Array-Syntax in F

Arrays als eigener Datentyp

Arrays sind in F ein- bis siebendimensionale rechteckige Anordnungen (Vektoren, Matrizen, ...) von Daten desselben Typs. Wie in anderen Programmiersprachen auch, können Arrays elementweise, durch Angabe der Indizes in den einzelnen Dimensionen, angesprochen und die Array-Elemente wie gewöhnliche skalare Variablen in Ausdrücken, Zuweisungen und Prozeduraufrufen verwendet werden. Darüber hinaus gibt es in F aber auch noch die Möglichkeit, mit Arrays *im ganzen* zu operieren. Dadurch entfällt oft die Notwendigkeit, `do`-Schleifen explizit auszuprogrammieren, und es entsteht ein wesentlich dichterer Code. Dies ist vor allem bei der Implementation von Vektor- und Matrix-orientierten Algorithmen der Linearen Algebra, bei der Lösung partieller Differentialgleichungen und in der Bildverarbeitung von Vorteil.

Arrays können in F, ohne Angabe von Indizes, in Ausdrücken sowie auf der rechten und linken Seite von Zuweisungen vorkommen. Damit wird in Kurzschreibweise angedeutet, daß die betreffenden Operationen *elementweise* auf einander ensprechende Elemente der Arrays anzuwenden sind. Natürlich müssen dazu Rang (Anzahl der Dimensionen) und Anzahl der Elemente der beteiligten Arrays übereinstimmen. Eine Ausnahme bilden Skalare, die auf der rechten Seite von Zuweisungen zu Arrays der passenden Größe ergänzt werden.

Es seien z.B. die Definitionen

```
integer::i,j,j1,j2,j3
integer,dimension(5)::iv
real::s
real,dimension(10)::u,v,w
real,dimension(10,10)::a,b,c
```

gegeben. Dann ist

```
u=v+w
```

die "Vektoraddition", die man sonst mit Hilfe einer Schleife als

```
do i=1,10
  u(i)=v(i)+w(i)
end do
```

schreiben müßte. Hingegen ist

```
a=b*c
```

keine Matrixmultiplikation (dafür gibt es eine eigene Funktion `matmul`), sondern das elementweise Produkt von `b` und `c`

```
do i=1,10
  do j=1,10
    a(i,j)=b(i,j)*c(i,j)
  end do
end do
```

Der Befehl

```
u=v+s
```

addiert den Skalar `s` zu allen Elementen des Vektors `v`.

Alle eingebauten Funktionen, bei denen dies sinnvoll ist, sind "elementweise" definiert, d.h. ruft man sie mit einem Argument vom Typ Array auf, so liefern sie auch ein Resultat vom entsprechenden Rang, bestehend aus der Anwendung der Funktion auf die einzelnen Elemente des Argument-Arrays. So ist z.B.

```
u=sin(w)
```

wieder äquivalent zu

```
do i=1,10
  u(i)=sin(w(i))
end do
```

Sowohl eingebaute als auch selbst geschriebene Funktionen dürfen Array-wertig sein, egal ob sie elementweise oder allgemeiner definiert sind; z.B. könnte man eine Funktion zur Invertierung quadratischer Matrizen schreiben, die den "Funktionswert"—also die Inverse der übergebenen Matrix—als *ein* Objekt vom Typ Matrix zurückgibt.

Auch bei der Ein- und Ausgabe können Arrays im ganzen angesprochen werden. So gibt etwa

```
write(unit=*,fmt=*) a
```

alle Elemente der Matrix `a` aus, allerdings in der Reihenfolge, in der Arrays in F intern gespeichert sind (also im Fall von Matrizen spaltenweise fortlaufend).

Teilobjekte

Durch die Angabe von Indizes oder Indexbereichen ist es möglich, auch Teilbereiche von Arrays mit Hilfe einer Kurzschreibweise anzusprechen.

Mit den Definitionen von vorhin ist zunächst

```
a(i,j)
```

ein einzelnes, skalares Element der Matrix `a`. Indexbereiche können mittels der `":"`-Notation angegeben werden, wobei

```
a(:,:)
```

für die ganze Matrix steht (also äquivalent zu `"a"`). Der `i`-te Zeilen- und `j`-te Spaltenvektor können mit Hilfe von

`a(i,:)` bzw. `a(:,j)`

selektiert und wie gewöhnliche eindimensionale Arrays verwendet werden. So wäre z.B. in

```
u=a(:,2)+a(:,3)
```

`u` die Summe aus dem zweiten und dritten Spaltenvektor von `a`. Ausschnitte aus Zeilen oder Spalten werden mit einer `do`-Schleifen-artigen Notation angegeben:

```
a(i,j1:j2:j3)
```

steht für alle Elemente der `i`-ten Zeile von `a` mit Spaltenindex von `j1` bis `j2` in Schritten von `j3` (wenn `:j3` fehlt, wird als Schrittweite `1` angenommen).

`a(i,:j2)` und `a(i,j1:)`

stehen für alle Elemente mit zweitem Index kleiner gleich `j2` bzw. größer gleich `j1`.

An Stelle von `i`, `j1`, `j2`, usw. können im allgemeinen natürlich ganzzahlige Ausdrücke treten. Außerdem kann in mehreren Dimensionen gleichzeitig selektiert werden; z.B. ist

```
a(:2,3:5)
```

eine `2*3` Teilmatrix von `a`. Zusätzlich zur Schleifenform können Elemente auch noch mit Hilfe eines (eindimensionalen) Indexvektors ausgewählt werden:

```
a(iv,j)
```

(Wenn diese Konstruktion auf der linken Seite einer Zuweisung steht, darf allerdings in `iv` kein Index mehr als einmal vorkommen.)

Die `where`-Konstruktion

Es kommt oft vor, daß Operationen nur für bestimmte Elemente eines Arrays ausgeführt werden sollen, um z.B. Division durch Null oder illegale Funktionsauswertungen zu vermeiden. Bei konventioneller Schleifenprogrammierung erfolgt dies in der Regel mit `if`-Blöcken innerhalb der Schleife über die Array-Elemente. In F lassen sich derartige Masken kompakt mit der `where`-Konstruktion ausdrücken.

`where(`*log_array_expr*`)`
  array_assignments_true
`elsewhere`
  array_assignments_false
`endwhere`

Dabei ist *log_array_expr* ein Array-wertiger logischer Ausdruck, und die Operationen und Zuweisungen in den *array_assignments_true* (*array_assignments_false*) werden nur für jene Array-Elemente durchgeführt, für die die entsprechenden Elemente im logischen Ausdruck wahr (falsch) sind. Der `elsewhere`-Zweig kann auch fehlen. Natürlich müssen die Dimensionen der Arrays im logischen Ausdruck und in den Zuweisungen übereinstimmen.

Das folgende Beispiel, in dem `a` und `b` entsprechend dimensionierte Arrays vom Typ `real` seien, zeigt, wie man es vermeidet, die Wurzel aus einer negativen Zahl zu ziehen.

```
where(b>0.0)
  a=sqrt(b)
elsewhere
  a=0.0
endwhere
```

Array-spezifische Funktionen

Neben den elementweise definierten Funktionen gibt es noch eine Reihe von Prozeduren zur Abfrage der Eigenschaften und zur allgemeinen Manipulation von Arrays. Es werden hier nur die wichtigsten angeführt.

Eigenschaften

Die Funktion

`size(`*array*`[,`*dim*`])`

liefert die Anzahl der Elemente des Arrays *array*, bzw. wenn das optionale Argument *dim* spezifiziert ist, die Anzahl der Elemente in der Dimension *dim*. Das Resultat von

`shape(`*array*`)`

ist ein Vektor vom Typ `integer`, dessen Komponenten die Anzahl der Elemente von *array* in den einzelnen Dimensionen angeben. Analog liefern

`lbound(`*array*`[,`*dim*`])`

und

`ubound(`*array*`[,`*dim*`])`

wenn *dim* fehlt, `integer`-Vektoren mit den unteren bzw. oberen Grenzen der Indizes in den einzelnen Dimension von *array*, sonst die untere bzw. obere Grenze in der Dimension *dim*.

Rang-verändernde Funktionen

Die Funktionen

`sum(`*array*`,[`*dim*`])`

und

`product(`*array*`,[`*dim*`])`

liefern Summe und Produkt aller Elemente von *array* bzw., wenn das optionale Argument *dim* angegeben wird, ein Objekt mit einem um `1` kleineren Rang als *array*, bestehend aus der Summe (dem Produkt) der Elemente in der Dimension *dim*. [Ist z.B. `a` wieder die `10*10` Matrix von vorhin, so wäre `sum(a,2)` ein Vektor, bestehend aus den Zeilensummen von `a`.] Analog liefern

`maxval(`*array*`,[`*dim*`])`

und

`minval(`*array*`,[`*dim*`])`

Maximum und Minimum der Elemente von *array* bzw. ein Unter-Array, bestehend aus den Maxima (Minima) in der Dimension *dim* des Originals. [Für die Matrix `a` wäre also `maxval(a,2)` der Vektor der Zeilenmaxima.]

[Um herauszufinden, wo in *array* Maximum oder Minimum auftreten, gibt es die Funktionen

`maxloc(`*array*`)`

und

`minloc(`*array*`)`

Das Resultat ist ein `integer`-Vektor mit den Indizes des maximalen (minimalen) Elements.]

Die Funktion

`spread(`*array*`,`*dim*`,`*copies*`)`

erhöht den Rang von *array* um `1`, indem entlang der Dimension *dim* eine Anzahl *copies* von Kopien angelegt wird. [Ist also `u` ein Vektor mit `10` Elementen, so macht `spread(u,2,3)` daraus eine `10*3` Matrix, deren Spaltenvektoren Kopien von `u` sind.]

Die Funktion

`reshape(`*array*`,`*shape*`[,...])`

transformiert *array* in ein Objekt, dessen Dimensionen durch die Elemente des `integer`-Vektors *shape* bestimmt sind. (Die Transformation kann auch noch durch weitere optionale Parameter modifiziert werden.)

Spezielle Matrix- und Vektorfunktionen

Das Skalarprodukt von zwei Vektoren *vector_1* und *vector_2* erhält man mittels

`dot_product(`*vector_1*`,`*vector_2*`)`

wobei *vector_1* und *vector_2* dieselbe Anzahl von Elementen haben müssen.

Die Matrixmultiplikation (im üblichen Sinn) zweier Matrizen *matrix_1* und *matrix_2* bewerkstelligt man mit

`matmul(`*matrix_1*`,`*matrix_2*`)`

Hier muß natürlich die zweite Dimension von *matrix_1* mit der ersten von *matrix_2* übereinstimmen.

Die Transponierte einer Matrix *matrix* erhält man mit

`transpose(`*matrix*`)`

Array-Konstruktoren

Arrays von Konstanten kann man mit Hilfe von Array-Konstruktoren erzeugen. Das sind eindimensionale Listen von Elementen desselben Typs

`(/`*element_1*`,`*element_2*`,.../)`

wobei *element_1*, *element_2*, usw. Konstante oder "implizite `do`-Schleifen" der Form

`(`*expr*`,`*int_variable*`=`*start*`,`*end*`[,`*step*`])`

sind. Im letzteren Fall ist *int_variable* eine Variable vom Typ `integer`, die (wenn angegeben, in Schritten von *step*) von *start* bis *end* läuft, und *expr* ein von der Laufvariablen abhängiger Ausdruck. Zum Beispiel:

```
(/1,2,3,5,8,13/)
(/(1.0/(1.0+real(i)**2),i=0,10)/)
```

Um auf diese Art mehrdimensionale Arrays zu erzeugen, verwendet man die `reshape`-Funktion, wobei zu beachten ist, wie in F mehrdimensionale Objekte im Speicher angeordnet werden. So ergibt

```
reshape((/(k=2,12,2)/),(/2,3/))
```

die Matrix

$$\begin{pmatrix} 2 & 6 & 10 \\ 4 & 8 & 12 \end{pmatrix}$$