Technische Universität Dresden

Skript:

Programmieren I

Verfasser Franziska Kühn

Daten Prof. Dr. Wolfgang V. Walter
Wintersemester 2008/09
Grundstudium

Inhaltsverzeichnis

1 Informatik

1.1 Gebiete der Informatik

1. Technische Informatik:
   - Hardware ($\longrightarrow$ E-Technik/Physik/...)
   - Rechnerarchitektur (parallele Architekturen, Netzwerke,...)
2. Praktische Informatik:
   - Software ($\longrightarrow$ Programmierung)
   - Compilerbau *(Aufruf g95: g95 -o prog prog.f95)*
3. Theoretische Informatik:
   - Automatentheorie
   - formale Sprachen
   - Logik
   - Kryptographie
   - Berechenbarkeit/Komplexität von Algorithmen
   - Einfluss auf Compilerbau
   - SW-Engineering
4. Angewandte Informatik:
   - Benutzer-Schnittstellen
   - Anwendungsprogramme
5. spezialisierte Informatik

1.2 Grundeinheiten

- bit: kleinste Informationseinheit $\{0, 1\}$
- nibble (4 bits): Hexadezimalziffer (0-15), $2^4 - 1$
- byte (8 bits): kleinste adressierbare Speichereinheit, Hexadezimalziffer (0-255), $2^8 - 1$
- allgemein: mit k bits können 2^k verschiedene Speicherzustände codiert werden

2 Fortran 95

2.1 Algorithmen

- Algorithmus = schrittweises Verfahren zur Berechnung von Ergebnisgrößen, jeder Schritt besteht aus einer ausführbaren eindeutigen Operation. Endliche, präzise Beschreibung.
- Algorithmus = endliches und vollständiges System formaler, allgemein verwendbarer regeln, die den Arbeitsgang zur Lösung einer Klasse von Problemen eindeutig bestimmen und in endlicher Zeit auf die Lösung des Problems führen.
- Strukturierungsmöglichkeiten: Zerlegen in Teilalgorithmen mit folgenden Grundstrukturen
  1. Sequenz (Hintereinanderausführung von Befehlen)
  2. Selektion: Verzweigung auf Basis einer Entscheidung (Speicherabfrage)
  3. Wiederhholung (Zyklus)/Iteraton: enthält mehrmals ausführbare Sequenz (while/repeat)

  beliebige Kombination der Grundstrukturen
- Schleifenformen:
  1. do

     DO
     IF (. . .) **exit**
     END DO
  2. while

     DO WHILE(. . .)

     END DO
  3. Iteration mit [x] als Schrittweite

     DO i = 1 , 5 [, 2]

     END DO

- Verzweigung: if

  IF (. . .) THEN

  ELSE [IF (. . .) THEN]

  END IF

2.2 Variablen

- Typen:
  1. logische Variable *(LOGICAL)*
  2. reelle Variable *(REAL)*
  3. ganzzahlige Variable *(INTEGER)*
  4. Zeichenkette *(CHARACTER)*
  5. komplexe Variable *(COMPLEX)*
- Eine Variable in einer imperativen Programmiersprache ist ein 5-Tupel: (Name, Typ, Gültigkeitsbereich, Left-value, Right-value):
  - Typ: legt Menge von möglichen Werten/Zuständen/Objekten fest
  - l-value: (Adressse) der Speicherzelle im Hauptspeicher
  - r-value: Wert der Variable in der Speicherzelle ($\longrightarrow$ zeitabhängige Veränderungen des r-value im Verlauf des Programms)
- Zeichensätze:
  - 7-bit ASCII (lower ASCII): 128 Zeichen (vorderstes bit=0), Standard
  - upper ASCII wird durch Codepage festgelegt (vorderstes bit=1), z.B. ISO xxxx
  - Unicode: 2^{16} Zeichen
- Blockstruktur der Programme:
  1. nicht ausführbare, deklarative Anweisungen (Deklarationsteil)
  2. ausführbare Anweisungen
- Gleitkommazahlen:
  - Format: R(b,l,e_{min}, e_{max}) mit Basis b, Länge l, Exponentenbereich $e_{max} - e_{min}$
    1. single: R(2,24,-125,128)
    2. double: R(2,53,-1021,1024)
  - Gleitkommazahl: entweder 0 oder $x = (-1)^{s_x} \cdot m_x \cdot b^{e_x}$ mit $s_x \in \{0, 1\}$, Mentisse $m_x = 0.m_1 m_2 \dots m_l$ mit $m_i \in \{0, 1, \dots, b-1\}$ und e_x mit $e_{min} \leq e_x \leq e_{max}$

– normalisierte Gleitkommazahl: $m_1 \neq 0$, damit eindeutige Darstellung

$$\frac{1}{b} \leq m_x \leq 1 \qquad m_x = 0.m_1 m_2 \ldots m_l = \sum_{i=1}^{l} m_i \cdot b^{-i}$$

– Basis-Konversion: Beispiele

1. 123_{10} konvertieren in binäre/duale Darstellung (b=2)

$$\begin{aligned} 123 : 2 &= 61 \text{ Rest } 1 \\ 61 : 2 &= 30 \text{ Rest } 1 \\ 30 : 2 &= 15 \text{ Rest } 0 \\ 15 : 2 &= 7 \text{ Rest } 1 \\ 7 : 2 &= 3 \text{ Rest } 1 \\ 3 : 2 &= 1 \text{ Rest } 1 \\ 1 : 2 &= 0 \text{ Rest } 1 \end{aligned}$$

   binär: 1111011

2. $0,1_{10}$ konvertieren in binäre Darstellung:

$$\begin{aligned} 0,1 \cdot 2 &= 0,2 \quad 0 \\ 0,2 \cdot 2 &= 0,4 \quad 0 \\ 0,4 \cdot 2 &= 0,8 \quad 0 \\ 0,8 \cdot 2 &= 1,6 \quad 1 \\ 0,6 \cdot 2 &= 1,2 \quad 1 \\ 0,2 \cdot 2 &= 0,4 \quad 0 \\ 0,4 \cdot 2 &= 0,8 \quad 0 \\ 0,8 \cdot 2 &= 1,6 \quad 1 \\ 0,6 \cdot 2 &= 1,2 \quad 1 \end{aligned}$$

   Ergebnis: $0.0001\overline{10011}$

– Gleitkomma-Arithmetik:
  * $\circ \in \{+, -, \cdot, /\}$
  * Rundungen:
    1. nearest: nächstgelegene GKZ
    2. zum kleinen Betrag (falls nicht exakt darstellbar)
    3. „nach links“
    4. „nach rechts“

– Auslöschung führender Ziffern in Summationsprozessen. Tritt auf, wenn das Endergebnis (die Summe) betragsmäßig (wesentlich) kleiner ist als die größten Beträge der Summanden (bzw. der Zwischenergebnisse)

Beispiel:

$$\begin{aligned} e^{-20} &= 1 - 20 + \frac{400}{2} - \frac{8000}{6} + \ldots \\ e^{x} &= 1 - x + \frac{x^2}{2} - \frac{x^3}{3!} + \ldots \end{aligned}$$

tritt kaum bei Multiplikation auf

- ganze Zahlen:
  - im 2er-Komplement
  - Beispiel: k=4bits

$$\begin{array}{lcl|lcl} 0111 & = & 7 & 1111 & = & -1 \\ 0110 & = & 6 & 1110 & = & -2 \\ 0101 & = & 5 & 1101 & = & -3 \\ 0100 & = & 4 & 1100 & = & -4 \\ 0011 & = & 3 & 1011 & = & -5 \\ 0010 & = & 2 & 1010 & = & -6 \\ 0001 & = & 1 & 1001 & = & -7 \\ 0000 & = & 0 & 1000 & = & -8 \end{array}$$

    asymmetrischer Zahlenbereich: $\{-2^{k-1}, \ldots, 2^{k-1}-1\}$ (nur eine Darstellung für 0)
  - Addition: über alle Bits (inkl. Vorzeichen) addieren, evtl. Überbetrag wird ignoriert
- lexikalische Einheiten: Symbole/Tokens
  1. Keywords (Schlüsselwörter, Wortsymbole): feste Buchstabenfolge mit festgelegter sprachlicher Bedeutung
  2. Identifiers (Namen): $B\{B|Z|\_\}_0^{30}$, Groß-und Kleinbuchstaben werden gleich interpretiert
  3. Literale (Konstanten):
     - ganzzahlig: $[+|-]z\{z\}$
     - reell
     - komplex (z.B. (-2.5,1.0))
     - LOGICAL .TRUE./.FALSE.
  4. Labels (Marken): Sequenz von bis zu 5 Dezimalziffern (z.B. für GOTO-Sprungziel)
  5. Seperatoren (Trennsymbole)

$$( ) \; (/ \; /) \; [ ] \; , \; ; \; \Rightarrow \; = \; : \; :: \; \%$$

  6. Operatoren:

$$+ \; - \; * \; / \; ** \; // \; == \; <= \; >= \; < \; > \; \backslash =$$

     sowie .NOT., .AND., .OR., .EQV., .NEQV., $.\{B\}_1^{31}$. (selbstdefinierte Operatoren)
- Syntax: definiert über

- Alphabet Σ ist eine endliche Menge von sprachbildenden Symbolen, den Terminalen (=Terminalsymbole = lexikalische Einheiten=Tokens), bestehend aus ≥ 1 Zeichen (entsprechend den Wörtern in einem Wörterbuch)
- Formale Sprache über dem Alphabet Σ ist eine Teilmenge aller möglichen endlichen Symbolfolgen über Σ,

  $$L(\Sigma) \subseteq \Sigma* = \text{Menge aller Folgen von Symbolen aus } \Sigma$$

  Diese Teilmenge wird durch eine Grammatik definiert, welche die Syntax der Sprache L festlegt.
- Eine Grammatik G ist ein 4-Tupel $G = (\Sigma, N, S, R)$ mit Σ = endliche Menge sprachlicher Symbole (Terminale), N=endliche Menge nicht sprachbildender Symbole (abstrakte Symbole, Nonterminale) Syntaxvariablen der Grammatik, S=Startsymbol $S \in N$ (z.B. ¡Fortran95-Programm¿), R=endliche Menge von (Syntax)Regeln R mit $R \subseteq N \times (\Sigma \times N)*$
- Bemerkung: Rekursion innerhalb der Syntaxregeln R macht die Sprache erst interessant und erlaubt unendlich viele verschiedene grammatikalisch korrekte „Sätze“.
- kompaktere Notation durch EBNF (extended Bachus-Naur form):
  * jedes Nonterminal wird durch genau 1 Regel $\in R$ definiert (in einer gewissen Notation)
  * wurde zur Grammatikdefinition von Algo 60 erfunden
  * definier eine Grammtik $G = (\Sigma, N, S, R)$, wobei die Nonterminale meist Syntaxvariablen genannt werden

2.3 Intrinsische Datentypen

1. INTEGER(KIND=k)
   - Konstanten: ±123_KIND
   - Wertemenge: $\{-2^{l-1}, \ldots, 2^{l-1} - 1\}$
   - Operatoren: + - * / **
   - Funktionen:
     - int(x)
     - floor(x)
     - mod(x)/modulo(x)

       $$MOD(x,y) = x - int(\frac{x}{y}) \cdot y \qquad MODULO(x,y) = x - FLOOR(\frac{x}{y}) \cdot y$$

     - ceiling(x)
     - NINT(x)
     - SIGN
     - MIN
     - MAX

- ABS

2. REAL(KIND=k)
   - Wertebereich: R(b,l,e_{min}, e_{max})
   - Konstanten:

$$[\pm]\{< z >\}_1^\infty.\{< z >\}_1^\infty[E/D[\pm]\{< z >\}_1^\infty]_k$$

   - Operatoren: + - * / **
   - Funktionen:
     - ABS
     - REAL (i/c [,kind])
     - MIN/MAX
     - AINT
     - SIN/COS/TAN
     - ASIN/ACOS/ATAN
     - ATAN2 $\sim ATAN(\frac{y}{x})$
     - SQRT
     - LOG ($\rightarrow$ ln)
     - LOG 10
     - EXP
     - SINH/COSH/TANH

3. COMPLEX(KIND=k)
   - KIND-Parmaterwerte identisch zu denen er erhaltenen REAL-Typen
   - Wertemenge: geordnete Paare von Gleitkommazahlen in einem festen Gleitkommaformat
   - Konstanten: (re,im)
   - Operatoren: + - * / **
   - Funktionen:
     - ABS ($\sqrt{a^2+b^2}$)
     - REAL ($\rightarrow$ a)
     - AIMAG ($\rightarrow$ b)
     - CONJG ($\rightarrow \bar{z}$)
     - SQRT
     - LOG
     - EXP
     - SIN/COS/TAN
     - CMPLX
   - Vergleichsoperatoren:

$$\begin{aligned} \mathbb{C} &: == \ / = \\ \mathbb{Z}, \mathbb{R} &: == <= >= \ / = < > \end{aligned}$$

- Numerische Ausdrücke: ($\circ \in \{+, -, *, /, **\}$)

$\circ$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{C}$
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{R}$	$\mathbb{C}$
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$\mathbb{C}$
$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$

  Bemerkungen:
  - bei $x^i \in \mathbb{Z}$ mit $x \in \mathbb{R}$ oder $x \in \mathbb{C}$ wird i nicht vor der Potenzierung zu $\mathbb{R}$ bzw. $\mathbb{C}$ konvertiert
  - implizite Typanpassungen: $\mathbb{Z} \to \mathbb{R} \to \mathbb{C}$ und in Richtung genauerer Zahldarstellung, z.B. single → double → quad
  - Wertzuweisung: es werden verengende Konversionen automatisch durchgeführt, z.B. $i = (3.7, 1)$ weist $i = 3$ zu

4. LOGICAL(KIND=k)
   - Wertebereich: {true,false}
   - Konstanten: { .TRUE., .FALSE. }
   - Operatoren: .NOT., .AND., .OR., .EQV., .NEQV.
5. CHARACTER(len=l)
   - Zeichenkette mit l (ASCII)Zeichen
   - default: l=1
   - Wertemenge: Zeichenfolge mit l Zeichen
   - Konstante: 'AB', „321“
   - Operatoren: <> und Verkettung: //
   - Zuweisung:

     ```
     CHARACTER(LEN=10):: z_k
     z_k = 'Wolf'
     ```

     Überlänge wird abgehackt
   - Teilzeichenketten (Substrings):

     ```
     z_k(2:4) = 'alt'
     ```

     Ergebnis: Walt
   - Funktionen:
     - CHAR(i)
     - ICHAR(c)
     - ACHAR(i)
     - IACHAR(c)
     - TRIM(c) (Kürzung um nachfolgende Leerzeichen)
     - LEN(stri) (inkl. Leerzeichen)
     - LENTRIM(stri) (exkl. Leerzeichen)
     - ADJUSTL(stri)
     - ADJUSTR(stri)
     - REPEAT(string,n)
     - INDEX/SCAN/VERIFY

2.4 Expressions

Bei der Auswertung von Ausdrücken wird mit den folgenden Prioritäten gearbeitet:

1. Objektauswertung, z.B. Bestimme Variablenwert, Teilobjekte, Konstanten
2. Geklammerte Ausdrücke (von innen nach außen)
3. Funktionsaufrufe(-auswertung)
4. Operatoren: höhere Priorität vor niedriger
5. Operatoren gleicher Priorität von links nach rechts außer **
6. Prioritäten:
   - (12) benutzerdefiniert unär
   - (11) **
   - (10) / *
   - (9) + - (unär)
   - (8) + - (binär)
   - (7) //
   - (6) < <= == >= > / =
   - (5) .NOT.
   - (4) .AND.
   - (3) .OR.
   - (2) .EQV. .NEQV.
   - (1) benutzerdefiniert binär

Ausdrücke können in Prefix- oder Postfixnotation ausgedrückt werden.

2.5 Globale Struktur eines F95-Programms

- Unterprogramme: Subroutinen oder Funktionen
- Aufruf: CALL sub(x,y,r,s) mit x,y,r,s als aktuelle Argumente (können von Subroutine geändert zurückgegeben werden!)
- Kopf: SUBROUTINE sub(a,b,c,d) mit a,b,c,d als formale Argumente
- für formale Argumente kann festgelegt werden:
  - INTENT(IN): Parameter nur lesen
  - INTENT(OUT): Parameter schreiben (min. 1!)
  - INTENT(INOUT): Parameter lesen und schreiben
- Aufrufmechanismus in 4 Phasen:
  1. Auswertung/Bestimmung der aktuellen Argumente

2. Assoziation der aktuellen Argumente mit den formalen Argumenten (per Referenz)
3. Unterprogramm-Sprung (transfer of control)
   (a) Ausführung der aufrufenden Routine wird temporär suspendiert
   (b) Ausführung des Unterprogramms
4. Rücksprung an die Aufrufstelle bei Erreichen einer Return-Anweisung im aufgerufenen Unterprogramm, Fortsetzung der aufrufenden Routine

- Parameterassoziation:
  - erfolgt per Referenz (=per Adresse), d.h. des vom Aufrufer übergebene aktuelle Argument wird mit dem entsprechenden formalen Argument assoziiert
  - Falls das aktuelle Argument ein echter Ausdruck (nicht nur eine Variable) ist, dann lagt das System eine anonyme temporäre Variable an, die mit dem Ergebniswert des Ausdrucks initialisiert wird. Diese anonyme temporäre Variable wird per Referenz an das formale Argument übergeben.
  - Jegliche Wertänderung eines solchen formalen Arguments ist ein Konzeptionsfehler im Programm.
  - Parametertypen müssen identisch sein (inkl. KIND); für CHARACTER kann CHARACTER(len=*) geschrieben werden
- Seiteneffekte:
  - Veränderung von Variable in einem Ausdruck durch Auswertung einer Funktion in diesem Ausdruck, welche diese Variable als aktuelles Argument erhält und verändert
  - Assoziation mehrerer formaler Argumente mit demselben Argument, wenn eines dieser formalen Argumente im Unterprogramm verändert wird (Aliasproblem).

2.6 Schnittstellen

1. generische Schnittstelle
   (a) generischer Prozedurname

```
INTERFACE genericfunc
        MODULE PROCEDURE specificfunc , sfunc2
END INTERFACE
```

   (b) Operator

```
INTERFACE OPERATOR (+-*)
        MODULE PROCEDURE f_1 , f_2
END INTERFACE
```

   (c) Zuweisung

```
INTERFACE ASSIGNEMENT(=)
        MODULE PROCEDURE subr1
END INTERFACE
```

2. spezifische Schnittstelle

```
INTERFACE
        FUNCTION f(x); REAL::x; COMPLEX:: f END FUNCTION
END INTERFACE
```

2.7 Selbstdefinierte Datentypen

- benutzerdefinierte Typen sind eine Art Speicherschablone/Datenobjektbeschreibung, die eine festgelegte innere Struktur von Objekten eines solchen Typs definiert. Diese besteht aus sogenannten Komponenten, welche jeweils einen Namen und einen Datentyp (intrinsisch/selbstdefiniert) besitzen.
- Typdefinition:

```
TYPE <Typname>
        [PRIVATE]
        {<Komponenten>}
END TYPE <Typname>
```

- Typdeklaration:

```
<Typ> [,Attributliste]:: <Variablenliste>
```

- Komponentenzugriff: mit % nach Variablenname, dahinter Komponentenname, z.B. datum%tag (Verkettung möglich)

2.8 Datenabstraktion

Ein abstrakter Datentyp (ADT) besteht aus einem selbstdefinierten Typ und einer wohldurchdachten Menge von auf Objekten dieses Typs definierten Grundoperationen. Ein abstrakter Datentyp wird immer in einem Modul definiert. Dabei sollte der Zugriff auf die Komponenten der Objekte dieses Typs außerhalb des Moduls verboten sein.

2.9 Arrays

- homogene Datenstruktur, d.h. alle Elemente haben denselben Typ
- ein- oder mehrdimensional (Anzahl der Dimensionen = Rang r $\in \{1, \ldots, 15\}$)
- Zugriff auf Element durch Indizes: vec(i), mat(3,2),...
- Ein Feldtyp (array type) wird charaktisiert durch:
  1. (festen) Rang r

2. (festen) Elementtyp (welcher selbst skalar sein muss, also kein Feldtyp sein darf)

- Ein Elementtyp darf ein beliebiger intrinsischer oder benutzerdefinierter Typ sein.
- Feldtyp aller Feldobjekt wird zur Compile-Zeit festgelegt.
- Jedes Feld hat eine (geometrische) Gestalt (shape), definiert durch Ausdehnung (extents) in den einzelnen Dimensionen (Festlegung statisch oder dynamisch)

```
REAL, DIMENSION([1:]5,0:4):: matrix55
REAL, DIMENSION(:,:), ALLOCATABLE:: dynamtrix
```

- Ausdehnung:

```
size(dynamtrix,i) = UBOUND(dynamtrix,i) - LBOUND(dynamtrix,i)+1
```

- Feldkonstruktor: definiert immer ein 1-dimensionales Feld

$$(/1,2,3,4/) = [1,2,3,4] = [(i, i = 1,4)]$$

  Daraus kann mit Hilfe der intrinsischen Funktion RESHAPE auch ein mehrdimensionales Feld gemacht werden.

- Speicherreihenfolge von Feldelementen spaltendominiert, d.h. die vorderen Indizes laufen schneller als die hinteren
- Speicherbeschaffung- und freigabe für dynamische Felder mit ALLOCATE und DEALLOCATE
- Zugriff auf Feldelemente:
  - echter Variablenzugriff (mit l-value und r-value)
  - Ein Feldelement wird über r Indexwerte, die jeweils im entsprechenden Indexbereich (l_i, u_i) (in der i-ten Dimension) liegen müssen, angesprochen.
- Subarrays:
  - Felder, bestehend aus einer Teilmenge der Elemente eines anderen Feldes
  - Beispiel:

```
REAL, DIMENSION(6):: v,w
v(2:6) = v(1:5) + w(2:6)
```

  - Indexmengen spezifiziert durch
    1. Indextripel (a,e,s) [Anfang, Ende, Schrittweite] mit $s \in \mathbb{Z}\backslash\{0\}$
    2. Indexvektor, z.B. (/1,3,5,7,9/); leere Teilfelder sind zulässig
- Funktionen:

- SUM(A): Summe aller Elemente
- SUM(A_i): Summe über Dimension i (Array mit Rang r-1)
- PRODUCT(A)/PRODUCT(A_i) entsprechend
- für Logical: ANY(A[,i]) = $\bigvee$ und ALL(A[,i]) = $\bigwedge$
- TRANSPOSE(A): Spiegelung der Matrix
- DOTPRODUCT(v,w)
- MATMUL(A,B) = $A \cdot B$ matriziell
- MATMUL(A,w) = $A \cdot w$
- MATMUL(v,A) = $v^T \cdot A$
- RESHAPE

2.10 Feldoperatoren

Alle intrinsischen Operatoren sind auch für beliebige gestaltkonforme Felder (bei binären Operatoren) bzw. beliebigen Feldern (bei unären Operationen) vordefiniert, wenn für deren (skalare) Komponententypen diese Operation vordefiniert ist. Dabei wird die skalare Operation komponentenweise auf sich in ihrer geometrischen Position entsprechenden Elemente (bei binären Operatoren) bzw. auf alle Elemente (bei unären Operatoren) angewandt.
Das Ergebnisfeld hat diesselbe Gestalt wie das/die Operandenfelder. Bei binären Operatoren darf ein Operand auch skalar sein (dessen Wert wird dann in jeder Operation mit einem Element des anderen Operanden verwendet).

$$\begin{pmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{pmatrix} \cdot \begin{pmatrix}2 & 5 & 11\\ 3 & 7 & 13\end{pmatrix} = \begin{pmatrix}2 & 15 & 55\\ 6 & 28 & 78\end{pmatrix}$$

$$\begin{pmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{pmatrix} < \begin{pmatrix}2 & 5 & 11\\ 3 & 7 & 13\end{pmatrix} = \begin{pmatrix}T & T & T\\ T & T & T\end{pmatrix}$$

$$2 \cdot \begin{pmatrix}3\\ 5\\ 2\end{pmatrix} = \begin{pmatrix}6\\ 10\\ 4\end{pmatrix}$$

2.11 Felder als Argumente und Funktionsergebnisse

1. Felder übernommener Gestalt (assumed-shape arrays):
   - formale Argumente, die ihre Gestalt an das jeweils übergebene aktuelle Feld „anpassen“
   - normalerweise werden dabei die Indexbereiche auf Bereiche der Untergrenze=1 verschoben, es sei denn man legt andere Untergrenzen explizit fest.
   - Beispiel:

```
FUNCTION matfun(a,b)
         REAL, DIMENSION([1]:,0:):: a,b
         REAL, DIMENSION(:SIZE(a,1),0:SIZE(a,2)-1):: matfun
```

2. Felder als Funktionsergebnisse:

   - Gestalt bzw. Indexgrenzen müssen durch Ausdrücke, die zum Aufrufzeitpunkt der Funktion bekannt sind, festgelegt werden. Es können z.B. ABfragefunktionen wie SIZE, LBOUND verwendet werden.

3. Automatische Felder:

   - sind lokale Variablen (Felder), welche diesselbe Art von Deklaration wie ein Funktionsergebnis (Feld) haben

2.12 Ein-/Ausgabe

- Eigentliche Aufgabe des I/O ist Datentransfer zwischen Dateien und internen Speicher (memory).
  - Lesen: Datei → Hauptspeicher
  - Schreiben: Hauptspeicher → Datei (Datei: auch Monitor)
- Datei: normalerweise extern (d.h. liegt nicht im Hauptspeicher); eine lineare Anordnung von Daten auf einem externen Speichermedium oder auch Monitor.
- gelegentlich auch intern: Zeichenkettevariable, die im Programm deklariert ist
- Datensatz (record): entweder
  1. formatiert: Daten werden als Sequenzen von Zeichen dargestellt. Zahlwerte müssen zwischen internen binärer und extern dezimaler Darstellung konvertiert werden.
  2. unformatiert: externe Daten sind direktes Abbild des internen Hauptspeichers (bit für bit)
- Datenzugriff:
  1. sequenziell: linearer Zugriff vom Anfang der Datei ausgehend, Existenz einer aktuelle (Byte)Position, Bewegung durch Datei immer „vorwärts“ (BOF → EOF)
  2. direkt: über record number(> 0) wird der gewünschte Datensatz angesprochen
- Standardzugriff:
  - sequenziell, nur datensatzweise (=zeilenweise)
  - mit ADVANCE='NO' wird aktuelle Zeilenposition beibehalten für nächstes READ/WRITE
- weitere I/O-Anweisungen:
  - INQUIRE(...)
  - REWIND([unit=]u, IOSTAT=iostat,ERR=err) (setzt Position an BOF)

- BACKSPACE([unit=]u,IOSTAT=iostat,ERR=err) (setzt Position an Zeilenanfang)
- ENDFILE(...) (schreibt EOF-Datensatz an aktuelle Position)

- Beispiel:

```
      TYPE person
            CHARACTER(len=25):: Vorname, Nachname
            INTEGER:: Alter
            REAL:: Groesse, Gewicht
            LOGICAL:: Geschlecht, Bart
      END TYPE person

      TYPE (person), DIMENSION(500):: Mitglieder

100   FORMAT(4I,1X,2A26,I3,F5.2,F8.3,2L2)
      WRITE(*,FMT=100) (i,Mitglieder(i),i=1,500) !Ausgabe
      READ(*,FMT=100) (i, Mitglieder(i),i=1,500) !Einlesen
```

2.13 Feld-Deskriptoren

- enthält alle laufzeitrelevanten Charakteristika eines Arrays, notwendig für „regelmäßige“ Teilfelder (mit konstanter Schrittweite in jeder Dimension sowie für Feldparameter mit übernommener Gestalt)
- Beispiele