweisungen**
4. **Aufrufende Umgebung**
   Variablen ausserhalb des Rumpfes, die zur Ausführungszeit der Prozedur verwendet und auch verändert werden können (→ **Seiteneffekte**)

Schema (Elemente) :

```
PROCEDURE name(parameter-list)
   Kopf (Deklarationen)
   Rumpf (Programmtext)
END name.
```

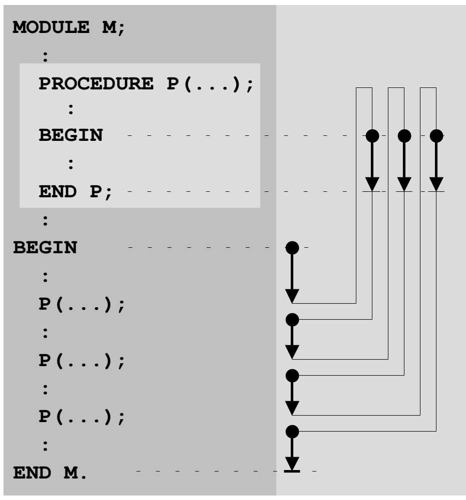
Ausführung wie ein eigenes Programm !

Ausführung einer Prozedur

Deklaration

```
MODULE M;
  :
  PROCEDURE P(...);
    :
  BEGIN
    :
  END P;
  :
BEGIN
  :
  P(...);
  :
  P(...);
  :
  P(...);
  :
END M.
```

Kontrollfluss



Übergabemechanismen für Parameter

Ersetzungsregeln für Parameter

1. Die **Ersetzung** der formalen Parameter durch aktuelle Parameter erfolgt zum Zeitpunkt des **Aufrufs** einer Prozedur (→ formale Parameter = Platzhalter)

Deklaration :

```
PROCEDURE foo(A, B, C);
   summe :  INTEGER;
BEGIN
   summe := A + B + C
END foo.
```

Aufruf :

```
    :
foo(a, b, c);

foo(1, 2, 3);
    :
```

2. Die **Prinzipien** der Ersetzung folgen festgelegten **Regeln**, die je nach Programmiersprache unterschiedlich sein können (→ folgt nach !)

Zuordnung aktueller zu formalen Parametern (Substitution)

1. Entsprechend der **Reihenfolge des Auftretens** in der Parameterliste

Deklaration :

1. 2. 3.

```
PROCEDURE foo(A, B, C);
   summe :  INTEGER;
BEGIN
   summe := A + B + C
END foo.
```

Aufruf :

```
    :
foo(a, b, c);
    :
foo(1, 2, 3);
    :
```

Mechanismus : z.B. in Pascal, Modula 2, etc.
Problem : **„Default“**-Parameter jeweils nur für das Ende der Parameter-Liste !

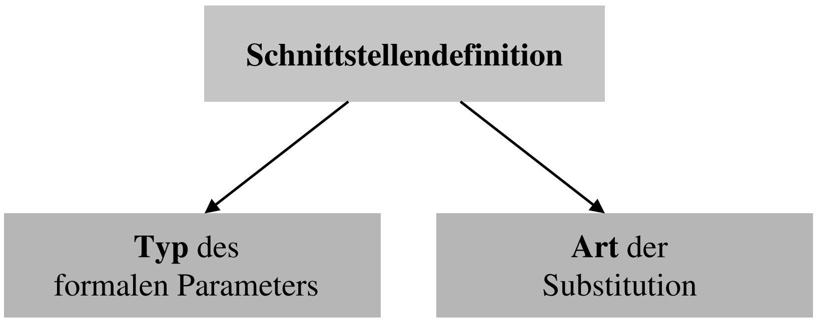
2. Via **Namens-Assoziation**

```
foo(a ⇒ A, b ⇒ B, c ⇒ C);
```

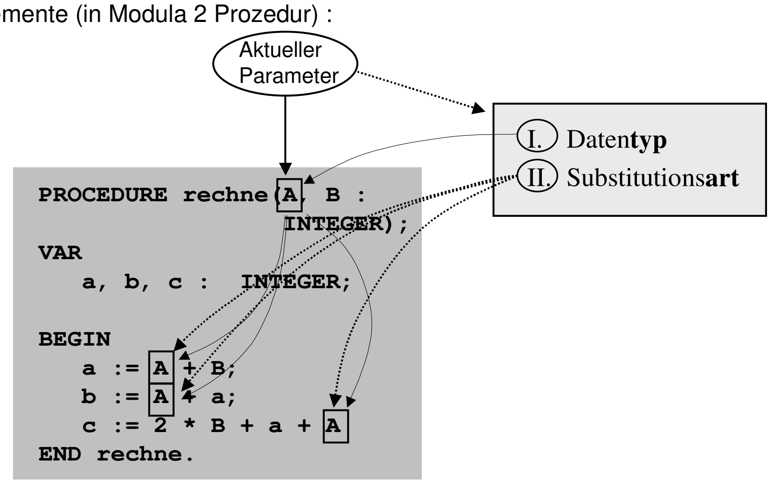
z.B. in Ada ;
hier sind **„Defaults“** beliebig !

Parametrisierung

Organisation :



Elemente (in Modula 2 Prozedur) :



1. **Datentypen** formaler Parameter

- Für die Definition der formalen Parameter müssen grundsätzlich die Datentypen deklariert werden – die Menge der Möglichkeiten umfasst die Datentypen der jeweiligen Programmiersprache;
- Bei der Zuordnung der aktuellen Parameter wird die Kompatibilität der Typen überprüft !

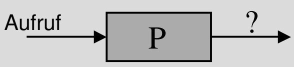
```
PROCEDURE geq(A, B :  INTEGER;
                 C :  BOOLEAN);

BEGIN
   IF A >= B THEN C := TRUE
             ELSE C := FALSE
END geq.
```

```
 INTEGER a, b;
 BOOLEAN eval;
    :
 geq(a, b, eval);
    :
```

2. Art der Substitution – oder Wie werden Ergebnisse zurückgeliefert ?

a) Bisher wurde noch nicht geklärt, in welcher Form die Parameter übergeben werden !
→ Art der Substitution ⇒ Parameter-**Auswertung** und –**Übergabe**

b) Wie werden Ergebnisse von Berechnungen in einer Prozedur an die aufrufende Umgebung mitgeteilt ?



Folgende Fälle treten häufig auf und
werden in gängigen Programmiersprachen behandelt :

1. 1 Ergebniswert eines einfachen **un**strukturierten Typs
   → Definition einer Prozedur, die selbst einen (Ergebnis-) Wert hat
2. Mehrere Ergebniswerte oder
   Ergebniswert eines strukturierten Typs
   → für ausgewählte Parameter sind nach Beendigung der Prozedur deren Wert in der Aufrufumgebung lesbar (auch 1. als Spezialfall !)
3. Ergebnis zeigt auf strukturierte Objekte (→ folgt später !)

Randbedingungen :

- **Eingabe**- und **Ausgabe**-Parameter
  → zunächst dienen Parameter der Eingabe für eine Prozedur, sie können aber auch zurückgelieferte Ergebnisse enthalten

Eingabe Rückgabe

```
PROCEDURE geq(A, B :   INTEGER;
                ↑C :   BOOLEAN);

BEGIN
   IF A >= B THEN C := TRUE
             ELSE C := FALSE
   END
END geq.
```

```
INTEGER a, b, i;
BOOLEAN eval;
   :
a := 27 * 5;
b := 0;
FOR i := 1 TO 27 DO
   b := b + 5
END;
geq(a, b, eval);
IF eval THEN ...
END;
   :
```

- **Einfache** oder **strukturierte Daten**
  → unterschiedliche Grössen der Parameterdaten

(→ Details in Parameter-Evaluation und -Übergabe !)

Arten von Prozeduren und Berechnung von Ergebnissen

1. Unterprogramme („subroutines“, „procedures“)

Rückgabe von Ergebnissen :

- Zuordnung eines Ergebnisses zu einem oder mehreren Parameter(n)
- Veränderung der Prozedur-Umgebung (Aufruf-Umgebung), d.h. Veränderung des Zustands des aufrufenden Programms

```
PROCEDURE sum(A, B, ↑C :
             INTEGER);
BEGIN
   C := A + B
END sum.
```

```
   :
sum(a, b, res);
   :
```

neuer Wert

```
PROCEDURE sum(A, B :
             INTEGER);
BEGIN
   result := A + B
END sum.
```

```
   :
sum(a, b);
   :
state := result + 7;
   :
```

Seiteneffekt !

2. Funktionen („functions“)

Werteliefernde Prozeduren, d.h. die Prozedur nimmt **selbst (!) einen Wert** an, der als Variable im aufrufenden Programm (z.B. innerhalb von Ausdrücken) verwendet werden kann !

Bsp.: mathematische Funktionen, z.B.
`cos(3.1415)` , **`exp(-x/2.0)`**

Gegenüberstellung :

```
PROCEDURE sum(A, B, ↑C :
              INTEGER);
BEGIN
   C := A + B
END sum.
```

```
   :
sum(a, b, res);
   :
```

Resultatwert

```
FUNCTION sum(A, B :
            INTEGER): INTEGER;
BEGIN
   RETURN A + B
END sum.
```

```
   :
state := sum(a, b) + 7;
   :
```

Rückgabewerte :

- **Einfache skalare Werte** (INTEGER, REAL, BOOLEAN, etc.)
  → z.B. in FORTRAN, Algol60
- ⊕ **Zeichenketten**
  → z.B. in PL/1
- ⊕ **Zeiger** (auf strukturierte Daten)
  → z.B. in PL/1, Pascal, Modula-2, C
- ⊕ **strukturierte Objekte**
  → z.B. in ELAN, Java

Parameter-Evaluation und -Übergabe

```
PROCEDURE geq(A, B :  INTEGER;
              C :  BOOLEAN);

BEGIN
   IF A >= B THEN C := TRUE
             ELSE C := FALSE
   END
END geq.
```

Eingabe Rückgabe

Auswertung der Parameter („parameter evaluation“)

- Identifikation jedes aktuellen Parameters und Zuordnung zu dem jeweils korrespondierenden formalen Parameter
- Auswertung jedes Parameters (an den Ersetzungsstellen)

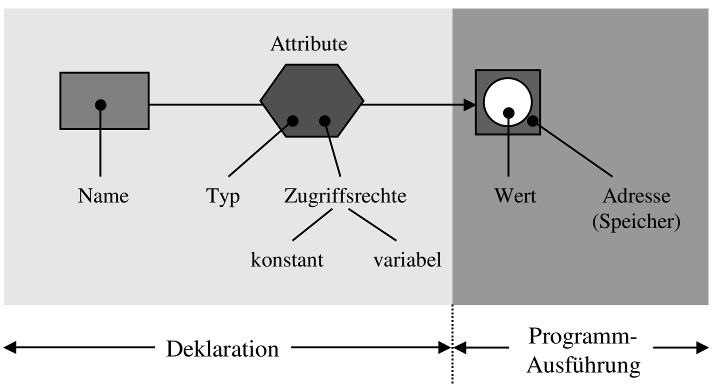
Übergabe der Parameter („parameter passing“)

→ Festlegung der Weise, wie ein aktueller Parameter an die Prozedur übertragen (~ **gebunden**) wird

zu klären : Struktur der Deklaration von Datenobjekten und Wertzuweisungen

Schema der Repräsentation von Objekten

→ <u>hier</u>: Variablen

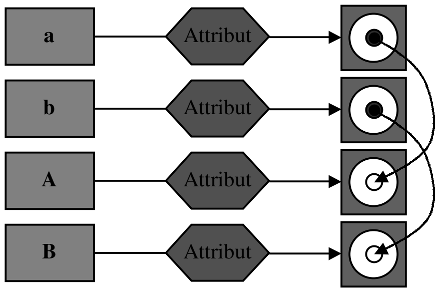


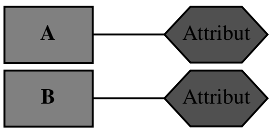
- **Werteübergabe** („**call-by-value**“)

Schema : Der aktuelle Parameter wird dereferenziert und ergibt einen Ergebnis**wert**, der anschliessend in die Orte (Platzhalter) der formalen Parameter **kopiert** wird

```
PROCEDURE sum(A, B :
             INTEGER):
              INTEGER;
BEGIN
   RETURN A + B
END sum.
```

```
    :
 c := sum(a, b);
    :
```





Einordnung :

- Ein aktueller Werte-Parameter erlaubt – aus Sicht der aufrufenden Umgebung – nur **Lesezugriff** („read-only value“)

  Lebensdauer des Parameters :
  → es ist keine Zuweisungen zur Kommunikation mit der Prozedur-Umgebung möglich; die „Lebensdauer“ eines aktuellen Parameters ist auf die Ausführungs-Dauer der Prozedur beschränkt und somit nur als **interne Variable** verwendbar !

- **Probleme** durch Kopieren aktueller Parameter
  → insbesondere bei strukturierten (Daten-) Objekten, z.B. Felder, Strukturen, kann das Anlegen von Kopien hohen Zeit- und Platzaufwand bedeuten !

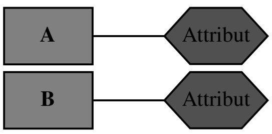
- Identifikation :
  - Standardübergabe in Pascal und Modula-2, d.h. ein formaler Parameter wird nicht weiter ausgezeichnet :
    ```
    <formal parameter>
    ```
  - In Algol60 :
    ```
    value <formal parameter>
    ```

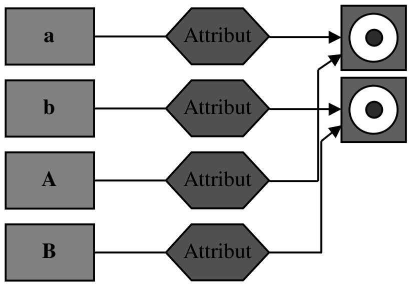
- **Adressübergabe** (Referenz-/Zeigerübergabe, „**call-by-reference**“)

Schema : Der aktuelle Parameter wird durch seine **Adresse** (Ort im Speicher), z.B. mittels eines Variablennamen, identifiziert. Diese Adresse wird direkt an den formalen Parameter gebunden

```
PROCEDURE exch(→A, →B :
               INTEGER);
   buf :  INTEGER;
BEGIN
   buf := →A;
   →A := →B;
   →B := buf
END exch.
```

```
      :
  exch(a, b);
      :
```





Einordnung :

- **Vorteile :**
  - **Steigerung der Effizienz** gegenüber Kopieren
  - Möglichkeit der Variablennutzung zur **Rückgabe** von Prozedur-Ergebnissen

    ⇒ ein für die Rückgabe von Ergebnissen genutzter formaler Parameter <u>muss</u> als Adress- (Referenz-) Parameter deklariert werden !

- **Nachteile :**
  - Die Veränderung des Wertes eines aktuellen Parameters – der als Referenz-Parameter deklariert wurde – führt **immer** zu **(kontrollierten) Seiteneffekten** !
  - **Problem „anonymer Variablen“** :

    <u>Bsp</u>. : Aufruf `exch(10, 12)` (`10`, `12` sind Konstanten <u>oder</u> Ergebnis eines Ausdrucks)

    . `10` und `12` werden in Speicherstellen abgelegt, deren Adresse an die formalen Parameter **A** bzw. **B** gebunden werden

    . Was passiert, wenn der Wert von **A** bzw. **B** verändert wird ?

    ⇒ die Speicherzelle dieser „anonymen Variable“ würde den neuen Wert erhalten; der Veränderung der Werte `10` bzw. `12` als Seiteneffekt muss geeignet entgegengewirkt werden ! (→ **Verboten in Modula-2** !)

- **Beispielprogramm in Modula-2**

  ⇒ Referenz auf das Datenobjekt wird bei der Compilation überprüft ! Bei Inkompatibilität, z.B. Bei Konstanten oder Ausdrücken, wird ein Syntaxfehler angezeigt !

```
MODULE AnonymeVariable;
  FROM InOut IMPORT WriteInt, WriteString, WriteLn;

VAR
  wert :  INTEGER;


PROCEDURE increment(
  VAR referenzParameter :  INTEGER);

BEGIN
  referenzParameter :=  referenzParameter + 1
END increment;


BEGIN (* -- Hauptprogramm 'AnonymeVariable' -- *)
  wert :=  1;
  WriteString("Print 'wert'      :  "); WriteInt(wert, 3); WriteLn;
  increment(wert);
  increment(wert);
  WriteString("Print 'wert + 2' :  "); WriteInt(wert, 3); WriteLn;

  (* increment(10); *)
  WriteString("Print '10 + 1'   :  "); WriteInt(10 + 1, 3); WriteLn
END AnonymeVariable.
```

auskommentiert, wg. Fehlermeldung

Erste Compilation ... führt zu Fehlermeldung (Syntaxfehler) !

Ausführungsergebnis (nach korrekter Übersetzung) !

```
23      increment(10);
                    ^ reference not to a variable
                    ^ modula revision: types must be identical
```

```
Print 'wert'      :    1
Print 'wert + 2' :    3
Print '10 + 1'   :   11
```

- Identifikation :
  - Standardübergabe in <u>FORTRAN</u> und <u>PL/1</u>, d.h. ein formaler Parameter wird nicht weiter ausgezeichnet :

    `<formal parameter>`

  - In <u>Pascal</u> und <u>Modula-2</u> :

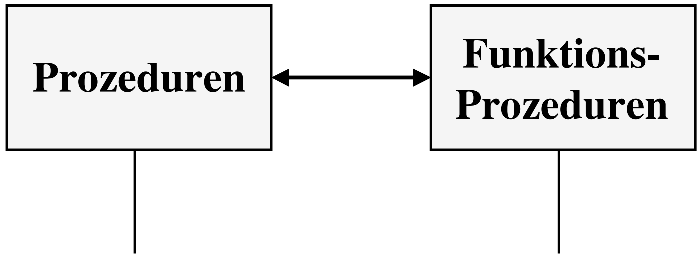
    `VAL <formal parameter>`

    (reserviertes Wort in Modula-2 ~ Terminalsymbol)

Prozeduren und Funktionen in Modula 2

Deklaration, Parameter und Substitution

Definition von Prozeduren



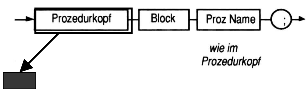
`PROCEDURE name(...);`

Anweisungssequenzen
OHNE
Ergebniswert

`PROCEDURE name(...): <result type>;`

Anweisungssequenzen
MIT
Ergebniswert

Vereinbarung (allgemein)

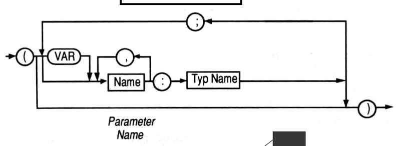


[redacted]

Syntax formaler und aktueller Parameterlisten

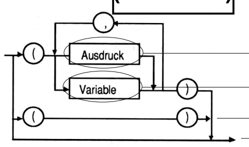
[redacted]

- **Formale Parameter (For Par L)**



[redacted]

- **Aktuelle Parameter (Akt Par L)**



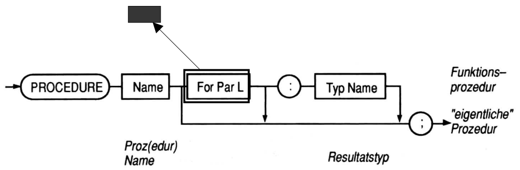
Aktueller Ausdruck (Werteparameter)
Aktuelle Variable (Referenzparameter)
Parameterlose Funktionsprozeduren
Parameterlose Prozeduren

Kopf von Prozeduren / Funktions-Prozeduren

1. **Prozedur** : abgeschlossener Programmteil als Einheit

2. **Funktion** : Prozedur, die mit einem **Wert** behaftet ist
(Werte-Rückgabe mittels RETURN-Anweisung !)

Prozedurkopf



Hinweis : **Ergebnis** einer **Funktions-Prozedur** darf nur ein
- einfacher Typ oder
- Zeiger-Typ (→ später !)

sein !

Notation verschiedener Parameter-Substitutionen

1. **Werte**-Parameter („call-by-value") :
(→ wird am häufigsten verwendet !)

   Die **Werte-Substitution** wird **automatisch** („default") angewandt, wenn keine Spezifikation der Substitutions-Art vorliegt !

   Hinweis : Formale **Werte**-Parameter stellen **lokale Variablen** dar, die – nachdem sie beim Prozedur-Aufruf Anfangswerte (durch die aktuellen Parameter) erhalten haben – keine weiteren Beziehungen mehr zu den aktuellen Parameter-Werten der Aufrufumgebung unterhalten !

2. **Referenz**-Parameter („call-by-reference") :
Die Referenz-Substitution (~ **Variablen-Substitution**) wird durch den Spezifikator der Substitutions-Art

   `VAR`

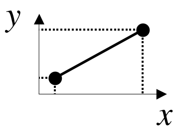
   festgelegt !

Beispiele für Prozedurköpfe :

```
PROCEDURE zeichneLinie(
    xAnf, yAnf,
    xEnd, yEnd :  INTEGER);

BEGIN
    :
END zeichneLinie;
```

```
PROCEDURE gauss(
    x, sigma :  REAL): REAL;

BEGIN
    :
END gauss;
```

Einfacher Typ



Referenz-Parameter für Ergebnis

```
PROCEDURE addVector(
    a, b    :  ARRAY [1..5] OF INTEGER;
    VAR res :  ARRAY [1..5] OF INTEGER);

BEGIN
    :
END addVector;
```

Beispiel – Auswirkung der Substitutionen :

```
MODULE werteVarParameter;
   :
VAR
  i :  INTEGER;
  a :  ARRAY [1..2] OF INTEGER;

PROCEDURE par(x :  INTEGER);

BEGIN
  x :=  x + 2
END par;

BEGIN (* -- Hauptprogramm *)
  a[1] :=  10;
  a[2] :=  20;
  i    :=   1;
  par(a[i])
END werteVarParameter.
```

```
MODULE werteVarParameter;
   :
VAR
  i :  INTEGER;
  a :  ARRAY [1..2] OF INTEGER;

PROCEDURE par(VAR x :  INTEGER);

BEGIN
  x :=  x + 2
END par;

BEGIN (* -- Hauptprogramm *)
  a[1] :=  10;
  a[2] :=  20;
  i    :=   1;
  par(a[i])
END werteVarParameter.
```

Ergebnis der Prozedur-Berechnung (Status) ?

Werte-Parameter :

`par(a[1])` → `par(10)`

`x = 10`

`x := x + 2` ⇒ `x = 12`

Resultat : `a = (10, 20)`

Referenz-Parameter :

`Par(`→`a[1])`

`x` → `a[1]` (`x` zeigt auf Adresse von `a[1]`)

`x := x + 2` ⇒ `a[1] := a[1] + 2`

Resultat : `a = (12, 20)`

Ungültige Prozedurköpfe

1. Fehlende **Typ**-Namen

falsch :
```
PROCEDURE name(VAR x);
```
richtig :
```
PROCEDURE name(VAR x :  INTEGER);
```

2. Deklaration **strukturierter Daten** (Records) ist <u>nicht</u> erlaubt
   (**<u>Achtung:</u>** Felder sind erlaubt !)

falsch :
```
PROCEDURE name(x :  RECORD
                      r, s :  REAL
                    END);
```
richtig :
```
TYPE tuple = RECORD
               r, s :  REAL
             END;
PROCEDURE name(x :  tuple);
```

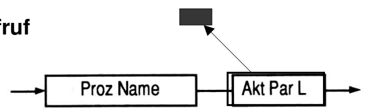
3. Aufbau der **Parameterliste**

falsch :
```
PROCEDURE name(VAR x, y :  INTEGER, z :  INTEGER);
```
richtig :
```
PROCEDURE name(VAR x, y :  INTEGER; z :  INTEGER);
```

<u>Aufruf und Termination</u>

Aufrufe

Prozeduraufruf



Prozedur : namentlicher Aufruf als **Programmzeile**, **mit** oder **ohne** aktuelle Parameter

Funktion : namentlicher Aufruf in einem **Ausdruck** mit aktuellen Parametern

∴ bei <u>leerer</u> Parameterliste :

```
i := Func()
```

Klammern mit leerer Parameter-Liste

→ dies hilft, eine Anweisung von einer einfachen Zuweisung (`i := Func`) zu unterscheiden !

Beenden einer Prozedur / Funktions-Prozedur

Prozedur : a) Abarbeitung des Programmteils bis

```
   :
END name.
```

b) **vorzeitiger Ausstieg**, z.B. bei Auswahlanweisungen, mit

```
RETURN
```

Funktion : **Rückgabewert** der Funktions-Prozedur

```
RETURN <Ausdruck>
```

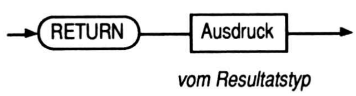
→ muss mindestens einmal vorhanden sein, meist direkt vor **`END`**-Anweisung; kann aber auch mehrmals an beliebigen Stellen auftreten, z.B. in Auswahlanweisungen

Syntax der **`RETURN`**-Anweisung :

- für **Prozeduren**

(für die vorzeitige Beendigung einer Prozedur an einem Terminationspunkt)

- für **Funktions-Prozeduren**



(muss immer angegeben werden – kann auch mehrmals auftreten !)

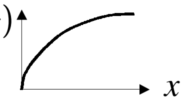
Anwendungen – (Funktions-) Prozeduren und Seiteneffekte

Prozeduren mit Rückgabewert und Funktions-Prozeduren

```
PROCEDURE gauss(
   x, sig  :  REAL;
   VAR res :  REAL);
VAR
   z, n :  REAL;
BEGIN
   z   := x * x;
   n   := 2.0 * sig * sig;
   res := exp(z/n)
END gauss;
```

```
PROCEDURE transform(
   x :  REAL): REAL;

BEGIN
   RETURN x / (1.0 + x)
END transform;
```

$RETURN = f(x)$



Hinweis : Eine Zuweisung innerhalb einer Prozedur / Funktions-Prozedur zu einer nicht-lokalen Variable oder einem VAR-Parameter führt zu einem **Seiteneffekt** !

Unkontrollierter Seiteneffekt

Kontrollierter Seiteneffekt → wird gemeinhin nicht weiter als Seiteneffekt bezeichnet !

Verwendung (Aufrufe)

Typ der aktuellen Parameter ⇔ Typ der formalen Parameter (laut Vereinbarung im Prozedurkopf)

Aufrufe :

```
   :
FOR i := 0 TO 10 DO
  gauss(FLOAT(i), 2.0, val);
  graph[i] :=  val
END;
   :
```

```
   :
i := 10;
WHILE i > 0 DO
  val       := exp(FLOAT(i));
  graph[i]  := transform(val);
  i         := i - 1
END;
   :
```

```
val       := exp(FLOAT(i));
graph[i]  := transform(val);
```

⟺

```
graph[i]  := transform(exp(FLOAT(i)));
```

Variablenveränderung mit und ohne Seiteneffekt

Aufgabe: Berechnung von x^y

Lösung: iterative Berechnung
x*x*...*x (y mal)

Erste Lösung enthält fälschlicherweise einen unkontrollierten Seiteneffekt !

Bsp.: x = 4, y = 3 ⇒ $4^3 = 64$

Ausgabe : 4 hoch 64 ist 64

```
MODULE Seiteneffekt;
  FROM InOut IMPORT WriteString, ReadInt, WriteInt, WriteLn;

(*
 * -- Beispielprogramm mit Seiteneffekt
 *)

VAR
  r, s, wert :  INTEGER;

PROCEDURE XhochY(
  x, y         :  INTEGER;
  VAR ergebnis :  INTEGER);
(* ergebnis := x hoch y (= x*x*x*..*x, y-mal) *)

VAR
  i :  INTEGER;

BEGIN
  r := 1;     (* ACHTUNG: r ist nicht lokal definiert, also global! *)
  FOR i := 1 TO y DO
    r := r * x
  END;
  ergebnis := r
END XhochY;

BEGIN (* -- Beispielprogramm 'Seiteneffekt' -- *)
  WriteString("Eingabe: Basis    :  "); ReadInt(s);
  WriteString("         Exponent :  "); ReadInt(r);
  WriteLn;

  XhochY(s, r, wert);

  WriteString("Ergebnis :  ");
  WriteInt(s, 3); WriteString(" hoch "); WriteInt(r, 3);
  WriteString(" ist "); WriteInt(wert, 6); WriteLn
END Seiteneffekt.
```

Korrektur :

Zweite Lösung ohne Seiteneffekt
– enthält lokale Variable !

Modula-2 Programme
`Seiteneffekt.m2` und
`SeiteneffektKorrekt.m2`
→ Vergleich der Ausgaben !

```
MODULE SeiteneffektKorrekt;
  FROM InOut IMPORT WriteString, ReadInt, WriteInt, WriteLn;

(*
 * -- Beispielprogramm mit korrigiertem Seiteneffekt
 *)

VAR
  r, s, wert :  INTEGER;

PROCEDURE XhochY(
  x, y         :  INTEGER;
  VAR ergebnis :  INTEGER);
(* ergebnis := x hoch y (= x*x*x*..*x, y-mal) *)

VAR
  i, r:  INTEGER;

BEGIN
  r := 1;
  FOR i := 1 TO y DO
    r := r * x
  END;
  ergebnis := r
END XhochY;

BEGIN (* -- Beispielprogramm 'SeiteneffektKorrekt' -- *)
  WriteString("Eingabe: Basis    :  "); ReadInt(s);
  WriteString("         Exponent :  "); ReadInt(r);
  WriteLn;

  XhochY(s, r, wert);

  WriteString("Ergebnis :  ");
  WriteInt(s, 3); WriteString(" hoch "); WriteInt(r, 3);
  WriteString(" ist "); WriteInt(wert, 6); WriteLn
END SeiteneffektKorrekt.
```

Variablen und Seiteneffekte

Manipulation von Variablen
in Prozedur mit Rückgabewert

```
MODULE Verzwickt;
  FROM InOut IMPORT WriteString, WriteInt, WriteLn;

(*
 * -- Beispielprogramm zur Veranschaulichung der Rolle
 * -- verschiedener Parameternamen
 *)

VAR
  k, l, m :  INTEGER;

PROCEDURE TueWas(
  x     :  INTEGER;
  VAR y :  INTEGER);

VAR
  z :  INTEGER;

BEGIN
  z := x;
  m := 17;
  y := z + 2;
  x := x + 5
END TueWas;

BEGIN (* -- Beispielprogramm 'Verzwickt' -- *)
  l := 1;
  k := 2;
  m := 0;
  TueWas(l, k);

  WriteString("Werte l, k, m ... : ");
  WriteInt(l, 4);
  WriteInt(k, 4);
  WriteInt(m, 4); WriteLn
END Verzwickt.
```

Resultat :

```
Werte l, k, m ... :   1  3 17
```

Blockstruktur und Schachtelung

Geschachtelte Prozeduren und lokale Objekte

Deklarationen und Funktionalität

```
MODULE M;
  :
  VAR a, b :  INTEGER;
      c, e :  REAL;

  PROCEDURE PA;
    VAR a, b, c, d :  CARDINAL;

    PROCEDURE PAA(
        x, VAR y :  CARDINAL);
      VAR a :  INTEGER;
    BEGIN
      a :=  5;
      y := a * x
    END PAA;

  BEGIN
    a :=  1;
    b :=  2;
    PAA(a, c);
    PAA(b, d);
    e :=  FLOAT(c + d)
  END PA;
```

```
c =  5
d = 10
```

```
  PROCEDURE PB(
      z :  INTEGER): REAL;
  BEGIN
    RETURN 5.0 * FLOAT(z)
  END PB;

BEGIN
  a :=  1;
  b :=  2;
  PA;
  c :=  PB(a) + PB(b);
  IF e = c THEN
    :
  ELSE
    :
  END
END M.
```

```
e = 15.0
c = 5.0 + 10.0
```

☑

Achtung : Es gibt namensgleiche Objekte, z.B. Variablen und Konstanten !

Blockstruktur – Lokale Objekte

1. Die Deklaration eines **`MODULE`**s oder einer **`PROCEDURE`** besteht immer aus
   - Kopf (Deklarationsteil) und
   - Rumpf

   Der **Rumpf** enthält Anweisungsfolgen, die durch **`BEGIN ... END`** geklammert sind.
   Innerhalb des **Kopf**es werden Daten- und Programm-Objekte deklariert, die innerhalb des **`MODULE`**s oder der **`PROCEDURE`** **lokal** gültig sind !

2. Definition **lokaler** Objekte, d.h. innerhalb des Textes der Prozedurvereinbarung gültige Objekte :

   Daten : CONST, VAR
   Typen : TYPE
   Programme : PROCEDURE (mit und ohne Wert)

3. In **`PROCEDURE`**s können selbst lokale **`PROCEDURE`**s enthalten sein, in denen wiederum lokale **`PROCEDURE`**s deklariert sind ... (bei beliebiger Schachtelungstiefe)

Geheimnisprinzip und Sichtbarkeitsbereiche

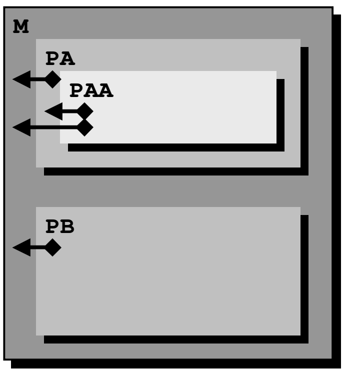
Nach innen ...

Lokal definierte Objekte (Daten, Prozeduren) sind von aussen, d.h. aus der Umgebung einer Prozedur, nicht sichtbar (**Geheimnisprinzip**)

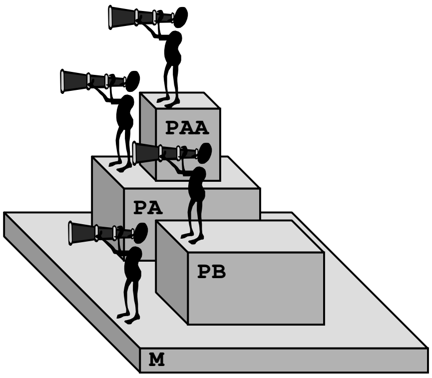
Nach aussen ...

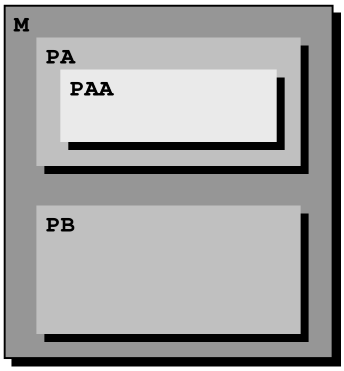
Aus einer Prozedur heraus sind alle Objekte der Umgebung sichtbar (**Sichtbarkeitsregeln**)

Block	Zugriff auf Objekte in den Blöcken
M	**in M : a, b (INTEGER),** **c, e (REAL)** **Umgebung :** importierte Prozeduren / Funktionen aus anderen Moduln
PA	**in PA : a, b, c, d (REAL)** **in M : e (REAL)**
PB	**in PB : z (INTEGER)** **in M : a, b (INTEGER),** **c, e (REAL)**
PAA	**in PAA: x, y (CARDINAL),** **a (INTEGER)** **in PA : b, c, d (CARDINAL)** **in M : e (REAL)**



- **Box- oder Stapel-Modell** (→ Sicht- oder Horizont-Metapher)





M sieht alle Objekte mit Eigenschaften in M
PA sieht alle Objekte mit Eigenschaften in PA und in M
PB sieht alle Objekte mit Eigenschaften in PB und in M
PAA sieht alle Objekte mit Eigenschaften in PAA, in PA und in M

- **Bezeichnungskonflikt – Verdeckungsregeln bei Namensgleichheit**

Was passiert mit
namensgleichen
Objekten, z.B. Variablen und Konstanten ?

im Detail ...

Die Bezeichnungen lokaler Grössen dürfen unabhängig von der Umgebung gewählt werden !

Bsp.: identische Bezeichnung x eines lokalen Objektes der (Funktions-) Prozedur P mit einem Objekt der Umgebung von P

$\Rightarrow$ **Bezeichnungskonflikt** !

➡ Bei Namensgleichheit von Objekten in verschiedenen Ebenen (Blöcke, Prozeduren) überdecken sich die Namen in der

Priorität von innen nach aussen

(~ „Höhe“ im Stapel-Modell)

Block	(Daten-) Objekte	überdeckte Objekte
`M`	`a, b, c, e`	`%`
`PA`	`a, b, c, d`	`a, b, c (in M)`
`PB`	`z`	`%`
`PAA`	`x, y, a`	`a (in PA)`

Beispiel-Prozeduren mit verschiedenen Parametrisierungsarten

```
MODULE ParameterArten;
  FROM InOut IMPORT WriteString, WriteInt, WriteLn;

(*
 * -- Beispielprogramm zur Veranschaulichung der Wirkung
 * -- verschiedener Parameterarten
 *)

VAR
  i :  INTEGER;

PROCEDURE keinParameter;

BEGIN
  WriteLn; WriteLn;
  WriteString("Wert von i in Prozedur 'keinParameter'            : ");
  WriteInt(i, 6); WriteLn;
  i := i + 1;
  WriteString("Letzter Wert von i in Prozedur 'keinParameter'    : ");
  WriteInt(i, 6); WriteLn
END keinParameter;

PROCEDURE valueParameter(
  i :  INTEGER);

BEGIN
  WriteLn; WriteLn;
  WriteString("Wert von i in Prozedur 'valueParameter'           : ");
  WriteInt(i, 6); WriteLn;
  i := i + 1;
  WriteString("Letzter Wert von i in Prozedur 'valueParameter'   : ");
  WriteInt(i, 6); WriteLn
END valueParameter;

PROCEDURE variableParameter(
  VAR i :  INTEGER);

BEGIN
  WriteLn; WriteLn;
  WriteString("Wert von i in Prozedur 'variableParameter'        : ");
  WriteInt(i, 6); WriteLn;
  i := i + 1;
  WriteString("Letzter Wert von i in Prozedur 'variableParameter': ");
  WriteInt(i, 6); WriteLn
END variableParameter;

BEGIN (* -- Beispielprogramm 'ParameterArten' -- *)
  i := 0;
  keinParameter;
  WriteString("Wert von i NACH Prozedur 'keinParameter'          : ");
  WriteInt(i, 6); WriteLn;

  i := 0;
  valueParameter(i);
  WriteString("Wert von i NACH Prozedur 'valueParameter'         : ");
  WriteInt(i, 6); WriteLn;

  i := 0;
  variableParameter(i);
  WriteString("Wert von i NACH Prozedur 'variableParameter'      : ");
  WriteInt(i, 6); WriteLn
END ParameterArten.
```

Prozeduren / Parametrisierungsarten (cont‘d)

Ergebnisse für

- Prozedur ohne Parameter
- Prozedur mit Werte-Parameter
- Prozedur mit VAR- (Referenz-) Parameter

```
Wert von i in Prozedur 'keinParameter'              :      0
Letzter Wert von i in Prozedur 'keinParameter'      :      1
Wert von i NACH Prozedur 'keinParameter'            :      1

Wert von i in Prozedur 'valueParameter'             :      0
Letzter Wert von i in Prozedur 'valueParameter'     :      1
Wert von i NACH Prozedur 'valueParameter'           :      0

Wert von i in Prozedur 'variableParameter'          :      0
Letzter Wert von i in Prozedur 'variableParameter': :      1
Wert von i NACH Prozedur 'variableParameter'        :      1
```

Parameterübergabe und Sichtbarkeit – Beispiel

Programm „Zaubertrank“

a) Welche Werte haben die **globalen Variablen** des Programms „Zaubertrank“ jeweils nach dem Aufruf der Prozedur „Nonsens“

b) Welche **Variablen** sind zu welchem Zeitpunkt aus einem Programm-segment sichtbar (und damit auch zugreifbar)?

Zaubertrank	Idefix Asterix Obelix	Majestix Miraculix Troubadix
Nonsens	Asterix Idefix	Obelix
Nonsens2	Asterix Grautvornix Obelix	Idefix

```
MODULE Zaubertrank;
  FROM InOut IMPORT WriteString, WriteInt, WriteLn;

(*
 * -- Beispielprogramm zur Veranschaulichung von Parameteruebergabe-
 * -- Mechanismen und Sichtbarkeitsregeln (Blockstruktur)
 *)

VAR
  Idefix, Asterix, Obelix,
  Majestix, Miraculix, Troubadix :  INTEGER;

(* -- Nonsens:   Block-Ebene 1 *)
PROCEDURE Nonsens(
  VAR Asterix :  INTEGER;
  Obelix      :  INTEGER);

VAR
  Idefix :  INTEGER;

 (* -- Nonsens2:  Block-Ebene 2 *)
 PROCEDURE Nonsens2(
   VAR Asterix, Grautvornix :  INTEGER;
   Obelix                   :  INTEGER);

 VAR
   Idefix :  INTEGER;

 BEGIN
   Grautvornix := Asterix + Obelix;
   Asterix     := Asterix + Grautvornix;
   Idefix      := Asterix + Majestix;
   Obelix      := Idefix  + Idefix
 END Nonsens2; (* Block-Ebene 2 *)

 BEGIN
   Idefix  := Asterix + Obelix;
   Nonsens2(Asterix, Idefix, Troubadix);
   Obelix  := Idefix  + Idefix;
   Asterix := Asterix + Obelix;
   Nonsens2(Miraculix, Miraculix, Troubadix)
 END Nonsens; (*  Block-Ebene 1 *)
```

„Zaubertrank“ (cont‘d)

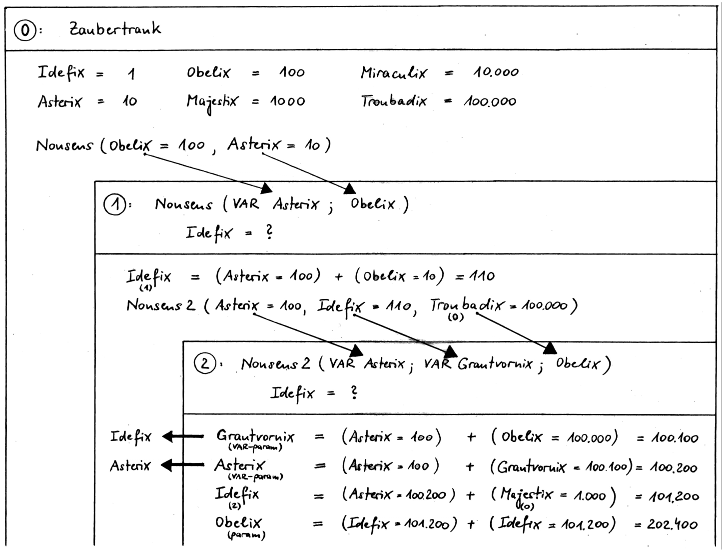
```
PROCEDURE DruckeVariablen;
BEGIN
  WriteString("| "); WriteInt(Idefix,    6); WriteString(" | ");
                     WriteInt(Asterix,   7); WriteString(" | ");
                     WriteInt(Obelix,    7); WriteString(" | ");
                     WriteInt(Majestix,  8); WriteString(" | ");
                     WriteInt(Miraculix, 9); WriteString(" | ");
                     WriteInt(Troubadix, 9); WriteString(" |"); WriteLn
END DruckeVariablen;

BEGIN (* -- Zaubertrank -- *)
  Idefix    :=      1;
  Asterix   :=     10;
  Obelix    :=    100;
  Majestix  :=   1000;
  Miraculix :=  10000;
  Troubadix := 100000;

  WriteString("+==============================================================+"); W
  WriteString("| Idefix | Asterix |  Obelix | Majestix | Miraculix | Troubadix |"); W
  WriteString("|==============================================================|"); W
  DruckeVariablen;

  Nonsens(Obelix, Asterix);
  DruckeVariablen;

  Nonsens(Obelix, Obelix);
  DruckeVariablen;

  Nonsens(Idefix, Idefix);
  DruckeVariablen;

  WriteString("+==============================================================+"); W
END Zaubertrank.
```

WriteLn
(abgeschnitten)

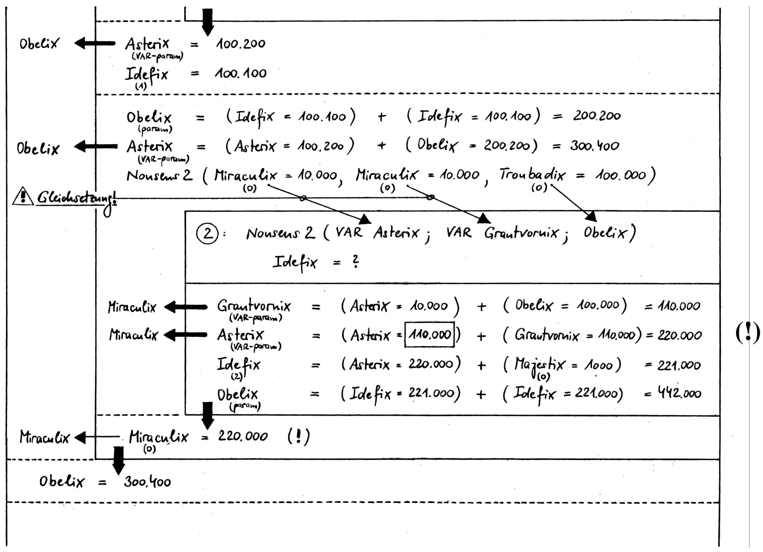
Ergebnisse „Zaubertrank“

```
+==============================================================+
| Idefix | Asterix |  Obelix | Majestix | Miraculix | Troubadix |
|==============================================================|
|      1 |      10 |     100 |     1000 |     10000 |    100000 |
|      1 |      10 |  300400 |     1000 |    220000 |    100000 |
|      1 |      10 | 1501600 |     1000 |    640000 |    100000 |
| 300004 |      10 | 1501600 |     1000 |   1480000 |    100000 |
+==============================================================+
```

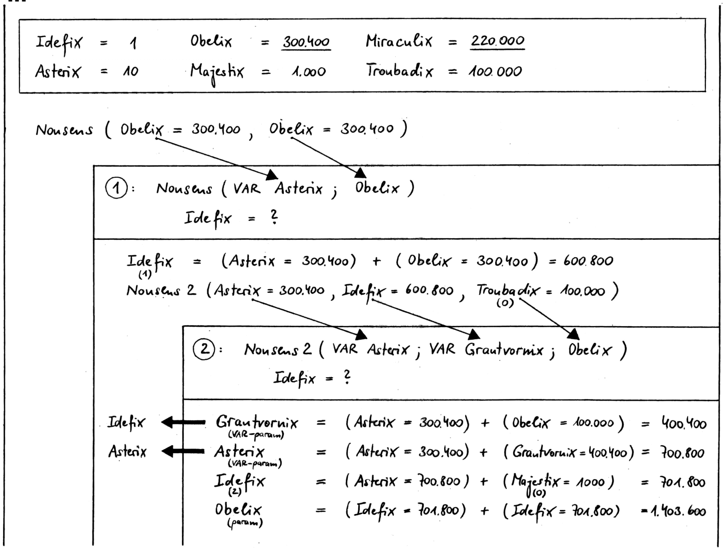
„Hand-Simulation“ des Programms „Zaubertrank“



Cont‘d ...

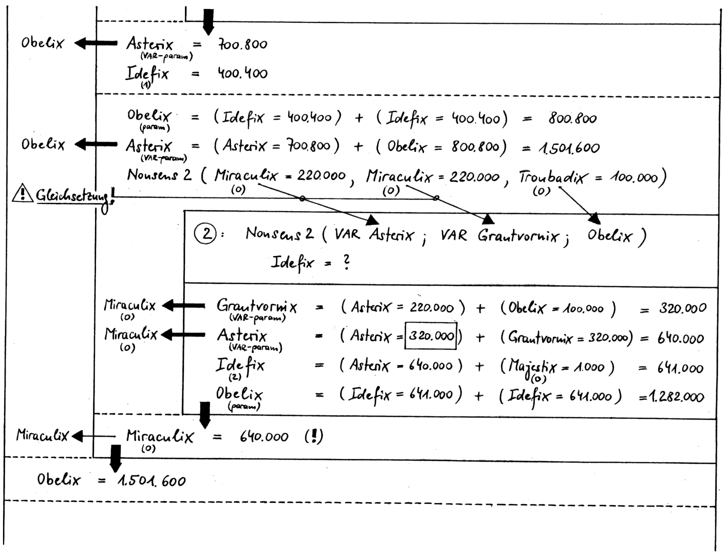


Cont‘d ...



Idefix = 1 Obelix = 300.400 Miraculix = 220.000
Asterix = 10 Majestix = 1.000 Troubadix = 100.000
Nousens (Obelix = 300.400 , Obelix = 300.400)
①: Nousens (VAR Asterix ; Obelix)
Idefix = ?
Idefix (1) = (Asterix = 300.400) + (Obelix = 300.400) = 600.800
Nousens 2 (Asterix = 300.400 , Idefix = 600.800 , Troubadix (0) = 100.000)
②: Nousens 2 (VAR Asterix ; VAR Grautvornix ; Obelix)
Idefix = ?
Idefix
Asterix
Grautvornix (VAR-param) = (Asterix = 300.400) + (Obelix = 100.000) = 400.400
Asterix (VAR-param) = (Asterix = 300.400) + (Grautvornix = 400.400) = 700.800
Idefix (2) = (Asterix = 700.800) + (Majestix (0) = 1000) = 701.800
Obelix (param) = (Idefix = 701.800) + (Idefix = 701.800) = 1.403.600

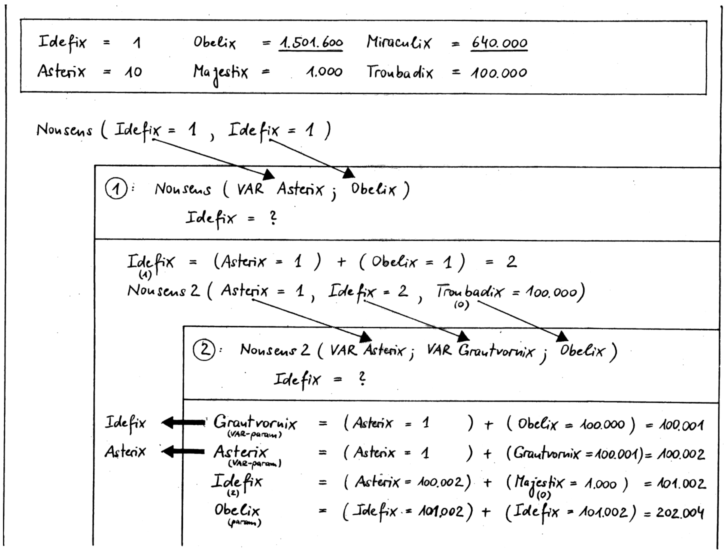
Cont‘d ...



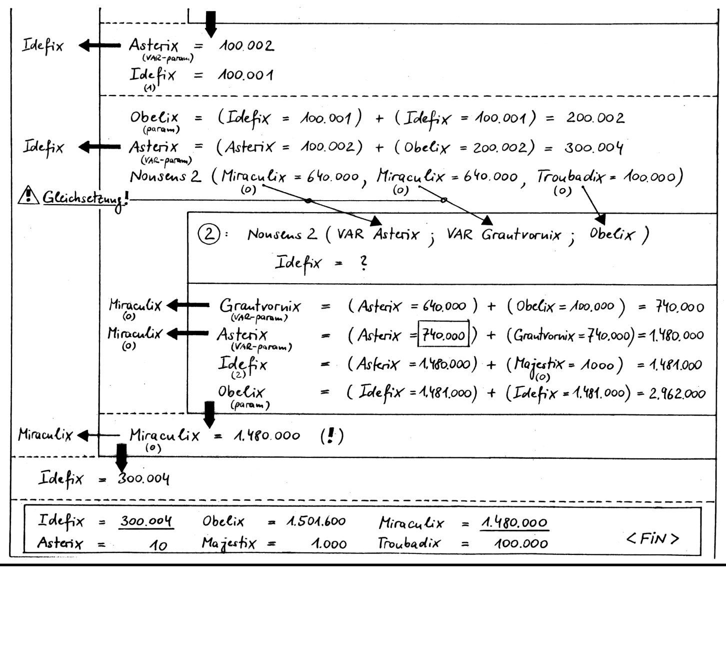
Obelix
Asterix (VAR-param) = 700.800
Idefix (1) = 400.400
Obelix (param) = (Idefix = 400.400) + (Idefix = 400.400) = 800.800
Obelix
Asterix (VAR-param) = (Asterix = 700.800) + (Obelix = 800.800) = 1.501.600
Nousens 2 (Miraculix (0) = 220.000 , Miraculix (0) = 220.000 , Troubadix (0) = 100.000)
Gleichsetzung!
②: Nousens 2 (VAR Asterix ; VAR Grautvornix ; Obelix)
Idefix = ?
Miraculix (0)
Miraculix (0)
Grautvornix (VAR-param) = (Asterix = 220.000) + (Obelix = 100.000) = 320.000
Asterix (VAR-param) = (Asterix = 320.000) + (Grautvornix = 320.000) = 640.000
Idefix (2) = (Asterix = 640.000) + (Majestix (0) = 1.000) = 641.000
Obelix (param) = (Idefix = 641.000) + (Idefix = 641.000) = 1.282.000
Miraculix
Miraculix (0) = 640.000 (!)
Obelix = 1.501.600

(!)

Cont‘d ...


Idefix = 1 Obelix = 1.501.600 Miraculix = 640.000
Asterix = 10 Majestix = 1.000 Troubadix = 100.000
Nousens (Idefix = 1 , Idefix = 1)
1: Nousens (VAR Asterix ; Obelix)
Idefix = ?
Idefix (1) = (Asterix = 1) + (Obelix = 1) = 2
Nousens 2 (Asterix = 1 , Idefix = 2 , Troubadix (0) = 100.000)
2: Nousens 2 (VAR Asterix ; VAR Grautvornix ; Obelix)
Idefix = ?
Idefix ← Grautvornix (VAR-param) = (Asterix = 1) + (Obelix = 100.000) = 100.001
Asterix ← Asterix (VAR-param) = (Asterix = 1) + (Grautvornix = 100.001) = 100.002
Idefix (2) = (Asterix = 100.002) + (Majestix (0) = 1.000) = 101.002
Obelix (param) = (Idefix = 101.002) + (Idefix = 101.002) = 202.004

Cont‘d


Idefix ← Asterix (VAR-param) = 100.002
Idefix (1) = 100.001
Obelix (param) = (Idefix = 100.001) + (Idefix = 100.001) = 200.002
Idefix ← Asterix (VAR-param) = (Asterix = 100.002) + (Obelix = 200.002) = 300.004
Nousens 2 (Miraculix (0) = 640.000, Miraculix (0) = 640.000 , Troubadix (0) = 100.000)
Gleichsetzung!
2: Nousens 2 (VAR Asterix ; VAR Grautvornix ; Obelix)
Idefix = ?
Miraculix (0) ← Grautvornix (VAR-param) = (Asterix = 640.000) + (Obelix = 100.000) = 740.000
Miraculix (0) ← Asterix (VAR-param) = (Asterix = 740.000) + (Grautvornix = 740.000) = 1.480.000
Idefix (2) = (Asterix = 1.480.000) + (Majestix (0) = 1000) = 1.481.000
Obelix (param) = (Idefix = 1.481.000) + (Idefix = 1.481.000) = 2.962.000
Miraculix ← Miraculix (0) = 1.480.000 (!)
Idefix = 300.004
Idefix = 300.004 Obelix = 1.501.600 Miraculix = 1.480.000
Asterix = 10 Majestix = 1.000 Troubadix = 100.000
<FIN>

(!)

Lebensdauer von Objekten

Einordnung

1. **Gültigkeitsbereich**
   → statische Eigenschaft von Objekten, die von der Deklaration im Quelltext abhängig ist
2. **Lebensdauer**
   → dynamische Eigenschaft von Variablen, die vom Programmablauf anhängig ist

Lebensdauer von Variablen

Globale Variable ...

- werden zu Beginn der Programmausführung angelegt, d.h. es wird Speicherplatz reserviert
- haben zunächst einen undefinierten Wert
- werden nach Beendigung des Programms „vernichtet“

Bei **Aufruf einer Prozedur** ...

- werden lokale Variable und formale Parameter angelegt
  (**Achtung :** VAR-Parameter werden **nicht** angelegt, die Daten sind bereits vorhanden, nur der Verweis auf deren Adresse muss gesetzt werden !)
- ist der Wert der lokalen Variablen zunächst undefiniert
- ist der Wert der formalen Parameter gleich dem Wert der korrespondierenden aktuellen Parameter

Ausführung einer Prozedur :
→ Anweisungen werden ausgeführt (inkl. Aufruf anderer Prozeduren !)
- globale und lokale Variable sowie formale Werte-Parameter sind vorhanden
- VAR-Parameter sind ebenso vorhanden

Beendigung einer Prozedur :
→ letzte Anweisung oder RETURN
- lokale Variable und Werte-Parameter werden „vernichtet“

Zusammenfassung und Übersicht

Prozeduren und Parameter

1. **Formale** und **aktuelle** Parameter :
   Formal-Parameter dienen als Platzhalter für den jeweils aktuell **substituierten** aktuellen Parameter
2. **Typ-Kompatibilität** :
   Die Datentypen der korrespondierenden **formalen** und **aktuellen** Parameter müssen gleich sein
3. **Variablen-** (**VAR**) Parameter (**Referenz**-Parameter) :
   VAR-Parameter können **Ergebnisse von Berechnungen** aus einer Prozedur heraus an ihre Aufrufumgebung transportieren
4. **Aktueller VAR**-Parameter :
   Ein aktueller VAR-Parameter darf **kein Ausdruck** (daher auch **keine Konstante**) sein, selbst wenn dem formalen Parameter keine Zuweisung erfolgt

Funktions-Prozeduren

1. **Resultat** :
   Eine Funktions-Prozedur liefert ein (und nur ein !) Resultat, das am Ort ihres Aufrufs Argument eines Ausdrucks ist
2. **Struktur** von **Resultaten** :
   Das Resultat einer Funktions-Prozedur kann selbst **nicht strukturiert** sein (**erlaubt** sind : einfacher Datentyp, Zeiger (folgt später !))
3. **Seiteneffekte** :
   Erzeugt eine Funktions-Prozedur sekundäre Resultate, so hat sie (unkontrollierte) Seiteneffekte. Diese müssen **äusserst vorsichtig benutzt** werden – generell ist dies nicht empfehlenswert! Stattdessen sollte
   a) eine reguläre Prozedur verwendet werden, die ihre Ergebnisse über **VAR**-Parameter zurückgibt, oder
   b) das Ergebnis der Funktions-Prozedur ein **Zeiger**-Typ auf eine Struktur (Feld, Verbund) sein

Lokalität, Blockstruktur und Sichtbarkeit

1. **Deklarationen** :
   Prozedur-Deklarationen können verschachtelt werden; entsprechend verschachtelt sind die **Sichtbarkeitsbereiche** (‚scopes') der lokal deklarierten Objekte (Konstante, Typen, Variable, Prozeduren). Lokale Prozedur-Deklarationen nehmen die Form eines separaten Programms an

2. **Sichtbarkeitsregeln – Gültigkeit von Objekten / Namen** :
   Prozedur-Aufruf → **Ausführung** der Anweisungsfolge einer Prozedur (~ **Block**) nach Übergabe von Objekten (~ aktuelle **Parameter**) der aufrufenden Umgebung

   Bestimmung des **Sichtbarkeitsbereiches** eines Objektes durch Bestimmung derjenigen Ebene, in der sich die Deklaration des betreffenden Objektes befindet (Suche **von Innen nach Aussen** ...!)

a) **Globale Objekte** :
   Objekte (Name und Typ !), die <u>ausserhalb</u> dieses Blocks definiert wurden, aber trotzdem innerhlab dieses Blocks verwendet werden;

   Formale VAR-Parameter (Referenz-Parameter) verhalten sich wie globale Objekte

b) **Lokale Objekte** :
   Objekte, die innerhalb dieses Blocks (im Deklarationsteil) vereinbart wurden und auchhierin verwendet werden – nach Beendigung der Abarbeitung des Blocks wird das Objekt „zerstört", d.h. Dessen Wert geht verloren;

   Formale Werte-Parameter verhalten sich wie lokale Objekte

c) **Standard-Namen** in Modula-2 werden – in einer imaginären, das Programm umschliessenden Prozedur – als bereits deklariert angesehen

3. Lokale Deklarationen

Vorteile lokaler Deklarationen :

a) Strukturierte Programmierung:
Verdeutlichung, dass ein Objekt auf eine Prozedur beschränkt bleibt, also gewöhnlich nur auf einen kleinen Teil des Gesamtprogramms

b) Fehlerdetektion:
Bei der Compilation kann die unbeabsichtigte Verwendung lokaler Objekte von anderen Programmteilen aus bemerkt werden

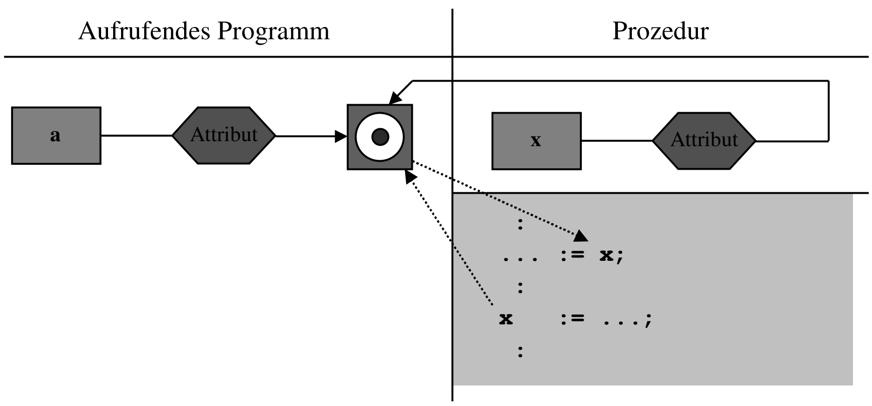
c) Speichereffizienz:
Bei der Implementierung erlauben lokale Deklarationen eine Verringerung des Speicherbedarfs (der Speicherplatz für eine lokale Variable kann für eine andere Variable verwendet werden)

Unterprogramm-Techniken

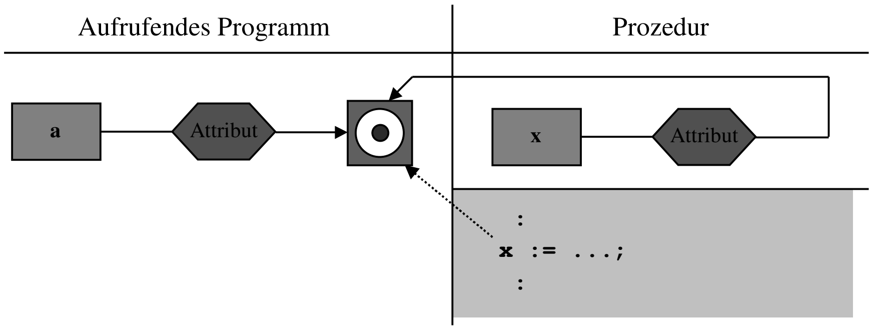
Techniken der Verwendung von Referenz-Parametern

Referenz- (Variablen-) Parameter

1. Transiente (= Zugriffs-) Parameter



2. Ergebnis- / Resultat-Parameter



→ ausschliesslich verändernde Wirkung;
der Wert der Parametervariablen beim Aufruf der Prozedur ist unerheblich !

⇒ Resultatparameter können als Hilfsvariable im Prozedurrumpf verwendet werden !

Aufspaltung eines transienten Parameters in

- einen gewöhnlichen Werte-Parameter („call by value") und
- einen Resultat-Parameter (Referenz-Parameter, „call by reference")

Bsp.:

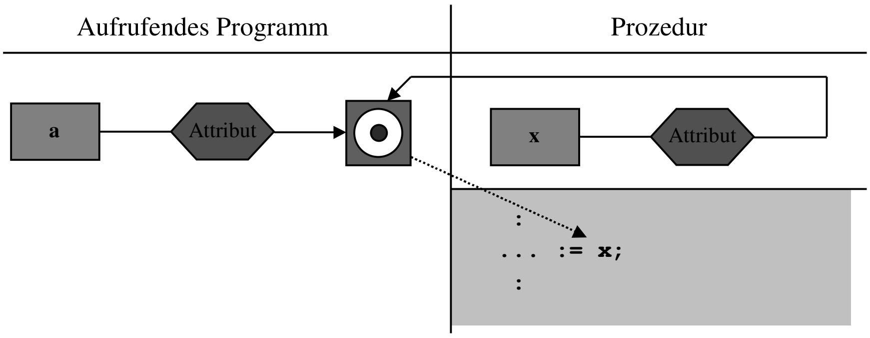
```
PROCEDURE incr1(VAR x :  INTEGER);

BEGIN
  x :=  x + 1
END incr1;
```

kann ersetzt werden durch

```
PROCEDURE incr2(     n :  INTEGER;
                 VAR x :  INTEGER);

BEGIN
  x :=  n + 1
END incr2;
```

Die gleichwertigen Aufrufe lauten : `incr1(a)` ⟺ `incr2(a, a)`

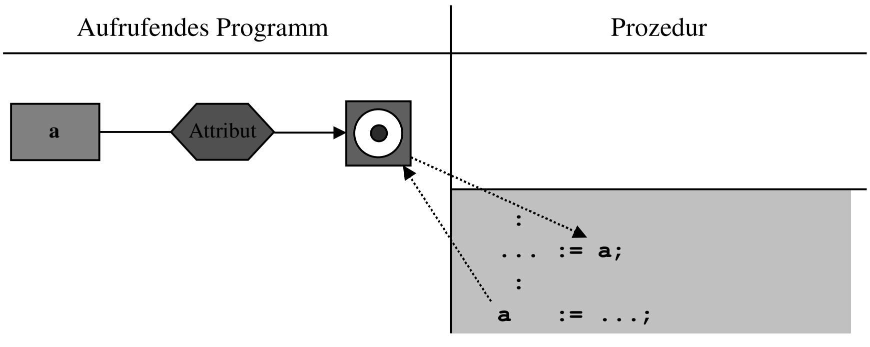
3. Eingabe-Parameter



→ gleiche Wirkung wie Werteparameter („call by value“), d.h. ausschliesslich rezipierende Wirkung

⇒ hier jedoch **ohne** Kopieren des Eingabewertes !

4. Unterdrückte Variablen-Parameter



→ Variablen-Parameter können unterdrückt werden, wenn ihre Veränderung nicht vorgesehen ist; die Unterdrückung von Variablenparametern führt bei Änderungen zur Manipulation (globaler) Variablen (~ unkontrollierte **Seiteneffekte**)

Achtung :
- häufig werden bestimmte Seiteneffekte übersehen
- führt schnell zu unübersichtlichen Programmen
- „Gleichsetzungstabu“ (siehe oben)

Beispiel für den sinnvollen Einsatz von Seiteneffekten

Berechnung der Binomialkoeffizienten (Pascal'sches Dreieck) → schrittweise Verfeinerung

Exkurs : Binomialverteilung

$$p_{\gamma,N}[k] = \binom{N}{k} \gamma^k (1-\gamma)^{N-k} \quad , k = 0, 1, ..., N$$

mit N : Gesamtzahl der Argumente
$\mu = \gamma N$: Mittelwert

Für $\gamma = 1/2$ ist die Binomialverteilung gerade (achsen-symmetrisch)

$$C_k^N \equiv \binom{N}{k} \stackrel{def}{=} \begin{cases} \dfrac{N!}{k!(N-k)!} & \text{für} \quad 0 \le k \le N \\ 0 & \text{für} \quad 0 \le N < k \end{cases}$$ (Binomialkoeffizient)

Pascal'sches Dreieck ($\delta = 1 - \gamma$) :

$$(\gamma + \delta)^N = C_0^N \gamma^N + C_1^N \gamma^{N-1} \delta + C_2^N \gamma^{N-2} \delta^2 + \ldots + C_N^N \delta^N$$

N	Werte der Binomialkoeff.	$\sigma = \sqrt{N\gamma(1-\gamma)}$
0	1	0
1	1 1	1/2
2	1 2 1	$1/\sqrt{2}$
3	1 3 3 1	$\sqrt{(3/2)}$
4	1 4 6 4 1	1
5	1 5 10 10 5 1	$\sqrt{(5/2)}$
6	1 6 15 20 15 6 1	$\sqrt{3}$
7	1 7 21 35 35 21 7 1	$\sqrt{(7/2)}$
8	1 8 28 56 70 56 28 8 1	2
9	1 9 36 84 126 126 84 36 9 1	$\sqrt{(9/2)}$
:	...	

Programmausdruck mit Ergebnissen

```
MODULE BinomialKoeff;
  FROM InOut IMPORT WriteInt, Write, WriteString, WriteLn;

CONST
  nmax =  10;

VAR
  binomi  :  ARRAY [0..nmax], [0..nmax] OF INTEGER;
  n, k, l :  INTEGER;

PROCEDURE binom;

BEGIN (* -- belegt das Feld 'binomi' *)
  FOR n := 0 TO nmax DO
    binomi[0, n] := 1;
    binomi[n, n] := 1
  END;

  FOR n := 1 TO nmax DO
    FOR k := 1 TO n - 1 DO
      binomi[n, k] := binomi[n-1, k-1] + binomi[n-1, k]
    END
  END
END binom;

BEGIN (* -- Hauptprogramm -- *)
  WriteString("Pascalsches Dreieck"); WriteLn;
  binom;
  FOR n := 1 TO nmax DO
    FOR l := 3 * n TO 3 * nmax DO
      Write(" ")
    END;
    FOR k := 1 TO n DO
      WriteInt(binomi[n, k], 6)
    END;
    WriteLn
  END
END BinomialKoeff.
```

Ergebnis-Berechnung :

```
Pascalsches Dreieck
                           1
                        1     1
                     1     2     1
                  1     3     3     1
               1     4     6     4     1
            1     5    10    10     5     1
         1     6    15    20    15     6     1
      1     7    21    35    35    21     7     1
   1     8    28    56    70    56    28     8     1
1     9    36    84   126   126    84    36     9     1
```

Probleme mit Referenz-Parametern

Gleichsetzungstabu für Referenz-Parameter

Zwei verschieden bezeichnete Referenz- (Variablen-) Parameter dürfen beim Aufruf **nicht** mit derselben Variablen (~ aktueller Parameter) besetzt werden !

Bsp.: Matrix-Multiplikation in der linearen Algebra

geg.: Matrizen A und B

$$A = \begin{pmatrix} 2 & 1 \\ -1 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

gesucht : $Z = A * B$

Schema :

$$*$$

$$X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \qquad Y = \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} \qquad Z = \begin{pmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{pmatrix}$$

Programmsegment

→ Entwurfskriterium **Effizienz**: Matrizen kopieren ist zeitaufwendig, daher verwenden wir hier Referenz-Parameter !

```
  :
TYPE
  MATRIX =  ARRAY [1..2][1..2] OF INTEGER;

PROCEDURE M(VAR X, Y, Z :  MATRIX);

BEGIN
 Z[1, 1] := X[1, 1] * Y[1, 1]  +  X[1, 2] * Y[2, 1];
 Z[1, 2] := X[1, 1] * Y[1, 2]  +  X[1, 2] * Y[2, 2];
 Z[2, 1] := X[2, 1] * Y[1, 1]  +  X[2, 2] * Y[2, 1];
 Z[2, 2] := X[2, 1] * Y[1, 2]  +  X[2, 2] * Y[2, 2]
END M;
```

Verknüpfungen :

Korrektes Resultat

Berechnung:		Aufruf:
$C := A * B$ ⇒	$C = \begin{pmatrix} 7 & 0 \\ 0 & 7 \end{pmatrix} = 7 \cdot \mathbf{I}$	$M(A, B, C)$
$A := A * B$ ⇒	$A = \begin{pmatrix} 7 & -5 \\ 0 & 6 \end{pmatrix}$	$M(A, B, A)$
$B := A * B$ ⇒	$B = \begin{pmatrix} 7 & 0 \\ -4 & 6 \end{pmatrix}$	$M(A, B, B)$

→ siehe Modula-2 Programm „`MatrixVariablenParam.m2`“

Korrekturmöglichkeit : Die Berechnungen liefern stets korrekte Resultate, wenn in M für X und Y die Werte-Substitution gewählt wird !

→ siehe Modula-2 Programm „`MatrixWerteParam.m2`“

Programmausdrucke mit Ergebnissen

Programm (mit Referenz- (Variablen-) Parametern) :

```
MODULE MatrixVariablenParam;
  FROM InOut IMPORT WriteString, WriteInt, WriteLn;

TYPE
  MATRIX =  ARRAY [1..2], [1..2] OF INTEGER;

VAR
  A, B, C :  MATRIX;

PROCEDURE InitMat(
  VAR mat            :  MATRIX;
      ul, ur, ll, lr :  INTEGER);

BEGIN
  mat[1, 1] := ul;   mat[1, 2] := ur;
  mat[2, 1] := ll;   mat[2, 2] := lr
END InitMat;

PROCEDURE M(
  VAR X, Y, Z :  MATRIX);   (* alle Argumente Variablen-Param. *)

BEGIN
  Z[1, 1] := X[1, 1] * Y[1, 1]  +  X[1, 2] * Y[2, 1];
  Z[1, 2] := X[1, 1] * Y[1, 2]  +  X[1, 2] * Y[2, 2];
  Z[2, 1] := X[2, 1] * Y[1, 1]  +  X[2, 2] * Y[2, 1];
  Z[2, 2] := X[2, 1] * Y[1, 2]  +  X[2, 2] * Y[2, 2]
END M;

PROCEDURE PrintMat(
  mat :  MATRIX);

BEGIN
  WriteString("mat[] = "); WriteInt(mat[1, 1], 3);
  WriteString(", ");       WriteInt(mat[1, 2], 3); WriteLn;
  WriteString("       "); WriteInt(mat[2, 1], 3);
  WriteString(", ");       WriteInt(mat[2, 2], 3); WriteLn; WriteLn
END PrintMat;

BEGIN
(* --   set 1   -- *)
  InitMat(A,  2,  1, -1,  3);
  InitMat(B,  3, -1,  1,  2);
  InitMat(C,  0,  0,  0,  0);

  WriteString("*** Test 1 ***"); WriteLn; WriteLn;
  M(A, B, C);
  WriteString("Resultat in Matrix C :"); WriteLn; PrintMat(C);
  M(A, B, A);
  WriteString("Resultat in Matrix A :"); WriteLn; PrintMat(A);

(* --   set 2   -- *)
  InitMat(A,  2,  1, -1,  3);
  InitMat(B,  3, -1,  1,  2);
  InitMat(C,  0,  0,  0,  0);

  WriteString("*** Test 2 ***"); WriteLn; WriteLn;
  M(A, B, C);
  WriteString("Resultat in Matrix C :"); WriteLn; PrintMat(C);
  M(A, B, B);
  WriteString("Resultat in Matrix B :"); WriteLn; PrintMat(B)
END MatrixVariablenParam.
```

Cont'd ...

Ergebnisse :

```
*** Test 1 ***

Resultat in Matrix C :
mat[] =   7,   0
          0,   7

Resultat in Matrix A :
mat[] =   7,  -5
          0,   6

*** Test 2 ***

Resultat in Matrix C :
mat[] =   7,   0
          0,   7

Resultat in Matrix B :
mat[] =   7,   0
         -4,   6
```

Cont‘d ...

Programm (mit Werte-Parametern) :

```
MODULE MatrixWerteParam;
  FROM InOut IMPORT WriteString, WriteInt, WriteLn;

TYPE
  MATRIX =  ARRAY [1..2], [1..2] OF INTEGER;

VAR
  A, B, C :  MATRIX;

PROCEDURE InitMat(
  VAR mat            :  MATRIX;
      ul, ur, ll, lr :  INTEGER);

BEGIN
  mat[1, 1] := ul;   mat[1, 2] := ur;
  mat[2, 1] := ll;   mat[2, 2] := lr
END InitMat;

PROCEDURE M(
      X, Y :  MATRIX;    (* Eingabe:  Werte-Param.             *)
  VAR Z    :  MATRIX);   (* Ergebnis: Variablen/Referenz-Param. *)

BEGIN
  Z[1, 1] := X[1, 1] * Y[1, 1]  +  X[1, 2] * Y[2, 1];
  Z[1, 2] := X[1, 1] * Y[1, 2]  +  X[1, 2] * Y[2, 2];
  Z[2, 1] := X[2, 1] * Y[1, 1]  +  X[2, 2] * Y[2, 1];
  Z[2, 2] := X[2, 1] * Y[1, 2]  +  X[2, 2] * Y[2, 2]
END M;
```

```
PROCEDURE PrintMat(
  mat :  MATRIX);

BEGIN
  WriteString("mat[] = "); WriteInt(mat[1, 1], 3);
  WriteString(", ");       WriteInt(mat[1, 2], 3); WriteLn;
  WriteString("        "); WriteInt(mat[2, 1], 3);
  WriteString(", ");       WriteInt(mat[2, 2], 3); WriteLn; WriteLn
END PrintMat;

BEGIN
(* --   set 1   -- *)
  InitMat(A,  2,  1, -1,  3);
  InitMat(B,  3, -1,  1,  2);
  InitMat(C,  0,  0,  0,  0);

  WriteString("*** Test 1 ***"); WriteLn; WriteLn;
  M(A, B, C);
  WriteString("Resultat in Matrix C :"); WriteLn; PrintMat(C);
  M(A, B, A);
  WriteString("Resultat in Matrix A :"); WriteLn; PrintMat(A);

(* --   set 2   -- *)
  InitMat(A,  2,  1, -1,  3);
  InitMat(B,  3, -1,  1,  2);
  InitMat(C,  0,  0,  0,  0);

  WriteString("*** Test 2 ***"); WriteLn; WriteLn;
  M(A, B, C);
  WriteString("Resultat in Matrix C :"); WriteLn; PrintMat(C);
  M(A, B, B);
  WriteString("Resultat in Matrix B :"); WriteLn; PrintMat(B)
END MatrixWerteParam.
```

Cont‘d ...

Ergebnisse :

```
*** Test 1 ***

Resultat in Matrix C :
mat[] =   7,   0
          0,   7

Resultat in Matrix A :
mat[] =   7,   0          ☑
          0,   7

*** Test 2 ***

Resultat in Matrix C :
mat[] =   7,   0
          0,   7

Resultat in Matrix B :
mat[] =   7,   0          ☑
          0,   7
```

- **Prozeduren** dienen der **Strukturierung** und stellen wiederverwendbare Programmabschnitte dar;
  formale Parameter dienen als Platzhalter für **aktuelle Parameter**
- Als **Substitutionsart** von Parametern werden **Werte-** und **Variablen-** (Referenz-) **Parameter** unterschieden
- **Seiteneffekte** werden erzeugt, wenn aus einem Block heraus die Zustände von Umgebungsvariablen verändert werden (→ stets **Vorsicht !**)
- **Funktions-Prozeduren** nehmen selbst einen **(Ergebnis-) Wert** an, dessen **Typ** ist auf einfache Typen und Zeiger-Typen beschränkt;
  Ergebnisse von Prozeduren werden über VAR-Parameter zurückgegeben
- Objekte (mit Namen) werden innerhalb von **Blöcken** (~ Prozedur) definiert (von der Programmumgebung aus nicht sichtbar), aus einem Block heraus sind Objekte der **Umgebung sichtbar**;
  bei **Namenskonflikten** werden Objekte in weiter aussen liegenden Sichtbarkeits-bereichen überdeckt
- Die **Lebensdauer** von Objekten ist auf die **Ausführungszeit eines Blocks** (~ Prozedur) beschränkt – nach Abschluss der Ausführung einer Prozedur werden alle Konstanten, Variablen, etc. eliminiert