Pascal-Kurs für Anfänger (Teil 3)

Lernen Sie den Gebrauch der Pascal-Funktionen kennen. Diese ragen weit über den Standard von Basic hinaus. Lesen Sie, wie Sie mit Pascal eigene Datentypen definieren und verarbeiten können.

Pascal besitzt, ähnlich wie Basic, die bekannten Operatoren »+«, »-« , »*« und »/«. Die Division mit »/« darf allerdings nur auf Werte vom Typ REAL angewendet werden. Für die ganzzahlige Division ist der Operator »DIV« zu verwenden. Den Rest einer ganzzahligen Division erhält man mit »MOD«.

Das sieht beispielsweise so aus:

10 DIV 3 ergibt 3
10 MOD 3 ergibt 1
3.15 MOD 5 ist nicht erlaubt
5/2 ergibt 2.5

Die letzte Operation ist gestattet, wenn das Ergebnis einer Real-Variablen zugewiesen wird. In Ausdrücken werden, wie in Basic auch, zuerst die Division und Multiplikation, dann die Addition und Subtraktion bearbeitet. Eine andere Vorrangregelung erreicht man durch das Setzen von Klammern.

Logische Ausdrücke

Ein logischer Ausdruck wird aus den Operatoren AND, OR und NOT gebildet. Er hat als Ergebnis den Wert TRUE oder FALSE und kann einer Variablen vom Typ Boolean zugewiesen werden.

Die folgende Wertetabelle zeigt, welche Ergebnisse die logischen Operatoren liefern. In logischen Ausdrücken werden zuerst NOT, dann AND und zuletzt OR abgearbeitet.

Beispiele:

```
a = FALSE und b = TRUE
und c = TRUE
NOT a AND b
```

ergibt TRUE

```
a OR b AND NOT c
```

ergibt FALSE

Vergleichsoperatoren haben ebenfalls ein boolesches Ergebnis. Folgende Vergleichsoperatoren sind vorhanden:

= gleich
> größer
< kleiner
>= größer gleich
<= kleiner gleich
<> ungleich

Innerhalb eines Ausdruckes werden Vergleichsoperatoren ganz zum Schluß abgearbeitet.

15 * 4 < 20 ergibt FALSE
4 <> 5 ergibt TRUE

Mit Vergleichsoperatoren dürfen alle einfachen Datentypen einschließlich den noch zu besprechenden Aufzählungs- und Ausschnittstyp verglichen werden.

Standardfunktionen

Pascal besitzt eine Reihe von Standardfunktionen für arithmetische Berechnungen und zum Umwandeln von Datentypen. Einen Überblick finden Sie im Bild 1. Vier Standardfunktionen liefern Ergebnisse vom Typ INTEGER:

ABS(X) liefert den absoluten Wert von X
SQR(X) liefert das Quadrat von X

Funktion	Parameter	Ergebnis	Bedeutung
ABS(x)	INTEGER	INTEGER	Absolutwert
ABS(x)	REAL	REAL	Absolutwert
SQR(x)	INTEGER	INTEGER	Quadrat
SQR(x)	REAL	REAL	Quadrat
SQRT(x)	REAL, INTEGER	REAL	Quadratwurzel
LN(x)	REAL, INTEGER	REAL	Nat. Logarithmus
EXP(x)	REAL, INTEGER	REAL	e hoch x
SIN(x)	REAL, INTEGER	REAL	Sinus
COS(x)	REAL, INTEGER	REAL	Cosinus
ARCTAN(x)	REAL, INTEGER	REAL	Arcustangens
TRUNC(x)	REAL	INTEGER	Ganzzahl von x
ROUND(x)	REAL	INTEGER	Rundung von x
CHR(x)	INTEGER	CHAR	Umwandlung eines ASCII-Wertes
ODD(x)	INTEGER	BOOLEAN	true falls ungerade
ORD(x)	skalar *	INTEGER	Position innerhalb eines Datentyps
SUCC(x)	skalar *	skalar	vorangehender Wert
PRED(x)	skalar *	skalar	folgender Wert

* darf nicht vom Typ REAL sein!

Bild 1. Überblick über arithmetische und Umwandlungsfunktionen

TRUNC(R) hat als Ergebnis den ganzen Teil eines REAL-Wertes
ROUND(R) ergibt den gerundeten Wert von R

Beispiele:

```
ABS(3) = 3
ABS(-3) = 3
SQR(2) = 4
TRUNC(5.45) = 5
TRUNC(-5.45) = -5
ROUND(6.5) = 7
ROUND(-6.5) = -7
```

Werden bei ABS und SQR REAL-Werte eingesetzt, erhält man auch ein REAL-Ergebnis. Bei den Funktionen SQRT, LN, EXP, SIN, COS und ARCTAN dürfen REAL- und INTEGER-Werte als Argument verwendet werden, das Ergebnis ist auf jeden Fall REAL.

Ein boolesches Ergebnis liefert die Funktion ODD. ODD(X) ist TRUE, falls X ungerade ist, sonst erhält man FALSE.

Zum Datentyp CHAR gibt es vier Standardfunktionen.

ORD(C): C ist vom Typ CHAR. Ergebnis ist die Ordnungszahl, durch die intern das Zeichen C dargestellt wird.

ORD('A') = 65

CHR(I) I ist eine positive ganze Zahl. Ergebnis ist das Zeichen, das der Ordnungszahl entspricht.

CHR(36)= "$"

ORD und CHAR dürfen auch auf Ausschnitts- und Aufzählungstypen angewendet werden. ORD ist gewissermaßen die Umkehrung von CHAR. Es gilt:

ORD(CHAR(I)) = I und
CHAR(ORD(C)) = C

In den Ausdruck PRED(C) ist C vom Typ CHAR. Ergebnis ist das Zeichen CHAR (ORD(C)-1)

Dies entspricht dem Zeichen, welches in der Ordnungszahl eines vorher liegt.

PRED('B') = 'A'

SUCC(C) Ergebnis ist das auf C folgende Zeichen.

SUCC('A') = 'B'

PRED und SUCC gelten für alle einfachen Datentypen außer REAL. So ergibt beispielsweise SUCC(10) die Zahl 11. Listing 1 enthält ein Beispiel für diese Funktion.

In der Variablenvereinbarung von Pascal muß für jede Variable der Datentyp definiert werden. Es gibt zwei Möglichkeiten, den Datentyp in der Variablenvereinbarung anzugeben. Die Typenangabe steht beispielsweise selbst in der Variablenvereinbarung. Eine weitere Möglichkeit besteht darin, in der Typenvereinbarung einen Datentyp festzulegen. Die Typenvereinbarung muß grundsätzlich vor der Variablenvereinbarung und nach der Konstantenvereinbarung stehen.

Mit der Typenvereinbarung kann der Programmierer eigene Datentypen selbst schaffen. Die Typenvereinbarung kann allerdings auch bei der Variablenvereinbarung getroffen werden. Sie ist eigentlich nur eine Schreiberleichterung.

Eine Pascal-Spezialität besteht darin, daß sich der Benutzer durch Aufzählung eigene Typen definiert. Beispiel:

```
TYPE farbe = (gruen, gelb,
rot, blau);
VAR  pinsel,topf : farbe;
```

Eine Zuweisung auf die Variable erfolgt dann so:

```
pinsel:= gelb;
topf:=gruen;
```

Die Vereinbarung eines Aufzählungstyps ohne Typenvereinbarung würde so aussehen:

```
VAR topf : (gelb,blau,rot)
```

In der Regel ist aber die erste Version vorzuziehen, weil sie leichter zu verstehen ist; vor allem wenn man mehrere Variablen vom selben Typ vereinbaren will.

Weitere Beispiele zum Aufzählungstyp:

```
TYPE karte = (pik, kreuz,
karo, herz);
woche = (MO,DI,MI,DO,FR,
SA,SO);
```

Beim Aufzählungstyp sind die Standardfunktionen SUCC, PRED und ORD erlaubt.

Es gilt zum Beispiel:

```
ORD(kreuz) = 1
```

(Das erste Element vom Typ Karte, nämlich pik, hat den Wert 0)

```
SUCC(DI)    = MI
PRED(FR)    = DO
```

Bei der Vereinbarung von Namen innerhalb der Aufzählung darf man einen Namen nur einmal verwenden. Aufzählungstypen können auch miteinander verglichen werden:

```
IF tag < SA THEN WRITELN
('Arbeitstag');
```

Auch innerhalb einer CASE- oder FOR-Anweisung können Aufzählungstypen vorkommen:

```
CASE pinsel of
  gelb: WRITE('Sonnenschein');
  blau: WRITE('Meer');
  gruen: WRITE('Wiese')
  END
FOR MO TO FR DO arbeiten;
```

Der Ausschnittstyp

Will man keinen neuen Typ definieren, sondern einen Unterbereich eines bereits existierenden skalaren Typs, so wird man einen Ausschnitt definie-

ren. In der Typenvereinbarung werden dann die untere und die obere Grenze des Ausschnitts festgelegt:

```
TYPE Typname =
Untergrenze .. Obergrenze;
```

Untergrenze und Obergrenze müssen vom gleichen Typ sein und die Untergrenze sollte kleiner als die obere sein. Zum Wertebereich dieses Typs gehören dann alle Elemente zwischen den Grenzen und die Grenzwerte selbst. Ausschnitte vom Typ REAL sind nicht erlaubt!
Beispiele:

```
TYPE ergebnis = 1..6;
 werktag  = MO .. FR;
 monat    = ( JAN, FEB,
 MRZ, APR, MAI, JUN,
 JUL, AUG, SEP, OKT,
 NOV, DEZ);
 sommer   = JUN .. AUG;
VAR  note : ergebnis;
     mo   : monat;  ....
```

Mit Aufzählungs- und Ausschnittstypen lassen sich Programme besser dokumentieren und lesbarer schreiben. Ein Algorithmus kann mit benutzerdefinierten Typen besser formuliert werden.

Variablen vom Typ Ausschnitt verhindern, daß mit fehlerhaften Werten gerechnet wird. Geht man beispielsweise vom deutschen Notensystem aus, so ist es unsinnig, mit einer Note 7 zu rechnen. Ist die entsprechende Variable wie oben definert, führt eine solche Wertzuweisung zu einer Fehlermeldung.

Strukturierte versus skalare Datentypen

Strukturierte Datentypen werden aus skalaren Datentypen gebildet. Es sind vier Arten von strukturierten Datentypen zu unterscheiden:

— SET (Menge)
— ARRAY (Feld)
— RECORD C (Variablenverbund)
— FILE (Datei)

Zum Datentyp ARRAY gibt es noch eine Sonderform, den Typ STRING. Er wird ähnlich wie ein String in Basic behandelt, ist aber in Standard-Pascal nicht vorgesehen. Zu jedem der strukturierten Typen sind Regeln zum Aufbau dieser Typen zu beachten.

Der wesentliche Unterschied zwischen skalaren und strukturierten Typen ist folgender:

Beim skalaren Typ hat jede Variable einen eigenen Namen.

Beim strukturierten Typ besteht eine Variable aus mehreren Komponenten. Diese Komponenten haben keinen eigenen Namen. Der Zugriff auf eine Komponente erfolgt unterschiedlich je nach dem verwendeten Typ.

Mit Mengen arbeiten

Von Mengen hörte man ja in der Schule mehr als genug. Trotzdem sollte dieser Datentyp nicht geringschätzig behandelt werden. Er eignet sich in vielen Fällen zur eleganten Formulierung von Programmen. Allerdings muß man sich darüber im klaren sein, daß alle Programme auch ohne den Typ SET geschrieben werden können.

Die Elemente einer Menge werden alle aus einem Grundtyp gebildet. Eine Variable vom Typ Set hat dann als Wert eine Teilmenge der gesamten Menge. Wer sich noch an Mengenlehre erinnert, dem wird der Begriff Potenzmenge vertraut sein. Diese Potenzmenge enthält als Elemente wiederum Mengen, und zwar alle, die sich aus einer gegebenen Menge bilden lassen. Dazu kommt noch die leere Menge.

Eine Menge muß in der Typenvereinbarung definiert werden: `TYPE menge = SET OF Grundmenge`

Die Grundmenge ist entweder vom Aufzählungstyp oder vom Ausschnittstyp. Es dürfen jedoch keine negativen Integer-Werte vorkommen. Der Wert einer Variablen vom Typ »menge« ist ein Element der Potenzmenge der Grundmenge. Beispiel:

```
TYPE menge = SET OF [1,2,3];
VAR s: menge;
```

Die Variable s vom Typ menge kann dann folgende Werte annehmen:
[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]

Die Anzahl der Elemente der Potenzmenge kann man sich leicht ausrechnen: Zwei hoch Anzahl der Werte der Grundmenge.
Weitere Beispiele für die Vereinbarung von Mengen:

```
TYPE lotto = SET OF 1..49;
     freunde = (Gaby, Peter,
     Josef, Georg);
   runde = SET OF freunde;
   buchstabe = 'A'..'Z';
VAR c,a,b: set of buchstabe;
     paar, gruppe: runde;
     spiel, zahl: lotto;
```

Einer Variablen vom Typ SET muß ein Wert im Anweisungsteil des Programms zugewiesen werden. Mengen werden dabei in rechteckige Klammern eingeschlossen. Aufeinanderfolgende Werte sind durch ein Komma zu trennen. Ausschnitte dürfen ebenfalls verwendet werden. Den vereinbarten Variablen werden Werte wie in den folgenden Beispielen zugewiesen:

```
a:= ['A'..'D'];
b:= ['B','E','J'];
gruppe:=[Peter, Josef,
Georg];
paar:=[Peter,Gaby];
c:=[];    {leere Menge}
```

Die untere und die obere Grenze einer Menge können durch Ausdrücke angegeben werden:

```
zahl:=[z DIV 2 .. z*2]
```

Das Ergebnis dieser Ausdrücke muß natürlich innerhalb der Grundmenge vorkommen.

Die bereits besprochenen Operatoren werden auch auf Mengen angewandt, haben aber dann eine andere Bedeutung.

Die folgenden Beispiele beziehen sich auf die bereits definierten Mengen.

+ entspricht der Vereinigung von Mengen: m:= a + b;
m hat den Wert ['A','B','C','D','E','J']

- entspricht der Differenz zweier Mengen: m:= a - b
m hat den Wert ['A','C','D']

***** entspricht dem Durchschnitt zweier Mengen: m:= a * b
(m hat hier den Wert ['B'])

Für die Vergleichsoperatoren gilt:

= entspricht der Gleichheit zweier Mengen.
a = b hat dann den Wert true, wenn beide Variablen die gleichen Elemente enthalten. Im vorliegenden Fall erhält man den Wert false.

<> fragt ab, ob zwei Mengen ungleich sind. Ungleichheit liegt vor, wenn nicht alle Elemente der beiden Mengen gleich sind.

<= entspricht der Beziehung Teilmenge.
['A','B'] <= a ist true, weil 'A' und 'B' in a enthalten sind.

>= entspricht ebenfalls der Teilmengen-Beziehung.
a>=b ist false, da b keine Teilmenge von a ist.

Beim Datentyp SET gibt es den zusätzlichen Operator »IN«. Er hat ebenfalls einen booleschen Wert als Ergebnis und stellt fest, ob ein Element in einer Menge enthalten ist. Das folgende Beispiel

```
'A' IN a
```

hat als Ergebnis true, da das Element 'A' in der Menge a enthalten ist. Für die Vergleichsoperatoren »<« und »>« gibt es keine Verwendung. Die entsprechende Mengenbeziehung echte Teilmenge kann man leicht mit den anderen Operatoren ausdrücken:

```
IF a<=b) AND (a<>b) THEN
```

Die Menge a ist nur dann eine echte Teilmenge, wenn sie in b enthalten und von b verschieden ist. Ein Vergleich der üblichen mathematischen Schreibweise mit der Pascal-Schreibweise ist im Bild 2 zu sehen.

Mengen können nicht direkt ausgegeben werden. Will man die Anzahl der Elemente einer Menge wissen, sind die Elemente der Menge zu zählen wie in Listing 2. Die Anzahl der Elemente darf nicht beliebig groß werden. So erlaubt Profi-Pascal maximal 256 Elemente, Oxford-Pascal nur 128 Elemente.

```
PROGRAM ZEICHEN;
(* LIEST ZEICHEN VON DER TASTATUR UND ZAEHLT DIE
  VORGEFUNDENEN ZEICHEN NACH KATEGORIEN *)
VAR GROSS, KLEIN, ZIFFER, SONDERZEICHEN: SET OF CHAR;
    G,K,Z,S:INTEGER;
    ZEICHEN: CHAR;
(* *)
BEGIN
(* INITIALISIEREN *)
GROSS:=ä'a'..'z'ü;
KLEIN:=ä'A'..'Z'ü;
ZIFFER:=ä'Ø'..'9'ü;
G:=Ø;
K:=Ø;
Z:=Ø;
S:=Ø;
ZEICHEN:=' ';
(*
   TEST UEBERNEHMEN UND AUSWERTEN
*)
WRITELN('GEBEN SIE DEN TEXT EIN:');
READ(ZEICHEN);
WHILE ZEICHEN <> '#' DO
  BEGIN
    REPEAT
        IF ZEICHEN IN GROSS THEN G:=SUCC(G)
        ELSE
        IF ZEICHEN IN KLEIN THEN K:=SUCC(K)
        ELSE
        IF ZEICHEN IN ZIFFER THEN Z:=SUCC(Z)
        ELSE S:=SUCC(S);
    READ(ZEICHEN);
    UNTIL EOLN;
   READLN
END;
(*
   ERGEBNIS AUSDRUCKEN
*)
WRITELN;
WRITELN('GEFUNDEN WURDEN');
WRITELN(G:5,' GROSSBUCHSTABEN');
WRITELN(K:5,' KLEINBUCHSTABEN');
WRITELN(Z:5,' ZIFFERN');
WRITELN(S:5,' SONDERZEICHEN');
WRITELN(G+K+Z+S:5,' ZEICHEN INSGESAMT')
END. (*ZEICHEN*)
```

Listing 1. Abfragen mit Mengen

Der Operator AND

a	b	a AND b
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Der Operator OR

a	b	a OR b
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Der Operator NOT

a	NOT a
FALSE	TRUE
TRUE	FALSE

Bild 2. Operationen mit Mengen

Mengen werden sehr kompakt als Bitfolge gespeichert. Jedem Bit entspricht ein Element der Menge. Das Bit zeigt an, ob das entsprechende Element vorhanden ist oder nicht. Wegen dieser Art der Speicherung sind Mengenoperationen sehr schnell.

Mengen kann man auch benutzen, ohne sie als Variable zu vereinbaren. Beispiel:

```
VAR test: 1..10;
test:= 2; .....
IF test IN [1..4] THEN .....
```

ist bedeutend besser als

```
IF (test=1) OR (test=2) OR
(test=3) OR (test=4) THEN ...
```

Das Beispiel in Listing 2 zeigt die Anwendung von Mengen. Es führt vor, wie elegant man mit Mengen Probleme lösen kann.

```
PROGRAM JOSEPH;
CONST ANZAHL=41;
      REST=2;
      ABZAEHL=3;
VAR I,J,K:INTEGER;
    REIHE:SET OF 1..ANZAHL;
BEGIN
(*INITIALISIEREN*)
  REIHE:=ä1..ANZAHLü;
 I:=ANZAHL;
 K:=ANZAHL;
 (*WIEDERHOLE BIS NUR NOCH 2
   ELEMENTE IN DER REIHE SIND
   DIESE GEBEN DIE GESUCHTEN
   POSITIONEN AN *)
 WHILE K>REST DO
   BEGIN
    FOR J:= 1 TO ABZAEHL DO
       REPEAT
         IF I<ANZAHL THEN I:=I+1
           ELSE I:=1;
       UNTIL I IN REIHE;
 (* DER JEWEILS DRITTE WIRD GESTRICHEN *)
    REIHE:=REIHE-äIü;
    K:=Ø;
 (* JETZT WIRD FESTGESTELLT, WIEVIELE ELEMENTE DIE
    MENGE NOCH ENTHAELT *)
    FOR J:= 1 TO ANZAHL DO
      IF J IN REIHE THEN K:=K+1;
 END; (* ENDE DER WHILE-SCHLEIFE*)
  FOR J:= 1 TO ANZAHL DO
    IF J IN REIHE THEN
      WRITELN('GESUCHTE POSITIONNR.: ',J)
END.
```

Listing 2. Das Problem des Josephus

Von dem jüdischen Historiker Josephus erzählt die Legende, daß er bei der Eroberung der Stadt Jotapater durch die Römer mit 40 anderen Juden vor den anstürmenden Feinden in ein Haus flüchtete. Josephus' Kameraden beschlossen, sich nicht den Römern zu ergeben und sich lieber selbst zu töten. Josephus und mit ihm sein Freund wollten am Leben bleiben. Joseph machte deshalb den Vorschlag, die Selbsttötung in einer gewissen Reihenfolge vorzunehmen. Alle sollten sich in einer Reihe aufstellen und dann sollte sich jeder dritte selbst töten. Am Ende der Reihe angelangt, würde das Verfahren von vorne beginnen.

Der Vorschlag wurde angenommen. Josephus und sein Freund nahmen diejenige Stellung ein, bei der sie beim Abzählen die letzten sein würden und blieben am Leben. Das Programm simuliert den Abzählvorgang, indem es eine Menge mit 41 Elementen verwendet. Dann wird jeweils bis drei gezählt und das entsprechende Element aus der Menge entfernt. Wenn sich nur noch zwei Elemente in der Menge befinden, wird der Abzählvorgang beendet. Das Programm läßt sich auch auf Probleme mit einer größeren Grundmenge, einer anderen Schrittweite und einem unterschiedlichen Rest anwenden.

Das Zeichen »#« beendet die Eingabe, da Oxford-Pascal kein EOF (Ende der Eingabedatei) von der Tastatur akzeptiert. Die Funktion eoln erkennt in einem Return das Zeilenende und veranlaßt einen Zeilenvorschub.

Das Programm enthält eine interessante Anwendung der Funktion »SUCC«. Sie wird hier statt des üblichen k:=k+1 verwendet. Das Beispielprogramm zeigt auch, wie man Mengen elegant für Abfragezwecke verwenden kann.

(S. Gutschmidt/A. Gruber/cg)

Streifzüge durch die Grafik-Welt (Teil 4)

In dieser Folge wird Ihnen ein Handwerkszeug vertraut werden, von dem Sie unter Umständen noch nie etwas gehört haben: die Matrix. Sie befinden sich aber in guter Gesellschaft, denn auch viele Leute, denen von Berufs wegen das Rechnen mit Matrizen Arbeitserleichterungen bringen würde, kennen oft nur den Namen dieser mathematischen Gebilde.

Nun könnten Sie es vielleicht mit der Angst zu tun bekommen, daß Ihnen hier höhere Mathematik zugemutet werde. Aber der Witz an Matrizen — jedenfalls in der Weise, in der wir uns damit befassen werden — ist gerade, daß weniger hohe Mathematik getrieben werden muß, wenn man sie auf bestimmte Fragestellungen anwendet als ohne die Matrizen. Weshalb begegnet man ihnen dann so selten, werden Sie fragen. Der Grund liegt vermutlich darin, daß die mit Matrizen mögliche Vereinfachung von Problemlösungen erkauft werden muß durch einen sehr viel höheren Rechenaufwand. Die Rechnungen sind dann zwar recht elementar, nämlich einfache Additionen, Subtraktionen und Multiplikationen, dafür werden es aber im Vergleich zu Lösungswegen aus Bereichen der höheren Mathematik viel mehr Rechenoperationen. Viele einfache Rechnungen miteinander zu verknüpfen, bedeutet aber auch eine Steigerung der Fehleranfälligkeit. Der Heim- und Personal-Computer ist zwar in der Lage, eine große Anzahl von solchen Rechnungen schnell und sicher durchzuführen, aber seine Existenz ist eben noch nicht sehr alt und im Denken vieler Menschen noch nicht präsent. Wie gut sich Computer zur Matrizenverarbeitung eignen, werden wir gleich noch sehen.

James Joseph Sylvester (1814—1897) führte 1850 den Begriff der Matrix ein. Lateinisch »mater« heißt »Mutter«, französisch »matrice« bedeutet auch »Gußform«. Sylvester wollte mit dieser Bezeichnung wohl eine häufige Verwendung von Matrizen charakterisieren: Dabei dient die Matrix gewissermaßen als eine Gußform, durch die Daten in gewisse neue Formen und Zusammenhänge gebracht werden können. Genug von Geschichte. Wo können Matrizen überall angewendet werden (außer in der Computergrafik, die unser Anliegen ist)? Dem Techniker und Ingenieur dienen sie beispielsweise zur Ermittlung von Eigenfrequenzen in der Schwingungstechnik, zu Netzberechnungen in der Elektrotechnik oder zur Berechnung statisch unbestimmter Systeme in der Baustatik. Der Physiker bedient sich ihrer in der Quantentheorie. Kaufleute, Betriebs- und Volkswirte erleichtern sich die Produktionsplanung, Materialplanung, Betriebskostenüberwachung damit. Matrizen sind aber auch Handwerkszeuge für die einfache Erfassung von komplexen Zusammenhängen: Man kann damit Verflechtungsbilanzen erstellen und untersuchen. Sehr interessant sind auch die vielfältigen Möglichkeiten bei Optimierungsproblemen. Jetzt müßte deutlich geworden sein, welch ein breites Anwendungsspektrum sich da offenbart. Allen ist eines gemeinsam: Es liegen sogenannte Vielfaktorenprobleme vor. Damit ist gemeint, daß