1.3.1.4 Der Datentyp integer

Zugrunde liegende Menge:
Z = { ..., -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, ... } mit üblicher Ordnung.

Nullstellige Operationen: Alle Zahlen.

Hinweis: Im Prinzip genügen die beiden Konstanten 0 und 1. Jede Zahl lässt sich dann durch Anwenden der Addition und der Bildung der negativen Zahl beschreiben:

3 durch 1 + 1 + 1 oder -4 = -(1+1+1+1).

Auch die 0 kann man wegen 0 = 1 - 1 noch weglassen.

First und Last gibt es hier nicht (wohl aber in konkreten Programmiersprachen, siehe z.B. in Ada).

Einstellige Operationen:

+	einstelliges Plus (wie Identität, verändert also nichts)
-	einstelliges Minus, Bildung der negativen Zahl, Negation
abs	Absolutbetrag einer Zahl
sgn	"Signum", also das Vorzeichen einer Zahl:

$$\text{sgn}(x) = \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \\ -1, & \text{falls } x < 0 \end{cases}$$

square Quadrat einer Zahl (square(x) = x^2)
odd: $\mathbf{Z} \to \mathbb{B}$ mit: odd(x) = true $\Leftrightarrow$ x ist ungerade $\Leftrightarrow$ x mod 2 = 1.
even: $\mathbf{Z} \to \mathbb{B}$ mit: even(x) = true $\Leftrightarrow$ x ist gerade $\Leftrightarrow$ x mod 2 = 0.

Zweistellige Operationen:

+	Addition zweier Zahlen
-	Subtraktion zweier Zahlen, Differenz zweier Zahlen
·	Multiplikation zweier Zahlen (in Programmiersprachen: *)
div	ganzzahlige Division (mit stets nichtnegativem Rest)
mod	Modulo-Funktion
/	ganzzahlige Division (wie bei reellen Zahlen, aber dann nächste ganze Zahl in Richtung zur Null nehmen)
rem	Rest bei der Division mit /
exp	Exponentiation (exp(x,y) = x^y, nur für y ≥ 0 definiert; in Programmiersprachen verwendet man das Zeichen **)

Alle sechs Vergleichsoperationen "=", "≠", "<", "≤", ">" und "≥" der Funktionalität $\mathbf{Z} \times \mathbf{Z} \to \mathbb{B}$ kommen hinzu.

Definition gewisser Operationen:

Alle Operationen werden als bekannt vorausgesetzt, nur div, mod, / und rem werden hier nochmals präzisiert. Diese vier Operationen sind nur für y≠0 definiert.

div und mod sind eindeutig festgelegt durch (für y≠0):

x = (x div y)·y + (x mod y) mit 0 ≤ x mod y < abs(y).

Die integer-Division "/" ist definiert durch:

x / y = sgn(x·y)·(abs(x) div abs(y))

Hieraus ergibt sich dann rem durch (für y≠0):

(x rem y) = x - (x / y)·y

Auf den natürlichen Zahlen stimmen div und / sowie mod und rem überein.

Machen Sie sich den Unterschiede genau klar: Beide Funktionspaare div und mod sowie / und rem erfüllen die Gleichung:

x = f(x,y)·y + g(x,y) mit 0 ≤ abs(g(x,y)) < abs(y).

Während mod stets nicht-negative Werte liefert, erhält man bei rem ein nicht-positives Resultat, falls der erste Operand negativ ist.

Beispiele:

16 div 5 = 3,	16 mod 5 = 1,	16 / 5 = 3,	16 rem 5 = 1,
16 div (-5) = -3,	16 mod (-5) = 1,	16 / (-5) = -3,	16 rem (-5) = 1,
(-16) div 5 = -4,	(-16) mod 5 = 4,	(-16) / 5 = -3,	(-16) rem 5 = -1,
(-16) div (-5) = 4,	(-16) mod (-5) = 4,	(-16) / (-5) = 3,	(-16) rem (-5) = -1.

Prioritäten der gängigen Operatoren (in Ada anders!):

Im "Alltag" lauten die Prioritäten:

1. Priorität:	Absolutbetrag abs
2. Priorität:	Exponentiation exp
3. Priorität:	·, div, mod, /, rem
4. Priorität:	+, - (als einstellige Operationen)
5. Priorität:	+, - (als zweistellige Operationen)
6. Priorität:	=, ≠, <, ≤, >, ≥
7. Priorität:	not
8. Priorität:	and
9. Priorität:	or
10. Priorität:	equiv, exor
11. Priorität:	impl

Beziehungen und Gesetzmäßigkeiten:

Die ganzen Zahlen bilden einen mathematischen Ring. Also gelten die üblichen Gesetze: Die Addition ist kommutativ und assoziativ mit der Einheit 0 und der Inversenbildung "-x", die Multiplikation ist kommutativ und assoziativ mit der Einheit 1, die beiden Operationen sind das Distributivgesetz miteinander verbunden. Weitere Gesetzmäßigkeiten lassen sich herleiten, zum Beispiel:

odd(x) = not even(x)

((-x) div y) = -1 - (x div y)

x mod y = y - ((-x) mod y)

Der Datentyp integer in der Sprache Ada 95

In Ada ist auch der Datentyp integer ein Aufzählungstyp, der von der kleinsten bis zur größten darstellbaren Zahl reicht:

type integer is (Integer'First, Integer'First+1, ..., Integer'Last);

Daher sind auch Min, Max, pred, succ, pos und val für Integer definiert, allerdings mit der Ausnahme, dass die Zahl 0 die Positionsnummer 0 erhält, d.h., für Integer gilt für alle Zahlen a:

pred(a) = a-1; succ(a) = a+1; pos(a) = a, val(a) = a.

In der Praxis orientiere man sich stets zuvor, welche Zahlen auf dem jeweiligen Rechner Integer'First und Integer'Last sind. Heute sind es meist die Zahlen -2_147_483_648 = -2^{31} und $2^{31}-1$ (dies entspricht einer 32-Bit-Darstellung im Zweierkomplement).

Mit Ausnahme von div, sgn, square, odd, even sind alle Operatoren in Ada vorhanden, wobei ** für die Exponentiation zu verwenden ist.

Gewisse Kombinationen sind in Ausdrücken ohne zusätzliche Klammerung verboten, insbesondere:
Folgt ein einstelliger Operator nach einem höher- oder gleichrangigen zweistelligen Operator, so müssen Klammern verwendet werden.
Da die Exponentiation nicht assoziativ ist, muss bei Mehrfachverwendung von ** geklammert werden.
Kombinationen aus NOT, ABS und ** dürfen nur mit Klammern verwendet werden.

Verboten sind also: x / -y, x**ABS y, x**y**z, ABS X**Y, NOT ABS(X) > 17.

Prioritäten der gängigen Operatoren in Ada 95:

1. Priorität:	ABS, **, NOT
2. Priorität:	*, MOD, /, REM
3. Priorität:	+, - (als einstellige Operationen)
4. Priorität:	+, - (als zweistellige Operationen)
5. Priorität:	=, ≠, <, ≤, >, ≥
6. Priorität:	AND, OR, XOR

Bei Integer ist (im Gegensatz zu Boolean) die Verwendung gleichrangiger Operatoren ohne Klammern erlaubt (siehe jedoch vorige Folie). Dies gilt auch für FLOAT.

Empfehlung: In Ada möglichst vollständig klammern!

1.3.1.5 Der Datentyp real

Zugrunde liegende Menge:
$\mathbb{R}$ = Menge der reellen Zahlen mit der üblichen Ordnung.
($\mathbb{R}$ ist die "Vervollständigung" von $\mathbb{Q}$, d.h., die Menge aller Grenzwerte konvergierender Folgen rationaler Zahlen, grafisch dargestellt durch den "Zahlenstrahl".)
Man beachte, dass diese Menge überabzählbar ist. Darstellen können wir immer nur eine höchstens abzählbare Teilmenge.

Nullstellige Operationen: Alle reellen Zahlen, die mit endlich vielen Ziffern und weiteren Symbolen (wie z.B. π, e, das Wurzelzeichen, die Potenzbildung) dargestellt werden können; hierzu gehören insbesondere alle rationalen Zahlen.

Einstellige Operationen:

+	einstelliges Plus (wie Identität, verändert also nichts)
-	einstelliges Minus, Bildung der negativen Zahl, Negation
abs	Absolutbetrag einer Zahl
square	Quadrat einer Zahl (square(x) = x^2)
sqrt	Positive Wurzel (square root) einer Zahl $x \geq 0$
sgn	$\mathbb{R} \to \mathbb{Z}$ "Signum", also das Vorzeichen einer Zahl:

$$\text{sgn}(x) = \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \\ -1, & \text{falls } x < 0 \end{cases}$$

Man kann sgn auch als Abbildung $\mathbb{R} \to \mathbb{R}$ auffassen.

Hinzu kommen weitere Funktionen wie log (Logarithmus zur Basis 10), ln (Logarithmus zur Basis e), sin (Sinus), cos (Cosinus), tan (Tangens), cot (Cotangens), arcsin (Arcussinus), arccos, arctan, arccot, sinh (Sinus hyperbolicus), cosh, tanh, coth, arcsinh, arccosh, arctanh, arccoth usw.

Einstellige Operationen (Fortsetzung):

Eine wichtige Funktion ist das Abschneiden der Nachkommastellen von reellen Zahlen, wobei man die nächst größere oder die nächst kleinere ganze Zahl erhält. Dies wird in der Mathematik durch die obere und die untere Gaußklammer und durch die "Truncate"-Funktion beschrieben:

$\lceil\ \rceil, \lfloor\ \rfloor : \mathbb{R} \to \mathbf{Z}$ (obere und untere Gaußklammer)

trunc: $\mathbb{R} \to \mathbf{Z}$ ("truncate" = abschneiden) mit
trunc(x) = if $x \geq 0$ then $\lfloor x \rfloor$ else $\lceil x \rceil$ fi.

Hinzu kommt das Runden zur nächsten ganzen Zahl:
round: $\mathbb{R} \to \mathbf{Z}$

Zweistellige Operationen:

+	Addition zweier Zahlen
-	Subtraktion zweier Zahlen, Differenz zweier Zahlen
·	Multiplikation zweier Zahlen (in Programmiersprachen: *)
/	Division (die Division durch 0 ist undefiniert)
exp	Exponentiation ($\exp(x,y) = x^y$; in Programmiersprachen verwendet man hierfür das Zeichen **)

Hinzu kommen alle sechs Vergleichsoperationen "=", "≠", "<", "≤", ">" und "≥" der Funktionalität $\mathbb{R} \times \mathbb{R} \to \mathbb{B}$.

Definition gewisser Operationen:

Die meisten Operationen werden als bekannt vorausgesetzt.

Eine wichtige Funktion ist das Abschneiden der Nachkommastellen von reellen Zahlen, wobei man die nächst größere oder die nächst kleinere ganze Zahl erhält. Dies wird in der Mathematik durch die obere und die untere Gaußklammer beschrieben:

$\lceil\ \rceil, \lfloor\ \rfloor : \mathbb{R} \to \mathbf{Z}$, die obere und untere Gaußklammer sind definiert durch

$\lceil x \rceil = \text{Min}\{\ z \in \mathbf{Z} \mid x \leq z\ \}$,

$\lfloor x \rfloor = \text{Max}\{\ z \in \mathbf{Z} \mid z \leq x\ \}$.

Es gilt stets $\lfloor x \rfloor \leq \lceil x \rceil$
und nur im Falle ganzer Zahlen x ist $\lfloor x \rfloor = x = \lceil x \rceil$.

Definition gewisser Operationen (Fortsetzung):

Will man die Nachkommastellen unabhängig vom Vorzeichen abschneiden, so nimmt man die Funktion trunc.

trunc: $\mathbb{R} \to \mathbf{Z}$ ("truncate" = abschneiden) mit
trunc(x) = if $x \geq 0$ then $\lfloor x \rfloor$ else $\lceil x \rceil$ fi.

Oft benötigt man das Runden zur nächsten ganzen Zahl:
round: $\mathbb{R} \to \mathbf{Z}$ ("round" = runden) mit

für $x \geq 0$: $\text{round}(x) = z \Leftrightarrow z \in \mathbf{Z}$ und $\text{abs}(z-x) < 0.5$ oder $\text{abs}(z-x) = 0.5$ und $z > x$,

für $x < 0$: $\text{round}(x) = -\text{round}(-x)$.

Z.B.: round(2.4) = 2, round(2.5) = 3, round(-2.4) = -2, round(-2.5) = -3.

Prioritäten der gängigen Operatoren:

Man führt keine zusätzlichen Prioritäten ein, sondern übernimmt die Reihung aus Folie 37.

Die Gaußklammern wirken wie Klammern und trunc und round schreibt man ebenfalls mit (runden) Klammern. Diese neuen einstelligen Funktionen bedürfen daher keiner Priorität.

Beziehungen und Gesetzmäßigkeiten:

siehe Mathematikbücher

Der Datentyp real in der Sprache Ada 95

In Ada müssen die reellen Werte durch endlich viele Zeichen dargestellt werden und zwar entweder als Gleitkommazahl oder als Festkommazahl.

Darstellung als Festkommazahl: Man gibt in Ada die kleinste ("unten") und die größte ("oben") darstellbare Zahl an und legt die feste Differenz "d" zwischen je zwei aufeinander folgenden Zahlen fest. Definitionsschema eines solchen "real-Datentyps":

type fkz is delta d range unten .. oben;

Beispiel: type km is delta 0.001 range -1000.0 .. 1000.0; Dies legt das Intervall von -1000.0 bis 1000.0 als Wertemenge fest, das mit der Schrittweite 0.001 durchlaufen wird. Dies sind insgesamt 2_000_001 Werte.

Darstellung von Festkommazahlen durch Dezimalziffern

Eine andere Möglichkeit, wie sie zum Beispiel im Geldverkehr gefordert wird, ist die Angabe der Nachkommastellen und der Gesamtzahl an Dezimalziffern. In Ada beschreibt man diesen durch Dezimalziffern festgelegten Festkommazahlentyp wie folgt (a ist eine natürliche Zahl, p eine Zehnerpotenz):

type dzz is delta p digits a;

Der Wertebereich sind alle Zahlen, die mit a Dezimalziffern beschrieben werden können, wobei das Komma so zu setzen ist, dass zwei aufeinander folgende Zahlen den Abstand p voneinander haben.

Beispiel: type kontostand is delta 0.01 digits 8;
Dies legt als Wertemenge genau die Zahlen des Intervalls von -999999.99 bis 999999.99 mit der Schrittweite 0.01 fest.

Darstellung von Gleitkommazahlen

Hier muss man zunächst nichts tun, denn in Ada ist der Typ FLOAT vordefiniert, der der Gleitkommadarstellung entspricht und mindestens eine Genauigkeit von 6 Dezimalziffern besitzen muss; dies entspricht einer Mantisse von mindestens 21 Binärstellen. Weiterhin gibt es einen Typ LONG_FLOAT mit mindestens 11 Dezimalziffern Genauigkeit.

Man kann die Genauigkeit vorgeben, z.B. durch

type MY_FLOAT is digits 14;

wodurch die Mantisse durch mindestens 48 Binärstellen dargestellt werden muss.

(Faustformel: Wegen $2^{10} \approx 10^3$ kann man von "digits a" auf rund $10 \cdot a/3 + 1$ Binärstellen schließen; "+1" wegen des Vorzeichens.)

Operationen auf den reellen Typen

Für Gleitkomma- und Festkommadarstellungen gibt es die einstelligen Operationen +, - und Absolutbetrag und die zweistelligen Operationen Addition +, Subtraktion -, Multiplikation *, Division / und Exponentiation ** (rechts von ** darf in Ada nur eine ganze Zahl stehen).

Weiterhin gibt es die sechs Vergleichsoperatoren: =, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist. Man sollte wegen der unvermeidlichen Rundungsfehler die Vergleichsoperatoren = und /= bei reellen Zahlen aber nicht verwenden.

Darüber hinaus gibt es eine Vielzahl weiterer Funktionen in speziellen Paketen für die reellen Datentypen, z.B. in Ada.Numerics.Generic_Elementary_Functions .

Zulässige Operanden

Ada ist "streng typisiert", was insbesondere bedeutet, dass die Datentypen der Operanden in Operationen genau festgelegt sind und beachtet werden müssen. Zum Beispiel ist in

```
X, Z: FLOAT;  K, L: INTEGER;  ...
Z := X + K;
```

die Wertzuweisung nicht zulässig, da der Operator "+" entweder für zwei ganze Zahlen oder für zwei reelle Werte definiert ist, aber nicht für einen Mix hieraus. Man muss zunächst die Datentypen der Operanden anpassen:

Z := X + FLOAT(K); oder L := INTEGER(X) + K;

FLOAT bewirkt die Umwandlung einer ganzen Zahl in die entsprechende reelle Zahl; INTEGER entspricht genau der Funktion round.

Zulässige Operanden

Dagegen ist die Multiplikation von ganzen mit reellen Zahlen erlaubt, ebenso die Division einer reellen Zahl durch eine ganze Zahl.

Die Prioritäten sind wie auf Folie 41 angegeben.

Auf weitere Details gehen wir hier nicht ein. Sie sind in jedem Buch über Ada aufgelistet und werden in den Übungen und den Programmierübungen "nebenbei" mitbehandelt.

Hinweis: In Ada 95 sind die Funktionen PRED und SUCC auch auf den real-Datentypen zugelassen. Sie liefern die vorherige bzw. nächst größere darstellbare Zahl.

1.3.2 Konstruktoren

Sind Datentypen gegeben, so kann man hieraus neue Datentypen erzeugen. Man orientiert sich hierbei an den mathematischen Operatoren auf Mengen. Solche Operatoren sind:

- Bildung von Unterbereichen (Intervalle, Projektion, "range"),
- kartesisches Produkt ("record"),
- kartesisches Produkt mit sich (Vektoren, Matrizen, "array"),
- disjunkte Vereinigung ("varianter record"),
- Folgen-Bildung (freies Monoid, "string"-Bildung),
- Menge der Teilmengen (Potenzmenge, "set of"),
- Menge von Abbildungen ("function").

Grundsätzliches Vorgehen:

Zu jedem Konstruktor K, der auf k Datentypen T_i wirkt,

$K\ (T_1, T_2, ..., T_k)$

wird angegeben:

- Wie sieht die Wertemenge aus (bezogen auf die Wertemengen der Datentypen T_i)?
- Wie kann man auf die einzelnen Komponenten T_i zugreifen?
- Welche Operationen der Datentypen T_i werden übernommen und in welcher Form?
- Werden neue Operationen hierdurch eingeführt?

Ein Beispiel, das Sie bereits kennen:

Der Konstruktor array wirkt auf zwei Datentypen:

array (T_1, T_2)

in Ada geschrieben als

ARRAY (T_1) OF T_2

Wenn T_1 die Wertemenge D mit n Elementen und T_2 die Wertemenge M besitzen, so ist M^D die zu array (T_1, T_2) gehörende Wertemenge (bis auf Isomorphie).

Für ein beliebiges Element $a \in D$ greift man auf den zu a gehörenden Funktionswert in M zu mittels

[a] (in Ada: (a)).

Ein Element aus array (T_1, T_2) ist somit eine als Tabelle dargestellte Abbildung von D nach M.

Ein Beispiel (Fortsetzung)

Oft ist die Wertemenge des Indexdatentyps T_1 eine Menge von Zahlen, z.B. die Menge {1, 2, ..., n}. In diesem Fall gehört zu

array (T_1, T_2)

die Wertemenge M^n (also ein Vektor über M). Man greift auf die einzelnen Komponenten zu mittels

[i] (in Ada: (i))

für eine Zahl $1 \le i \le n$.

Außer dieser Zugriffsoperation werden durch den Konstruktor array keine neuen Operationen eingeführt.

1.3.2.1 Unterbereiche (vgl. 1.1.2.20)

Unterbereich oder Intervall $[a .. b] = \{x \in M \mid a \le x \le b\}$.
Speziell: Intervallbildung auf diskreten Mengen:
Wenn $M = \{..., m_1, m_2, m_3, ..., m_q, ...\}$ eine abzählbare Menge ist mit $m_1 < m_2 < m_3 < ... < m_n$, dann ist ein Intervall eine Teilmenge $[m_i, m_k] = \{m_i, m_{i+1}, m_{i+2}, ..., m_k\} \subseteq M$ mit $1 \le i \le k \le n$.

Definition eines Datentyps "intervall" als Unterbereich des Datentyps T:

type intervall = T $[m_i .. m_k]$;

Alle Operationen von T gelten auch für "intervall", sofern der Unterbereich hierbei (und bei Zwischenrechnungen) nicht verlassen wird (letzteres kann man auch anders festlegen; z.B. beachtet Ada die Zwischenrechnungen nicht).

type natural = integer [0 .. flex];

Mit Hilfe des Symbols "flex" (=flexibel) lässt man die jeweilige Grenze offen. Dies kann man auch nach unten verwenden:

type kleiner80 = natural [flex..79];

type integer2 = integer [flex..flex];
Dies definiert einen Datentyp, der genau wie die ganzen Zahlen aufgebaut ist, aber zunächst nicht "kompatibel ist. D.h. für var X: integer; Y: integer2; ... ist die Zuweisung X:=Y verboten.
(integer könnte für die Maßeinheit "Meter" und integer2 für "Fuß" stehen; man darf nicht einfach ohne Anpassungen zwischen verschiedenen Maßsystemen umschalten.)

Die Unterbereichsbildung in Ada:

Unterbereiche werden in Ada als "subtype" eines anderen Datentyps bezeichnet. Die allgemeine Form lautet:

subtype <Name> is <Datentyp> [<constraint>]

Die Beschränkung <constraint> gibt den Bereich ("range") von unterer bis oberer Grenze an:
<constraint> ::= range <Ausdruck> .. <Ausdruck>

Alle Operationen des Typs <Datentyp> sind auch für den Unterbereichsdatentyp gültig.

Die Beschränkung darf fehlen; dann hat der Unterdatentyp die gleiche Wertemenge wie der <Datentyp>.

Basistyp

In der Definition

subtype U is T <constraint>

ist U der Unterdatentyp von T und T der Oberdatentyp von U. Der Datentyp, von dem U letztlich abgeleitet wurde, heißt Basisdatentyp zu U. Es sei T ein elementarer Datentyp. Im Falle

subtype U is T <constraint>
subtype V is U <constraint>
subtype W is V <constraint>

ist W Unterdatentyp von V, von U und von T. U ist Obertyp von V und von W. Der Basisdatentyp von U, V und W ist T.

Operationen

"subtype" bezieht sich ausschließlich auf die Wertebereiche. Alle Operationen des Obertyps werden vom Unterdatentyp übernommen.

In Ada wird jede Variable eines Unterdatentyps stets auch als Variable des Basistyps aufgefasst, so dass man also rechnen kann, als ob man im Obertyp wäre; erst bei der Zuweisung des Ergebnisses wird geprüft, ob das Ergebnis die Beschränkung erfüllt.

Beispiele

subtype Monatstage is integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

Wenn man nun zu einem Tag vom Typ Monatstage den Tag der folgenden Woche (plus 7) berechnen will, so könnte man schreiben:

X, Z: Monatstage; .....
Z := X + 7;
if Z > 31 then Z := Z-31 end if;

Dies führt natürlich zu einem Fehler, falls X mindestens den Wert 25 besaß, da dann Z den Unterbereich verlassen würde. Dagegen führt if X > 24 then Z := X + 7 - 31 end if; nicht einem Fehler, auch wenn zwischenzeitlich der Bereich verlassen wurde, da Ada den Ausdruck X+7-31 im Obertyp integer berechnet.

Zuweisungen an Variablen der Unterdatentypen sind immer erlaubt, wenn der zuzuweisende Wert im Unterbereich liegt:

subtype Monatstage is integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

X, Y, Z: Monatstage; A: MinMonatstage; H, J: integer; .....
H := 50; X := 30; Z := 20;

Erlaubt sind dann (beachte: (H*Z) mod 30 +1 ist ein Ausdruck in den ganzen Zahlen, der dort berechnet wird, erst das Ergebnis wird der Variablen Y zugewiesen):

A := Z; Y := H-20; Y := (H*Z) mod 30 +1; J := X; A := X-Z;

Verboten sind dann (d.h., zu Bereichsüberschreitungen führen):

A := X; Z := H; A := Z*Z;

Wichtige vordefinierte Unterdatentypen in Ada sind:

subtype natural is integer range 0..integer'Last;

subtype positive is integer range 1..integer'Last;

Unterbereiche kann man auch für den Typ float einführen, sofern die Grenzen Ausdrücke sind, deren Ergebnisse reellwertig sind:

subtype einheitsintervall is float range 0.0 .. 1.0;

1.3.2.2 Felder (vgl. 1.1.2.21)

Für Elemente der Wertemengen M^n benötigt man n Variablen der gleichen Art, auf die mit einem Index zugegriffen werden kann. Diese fasst man unter einem Namen zu einem "Feld" (oder Vektor) zusammen. Man muss den Indexdatentyp und den Wertedatentyp angeben: array <Indexdatentyp> of <Datentyp>.

Hierbei 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>
und nennt dies ein zwei-dimensionales Feld. Wiederholt man dies, so lassen sich d-dimensionale Felder deklarieren, d ≥ 1.

Beispiel aus 1.1.2.21: (In Ada ist die Darstellung etwas anders!)

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

Hierzu bilden wir den Unterbereich:

type Arbeitstage = Wochentage [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[J] zugegriffen. Dabei durchläuft J den Wertebereich des Datentyps Arbeitstage, d.h. J∈ {Mo, Di, Mi, Do, Fr}.

Ein weiteres Beispiel "skalarprodukt1" finden Sie unter 1.1.2.21.

Unendliche Indexbereiche:

In der Programmierung sind Wertebereiche der Form M∞ ungebräuchlich, auch wenn sie in der Mathematik eine bedeutende Rolle spielen. Will man solche Wertebereiche jedoch verwenden, um eine Folge von Werten darzustellen, so sollte man die Folgenbildung (siehe später) benutzen.

Man könnte aber auch einen Datentyp

array [1..flex] of ...

einführen, wobei das Symbol "flex" (=flexibel) bedeutet, dass die obere Grenze offen bleibt und sich während der Berechnungen ständig ändern kann.

Diese Möglichkeit gibt es in manchen Sprachen, z.B. in Algol 68 und in gewisser Form auch Ada.

Felder in Ada:

Grundsätzliche Form zur Einführung eines Feld-Datentyps:

type <name> is array (<Indexdatentyp>) of <Datentyp>

Beispiele:

type Hvektor is array (integer range 1..100) of float;

Da durch 1..100 bereits der Datentyp integer festgelegt ist, darf man in Ada hierfür auch kürzer schreiben:

type Hvektor is array (1..100) of float;

type Hmatrix is array (1..50) of Hvektor;

type Bundesligatabelle is array (1..18) of fussballverein;

type codierung is array (Character) of Character;

Die iterierte array-Bildung kürzt man ebenfalls ab, indem alle Indexbereiche in eine Definition geschrieben werden:

Statt

type Hvektor is array (1..100) of float;
type Hmatrix is array (1..50) of Hvektor;
type Hvolume is array (1..80) of Hmatrix;

schreibt man also kurz:

type Hvolume is array (1..100,1..50,1..80) of float;

Die Zahl der hierbei verwendeten Indexbereiche heißt die *Dimension* des Feldes. Hvolume ist also ein 3-dimensionales Feld.

Statt Konstanten dürfen in den Indexbereichen auch Ausdrücke verwendet werden, sofern jeder Ausdruck in dem Augenblick, in dem die Deklaration erreicht wird, auch tatsächlich ausgerechnet werden kann.

Beispiele (wobei f und g Funktionen vom Ergebnistyp integer sein sollen):

type Hvektor is array (1..N, x..I*J) of Boolean;
type ausschnitt is array (f(unten)..g(oben)) of integer;
type sonstiges is array ((oben-unten) div 2..f(g(unten*oben))) of float;

Ein Feld heißt *statisch*, wenn zur Übersetzungszeit (also unabhängig von Eingabewerten) alle Feldgrenzen bekannt sind. Wird mindestens eine Feldgrenze erst zur Laufzeit des Programms berechnet, so heißt das Feld *dynamisch*.

Auch *unspezifizierte* Feldgrenzen sind zulässig. Man trägt dann nur den Datentyp des Index ein und fügt ein

range <> ("<>" sprich "box")

hinzu. Solche unspezifizierten array-Deklarationen *muss* man bei ihrer Verwendung spezifizieren, z.B. bei der Deklaration von Variablen und beim konkreten Parameteraufruf.

Beispiele:

type text is array (natural range <>) of Character;
type raster is array (integer range <>, integer range <>) of Boolean;
type ganzzahlvektor is array (integer range <>) of integer;

Bei der Deklaration von Variablen gibt man dann die konkreten Grenzen statisch oder dynamisch vor, z.B.:

X: ganzzahlvektor (-10..10) oder Y: ganzzahlvektor (I..J)

In Ada sind weitere Operationen mit arrays verbunden (in Ada werden sie als Attribute bezeichnet):

Die Konstante "Range(i)" gibt zu jeder Feld-Variablen den Indexdatentyp ihrer i-ten Dimension an. Analog bezeichnet "Length(i)" die Anzahl der Elemente des Indexdatentyps in der i-ten Dimension. Mittels First(i) und Last(i) erhält man das erste bzw. letzte Element des Indexdatentyps der i-ten Dimension. Hat man nur eine Dimension, so darf man "(1)" weglassen.

Beispiel:

```
type Codes is (character range 'A'..'Z', 1..32) of
                                    character range 'B'..'z';
W: Codes;                Dann gilt:

W'Range(2) = 1..32,      W'Length(1) = 26,
W'First(1) = 'A' ,       W'Last(2) = 32
```


Weiterhin darf man Feldvariablen, die *vom gleichen Datentyp* sind, einander zuweisen.

```
type vektor is array (1..100) of character;

X, Y: vektor;

....

X := Y;
```

Bedeutung dieser Zuweisung: Der gesamte Inhalt von Y wird komponentenweise in die Variable X kopiert.

Bei Ada gibt es eine klare restriktive Regel, wann zwei Variablen A und B vom gleichen Datentyp sind: Sie müssen den gleichen "benannten" Datentyp haben, d.h., sie besitzen entweder den gleichen vordefinierten Datentyp (Boolean, character, integer, natural, positive, float usw.) oder es gibt eine Typ- oder Untertyp-Definition mit einem Namen und beide Variablen sind mit diesem Namen deklariert worden.

"Benannt" bedeutet, dass dem Datentyp ein Name per Datentypdefinition zugewiesen ist. In Ada kann man auch Feld-Variablen unbenannt deklarieren:

```
X: array (1..100) of character;
```

Solch eine Deklaration wird stets als neue, noch nicht vorhandene Feld-Datentypdefinition aufgefasst (s.u.).

Gleichen Datentyp haben z.B.:

```
type zehnziffern is array(1..10) of 0..9;
A, B: integer; A, B: positive;
X, Y: zehnziffern;
```

Dagegen haben *nicht* den gleichen Datentyp in Ada:

```
D: zehnziffern; E: array (1..10) of 0..9;
F, G: array(1..10) of 0..9;
H, I: 0..50;
```

denn Ada fasst H, I: 0..50 auf als die Deklarationsfolge

```
H: 0..50;
I: 0..50;
```

und diese beiden Datentypen sind verschieden, da sie nicht denselben vom Benutzer vergebenen Namen besitzen.

Weiteres Beispiel:

```
type Codes is (character range 'A'..'Z', 1..32) of
                                    character range 'B'..'z';
V, W: Codes;  R, S: character range W'Range(1);
begin
for I in W'Range(1) loop
    for K in W'Range(2) loop
        W(I,K) := val(pos(I) + K);  end loop;  end loop;
V := W;      -- dies bewirkt, dass das ganze Feld W nach V kopiert wird
R := W('C',17);   -- es wird geprüft, ob das Ergebnis im Unterdatentyp liegt
S := W(R,32);
...
end;
```