Teil VI: Prozeduren – Feld-Parameter & Typen

1. Offene Array-Parameter
2. Prozedurtypen und -variablen

Offene ARRAY-Parameter

Motivation

Problem : geg.: mehrere deklarierte Felder unterschiedlicher Längen, die als Variablen neuer Datentypen definiert wurden

```
TYPE
  Vektor  =  ARRAY [1..20] OF INTEGER;
  Zahlen  =  ARRAY [0..99] OF INTEGER;
  Nummern =  ARRAY [`A`..`Z`] OF INTEGER;
```

Aufgabe : Implementierung einer Prozedur, die nach einer bestimmten Zahl in Variablen der o.g. Datentypen sucht !

Beobachtung : Wegen der strengen Forderung von Typkompatibilität sind **drei (!)** Prozeduren für die Berechnungen notwendig :

```
PROCEDURE IstInVektor(x         :  Vektor;
                      suchElem  :  INTEGER)  :  BOOLEAN;
PROCEDURE IstInZahlen(x         :  Zahlen;
                      suchElem  :  INTEGER)  :  BOOLEAN;
PROCEDURE IstInNummern(x        :  Nummern;
                       suchElem :  INTEGER)  :  BOOLEAN;
```

Untersuchung

Der einzige Unterschied zwischen den Prozeduren besteht in dem Intervall, in dem die Suche stattfindet (~ Länge der Felder)

```
1..20          (IstInVektor)
0..99          (IstInZahlen)
`A`..`Z`       (IstInNummern)
```

Hinweis : Dasselbe Problem würde vorliegen, wenn nur Variablen deklariert worden wären, die als ARRAYs von INTEGER-Elementen, aber unterschiedlicher Länge definiert wären :

```
VAR
  Vektor   :  ARRAY [1..20] OF INTEGER;
  Zahlen   :  ARRAY [0..99] OF INTEGER;
  Nummern  :  ARRAY [`A`..`Z`] OF INTEGER;
```

Man kann **nicht** einfach eine Auswerte-Prozedur definieren, bei der ein ARRAY maximaler Länge definiert wird, z.B.

```
PROCEDURE IstInVektor(x         :  ARRAY [0..1000];
                      suchElem  :  INTEGER)  :  BOOLEAN;
```

Vereinigung der Lösungen mittels offener ARRAY-Paramete

```
PROCEDURE IstInFeld(x         :  ARRAY OF INTEGER;
                    suchElem  :  INTEGER) :  BOOLEAN;

VAR
  i :  CARDINAL;

BEGIN
  FOR i := 0 TO HIGH(x) DO
    IF x[i] = suchElem THEN
      RETURN TRUE
    ELSE
      RETURN FALSE
    END
  END
END IstInFeld;
```

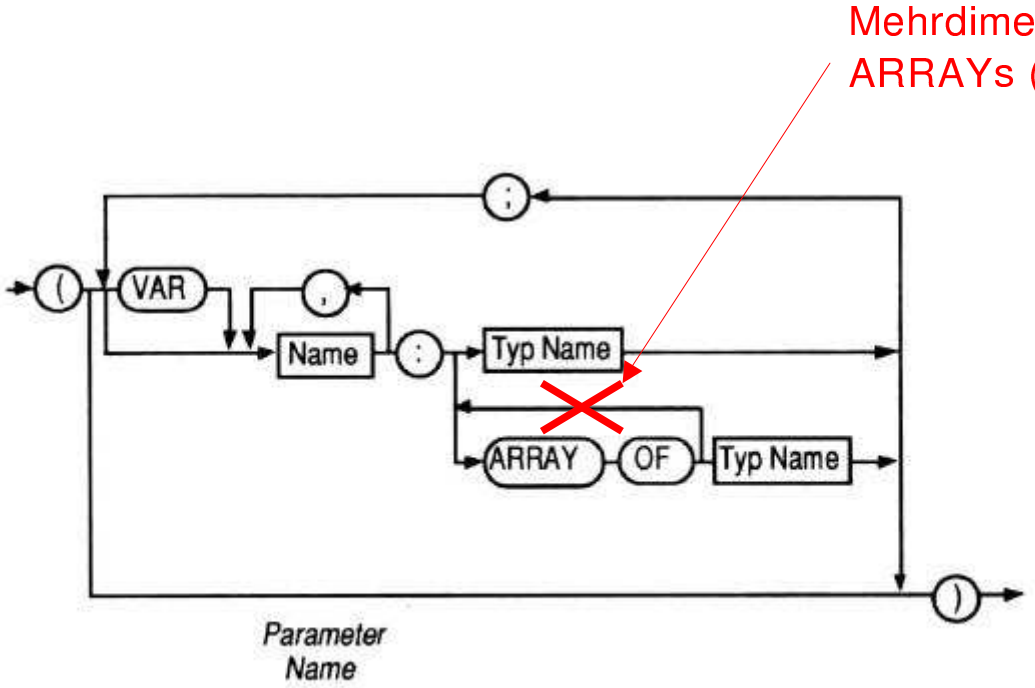
Dimensionen werden „offen“ gelassen

Standard-Prozedur in Modula-2 zur Bestimmung des obersten Elements eines ARRAYs (Ergebnis vom Typ CARDINAL)

Hinweis: für arr : ARRAY [min..max] OF T
liefert HIGH(arr) = ORD(max) – ORD(min)

Syntax

Definition



Erläuterungen

Formal-Parameter : **`ARRAY OF T`**

Aktueller Parameter : **`ARRAY Index-Typ OF T`**

beliebig !

Prozedurinterne Behandlung offener Arrays **`arr`**

untere Indexgrenze : 0

obere Indexgrenze : HIGH(arr) = ORD(max) – ORD(min) (= length(arr) – 1)

Mehrdimensionale offene ARRAY

→ In verschiedenen Realisierungen (Implementierungen der Sprache) sind **mehrdimensionale** offene ARRAYs möglich !

Für die Bestimmung der Anzahl der Elemente des offenen ARRAYs **`arr`** in den einzelnen Dimensionen 2, 3, etc. wird die Funktion HIGH für **`arr[0]`** , **`arr[0,0]`**, etc. aufgerufen !

(→ siehe Erläuterungen in
J. Puchan et al. (1994) Programmieren mit Modula-2. Stuttgart, Teubner)

Hinweis

In der Ulmer Modula-2 Implementierung sind mehrdimensionale offene ARRAYs nicht erlaubt !

Beispiel

Von einem beliebigen (1-D) Eingabe-Feld wird dessen Länge bestimmt

2- und 3-dimensionale offene ARRAYs sind in dieser Modula-2 Implementierung nicht erlaubt !

```
MODULE OffeneArrays;
  FROM InOut IMPORT WriteString, WriteCard, WriteInt, WriteLn;

CONST
  min =  -7;
  max =   2;

TYPE
  FARBE =  (rot, gelb, blau);

VAR
  Feld1D :  ARRAY [1..10] OF CARDINAL;
  Feld2D :  ARRAY [1..5] OF
                  ARRAY [1..3] OF CARDINAL;
  Feld3D :  ARRAY [min..max] OF
                  ARRAY BOOLEAN OF
                        ARRAY FARBE OF REAL;

PROCEDURE ArrayDimensionen1(
  f :  ARRAY OF CARDINAL);

BEGIN
  WriteString("Bestimmung der Dimension des (offenen) 1-D Feldes ...");
  WriteLn;
  WriteInt(HIGH(f) + 1, 5)
END ArrayDimensionen1;

PROCEDURE ArrayDimensionen2(
  f :  ARRAY OF ARRAY [1..3] OF CARDINAL);

BEGIN
  WriteString("Bestimmung der Dimension des (offenen) 2-D Feldes ...");
  WriteLn;
  WriteInt(HIGH(f) + 1, 5)
END ArrayDimensionen2;

(*PROCEDURE ArrayDimensionen3(
  f :  ARRAY [min..max] OF ARRAY BOOLEAN OF ARRAY FARBE OF REAL);

BEGIN
  WriteString("Bestimmung der Dimensionen des (offenen) n-D Feldes ...");
  (*WriteLn;
  WriteInt(HIGH(f), 5);
  WriteInt(HIGH(f[0]), 5);
  WriteInt(HIGH(f[0, 0]), 5)*)
END ArrayDimensionen3;*)


BEGIN
  ArrayDimensionen1(Feld1D);
  WriteLn
END OffeneArrays.
```

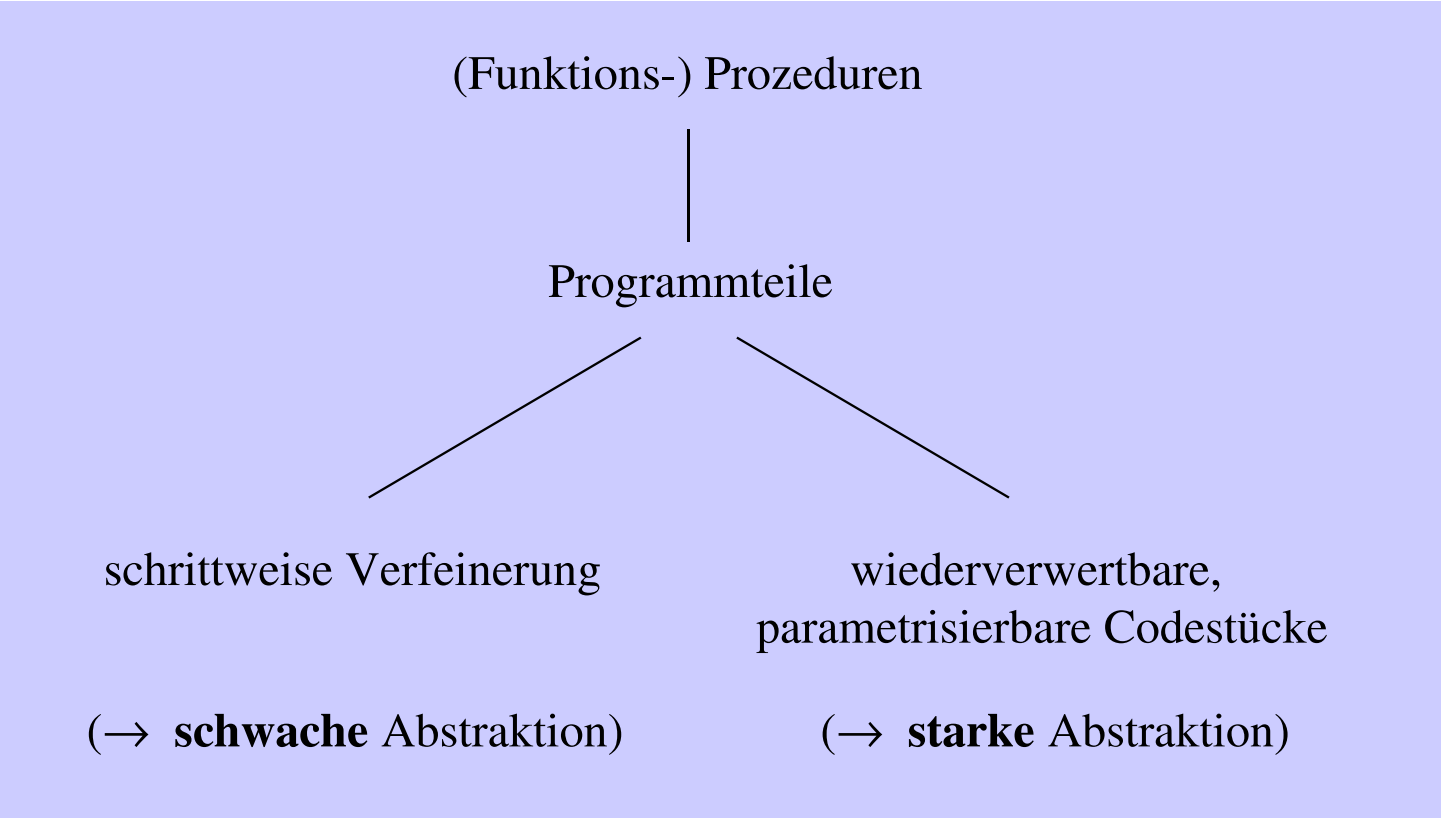
Ergebnis :

```
Bestimmung der Dimension des (offenen) 1-D Feldes ...
   10
```

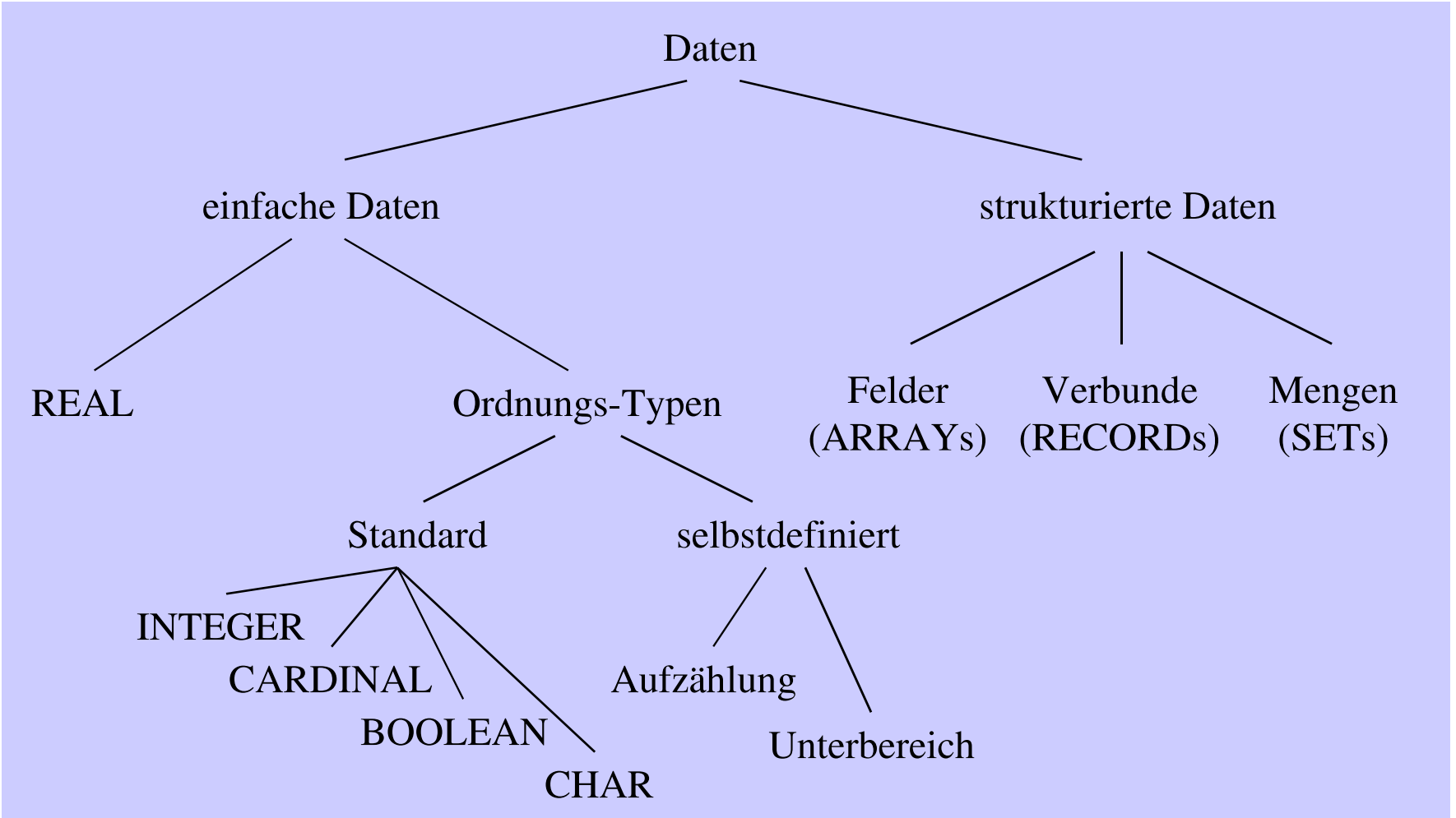
Prozedurtypen und -variablen

Einordnung: Objekte und Abstraktion

Prozeduren für schwache und starke Abstraktion



Daten-Typen


Daten
einfache Daten
strukturierte Daten
REAL
Ordnungs-Typen
Felder
(ARRAYs)
Verbunde
(RECORDs)
Mengen
(SETs)
Standard
selbstdefiniert
INTEGER
CARDINAL
BOOLEAN
CHAR
Aufzählung
Unterbereich

Anwendungsproblem

Aufgabe 1

Für **verschiedene Funktionen** $y = f(x)$ sollen Felder (ARRAYs) mit **Funktionswerten für diskrete Argumente** generiert werden

Bsp.: $f_1(x)$: Gauss'sche Normalverteilungs-Fkt.
$f_2(x)$: sin-Fkt.
$f_3(x)$: Produkt aus Gauss-Fkt. und sin-Fkt., $f_3(x) = f_1(x) * f_2(x)$

Naiver Lösungs-Ansatz :

Für jede der Funktionen wird **jeweils eine Prozedur** definiert, die alle das **Feld mit Funktionswerten (Koeffizienten)** mittels eines formalen Variablen-Parameters zurückgeben !

beispielsweise ...

```
  :
CONST
  PI   =    3.14159;
  XMIN =  -20;
  XMAX =   20;

TYPE
  mask = ARRAY [XMIN..XMAX] OF REAL;

PROCEDURE DiscreteGaussian(    mean, std :  REAL;
                           VAR y         :  mask);

VAR
  k        :  INTEGER;
  x, norm  :  REAL;

BEGIN
  norm :=  1.0 / (sqrt(2.0 * PI) * std);
  FOR k := XMIN TO XMAX DO
    x    :=  FLOAT(k) - mean;
    y[k] :=  norm * exp(- x * x / (2.0 * std * std))
  END
END DiscreteGaussian;
  :
```

$$f(x) = \frac{1}{\sqrt{2\pi}\cdot\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Nachteil

Für jede (mathematische) Funktion muss eine eigene PROZEDUR geschrieben werden, in deren Rumpf die Werte des Feldes berechnet und zugewiesen werden

→ die Realisierung ist eine **Mischung** aus der **Definition der mathematischen Funktion** und der iterativen Lösung zur **Bestimmung der diskreten Funktionswerte** und ihrer **Zuweisung zu indizierten Feldkomponenten**

Wünschenswert

Trennung in

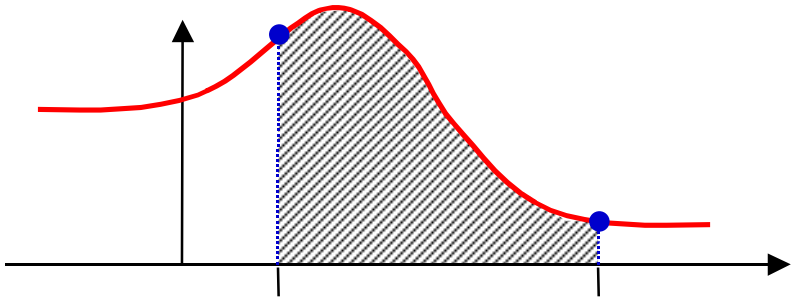
- Definition der mathematischen Funktion und
- „generischer“ Routine, die für jedes Argument der gegebenen Funktion den berechneten Wert zuweist

Aufgabe 2

Für verschiedene Funktionen $y = f(x)$ soll für ein gegebenes Intervall x_{beg} und x_{end} **numerisch die Fläche der Funktion** bestimmt werden

Als **numerische Integrationsverfahren** stehen verschiedene Ansätze zur Verfügung, z.B. Auswertung der

- Simpson-Methode oder
- Romberg-Methode



Lösungs-Ansatz (→ Aufgabe 1)

Für jedes numerische Verfahren wird eine Prozedur definiert, die die jeweils zu integrierenden Funktion als Eingabe erhält !

Wünschenswert

Definition einer „generischen“ Lösung, die als Eingabe die **Funktion** sowie die **Integrationsmethode** erhält !

Definition von Prozedur-Typen

Weitere Abstraktionsmöglichkeit ..

- Prozeduren sind selbst Objekte (~ Platzhalter mit Wert)
- Zuweisung an Variablen

  ⇒ Prozedur-Deklaration :
  Spezialfall der Konstanten-Deklaration;
  Wert der Konstanten ist eine Prozedur !

Deklaration

- Festlegung der **Anzahl** und **Typen** der formalen Parameter
- **Art** der formalen Parameter : Wert - oder
  Variablen- (Referenz-) Parameter
- bei Funktions-Prozeduren → Rückgabetyp

Beispiele

```
TYPE
  PROC      =  PROCEDURE;

  ProcTyp1  =  PROCEDURE(INTEGER,
                         REAL);
  ProcTyp2  =  PROCEDURE(VAR REAL,
                             CHAR);
  ProcTyp3  =  PROCEDURE(ARRAY OF CHAR);

  FuncTyp1  =  PROCEDURE(CARDINAL,
                         REAL) :  BOOLEAN;
  FuncTyp2  =  PROCEDURE();
  :
```

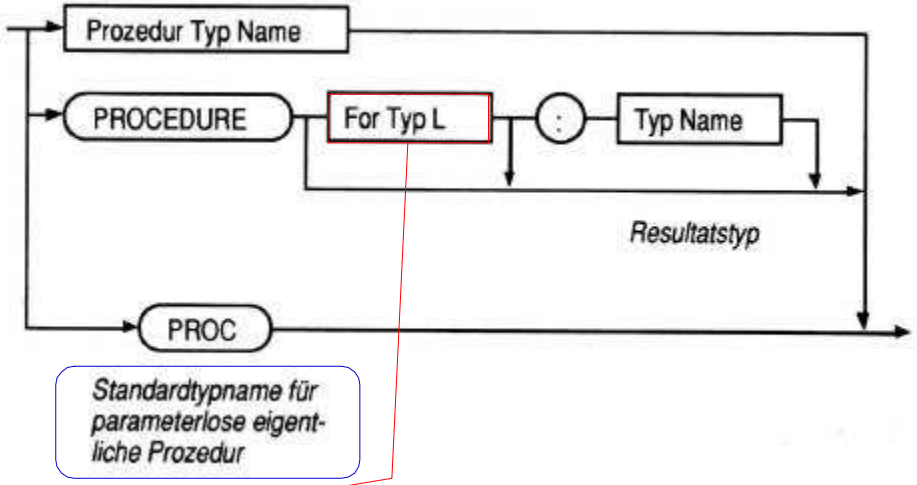
Standardtyp (→ bereits definiert !)

Proz. ohne Parameter

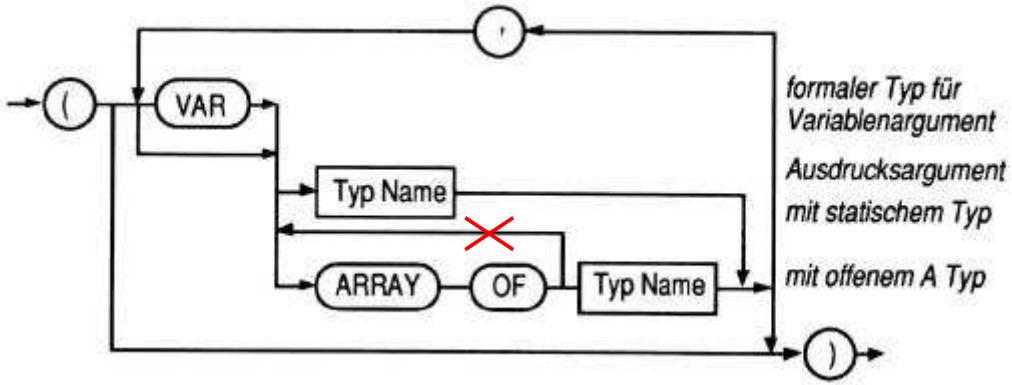
Leere Parameterliste für Funktions-Prozeduren

Synta

Prozedur-Typ



Formale Typliste (For Typ L)



Besonderheite

- Zuweisung von Prozeduren zu Prozedur-Variablen
  → **nur** für Prozeduren erlaubt, die **nicht** lokal in anderen Prozeduren deklariert sind
- **Nicht erlaubt :** Zuweisung von Standard-Prozeduren, z.B. exp(), sin(), ...
  (→ siehe Liste der Modula-2 Standard-Prozeduren !)
- **Prozedur-Kompatibilität** zwischen Prozedur-Variable **pv** und Prozedur-Typ **P**
  - **Pv** und **P** haben die gleiche Anzahl formaler Parameter
  - Die formalen Parameter von **pv** und **P** stimmen paarweise überein, d.h.
    der i-te Parameter von **pv** und
    der i-te Parameter von **P**
    sind
    . Vom selben Datentyp
    . Von gleicher Art (beide Werte- oder VAR-Parameter)
  - Bei **Funktions-Prozeduren** müssen die Datentypen der Ergebniswerte identisch sein

Prozeduren als Variable und Komponentenvariabl

Variable

Vordefinierter Typ parameterloser Prozeduren

vorher definierter Funktions-Prozedur Typ

→ Deklaration von Variablen eines Typs, der zuvor als Prozedurtyp definiert wurde

Beispiel :

```
  :
TYPE
  FUNC =  PROCEDURE(REAL): REAL;

PROCEDURE halt;

VAR
  ch :  CHAR;

BEGIN
  WriteString("Eingabe (Abbruch mit `q`) :  ");
  Read(ch);
  IF ch = `q` THEN
    HALT
  END
END halt;
```

← Funktions-Prozedur Typ

```
VAR
  f           :  FUNC;
  abbruch     :  PROC;
  erg1, erg2  :  REAL;

BEGIN
  f       :=  sin;
  abbruch :=  halt;

  erg1 :=  sin(0.2);
  erg2 :=  f(0.2);

  IF erg1 = erg2 THEN
    abbruch
  END
  :
END ProzVar.
```

← ~ halt-Prozedur

Komponentenvariabl

Bisher : strukturierte (Daten-) Objekte (z.B. ARRAY, RECORD) waren Kollektionen von Elementen einfacher oder ebenso strukturierter Daten

a) Felder (ARRAY) : Elemente vom selben Typ

b) Verbunde (RECORD) : Elemente unterschiedlicher Typen

Felder von Prozedur-Typen (→ Felder eines zuvor definierten Prozedur-Typs)

Bsp.:

```
TYPE
  OpProc   =  PROCEDURE(REAL);
  RealFunc =  PROCEDURE(REAL): REAL;

VAR
  ProcArr :  ARRAY [0..5] OF RealFunc;
  FuncArr :  ARRAY BOOLEAN OF OpProc;
  :
```

Verbunde mit Prozedur-Komponenten

Beschreibung von Objekten mit Datenattributen und Prozeduren (~ Operationen)

Beispiele – Verwendung von Prozedur-Typen

Beispiel 1 : Prozedur-Typen und Variablen von Funktions-Typen

```
MODULE ProcTypen;
  FROM InOut     IMPORT WriteString, WriteLn;
  FROM RealInOut IMPORT ReadReal, WriteReal;

CONST
  len =  31;

TYPE
  PrintStrProc =  PROCEDURE(ARRAY OF CHAR);
  RealFunc     =  PROCEDURE(REAL) :  REAL;
  CopyProc     =  PROCEDURE(    ARRAY OF CHAR,
                            VAR ARRAY OF CHAR);
```

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

BEGIN
  RETURN (x * x  -  2.0 * x  + 1.0)
END SquareFunc;
```

```
PROCEDURE CopyString(
      from :  ARRAY OF CHAR;
  VAR to   :  ARRAY OF CHAR);

VAR
  i :  CARDINAL;

BEGIN
  i := 0;
  WHILE ( from[i] <> 0C ) & ( i <= HIGH(from) ) & ( i <= HIGH(to) ) DO
    to[i] := from[i];
    i     := i + 1
  END;
  IF i <= HIGH(to) THEN
    to[i] := 0C
  END
END CopyString;
```

```
VAR
  string1, string2 :  ARRAY [0..len-1] OF CHAR;
  cp               :  CopyProc;
  ps               :  PrintStrProc;
  f                :  RealFunc;
  ff               :  PROC;
  x                :  REAL;

BEGIN (* -- Hauptprogramm -- *)
  string1 := "Reelle Zahl eingeben :  ";
  cp      := CopyString;
  ps      := WriteString;
  ff      := WriteLn;
  f       := SquareFunc;

  cp(string1, string2);
  ps(string2); ReadReal(x); ff;

  ps("Ergebnis :  x^2 - 2x + 1 = "); WriteReal(f(x), 16); ff
END ProcTypen.
```

Beispiel 3 : Prozedur-Typen, Variablen von Funktions-Typen und **generische Funktionen** (→ vgl. Aufgabenstellung zur Motivation)

```
MODULE KoeffFelder;
  FROM InOut     IMPORT WriteString, WriteLn;
  FROM RealInOut IMPORT WriteFloat;
  FROM MathLib   IMPORT exp, sqrt;

CONST
  KMin =  -10;
  KMax =   10;

TYPE
  KERNEL =  ARRAY [KMin..KMax] OF REAL;
  FUNC   =  PROCEDURE(REAL, REAL) : REAL;
```

```
PROCEDURE box(
  x   : REAL;
  rad : REAL) : REAL;

BEGIN
  IF ABS(x) <= rad THEN
    RETURN 1.0
  ELSE
    RETURN 0.0
  END
END box;
```

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

CONST
  pi =  3.14159;

BEGIN
  IF x <= 3.0 * sigma THEN
    RETURN ( 1.0 / (sqrt(2.0 * pi) * sigma) *
             exp(- x * x / (2.0 * sigma * sigma)) )
  ELSE
    RETURN 0.0
  END
END gauss;
```

$$f(x) = \frac{1}{\sqrt{2\pi}\cdot\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right)$$

```
PROCEDURE Dlgauss(
  x     : REAL;
  sigma : REAL) : REAL;

BEGIN
  RETURN ( - x / (sigma * sigma) * gauss(x, sigma) )
END Dlgauss;
```

```
PROCEDURE zeroKernel(
  VAR kern : ARRAY OF REAL);

VAR
  i : CARDINAL;

BEGIN
  FOR i := 0 TO HIGH(kern) DO
    kern[i] := 0.0
  END
END zeroKernel;
```

```
PROCEDURE makeKernel(
  VAR kern     : ARRAY OF REAL;
      min, max : INTEGER;
      genFunc  : FUNC;
      param    : REAL);

VAR
  i, len : INTEGER;

BEGIN
  len := HIGH(kern) + 1;
  IF (len < max - min + 1) THEN
    max := len DIV 2;
    min := -max
  END;

  FOR i := min TO max DO
    kern[i - min] := genFunc(FLOAT(i), param)
  END
END makeKernel;
```

```
PROCEDURE PrintKernel(
  kern : ARRAY OF REAL);

VAR
  i, len : CARDINAL;

BEGIN
  len := HIGH(kern) + 1;
  FOR i := 0 TO (len - 1) DIV 2 - 1 DO
    WriteFloat(kern[i], 3, 3)
  END;
  WriteString(" |");
  WriteFloat(kern[(len - 1) DIV 2], 3, 3);
  WriteString(" |");
  FOR i := (len - 1) DIV 2 + 1 TO len - 1 DO
    WriteFloat(kern[i], 3, 3)
  END;
  WriteLn
END PrintKernel;
```

```
VAR
  ff                                  : PROC;
  funcBox, funcGauss, funcDlGauss     : FUNC;
  kernelBox, kernelGauss, kernelDlGauss : KERNEL;
  boxRad, sigma                       : REAL;

BEGIN (* -- Hauptprogramm -- *)
  ff          := WriteLn;
  funcBox     := box;
  funcGauss   := gauss;
  funcDlGauss := Dlgauss;

  zeroKernel(kernelBox);
  zeroKernel(kernelGauss);
  zeroKernel(kernelDlGauss);

  boxRad :=  5.0;
  sigma  :=  1.5;
  makeKernel(kernelBox, KMin, KMax, funcBox, boxRad);
  makeKernel(kernelGauss, KMin, KMax, funcGauss, sigma);
  makeKernel(kernelDlGauss, KMin, KMax, funcDlGauss, sigma);

  WriteString("Koeffizienten in Listen : "); ff; ff;
  PrintKernel(kernelBox); ff;
  PrintKernel(kernelGauss); ff;
  PrintKernel(kernelDlGauss); ff
END KoeffFelder.
```

Resultate :

- Box-Funktion (Radius = 5)
- Gauss‘sche Dichtefunktion (Mittelwert = 0, Standardabweichung = 1.5)
- Ableitung 1. Ordnung der Gauss‘schen Dichte

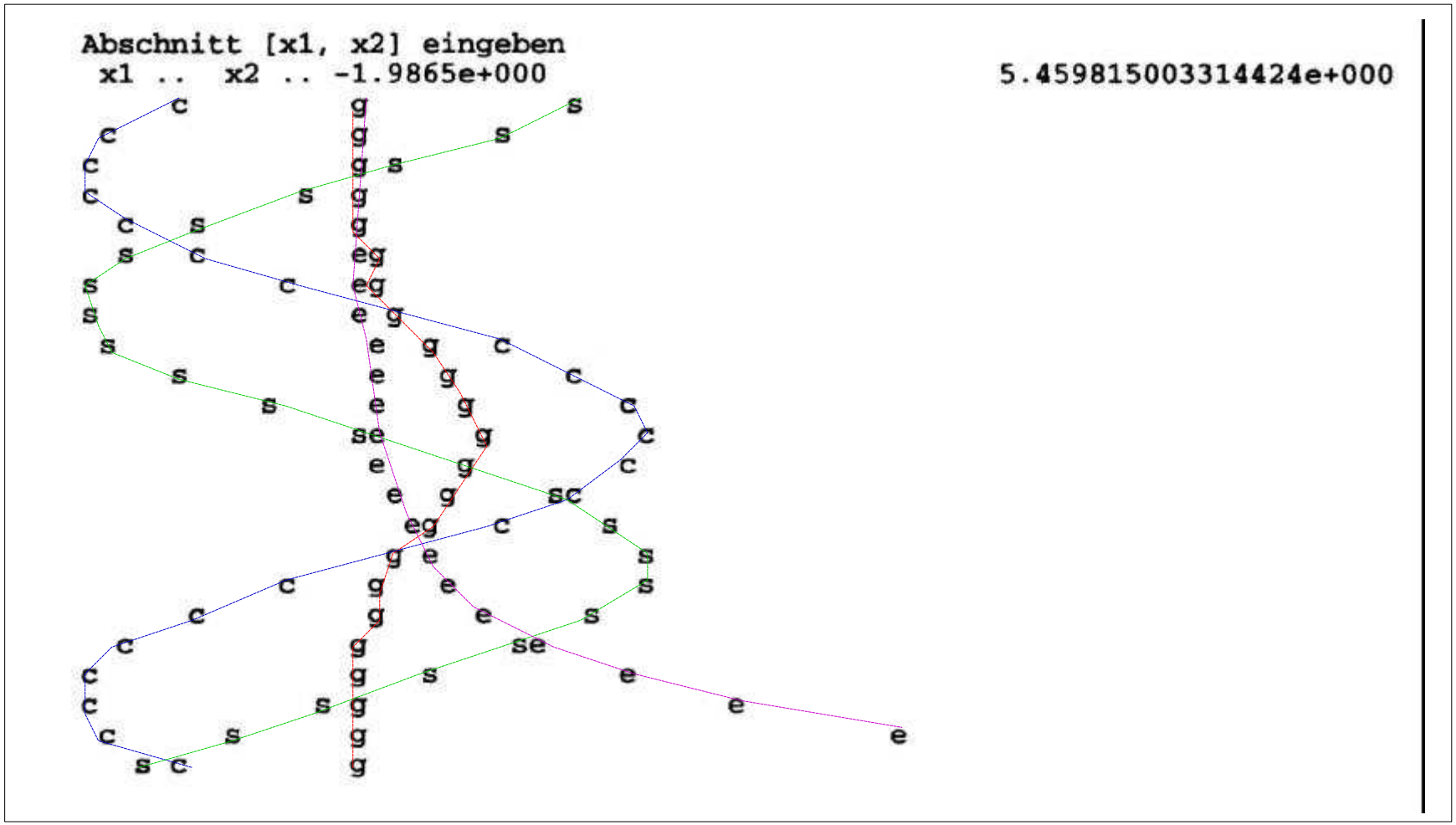
Koeffizienten in Listen :

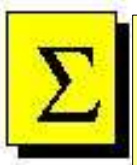
0.000	0.000	0.000	0.000	0.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.001	0.008	0.036	0.109	0.213	0.266	0.213	0.109	0.036	0.008	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.002	0.014	0.048	0.097	0.095	0.000	-0.095	-0.097	-0.048	-0.014	0.000	0.000	0.000	0.000	0.000	0.000

Beispiel 2 : Prozedur-Typen, Funktionen als **Komponentenvariablen** und generische Funktionen (→ vgl. Aufgabenstellung zur Motivation)

```
MODULE FktGraphen;
  FROM InOut     IMPORT WriteInt, Write, WriteString, WriteLn;
  FROM RealInOut IMPORT ReadReal, WriteReal;
  FROM MathLib   IMPORT exp, sin, cos, sqrt;

CONST
  ZeilenLaenge =  60;
  AnzFkt       =   4;

TYPE
  Fkt =  PROCEDURE(REAL) :  REAL;

VAR
  FktFeld     :  ARRAY [0..AnzFkt-1] OF Fkt;
  CharFeld    :  ARRAY [0..AnzFkt-1] OF CHAR;
  ScalingFeld :  ARRAY [0..AnzFkt-1] OF REAL;
  x1, x2      :  REAL;

PROCEDURE Graphen(
  funcArr :  ARRAY OF Fkt;
  charArr :  ARRAY OF CHAR;
  scalArr :  ARRAY OF REAL;
  x1, x2  :  REAL;
  n       :  CARDINAL);

CONST
  zMAX =  79;

VAR
  dx, fmax, fmin, scale, fx, x :  REAL;
  i, j, nr                     :  CARDINAL;
  zeile                        :  ARRAY [1..zMAX] OF CHAR;

BEGIN
  fmin := 0.0;
  fmax := 0.0;
  dx   := (x2 - x1) / FLOAT(n);

  FOR i := 0 TO n DO
    x := x1 + FLOAT(i) * dx;
    FOR nr := 0 TO HIGH(funcArr) DO
      fx   := funcArr[nr](x) * scalArr[nr];
      IF fx < fmin THEN
        fmin := fx
      END;
      IF fx > fmax THEN
        fmax := fx
      END
    END (* FOR nr *)
  END; (* FOR i *)

  WriteReal(fmin, 12);
  WriteReal(fmax, ZeilenLaenge-1-12); WriteLn;
  scale := FLOAT(ZeilenLaenge-1) / (fmax - fmin);

  FOR i := 0 TO n DO
    FOR j := 1 TO ZeilenLaenge - 1 DO
      zeile[j] := ' '
    END;

    x := x1 + FLOAT(i) * dx;
    FOR nr := 0 TO HIGH(funcArr) DO
      zeile[TRUNC((funcArr[nr](x) * scalArr[nr] - fmin) * scale) + 1] :=
        charArr[nr]
    END;

    FOR j := 1 TO ZeilenLaenge - 1 DO
      Write(zeile[j])
    END;
    WriteLn
  END
END Graphen;

PROCEDURE gauss(
  x :  REAL) :  REAL;

CONST
  pi =  3.14159;

BEGIN
  RETURN ( exp(- x * x / 2.0) / sqrt(2.0 * pi) )
END gauss;

BEGIN (* -- Hauptprogramm -- *)
  WriteString("Abschnitt [x1, x2] eingeben "); WriteLn;
  WriteString(" x1 .. "); ReadReal(x1);
  WriteString(" x2 .. "); ReadReal(x2);

  FktFeld[0] := exp;     CharFeld[0] := 'e';   ScalingFeld[0] := 0.1;
  FktFeld[1] := sin;     CharFeld[1] := 's';   ScalingFeld[1] := 2.0;
  FktFeld[2] := cos;     CharFeld[2] := 'c';   ScalingFeld[2] := 2.0;
  FktFeld[3] := gauss;   CharFeld[3] := 'g';   ScalingFeld[3] := 2.0;

  Graphen(FktFeld, CharFeld, ScalingFeld, x1, x2, 22)
END FktGraphen.
```

Ausgabe von Funktions-Graphen





- Zwecks flexibler Berechnung auf Daten-Feldern mit verschiedener Anzahl von Elementen sind **ARRAY-Parameter** mit **offener Länge** möglich – die Element-Typen der formalen und aktuellen Parameter müssen kompatibel sein !
- Ähnlich wie selbstdefinierte Datentypen können **Prozeduren** und **Funktions-Prozeduren** als **neue Typen** definiert werden, von denen auch Variable definiert werden können;
- Variablen dieser Prozedur-Typen können an **Prozeduren als aktuelle Parameter** übergeben werden – formale und aktuelle Parameter müssen auch hier kompatibel sein
- Prozeduren / Funktions-Prozeduren können **Elemente strukturierter Daten**, z.B. Felder oder Verbunde, sein