Programmierung Paralleler Systeme

3. Programmiermodell “Single Programm Multiple Data”

3.1 Fortran95

3.1.1 Geschichte

3.1.2 Basiskonzepte

3.1.3 Typvereinbarungen und Spezifikationen

Claus-Uwe Linster

IMMD IV, Lehrstuhl für Betriebssysteme
Universität Erlangen-Nürnberg

linster@informatik.uni-erlangen.de

Programmiersprachen

- 3.1 FORTRAN95
- 3.2 HPF - High-Performance-Fortran
- 3.3 Übungen Fortran95/HPF

3.1 FORTRAN95

- 3.1.1 Geschichte
- 3.1.2 Basiskonzept
- 3.1.3 Typvereinbarungen und Spezifikationen
- 3.1.4 Strukturen (selbstdef. Typen) und Module
- 3.1.5 Zeiger (Pointer)
- 3.1.6 Unterprogramme
- 3.1.7 Feldverarbeitung
- 3.1.8 Dynamische Speicherverwaltung
- 3.1.9 Bedingte Feldzuweisungen
- 3.1. 10 E/A und Dateisystem
- 3.1.11 Module

3.1.1 Geschichte

- In den 50er-Jahren wurden Rechenmaschinen ausschließlich in Maschinensprache programmiert (Single User, Single Job);
  dem Programmierer oblag die vollständige Steuerung des Prozessors (Rechenwerks), d.h. er benötigte detaillierte Hardwarekenntnisse.
  Die Programme waren extrem effizient!
- Anwendungen waren vorwiegend numerischer Art; der Rechner wurde tatsächlich als **Rechner**(Computer) genutzt!
- In der 2. Hälfte der 50er-Jahre entwarf eine Forschungsgruppe bei IBM unter der Leitung von **John Backus** eine “höhere” Programmiersprache, die dem Programmierer auf einfache Weise gestatten sollte, seine mathematischen Formeln in gewohnter Schreibweise zu formulieren. Eine spezielles Übersetzungsprogramm(Compiler) transformierte die Formeln in Maschinensprache. (**For**mula **Tran**slation --> Fortran)

Geschichte

- Die Sprache wurde sehr schnell auch von anderen Forschergruppen und Rechnerherstellern aufgegriffen und - da natürlich alles zu verbessern ist - an vielen verschiedenen Orten unabhängig voneinander erweitert, mit der Folge: Kein Fortran-Programm war maschinenunabhängig (kompatibel). Man sprach von Dialekten der Sprache Fortran.
- 1966 wurde vom American National Standards Institute (ANSI) das erste Mal eine Programmiersprache standardisiert; es war Fortran66. Letztendlich handelte es sich um einen standardisierten Subset einer Anzahl von Dialekten, die die einzelnen Firmen (auch aus einem "Abschottungsbestreben" heraus) nicht daran hinderte, ihre eigenen Dialekte weiter zu entwickeln. Der Begriff "Offenes System" war seinerzeit noch nicht geprägt und wäre den einzelnen Herstellern (IBM, CDC, Siemens, Telefunken, Rank Xerox, Honeywell Bull, RCA?...) zutiefst suspekt gewesen
  Immerhin - wer wollte - der konnte portable Programme schreiben.

Geschichte

- Es gab natürlich auch sinnvolle Erweiterungen; schon aufgrund der Notwendigkeit zur effizienteren Nutzung inzwischen entwickelter neuer Rechensysteme. So wurde1978 eine neuer Standard verabschiedet: Fortran77.
  Es sollte bis Mitte der 80er-Jahre dauern, bis der Übergang von Fortran66 zu Fortran77 vollzogen war - im wesentlichen auch erzwungen, durch die Weigerung der Hersteller, die alten Compiler weiter zu pflegen.
- Aufgrund der inzwischen vielfältigen Entwicklung "modernerer" Programmiersprachen, war schon zur gleichen Zeit klar, daß Fortran weiterentwickelt werden muß. Ziel war es Fortran8x zu entwickeln, später Fortran88 genannt. Als dieser Standard endgültig 1992 verabschiedet wurde, nannte man ihn Fortran90 (ANSI X3.198-1992).
- Fortran90 enthält Fortran77 als Subset; es sind aber bereits jetzt die Sprachkonstrukte als "obsolet" klassifiziert, die bei der nächsten Festlegung eines Standards namens Fortranxx wegfallen sollen. Neue Programmsysteme sollten in jedem Fall nur in Fortran90 formuliert werden.

3.1.2 Basiskonzepte

fortran90/95-Zeichensatz

- 26 Klein/Großbuchstaben des englischen Alphabets; jedoch A = a!!
  10 Ziffern (0-9)
  Unterstreichungsstrich
  Zeichen mit besonderer Bedeutung (Sonderzeichen, special character)

Sonderzeichen:	blank	: colon	= equals sign
+ plus sign	- minus sign	* asterisk	/ slash
(left parenthesis	) right parenthesis	, comma	. decimal point
’ apostrophe	! exclamation mark	" quotation mark	% percent
& ampersand	; semicolon	< less than	> greater than
? question mark	$ currency symbol		

- Token
  - Zeichenfolgen, die durch Trennzeichen getrennt sind

Basiskonzepte

Quellcode-Layout

- spaltengerechte Schreibweise (Lochkarte) Kompatibilitätsmode f77 (wird im Rahmen dieser Vorlesung nicht weiter behandelt)
- spaltenunabhängiges Quellprogramm f90
  - max. 132 Zeichen pro Zeile; bis zu 39 Fortsetzungszeilen möglich
  - normalerweise: 1 Zeile - 1 Statement
  - ’blank’ ist Separatorzeichen; beliebige Anzahl --> Lesbarkeit des Codes
  - "!": Kommentar bis zum Zeilenende; beliebige Zeichenfolgen
  - "&": Fortsetzungszeilen; sollten der besseren Lesbarkeit wegen ausgerichtet sein

Basiskonzepte

Typkonzept

Ein Datentyp ist durch das Tupel (Wertemenge; Operationen) definiert.

- vordefinierte Datentypen
  - numerische Datentypen
    - ganzzahlig (INTEGER)
    - reell (REAL)
    - reell, doppelt genau (DOUBLE PRECISION)
    - komplex (COMPLEX)
  - nichtnumerische Datentypen
    - logisch (LOGICAL)
    - Zeichen (CHARACTER)
  - Alle vordefinierten Datentypen sind parametrisierbar, d.h. man kann ihren Wertebereich spezifizieren .

Basiskonzepte

Konstanten

- Konstante sind Größen, die zur Laufzeit unveränderbar sind; deshalb stehen sie nie auf der linken Seite einer Zuweisung
- Eine Konstante hat einen Typ und einen Wert, ggf. einen Typparameter oder eine Zeichendatenlänge
- Namenskonstane: Konstante mit Namen, sie werden explizit mit einer PARAMETER-Anweisung vereinbart.
- literale Konstante: auch konstruierte Konstante; der Typ geht implizit aus der Schreibweise hervor

Basiskonzepte

Variable

- Einer Variablen ist ein Speicherplatz zugeordnet, dessen Länge
  - vom Typ der Variablen und
  - von der Realisierung des Typs auf der konkreten Maschine (Standardisierung; kompatible Programmierung)

  abhängig ist.
- Der Name der Variablen ist eigentlich die symbolische Bezeichnung einer Adresse. Steht der Variablenname auf der linken Seite einer Zuweisung "meint" man die Adresse der Variabalen (dem zugeordneten Speicherplatz soll ein Wert zugewiesen werden). Steht der Variablenname auf der rechten Seite einer Zuweisung so "meint" man den Wert der Variablen.
  (Lies: "=" als "ergibt sich aus")
- In Fortran90 ist der direkte Zugriff auf die Adresse einer Variablen (und Adreßarithmetik) nicht möglich - im Gegensatz zu "C".

Basiskonzepte

Variable(cont)

- Man unterscheidet Variable
  - mit eigenen Namen
    - skalare Variable
    - Feldvariable
  - ohne eigenen Namen; Teilobjekte, wie
    - Feldelemente
    - Strukturelemente
- Eine Variable kann man als Tupel auffassen:
  - (Typ, Wertebereich,Gültigkeits-(Geltungs-)bereich, Lebensdauer)

© Claus-Uwe Linster – pps3.1-1_spmd_f90: 16/3/98

Basiskonzepte

Variable(cont)

- In Fortran gibt es (leider) eine große Anzahl von impliziten Voreinstellungen, gerade auch Datentypen betreffend. So sind die Variablen deren Namen mit den Buchstaben i, j, k, l, m, n beginnen vom Typ INTEGER, alle anderen vom Typ REAL es sei denn:
  - diese impliziten Vereinbarungen sind mit der IMPLICIT-Anweisung redefiniert worden oder
  - die Variable selbst wurde explizit redefiniert.
- Die Einhaltung der Regeln in größeren Programmsystemen zu durchschauen, insbesondere dann, wenn man sie nicht selbst geschrieben hat, ist ausgesprochen schwierig. Spätere Änderungen durch Dritte haben häufig Fehler zur Folge, die ausgesprochen schwer zu lokalisieren sind.

Deshalb: Im Rahmen dieses Kurses ist das 2. Statement eines Programms: "IMPLICIT NONE". Hierdurch werden explizite Typvereinbarungen erzwungen.

Basiskonzepte

INTEGER

- Ganzahliger Datentyp (INTEGER)
  - Operationen: `+ - * / **`
  - Wertebereich: abhängig von der Wortbreite der Maschine -2^{n-1} bis $+ 2^{n-1} -1$; z. B. bei 16 Bit Wortbreite: -32768 bis 32767
  - Schreibweise der literalen Konstanten:

```
decimal:                1
                        0
                        -199
                        32612
binary (base 2):        B'01101111'
octal (base 8):         O'07631'
hexadecimal (base 16):  Z'10BC'
```

  - Typvereinbarung von skalaren Variablen

```
INTEGER a, index, paula
```

Basiskonzepte

- Reeller und doppeltgenauer Datentyp
(REAL; DOUBLE PRECISION)
  - Operationen: + - * / **
  - Wertebereich: abhängig von der Wortbreite der Maschine
heute üblich: REAL 32 Bit; DOUBLE PRECISION 64 Bit
Zahlendarstellung:
Koeffizient: Dezimalzahl der Form d. d.d oder .d; d Ziffernfolge
Exponent: ganzahliger Exponent Basis 10

Schreibweise der literalen Konstanten:

```
7.5       35.E1      (=350.)
-5.34     .0016E+4   (= 16.)
+1832.    2443.E2    (= 244300)
.3        50.E-2     (= .5)
```

- Typvereinbarung von skalaren Variablen

```
REAL a, leistung, paula
DOUBLE PRECISION b, ertrag, skonto
```


Basiskonzepte

- Complexer Datentyp (COMPLEX)
  - Operationen: + - * / **
  - Zahlen des Typs COMPLEX sind Paare (real, imaginär). Beide Komponenten können vom Typ INTEGER, REAL oder DOUBLE PRECISION sein, wobei die Komponten von verschiedenem Datentyp sein dürfen.

Schreibweise der literalen Konstanten:

```
(1.,3.2)
(4, 10.3E-10)
(7.9,25)
```

- Typvereinbarung von skalaren Variablen

```
COMPLEX a, leistung, paula
```

Basiskonzepte

Zeichendatentyp (CHARCTER)

- Operationen: Verkettung `//`
- Wertebereich: Zeichenfolgen aus der Menge der darstellbaren Zeichen; z. B. ASCII-Zeichensatz (Tabelle Seite 356 RRZN-Nachschlagewerk) Eine Zeichenfolge hat eine Länge; die Länge Null ist zulässig

Schreibweise der literalen Konstanten:

```
'STRING'
'string' verschieden von 'STRING'!!!!!
'Hello world'
```

`"This string contains a ' !"` hat den Wert: This string contains a ' !

`'Isn''t it a nice day?'` hat den Wert: Isn't it a nice day?

- Durch die Verschlüsselung der Zeichen ist eine Reihenfolgerelation definiert der Art:
  0 < 1 < 2 .....A < B < C ......a < b < c... (Siehe ASCII-Tabelle)
- Typvereinbarung von Variablen

```
CHARACTER *5 poem, gruss, char!(alle Variablen enth. 5 char)
```


Basiskonzepte

Logischer Datentyp (LOGICAL)

- Operationen: `.NOT. .AND. .OR. .EQV. .NEQV.`
- Wertebereich: true, false

Schreibweise der literalen Konstanten:

```
.FALSE.
.TRUE.
```

- Typvereinbarung von Variablen

```
LOGICAL epsilon, bedingung
```

Basiskonzepte

Namen

- Namen bestehen aus 1 bis 31 alphanumerischen Zeichen, der Unterstreichungsstrich ist eingeschlossen. Sie müssen mit einem Buchstaben beginnen und dürfen keine *blanks* enthalten. **Es gibt keine geschützten Schlüsselworte.**

Gültige Namen:

```
b
a327
masse
abfahrts_zeit
```

```
Ungültige Namen:

1a        beginnt mit einer Ziffer
a source  enthält ein blank
$help     enthält ein Sonderzeichen
```

- **Vereinbarung:** Schlüsselworte werden groß geschrieben, alles andere klein; trotzdem keine Schlüsselworte als Variablennamen verwenden! Namen sollten signifikant bezüglich ihrer Bedeutung sein.

Basiskonzepte

Datenfelder (arrays)

```
Syntax:Typ, DIMENSION(dim[,dim]...):: varname [..]
```

- eindimensionale Felder (Vektoren)

  Beispiel:

  ```
  REAL, DIMENSION(10) :: array
  ```

  Ein Vektor der Länge 10 mit skalaren Elementen des Typs REAL.
  Die Untergrenze der Indizes ist voreingestellt (default) 1.
  Die Einzelelemente: **`array(1) array(2) .... array(10)`**

  ```
  INTEGER DIMENSION(-5:10) :: vek
  ```

  Ein Vektor der Länge 16: lower bound: -5; upper bound: 10
  Die Einzelelemente: **`vek(-5)...vek(0) vek(1)....vek(10)`**

- mehrdimensionale Felder

  Beispiel:

  ```
  REAL, DIMENSION(5,10) :: matrix
  DOUBLE PRECISION, DIMENSION(-10:5,-5,10,0:16) :: grid
  ```

© Claus-Uwe Linster – pps3.1-1_spmd_f90: 16/3/98

Basiskonzepte

Ausdrücke und Zuweisungen (einfache)

- Ein Ausdruck ist im allgemeinen eine Formel zur Berechnung eines Wertes. Er besteht aus Operanden, Operatoren und/oder Klammern.
  - Beispiele:

```
-a + b/z
(a**b) + (a*b)
-a+b+c
17 + 4 <= carddeck          (Vergleichsausdruck)
x .AND. y                   (logischer Ausdruck)
'Eisen' // 'bahn'           (Zeichenausdruck)
```

- Ausdrücke werden bei gleicher Wertigkeit (precedence) von links nach rechts ausgewertet. Durch Setzten von Klammern kann man die Reihenfolge steuern. Zur Wertigkeit der Operatoren: siehe "Precedence Table".
  - Beispiel: `-a+b+c` wird wie folgt ausgewertet: `((-a)+b)+c`

Basiskonzepte

skalare Zuweisung (Assignment)

- skalare Zuweisung (Assignment)

`Syntax:variable = expr`

Beispiel: a = 25.5 * 3.1E9

- Eine Variable ist erst dann definiert, wenn ihr ein Wert zugewiesen wird. Bei der Typvereinbarung der Variablen wird keine automatische Initialisierung vorgenommen; man kann sie jedoch bei der Vereinbarung explizit initialisieren.
  Beispiel:

```
INTEGER :: index1=0, index2=0
```

Arithmetik mit Daten vom Typ INTEGER

Bei der Integerdivision wird der Bruchanteil immer abgeschnitten
Beispiele:

```
9 / 3 ist 3          2**3 ist 8
11 / 3 ist 3         2**(-3) ist 1 / (2**3) also = 0!
-11 / 3 ist -3
```

Basiskonzepte

Anweisungsmarke und Assign-Anweisung

- Anweisungsmarke (label) und Assign-Anweisung
  - Programmzeilen können mit Marken der Form

```
Syntax:zzzz        statement; zzzz: ganze positive Zahl
```

identifiziert werden.

Beispiel: `1140       x = y + 1.0`

- Die Assign-Anweisung weist einer Variablen vom Typ INTEGER eine Anweisungsmarke zu.

```
Syntax:ASSIGN label TO v_INTEGER
```

Beispiel:

```
   INTEGER abbruch, errorcode
      ASSIGN 2000 TO abbruch
      ...
      GO TO abbruch
      ...
2000  errorcode = 100
      ...
```


Basiskonzepte

Programmsteuerungsanweisungen (Control Statements)

- Normalerweise werden die Anweisungen eines Programms Zeile für Zeile ausgeführt. Mit Programmsteueranweisung kann man den Kontrollfluß ändern, unterbrechen oder beenden.

GO TO statement

```
Syntax:GO TO label
```

- In FORTRAN90 hat der "unbedingte Sprung" an Bedeutung verloren und sollte im wahrsten Sinne des Wortes nur zur Ausnahmebehandlung (exception handling) verwendet werden. Unnötiger Einsatz von GO TOs führt in der Regel zu unstrukturiertem Programmcode.

IF statement

```
Syntax:IF (scalar-logical-expr) action-stmt
```

Beipiel:

```
IF (flag) index = 0
IF (x-y <= schranke) GO TO ende
```

Basiskonzepte

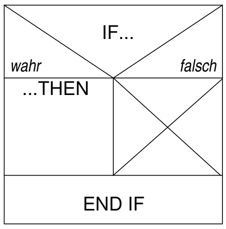
- IF construct (einseitig)

```
Syntax:[name:] IF (scalar-logical-expr) THEN
                    block¹
              END IF [name]²
```

Beispiel:

```
swap:IF (x < y) THEN
          temp = x
          x = y
          y = temp
     END IF swap
```

Symbol:



1.block: ein oder mehrere Anweisungen
2.Die Notation [...] bedeutet optional!

Basiskonzepte

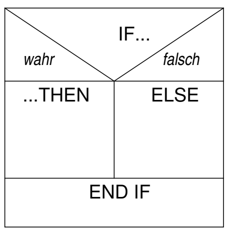
- IF construct (zweiseitig)

```
Syntax:[name:] IF (scalar-logical-expr) THEN
                    block1
              ELSE [name]
                    block2
              END IF [name]
```

Beispiel:

```
IF (x < y) THEN
   x = -x
ELSE
   y = -y
END IF
```

Symbol:

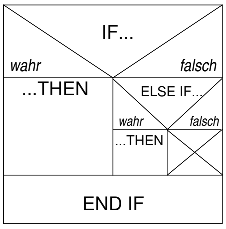


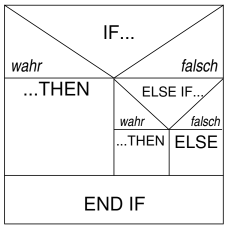
Basiskonzepte

▲ IF...ELSE IF constuct

```
Syntax:
[name:] IF (scalar-logical-expr) THEN
             block
 [ELSE IF (scalar-logical-expr) THEN [name]
             block]...[1]
 [ELSE [name]
             block]
 END IF [name]
```

Symbol:





1.... bedeutet: pot. beliebig oft

Basiskonzepte

▲ IF...ELSE IF construct (cont)

- IF constructs dürfen geschachtelt sein. Das Einspringen in einen Block von außenhalb ist nicht gestattet.
- Um die Lesbarkeit des Programm zu erhöhen soll die Blockstrucktur des Codes durch Zeileneinrücken (tabs) deutlich gemacht werden.

Basiskonzepte

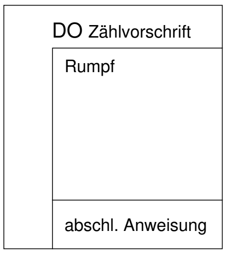
DO construct

- Viele Lösungen mathematischer Probleme werden iterativ durchgeführt. Das DO construct unterstützt diese Lösungsverfahren.

```
Syntax:[name]      DO variable = expr1,expr2 [,expr3]
                            block
                   END DO [name]
```

expr1: Anfangswert
expr2: Endwert
expr3: Schrittweite (default:1)

SYMBOL:



Basiskonzepte

DO construct

Beipiel: Summation von 10 Feldelementen

```
sum = 0
DO i = 1,10
   sum = sum + feld(i)
END DO
```

- Die Anzahl der Iterationen ist gegeben durch die Formel

Max((expr2-expr1+expr3)/expr3,0)

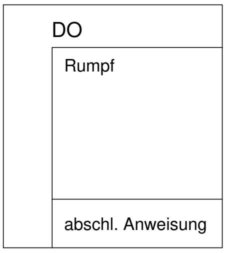
Für die Anweisung DO i = 1, n; n <= 0, wird der Rumpf nicht ausgeführt!

Basiskonzepte

- DO - EXIT construct (Endlosschleife)

```
Syntax:[name:]     DO
               block
               .
               .
               EXIT [name]
               .
             END DO [name]
```

Symbol:



Basiskonzepte

- Welchen Wert hat die Indexvariable i nach dem verlassen einer DO-Loop?

```
DO i = 1, n, m
   .
   .
   If (i == j) EXIT
   .
   .
END DO
k = i
```

n <= 0;   k = 1
i = j;   k = j
0 < n < j; k = n + m

- Ein Einsprung in den Rumpf eines DO constructs ist nicht erlaubt.

Basiskonzepte

▲ DO construct(cont)

- DO constructs dürfen geschachtelt sein
  Beispiel:

```
! Matrixmultiplikation

DO i = 1, n
  DO j = 1, m

     a(i,j) = 0.0
     DO l = 1, k

        a(i,j) = a(i,j) + b(i,l) * c(l,j)
     END DO
  END DO
END DO
```

▲ CYCLE-Statement

```
Syntax:CYCLE [name]
```

- Mit dem CYCLE-Statement wird zum Ende der DO-Schleife gesprungen und mit der nächsten Iteration begonnen

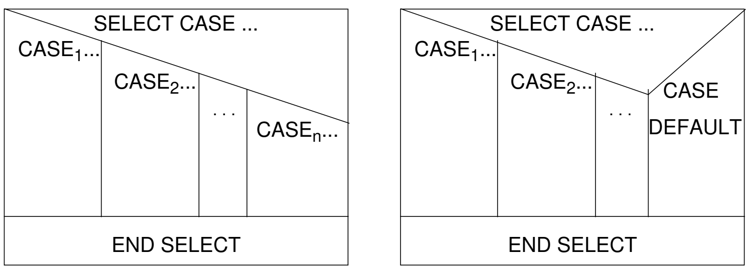
Basiskonzepte

■ CASE construct

```
Syntax:[name:] SELECT CASE (expr)
                 [CASE selctor [name]
                       block]...
                END SELECT [name]
```

- `expr:` Scalar vom Typ INTEGER, CHARACTER oder LOGICAL
- `selector`: Wert muß vom gleichen Typ wie `expr` sein
  für den Typ INTEGER UND CHARACTER kann ein Interval spezifiziert werden. z. B.: `CASE (low:high)`

Symbol:



Basiskonzepte

▲ CASE construct(cont)

– Beispiel

```
SELECT CASE (number)! number vom Typ INTEGER
  CASE (:-1)    ! alle Werte < 0
     n_sign = -1
  CASE (0)      ! nur die 0
     n_sign = 0
  CASE (1:)     !alle Werte >0
     n_sign = 1
END SELECT
```

- Sind die Selektoren so gewählt, das nur eine Teilmenge der Möglichkeiten abgedeckt ist, so wird mit dem nächsten Statement nach der select-Anweisung fortgefahren, wenn kein Selektor die Bedingung erfüllt, es sei denn, man verwendet den Selktor “DEFAULT”.

Basiskonzepte

▲ CASE construct(cont)(

- Die generelle Form des Selektors ist eine Liste von sich nicht überlappenden Werten oder Intervallen

  `CASE (1, 2. 7, 10:25, 50)`

- Die Form

  ```
  CASE DEFAULT
  ```

  ist die Restmenge aus dem Wertebereich minus der Werte in den Selektoren. Beispiel:

  ```
  SELECT CASE (ch)        ! ch vom Typ CHRACTER
     CASE ('a','d','r':)
        ch_type = .TRUE.
     CASE ('i' : 'n')
        int_type = .TRUE.
     CASE DEFAULT
        real_type = .TRUE.
  END SELECT
  ```

Basiskonzepte

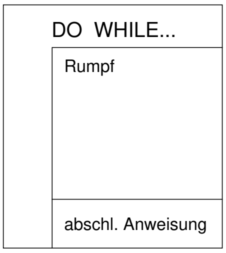
- DO...WHILE

```
Syntax:[name]DO[label] [,] WHILE(scal.-logical-expr)
 ...
 END DO
```

ist equivalent zu

```
[name]  DO
           IF (.NOT.scalar-logical-expr) EXIT
           ....
        END DO [name]
```

Symbol:



Basiskonzepte

- DO...WHILE(cont)

Beispiel: ...

```
DO WHILE (i > 0)! i,j: INTEGER
   j = i        !Vorsicht erst mit 1000er-
   i = i + 1    !Schritten testen!
END DO
PRINT *,'Die grösste ganze Zahl ist:',j
Print *,'Die kleinste ganze Zahl ist:',i
.....
```

Basiskonzepte

▲ Einfache Ein/Ausgabe (listengesteuert, list-directed)

Wird normalerweise zur Ein/Ausgabe kleiner Datenmengen, etwa für Testaus/eingaben verwendet.

- Standard E/A-Unit ist das Terminal
  Es ist nicht notwendig das Gerät (unit) explizit zu definieren.
- PRINT * steuert die Ausgabe, wobei das richtige Format automatisch zur Laufzeit in Abhängigkeit vom Datentyp bestimmt wird.
  - Ausgabeliste: Variable; Ausdrücke (Konstanten)
  - Beispiel

```
REAL x, erg

...

PRINT *, 'Squareroot of ',x, ' = ' erg
...


Ausgabe z. B.: Squareroot of 9 = 3
```


Basiskonzepte

▲ Einfache Ein/Ausgabe (list-directed)(cont)

- READ * steuert die Eingabe, wobei das richtige Format automatisch zur Laufzeit in Abhängigkeit vom Datentyp bestimmt wird.
  - Eingabeliste: nur Variable!
  - Eingabetrennzeichen: <b> , /

```
INTEGER i
REAL a
LOGICAL flag
CHARCTER (Len=12) titel
CHARACTER (Len=4) word
....
  READ *,i,a,flag,titel,word
...
```

Eingabe:10 5.60 F 'Hello world' test oder

Eingabe:10,5.6,.FALSE.,,test (hier bliebe titel unverändert)

Basiskonzepte

Programmaufbau

- Hauptprogramm
  Jedes vollständige Programm enthält genau ein Hauptprogramm

```
Syntax:[PROGRAM program-name]
                [specification-statements]
                [executable-statements]
       [CONTAINS
                internal-subprograms]
       END [PROGRAM [program-name]]
```

"PROGRAM program-name" sind zwar optional, es ist aber guter Programmierstil, sie zu verwenden!

Basiskonzepte

Programmaufbau(cont)

- Hauptprogramm (main program)
  - internal subprograms
- Externe Unterprogramme (external subprogram)
  - internal subprograms
- Module
  - module subprograms
    - internal subprograms
- subprograms
  - subroutine
  - function

Vertiefung: Siehe Kapitel 3.1.6

Basiskonzepte

- Zusammenfassung
- Quellcode
  - Zeichensatz, Zeile, Fortsetzungszeile, ’A’ gleich ’a’,
  - formatfreies Quellprogramm
  - Token, Statement, Separatoren (Trennzeichen), Sonderzeichen, Schlüsselworte
- Datenstrukturen
  - Tykonzept; Typen: vordefinierte, benutzerdefinierte, numerisch, nicht numerisch
  - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL
  - Wertebereich, Genauigkeit, Gültigkeitsbereich, Lebensdauer
  - Namen, Konstante, Variable, Felder, DIMENSION, Rang, Vektor, Matrix

Basiskonzepte

- Programmstrukturen
  - Namen, Ausdrücke, Zuweisungen, Label
  - GOTO, ASSIGN
  - IF-statement; IF THEN; IF THEN ELSE; IF THEN ELSE IF..... ELSE; END IF;
  - SELECT CASE; END SELECT
  - DO i = anfang, ende, inkrement; CYCLE; EXIT; DO WHILE; END DO
  - PRINT *; READ *; PROGRAM; END PROGRAM

3.1.3 Typvereinbarungen/Spezifikationen

- Vereinbarungen und Spezifikationen
  - Deklarations- und Spezifikations-Anweisungen sind nicht ausführbare Anweisungen
  - Zum Sprachgebrauch:
    - Datentypen werden ***definiert*;** damit wird ihre Struktur festgelegt Vordefinierte Typen (INTEGER, REAL,..) können nicht explizit definiert werden.
    - Variablen oder Namenskonstanten werden ***vereinbart*** (oder ***deklariert***; eng. declared). Der Typ kann explizit mit einer Typvereinbarungsanweisung vereinbart werden (oder implizit vorgegeben sein)
    - Wir werden im Rahmen dieses Kurses die explizite Form wählen. Für literale Konstanten ist allerdings nur die implizite Methode möglich.

Typvereinbarungen/Spezifikationen

- Zum Sprachgebrauch (cont):
  - Die Eigenschaften einer Variablen oder Namenskonstanten, also ihre Attribute können
    - mit der Typvereinbarung zusammen oder
    - durch Spezifikations-Anweisungen gesondert

    spezifiziert werden.
  - Variablen sind ***definiert***, wenn ihnen ein Wert zugewiesen worden ist, sonst sind sie undefiniert (und dürfen nicht auf der rechten Seite einer Zuweisung auftreten).

▲ In einfachen Fällen sollte die *Spezifikation* zusammen mit der Typvereinbarung vorgenommen werden.
Bevorzugt man die getrennte Schreibweise von Typ-Vereinbarung und Attribut-Spezifikation, dann sollte die Spezifikations-Anweisung unmittelbar der Typvereinbarungsanweisung folgen!

Vereinbarungen und Spezifikationen

- Typvereinbarungsanweisungen
  - Typvereinbarungsanweisungen; allgemeine Form

```
typ [(typparameter | typname)][,attribut]...[::]
datengröße [,datengröße]...
```

  - Dabei ist **TYP** eine der folgenden Typspezifikationen
    - INTEGER
    - REAL
    - DOUBLE PRECISION
    - COMPLEX
    - LOGICAL
    - CHARACTER
    - TYPE !selbstdefinierter Typ (Siehe Kapitel 3.1.4)

Vereinbarungen und Spezifikationen

- Typvereinbarungsanweisungen(cont)
  - typparameter | typname
    - Mit **typparameter** wird der Wertebereich des Typs spezifiziert
      - **[KIND =] kind; kind:** gleich ganzzahliger skalarer Initialisierungsausdruck
      - die zulässigen Typparameterwerte sind systemabhängig

    Im Rahmen dieses Kurses werden wir ausschließlich die “Normal”-formen (keine Angabe) verwenden, wobei davon ausgegangen werden kann, daß bei Rechnern gleicher Klasse (z. B. 32-Bit-Architektur) die Normalformen gleich sind. Wenn man gezwungen ist Typparameter zu verwenden, dann in der symbolischen Form wie ’BYTE’; ’INT32’; ’INT64’;..

      - **typname** ist die Bezeichnung eines selbstdefinierten Datentyps; also nur bei der Typvereinbarung **TYPE** anzuwenden (Siehe Kapitel 3.1.4).

Vereinbarungen und Spezifikationen

■ Attribute

▲ Typvereinbarungsanweisungen; allgemeine Form:

```
typ [(typparameter | typname)][,attribut]..[::]&
datengröße [,datengröße]...
```

- mit dem `attribut` werden die Datentypen näher spezifiziert und zwar:

```
– ALLOCATABLE              ! für dynamische Felder
                           ! (Kapitel 3.1.8: Dyn. Speicherv.)
– DATA(...)                ! Initialisierung von Variablen;
                           ! tritt nicht direkt als
                           ! 'Attribut' auf.
– DIMENSION(...)           ! als vorangestellte
                           ! Feldspezifikation
– EXTERNAL                 ! für Namen externer Funktionen
                           ! (Kapitel 3.1.11: Module..)
– INTENT (IN | OUT | INOUT)! für Formalparameter
                           ! (Kapitel 3.1.6: Unterprogramme)
– INTRINSIC                ! für Namen vordef.Funktionen
                           ! (Kapitel n.b.: Bibliotheken..)
```


Vereinbarungen und Spezifikationen

■ Attribute(cont)

▲ Typvereinbarungsanweisungen; allgemeine Form:

```
typ [(typparameter | typname)][,attribut]...[::]
datengröße [,datengröße]...
```

– mit dem `attribut` werden die Datentypen näher spezifiziert und zwar:

```
– OPTIONAL                 ! für optionale Formalparameter
                           ! (Kapitel 3.1.6: Unterprogramme)
– PARAMETER                ! für Namenkonstanten
– POINTER                  ! für Zeiger (Kapitel 3.1.5:
                             Zeiger)
– PRIVATE                  ! für unsichtb. Größen im Modul
                           ! (Kapitel 3.1.11: Module..)
– PUBLIC                   ! für sichtb. Größen im Modul
                           ! (Kapitel 3.1.11: Module..)
– SAVE                     ! für gesicherte lokale Größen
                           ! in Unterprogrammen
                           ! (Kapitel 3.1.6: Unterprogramme)
– TARGET                   ! für Ziele von Pointern
                           ! (Kapitel 3.1.5: Zeiger)
```

Vereinbarungen und Spezifikationen

- Attribute(cont)

▲ Typvereinbarungsanweisungen; allgemeine Form:

`typ [(typparameter | typname)][,attribut]...[::]`
`datengröße [,datengröße]...`

- `'::'` darf fehlen, wenn kein Attribut vorhanden ist und keine Initialisierung mit `'='` erfolgt.
- die **`datengrößen`** haben folgende allgemeine Form:

`name [(dim [,dim]...)] [*elänge] [= initialisierungsausdruck]`

  - name = Name der Variablen
  - dim = spezifische Angabe einer Feldspezifikation; “eigene” Feldspezifikation “überschreibt”, falls vorhanden, eine “vorangestellte” **`DIMEMSION`**-Spezifikation für diese Variable;
  - **`*elänge`** nur in Verbindung mit der **`CHARACTER`**-Vereinbarung
  - **`= initialisierungsausdruck:`** der Wert der Variablen wird mit dem Wert des Ausdrucks initialisiert

Vereinbarungen und Spezifikationen

- Typvereinbarungsanweisungen
  - Implizite Typvereinbarungsanweisungen (aus Kapitel 3.1.2): In Fortran gibt es (leider) eine große Anzahl von impliziten Voreinstellungen, gerade auch Datentypen betreffend. So sind die Variablen deren Namen mit den Buchstaben i, j, k, l, m, n beginnen vom Typ INTEGER, alle anderen vom Typ REAL es sei denn:
    - diese impliziten Vereinbarungen sind mit der IMPLICIT-Anweisung redefiniert worden oder
    - die Variable selbst wurde explizit redefiniert.
  - Die Einhaltung der Regeln in größeren Programmsystemen zu durchschauen, insbesondere dann, wenn man sie nicht selbst geschrieben hat, ist ausgesprochen schwierig. Spätere Änderungen durch Dritte haben häufig Fehler zur Folge, die ausgesprochen schwer zu lokalisieren sind.
  - Deshalb: Im Rahmen dieses Kurses ist das 2. Statement eines Programms: “IMPLICIT NONE”. Hierdurch werden explizite Typvereinbarungen erzwungen.

Vereinbarungen und Spezifikationen

- Vereinbarung von Skalarer Variablen
  - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL

    Beispiele (Siehe auch Kapitel 1.2):

```
INTEGER index1, nn, number
REAL einkommen, ausgaben, summe
DOUBLE PRECISION skonto, rabatt, verlust
COMPLEX leistung
LOGICAL flag, error, panic
```


Vereinbarungen und Spezifikationen

- Feldvereinbarungen
- INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL

```
Syntax:Typ, DIMENSION(dim[,dim]...):: varname [..]
```

- DIMENSION ist ein vorangestelltes Attribut (Spezifikation), die durch eine zugeordnete (eigene) Spezifikation überschrieben wird. Im Rahmen dieses Kurses soll die Mischung beider Schreibweisen in einem Statement vermieden werden.
- eindimensionale Felder (Vektoren)

  Beispiel:

```
REAL, DIMENSION(10) :: array
! oder
REAL array(10)
```

Ein Vektor der Länge 10 mit skalaren Elementen des Typs REAL.
Die Untergrenze der Indizes ist voreingestellt (default) 1.
Die Einzelelemente: **`array(1) array(2) .... array(10)`**

Vereinbarungen und Spezifikationen

- Feldvereinbarungen
- INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL (cont)

```
Syntax:Typ, DIMENSION(dim[,dim]...):: varname [..]
```

- eindimensionale Felder (Vektoren), Beispiel:

```
INTEGER DIMENSION(-5:10) :: vek
! oder
INTEGER vek(-5:10)
```

Ein Vektor der Länge 16: lower bound: -5; upper bound: 10

Die Einzelelemente: **vek(-5)...vek(0) vek(1)....vek(10)**

- mehrdimensionale Felder, Beispiel:

```
REAL, DIMENSION(5,10) :: matrix
! oder
REAL matrix(5,10)
DOUBLE PRECISION, DIMENSION(-10:5,-5,10,0:15) :: grid
! oder
DOUBLE PRECISION grid(-10:5,-5,10,0:15)
```


Vereinbarungen und Spezifikationen

- CHARACTER Variable
  - Die Typvereinbarungs-Anweisung der Character Variable unterscheidet sich hinsichtlich des “zeichentypparameters” von den sonstigen Typvereinbarungs-Anweisungen.

```
Syntax:typ [zeichentypparameter][,attribut]...[::]&
   datengröße [,datengröße]...
```

  - Der Zeichentypparameter bestimmt die Größe (in BYTES) des, der Variablen zugeordneten, Speichers; was der Länge des abzuspeichernden “charcterstrings” entspricht.

Vereinbarungen und Spezifikationen

- CHARACTER Variable
  - Man unterscheidet vorangestellte Längenangaben

    `(LEN = vlänge) oder *vlänge`

  oder variablenspezifische (eigene) Längenangaben

    `*elänge`

  die die vorangestellte Längenangabe “überschreibt”.

  - Tritt als Längenangabe nur ein *, (LEN=*) oder *(*) auf, so handelt es sich um eine “übernommene” Länge, z. B. bei Namenskonstanten (siehe dieses Kapitel: PARAMETER-Attribut) oder Formalparameter (siehe Kapitel 3.1.6: Unterprogramme).

Vereinbarungen und Spezifikationen

- CHARACTER Variable
  - Beispiele:

```
CHARACTER(LEN=5) person5, person10 *10, vorname
CHARACTER *10, DIMENSION (10,10) ::text1, text2, text3 *4
! oder (besser)
CHARACTER *10, DIMENSION (10,10) ::text1, text2
CHARACTER *4, DIMENSION (10,10) ::text4
! oder
CHARACTER *10,text1(10,10), text2(10,10)
CHARACTER *4, text3(10,10)
```

  - Im Rahmen dieses Kurses soll die Mischung von vorangestellten Längenangaben und eigenen Längenangaben in einem Statement vermieden werden.

Vereinbarungen und Spezifikationen

- Initialisierung
  - Initialisierung von Variablen
    - Wenn einem Namen ein Gleichheitszeichen folgt, wird entweder eine Namenkonstante definiert (Attribut **`PARAMETER`**) oder es wird eine Variable initialisiert; sie erhält einen Anfangswert.
    - Folgende Größen dürfen nicht Initialisiert werden (Sämtlich Gegenstand späterer Kapitel).
      - Formalparameter
      - Funktionswerte
      - Variable im COMMON-Block; es sei denn es handelt sich um einen benamten COMMON-Block in einem BLOCK DATA
      - dynamische Felder
      - Zeiger (Pointer)
      - automatische Variable
    - Für den **`initialisierungsausdruck`** gelten die gleichen Regeln wie für Ausdrücke allgemein.

Vereinbarungen und Spezifikationen

- Initialisierung
  - Mit der Typvereinbarung
    - zu bevorzugen, bei überschaubaren Strukturen
  - Mit der DATA-Anweisung
    - zu bevorzugen bei komplexeren Strukturen; jedoch in unmittelbarer Nähe der Typvereinbarung
  - Mit der Typvereinbarung

```
Syntax:skalar_varname = ausdruck |
      feld_varname = (/werteliste/)
```

Vereinbarungen und Spezifikationen

- Initialisierung
  - skalare Variable:

```
Syntax:skalar_varname = ausdruck
```

Beipiele:

```
INTEGER :: index = 0, anfangswert = 2**2 + 1
CHARACTER (LEN=3) :: der = 'der', die = 'die'
REAL :: skonto = 3.5
LOGICAL :: flag = .TRUE.
```

  - Feldvariable:

```
Syntax:feld_varname = (/werteliste/)
       werteliste = (/ausdruck[, ausdruck] ... /)
```

Beispiele:

```
INTEGER, DIMENSION (10):: feld1 = (/1,2,3,4,5,6,7,8,9,10/)
REAL, DIMENSION (3) :: rand1 = (/1000.,100.,10./)
CHARACTER *5, DIMENSION(3) :: person = &
                              (/'meier','meyer',maiyr'/)
```


Vereinbarungen und Spezifikationen

- Initialisierung
  - Feldvariable (cont):
    - implizite Schleifen

```
Syntax:feld_varname = (/werteliste/)
       werteliste = (wirkungsbereich, laufvar =
       anfpar,endpar,[inkrementpar])(analog DO-Loop)
```

Beispiele mit Werteliste:

```
INTEGER, DIMENSION (10):: feld1 = (/1,2,3,4,5,6,7,8,9,10/
```

mit Wiederholungsfaktor; k*:

```
INTEGER, DIMENSION (15) :: feld2 = (/15 * 33/)
```

mit impliziter Schleife:

```
INTEGER, DIMENSION (10) :: feld1 = (/ (i, i = 1,10)/)
```

Kontrollausdruck für dieses Feld:

```
PRINT *, 'feld1(i):', (feld1(i), i = 1,10)
REAL, DIMENSION (28,100) :: x = (/(0.0, i = 1,28, j=1,100)/)
```

Vereinbarungen und Spezifikationen

- Initialisierung
  - Mit der DATA-Anweisung
    - skalare Variable:

```
Syntax:DATA varlist /konstlist/[,varlist /konstlist/]..
```

Beispiele:

```
INTEGER index1, index2, index3
DATA index1, index2, index3 /1, 100, 1000/
REAL skonto, rabatt
DATA skonto, rabatt /3.0, 10.0/
```

obige Typvereinbarungen vorausgesetzt, sind auch folgende Schreibweisen zulässig:

```
DATA index1, index2, index3, skonto, rabatt /1,100,1000,3.0,10.0/
oder:
DATA skonto, rabatt /3.0, 10.0/,index1, index2, index3 /1,100,1000/
```

Empfehlung: Typmischungen vermeiden

Vereinbarungen und Spezifikationen

- Initialisierung
  - Mit der DATA-Anweisung

```
Syntax:DATA varlist /konstlist/[,varlist /konstlist/]..
```

  - Feldvariable:

Beispiele:

```
INTEGER, DIMENSION (10):: feld1
DATA feld1 /1,2,3,4,5,6,7,8,9,10/
REAL, DIMENSION (3) :: rand1, rand2
DATA rand1, rand2 /1000.,100.,10., 10., 100., 1000./
CHARACTER *5, DIMENSION(3) :: person
DATA person /'meier','meyer',maiyr'/
```

auch Mischform möglich:

```
DATA person /'meier','meyer',maiyr'/, &
                         feld1 /1,2,3,4,5,6,7,8,9,10/
```

© Claus-Uwe Linster – pps3.1-1_spmd_f90: 16/3/98

Vereinbarungen und Spezifikationen

- Initialisierung
  - Mit der DATA-Anweisung

```
Syntax:DATA varlist /konstlist/[,varlist /konstlist/]..
```

  - Feldvariable:

mit Wiederholungsfaktor; k*

```
INTEGER, DIMENSION (10):: feld1
DATA feld1 /10*0/
REAL epsilon1, epsilon2, epsilon3
DATA epsilon1 epsilon2 epsilon3 /3*1.0E-6)
```


Vereinbarungen und Spezifikationen

- Initialisierung
  - Feldvariable (cont):
    - implizite Schleifen

```
Syntax:(wirkungsbereich, laufvar = anfpar, endpar
     ,[inkrementpar])(analog DO-Loop)
```

Beispiele mit impliziter Schleife:

```
INTEGER, DIMENSION (10) :: feld1
DATA (feld(i), i = 1,10) /10 * 3/
REAL, DIMENSION (28,100) :: x
DATA ((x(i,j), i = 1,28),j = 1,100) /2800 * 0.0/)

Character *2, DIMENSION (300) :: alpha
DATA (alpha(i), i = 1,100) /100 * 'aa'/ (alpha(i), i= &
101,200)/100 * 'bb'/ (alpha (i), i= 201,300) /100 * 'cc'/
```

Vereinbarungen und Spezifikationen

Beispiele mit impliziter Schleife(cont):

```
REAL DIMENSION (3,4) :: feld2
DATA ((feld2(i,j), i = 1,3), j = 1,4) &
     /3 * 2.5, 3 * 5.0, 3 * 7.5, 3 * 10.0/
entspricht:
DATA feld2 /3 * 2.5, 3 * 5.0, 3 * 7.5, 3 * 10.0/
```

- Initialisierung
  - Selbstdefinierte Typen (Siehe Kapitel 1.4)

Vereinbarungen und Spezifikationen

- Namenskonstanten
  - Namenskonstanten werden mit dem Attribut “PARAMETER” spezifiziert, wobei beide Formen (mit der Typvereinbarung und explizit) etwa der gleichen Syntax gehorchen:

```
PARAMETER (name = ausdruck [,name =ausdruck...)
```

oder

```
typ , PARAMETER :: name = ausdruck [,name =ausdruck]...
```

Beispiele:

```
REAL, PARAMETER :: pi = 3.14, epsilon = 1.0E-5
INTEGER, PARAMETER :: max = 1000, min = -1000
INTEGER, DIMENSION (max) :: vektor
INTEGER, DIMENSION (max:min) :: matrix
CHARACTER fmt *(*)
INTEGER n, nn, nnn
PARAMETER (n = 5, nn = n*n, nnn = nn+n, fmt = 'MEYER')
```

Vereinbarungen und Spezifikationen

Namenskonstanten

Beispiele (cont):

```
CHARACTER (LEN=*), PARAMETER :: j = 'Januar', f = 'Februar'
CHARACTER *, PARAMETER :: m = 'Mai', j = 'Juni'

INTEGER n, nn, nnn
PARAMETER (n = 5, nn = n*n, nnn = nn*n)

REAL vektor(n), matrix(n, n), grid(n, n, n)
DATA f /n*0./, matrix /nn*1./, grid nnn*2./
```

Geschickter Einsatz von Namenskonstanten bei der Konstruktion und Spezifikation von Datenstrukturen ermöglicht die zentrale Restrukturierung für den Fall veränderter Anforderungen. (Änderungsfreundlichkeit!)