Die Darstellung der rationalen und der reellen Zahlen ($\mathbb{Q}$ und $\mathbb{R}$).
Rationale und reelle Zahlen werden meist als Dezimalzahlen mit Nachkommastellen geschrieben. So ist 1/8 = 0,125 oder 2/3 = 0,666666... oder $\sqrt{2}$ = 1,414241... .

1.1.2.12: Die Darstellung zu einer Basis geschieht ebenso wie bei den natürlichen Zahlen, nur dass man nun auch negative Exponenten zulässt:
$z = z_{n-1}z_{n-2} \ldots z_1 z_0 \cdot z_{-1}z_{-2} \ldots z_{-k}$ .... bezeichnet also die Zahl

$$\phi(z) = \sum_{i=-\infty}^{n-1} \beta(z_i)\cdot b^i \qquad \text{(vgl. 1.1.2.8)}$$

Beispiel: $(101.1001)_2 = 1\cdot 2^2 + 1\cdot 2^0 + 1\cdot 2^{-1} + 1\cdot 2^{-4} = (5.5625)_{10}$.

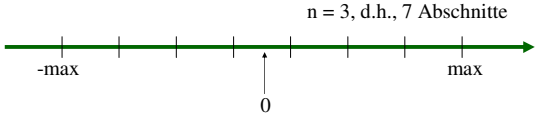
Darstellung in der Programmierung:

Zahlen werden in der Regel im Rechner mit 32, 64 oder 128 Binärstellen beschrieben, was für die meisten Anwendungen ausreicht. Hiermit kann man aber nur 2^{32} bzw. 2^{64} bzw. 2^{128} verschiedene Zahlen erfassen. Jedes endliche Intervall in $\mathbb{Q}$ und $\mathbb{R}$ ist jedoch unendlich groß und fast alle hierin liegenden Zahlen lassen sich nicht durch eine beschränkte Zahl von Ziffern beschreiben. Folglich lassen sich rationale und reelle Zahlen mit einer beschränkten Zahl an Stellen nur annähern, aber nur selten exakt beschreiben.

Wie kann man ein rationales oder reelles Intervall von -max bis +max mit 2^n Zahlenwerten möglichst gut annähern (n=32, 64 oder 128)? Hierbei sei max die größte Zahl, die man darstellen möchte.

Wir betrachten einige naheliegende Möglichkeiten.

Fall 1a: gleichmäßige Aufteilung des Intervalls [-max, max]. Dieses Intervall wird in (2^n-1) Abschnitte der Länge $2\cdot max/(2^n-1)$ unterteilt:

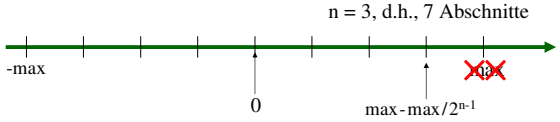


Die 2^n darstellbaren Zahlen sind dann:
$-max + i\cdot 2\cdot max/(2^n-1)$ für i = 0, 1, 2, ..., 2^n-1.

Nachteil: Die Zahl 0 ist nicht exakt darstellbar. Dieser Nachteil ist in der Praxis nicht akzeptabel.

Daher verzichtet man auf die Darstellung der Zahl max und endet bei $max - 2\cdot max/2^n$.

Fall 1b: gleichmäßige Aufteilung des Intervalls [-max, max). Das Intervall von -max bis $max - 2\cdot max/2^n$ wird in 2^n-1 Abschnitte der Länge $2\cdot max/2^n = max/2^{n-1}$ unterteilt:



Die 2^n darstellbaren Zahlen sind dann:
$-max + i\cdot 2\cdot max/2^n$ für i = 0, 1, 2, ..., 2^n-1.

Hierbei wird die Null stets exakt dargestellt.

Der Vorteil ist, dass man die Darstellungen für die ganzen Zahlen auch für die Annäherung der reellen Zahlen verwenden kann; denn die Zahl
$-max + i\cdot 2\cdot max/2^n = (i-2^{n-1})\cdot max/2^{n-1} = j\cdot(max/2^{n-1})$
wird durch die ganze Zahl j eindeutig bestimmt; dabei läuft j von -2^{n-1} bis $+2^{n-1}$-1 (vgl. Zwei-Komplementdarstellung 1.1.2.10).

$max/2^{n-1}$ heißt Schrittweite. Sie ist kleinste Zahl in diesem System, um die sich zwei Zahlen unterscheiden.

Im Falle $max = 2^{n-1}$ erhält man die Schrittweite 1 und somit die ganzen Zahlen von -2^{n-1} bis $+2^{n-1}$-1.

In der Praxis wählt man max stets als Zweierpotenz, sodass $(max/2^{n-1}) = 1/2^d$ ist. Dadurch erhält jede darstellbare Zahl im Binärsystem die Form

<Vorzeichen> <Folge von n-1-d Nullen und Einsen> . <Folge von d Nullen und Einsen>

Man kann dann die reellen Zahlen als ganze Zahlen schreiben, wobei sich aber an einer festen Stelle (zwischen der d-t und der (d+1)-letzte Position) der Binärpunkt, also der Beginn der Nachkommastellen befindet. Dies führt zur

1.1.2.13: Festpunktdarstellung

Reelle Zahlen werden hierbei wie ganze Zahlen dargestellt, wobei ein gedachter Binärpunkt die Nachkommastellen bezeichnet. (Man kann auch eine andere Basis statt 2 nehmen.)

Die Zahl der Nachkommastellen d muss bei der Deklaration angegeben werden, z.B.:
var Y: real binary fixed (d) % Darstellung mit Binärziffern
var X: real decimal fixed (d); % Darstellung mit Dezimalziffern

In Ada verwendet man hierfür das Schlüsselwort DELTA und gibt hiermit exakt den Wert $\max / 2^{n-1}$ und zusätzlich das Intervall durch seine Grenzen an. Jede Programmiersprache hat hier andere Darstellungsmöglichkeiten.

Beispiel:

```
var U,V: real binary fixed (3);
U := 1011001;        % U erhält die Zahl  11,125
V := 10101;          % V erhält die Zahl  2,625
U := U + V;          % U besitzt nun die Zahl 13,75
                       (binär:  1101.110)
```

Für Zahlen in U und V sind also die drei letzten Stellen Nachkommastellen.

Der Vorteil dieser Festpunktdarstellung ist, dass man wie mit ganzen Zahlen rechnen kann: Die Zahlen lassen sich im Zwei-Komplement darstellen, um Subtraktion wie Addition behandeln zu können, und die Operationen +, -, $*$ lassen sich einfach im Computer realisieren. Anwendungen liegen z.B. im Banken-bereich, wo mit zwei Nachkommastellen gerechnet wird.

Dieses Vorgehen, das gesamte Intervall gleichmäßig mit 2^n Zahlen zu überstreichen, hat aber den Nachteil, dass der Bereich um die Zahl 0, in dem fast immer wesentlich genauer als in anderen Bereichen gerechnet werden muss, genau so grob aufgeteilt wird wie die weniger interessanten Bereiche in der Nähe der Intervallgrenzen. Man verwendet daher in der Praxis andere Darstellungen für die rationalen bzw. die reellen Zahlen.

1.1.2.14 Fall 2: Gleitpunktdarstellung

Reelle Zahlen $z \neq 0$ werden hierbei in folgender Form dargestellt $z = m \cdot b^e$ mit $m \in \mathbb{R}$, $1/b \le |m| < 1$, $b, e \in \mathbb{Z}$, $b \ge 2$.

Im Falle $z = 0$ setzen wir $m = 0$. In diesem Fall ist die Darstellung nicht eindeutig, da nun ein beliebiger Exponent e verwendet werden kann. (Dies stört aber im Weiteren nicht.)

m heißt Mantisse, e heißt Exponent von z bzgl. der Basis b.

Wir müssen uns davon überzeugen, dass die Definition der Gleitpunktdarstellung "sinnvoll" ist. Von einer solchen Darstellung werden wir verlangen, dass es sie immer gibt und dass sie für jede Basis und jede reelle Zahl eindeutig ist.

Den Fall $z = 0$ haben wir bereits betrachtet ($z=0 \Leftrightarrow m=0$).

Hilfssatz 1.1.2.15: Existenz und Eindeutigkeit

Es sei $b \ge 2$ eine natürliche Zahl. Zu jeder reellen Zahl $z \neq 0$ gibt es genau ein $m \in \mathbb{R}$ mit $1/b \le |m| < 1$ und genau ein $e \in \mathbb{Z}$ mit $z = m \cdot b^e$.

Beweis: Wir zeigen zunächst die Eindeutigkeit der Darstellung: Wenn für zwei von Null verschiedene Zahlen $z_1 = m_1 \cdot b^{e_1}$ und $z_2 = m_2 \cdot b^{e_2}$ gilt $z_1 = z_2$ (o.B.d.A. sei $e_1 \le e_2$), so muss $m_1 \cdot b^{e_1} - m_2 \cdot b^{e_2} = (m_1 \cdot b^{e_1} - m_2 \cdot b^{e_2-e_1+e_1}) = (m_1 - m_2 \cdot b^{e_2-e_1}) \cdot b^{e_1} = 0$ sein. Wegen $b^{e_1} \neq 0$ muss $m_1 = m_2 \cdot b^{e_2-e_1}$ gelten.

Wäre nun $e_1 < e_2$, also $b^{e_2-e_1} \ge b$, so würde $1/b \le |m_1| < 1$, aber $1 \le |m_2| \cdot b^{e_2-e_1}$ gelten (wegen $1/b \le |m_2|$). Dann können aber m_1 und $m_2 \cdot b^{e_2-e_1}$ nicht gleich sein.

Folglich muss $e_1 = e_2$ sein, woraus sofort $m_1 = m_2$ folgt. Also gilt $z_1 = z_2$ gleich und die Darstellung ist eindeutig.

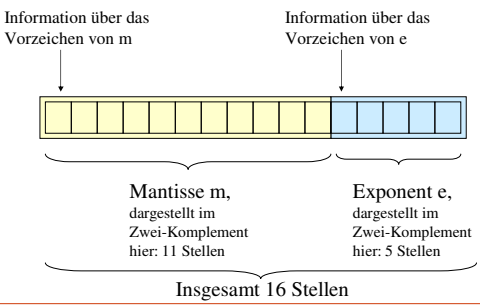
Fortsetzung des Beweises:

Wir müssen noch die Existenz einer solchen Darstellung zeigen. Betrachte $|z|$. Wegen $z \neq 0$ ist $|z| > 0$.

Falls $|z| \ge 1$, so bestimme die größte negative ganze Zahl e', für die $|z| \cdot b^{e'} < 1$ ist. (D.h., dividiere $|z|$ ständig durch b, bis der Wert kleiner als 1 geworden ist.) Setze dann $m := z \cdot b^{e'}$. Mit z und $b^{e'}$ ist auch m eine reelle Zahl, und da e' maximal gewählt wurde, muss $1/b \le |m| = |z| \cdot b^{e'} < 1$ gelten. Folglich existiert die Gleit-punktdarstellung $z = m \cdot b^{-e'}$ in diesem Fall.

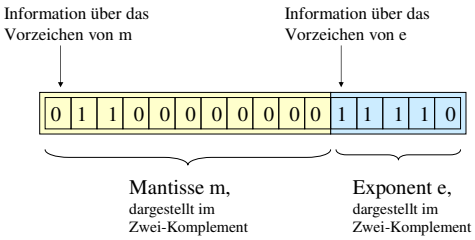
Falls $|z| < 1$ ist, so bestimme die kleinste natürliche Zahl $e' \ge 0$, für die $|z| \cdot b^{e'} < 1$, aber $|z| \cdot b^{e'+1} \ge 1$ ist. Setze $m := z \cdot b^{e'}$. Dann ist m eine reelle Zahl, und da e' minimal gewählt wurde, muss $1/b \le |m| < 1$ gelten. Folglich existiert auch in diesem Fall die Gleitpunktdarstellung $z = m \cdot b^{-e'}$. $(e := -e')$ ■

Beispiel: Betrachte n = 16, m ist 11-stellig, e ist 5-stellig.



Beispiel: Betrachte n = 16, m ist 11-stellig, e ist 5-stellig.

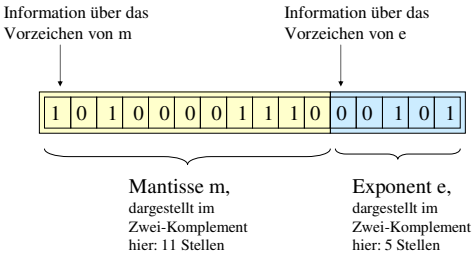
Sei b = 2 und z = 0,1875 = 1/16 + 1/8 = $(0,0011)_2$. Dann ist $m=(0,11)_2=0,75$ und $2^e = 1/4$, also e = -2. Gleitpunktdarstellung:



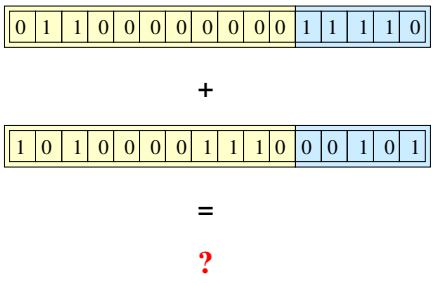
Proberechnen: $(0110...)_2 = 0,5+0,25 = 0,75$ ⇒ $z = 0,75 \cdot 2^{-2} = 0,1875$
Zwei-Komplement für e: -16+8+4+2 = -2

Beispiel: Betrachte n = 16, m ist 11-stellig, e ist 5-stellig, b=2.

Sei z = -23,6875 = -(23 + (1/2 + 1/8 + 1/16) = $-(10111.1011)_2$. Dann ist m = $-(0.101111011)_2$ und $2^e = 32$, also e = 5. Zwei-Komplement zu $-(0.1011110110)_2$ liefert 10100001110.



Was ist nun die Summe der beiden Zahlen: 0,1875 + (- 23,6875) ?



1.1.2.16: Addition zweier positiver reeller Zahlen

Betrachte zwei positive Zahlen z_1 und z_2 und ihre Gleitpunktdarstellungen $z_1 = m_1 \cdot b^{e_1}$ und $z_2 = m_2 \cdot b^{e_2}$.

Falls die Exponenten e_1 und e_2 verschieden sind, so wähle als Zahl z_2 die Zahl, deren Exponent der kleinere ist. Es gilt also $e_1 \geq e_2$, und somit $m_2 \cdot b^{e_2-e_1} \leq m_2 < 1$.

$$\begin{aligned} z_1 + z_2 &= m_1 \cdot b^{e_1} + m_2 \cdot b^{e_2} \\ &= m_1 \cdot b^{e_1} + m_2 \cdot b^{e_2-e_1+e_1} \\ &= (m_1 + m_2 \cdot b^{e_2-e_1}) \cdot b^{e_1} \end{aligned}$$

Also führt man die Addition folgendermaßen durch:

Verschiebe die Mantisse der kleineren Zahl um e_1-e_2 Stellen nach rechts (schiebe hierbei vorne Nullen nach und lass die rechts rausgeschobenen Ziffern weg),
addiere die Mantissen: $m = m_1+m_2$,
falls $m > 1$ ist, verschiebe m um eine Stelle nach rechts (schiebe vorne eine Null nach und lass die rechts rausgeschobene Ziffer weg) und erhöhe e_1 um 1,
das Resultat der Addition ist die Zahl $m \cdot b^{e_1}$, d.h., die Zahl, deren Mantisse m und deren Exponent e_1 ist.

Liegen negative Zahlen vor, so muss man dieses Verfahren modifizieren.
Die Subtraktion lässt sich in ähnlicher Weise behandeln.
Beispiele siehe Übungen.

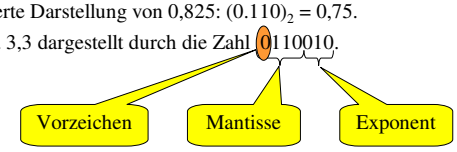
Fazit: Reelle Zahlen können nur angenähert dargestellt werden. Rationale Zahlen werden nicht gesondert betrachtet, sondern werden entweder als reelle Zahlen aufgefasst oder müssen als eigener Datentyp definiert werden (siehe später, Kap. 1.3).

Die Darstellung einer reellen Zahl r durch die am nächsten zu r liegende Zahl $m \cdot b^e$ bezeichnet man als Rundung. Die hierbei auftretende Differenz $|r-m \cdot b^e|$ nennt man Rundungsfehler.

Die Bezeichnung "Rundungsfehler" verwendet man auch für die Fehler, die im weiteren Verlauf von Berechnungen auftreten und die sich zu sehr großen Zahlen aufschaukeln können.

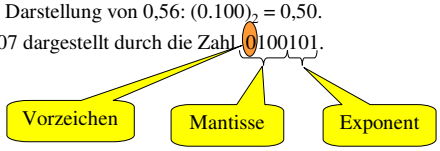
Beispiel: Basis 2, Gesamtstellenzahl n = 7,
Mantissenlänge (mit Vorzeichen) = 4, Exponentenlänge = 3.
Betrachte die reellen Zahlen 3,3 und 0,07.

Darstellung von 3,3 = 0,825 * 2^2; 3.3 = $(11.01001100110011...)_2$
Der Exponent ist somit 2.
Angenäherte Darstellung von 0,825: $(0.110)_2$ = 0,75.
Also wird 3,3 dargestellt durch die Zahl 0110010.



Beispiel: Basis 2, Gesamtstellenzahl n = 7,
Mantissenlänge (mit Vorzeichen) = 4, Exponentenlänge = 3.
Betrachte die reellen Zahlen 3,3 und 0,07.

Darstellung von 0,07 = 0,56 * 2^{-3}; 0.07 = $(0.00010001111101...)_2$
Der Exponent ist somit -3 = $-(011)_2$; Zwei-Komplement: 101.
Angenäherte Darstellung von 0,56: $(0.100)_2$ = 0,50.
Also wird 0,07 dargestellt durch die Zahl 0100101.



Beispiel: Basis 2, Gesamtstellenzahl n = 7,
Mantissenlänge (mit Vorzeichen) = 4, Exponentenlänge = 3.
Betrachte die reellen Zahlen 3.3 und 0.07 .

Addiere nun 0111010 und 0100101 .
Die kleinere Zahl ist 0100101 mit dem Exponenten -3; die größere ist 0111010 mit dem Exponenten 2.
Schiebe daher 0100 um 2-(-3) = 5 Stellen nach rechts und schiebe vorne Nullen nach. Ergebnis: 0000101.
Addiere nun die Mantissen. Dies ergibt 0111.
Überlauf links durch eine 1 liegt nicht vor.
Ergebnis ist also 0111010.
Folglich ist bei dieser Darstellung 3.3 + 0.07 = 3.3.

Daher: Vorsicht beim Rechnen mit reellen Zahlen!

1.1.2.17: Die **Rundungsfehler** sinken mit der Anzahl der Stellen n und sie wachsen mit der Anzahl der Operationen, die während einer Berechnung angewendet werden.

Die besondere Warnung: Bei der Division durch betragsmäßig kleine Zahlen entstehen oft in nur wenigen Schritten schon sehr große Rundungsfehler.

Wer reelle Zahlen benutzt, sollte daher stets mit einer möglichst großer Stellenzahl n arbeiten. Die meisten Programmier-sprachen erlauben es, die Standarddarstellungen (meist n=32) auf n=64 oder n=128 usw. zu erhöhen. Beispiele siehe Vorlesungen über Mathematik, Numerik, Simulation usw.

1.1.2.18 Selbstdefinierter elementarer Datentyp

Wir vereinbaren nun noch, dass man einen neuen Datentyp im Deklarationsteil eines Programms mit Hilfe des Schlüssel-wortes "type" einführen darf in der Form:

type <Name des Datentyps> = (<Liste der Elemente>)

Die Reihenfolge in der Liste gibt zugleich die Anordnung der Elemente an.

Beispiele:

type Wochentage = (Mo, Di, Mi, Do, Fr, Sa, So);

type Farbe = (weiß, gelb, grün, rot, blau, schwarz);

type erste_zehn_Primzahlen = (2,3,5,7,11,13,17,19,23,29)

Welche Konstruktoren gibt es nun, um aus elementaren Datentypen kompliziertere Strukturen zu erhalten?

1.1.2.19: Hier folgt man den mathematischen Strukturen:
Unterbereiche, Intervalle (z.B. Einschränkung auf [... , ...]),
kartesische Produkte (Datensätze, "record"),
n-faches Produkt einer Menge (Feld, Vektor, "array"),
(disjunkte) *Vereinigung* von Mengen (varianter record, union),
Potenzmenge ("set of "),
Funktionen zwischen Mengen (function, procedure),
Graphen, Relationen (realisiert durch Zeiger, pointer, "reference").

Wir betrachten hier nur kurz Unterbereiche und Felder. Mit den anderen Datentypen beschäftigen wir uns in Kapitel 1.3

1.1.2.20 Unterbereiche

Wenn M eine angeordnete Menge ist und a und b zwei Elemente aus M sind, dann ist

$$[a \,..\, b] = \{x \in M \mid a \le x \le b\}$$

der Unterbereich von a bis b der Menge M.

Gilt $M \neq \emptyset$ und $a \le b$, so ist $[a \,..\, b] \neq \emptyset$.
Im Falle $a > b$ ist $[a \,..\, b] = \emptyset$.

Unterbereiche von Zahlen sind Intervalle, z.B.
[17 .. 35] oder [-23 .. -4] oder [-23.6875 .. 0.1875].

In den meisten Programmiersprachen sind nur Unterbereiche zulässig, die endlich sind; insbesondere erlaubt man meist keine Unterbereiche reeller Zahlen.

Zur Definition verwenden wir wieder das Schlüsselwort type.

Wenn man eine Menge selbst definiert, so wird die Reihenfolge in der Definition als Anordnung dieser Menge aufgefasst (siehe 1.1.2.18).

Vereinbare erneut folgende Menge:
type Wochentage = (Mo, Di, Mi, Do, Fr, Sa, So) ,
dann gilt Mo < Di < Mi < Do < Fr < Sa < So.

Unterbereiche hiervon können sein
type Arbeitstage = [Mo .. Fr] oder
type Ausgehtage = [Fr .. Sa].

Ein anderes Beispiel lautet
Großbuchstaben = ['A' .. 'Z']
als Unterbereich des Datentyps character.

1.1.2.21 Felder

Sehr häufig benötigt man n Variablen der gleichen Art, auf die mit einem Index zugegriffen werden kann, vgl. Beispiel 1.1.1.8. Diese fasst man unter einem Namen zu einem "Feld" (oder Vektor) zusammen.

Zur Angabe solcher Felder benötigt man den Datentyp, den jede Variable besitzen soll, und einen Bereich für den Index. Man verwendet das Schlüsselwort "array", um ein Feld zu bezeichnen, gibt dann den Indexbereich an (meist als Unterbereich) und fügt daran den gemeinsamen Datentyp der einzelnen Komponenten an, abgetrennt durch "of". Darstellung in einer Deklaration:

var X: array <Datentyp des Index> of <Datentyp>.

Beispiel: Wir übernehmen den selbstdefinierten Datentyp:

type Wochentage = (Mo, Di, Mi, Do, Fr, Sa, So)

Hierzu bilden wir den Unterbereich:

type Arbeitstage = [Mo .. Fr]

und dann folgendes Feld aus fünf Elementen

type Arbeitsbeginn = array Arbeitstage of [0 .. 23]

Variablen dieses Typs werden deklariert durch

var X: Arbeitsbeginn;

Auf ihre Komponenten wird mittels X[i] zugegriffen. Dabei durchläuft i den Wertebereich des Datentyps Arbeitstage, d.h. $i \in \{Mo, Di, Mi, Do, Fr\}$.

1.1.2.22 Skalarprodukt im $\mathbb{R}^n$

Im n-dimensionalen Vektorraum $\mathbb{R}^n$ werden Punkte durch ihren Koordinaten beschrieben, also durch den Vektor $(x_1,x_2,...,x_n)$. Der Abstand vom Nullpunkt ist dann die Wurzel aus dem Skalarprodukt mit sich selbst $x_1^2 + x_2^2 + ... + x_n^2$.

Allgemein ist das Skalarprodukt zweier Vektoren $(x_1,x_2,...,x_n)$ und $(y_1,y_2,...,y_n)$ definiert als $x_1 \cdot y_1 + x_2 \cdot y_2 + ... + x_n \cdot y_n$. Folgendes Programm berechnet dieses Skalarprodukt.

Wir nehmen an, die Zahl n, der Vektor x und der Vektor y stehen in dieser Reihenfolge in der Eingabe. Wir lesen diese Werte ein, berechnen dann das Skalarprodukt und geben es aus.


```
program skalarprodukt1 is
var i, n: natural; s: real;
    x, y: array [1..n] of real;
begin  read(n);
   if n > 0 then
      for i := 1 to n do read (X[i]) od;
      for i := 1 to n do read (Y[i]) od;
      s := 0.0;
      for i := 1 to n do  s := s + X(i) * Y(i)  od;
      write (s)
   fi
end
```

Hinweis: Grundsätzlich strebt man in der Programmierung an, dass alle Werte, die an einer Stelle es Programms verwendet werden, bekannt sind, wenn man sich dort befindet.

Das Programm skalarprodukt1 erfüllt diese Forderung nicht, weil an der Stelle " x, y: array [1..n] of real; " der Wert von n noch nicht bekannt ist; denn er wird erst später eingelesen.

Daher lässt sich das Programm skalarprodukt1 nicht direkt in jede Programmiersprache übertragen. Meist muss ein *Blockkonzept* vorhanden sein, das wir in Kapitel 1.4 behandeln. Wir weisen hier auf diese Unkorrektheit hin, bereinigen sie aber erst später.

In der Feld-Deklaration

array <Datentyp des Index> of <Datentyp>

darf der <Datentyp> der Komponenten erneut eine Felddeklaration sein:

array <Datentyp des 1.Index> of
array <Datentyp des 2.Index> of <Datentyp>

Dies kürzt man meist ab, z.B. durch
array <Datentyp des 1.Index, Datentyp des 2.Index> of <Datentyp>

bzw. man schreibt bei Unterbereichen anstelle von
array [3 .. 25] of array [-4 .. 7] of real

die Deklaration:
array [3 .. 25, -4 .. 7] of real
und nennt dies ein zwei-dimensionales Feld. Wiederholt man dies, so lassen sich d-dimensionale Felder deklarieren.