

Fortran 90 - die neuen Konzepte in Beispielen

```
program test
  ! uses allocatable array of rang 2

  real, dimension(:,:), allocatable :: arM
  integer :: i1, i2, iAllocStatus, iDeallocStatus, n

  read *, n                                ! reads from stdin

  allocate(arM(n,n), stat=iAllocStatus) ! creates dynamic array
  if (iAllocStatus /= 0) then
    print *, "Error: allocate failed: ", iAllocStatus
  else
    do i2=1, n
      do i1=1, n
        arM(i1,i2)=10*i1+i2
        print *, 'arM(',i1,',',i2,') = ', arM(i1,i2)
      end do
    end do
  end if

  ! DO SOMETHING ...

  deallocate(arM,stat=iDeallocStatus)
  if ((iAllocStatus = 0) .and. (iDeallocStatus /= 0)) then
    print *, "deallocate failed: ", iDeallocStatus
    return
  end if

end program test
```

Fortran 90 - die neuen Konzepte in Beispielen

Frank Elsner
Universität Osnabrück
- Rechenzentrum -
Albrechtstraße 28
D-49076 Osnabrück

E-Mail: elsner@dosuni1.bitnet (BITNET)
elsner@titan.rz.uni-osnabrueck.de (INTERNET)

Version: 1.2
Stand: 11/29/96
Datei: L:\DOC\FORTRAN\HB_F90.DOC

Verfügbar im Postscript Format:
aFTP Server <ftp.rz.uni-osnabrueck.de>
Datei pub/reports/rz/f90_tutorial-1.2.ps

Inhaltsverzeichnis

1. Einleitung.....	5
2. Überblick.....	6
2.1. Entwicklungsziele für Fortran 90.....	6
2.2. Wesentliche Neuheiten in Fortran 90.....	6
2.3. Migration von FORTRAN 77 nach Fortran 90.....	7
3. Lexikalische Erweiterungen.....	9
3.1. Zeichensatz und Namen.....	9
3.2. Formatfreie Form des Quelltextes.....	10
3.3. Einfügen von Quelltext.....	11
3.4. Gegenüberstellung.....	12
4. Typvereinbarung.....	13
4.1. IMPLICIT NONE.....	13
4.2. Benutzerdefinierte Datentypen.....	13
4.3. Attribute und Anfangswerte in einer Typvereinbarung.....	15
4.4. Überblick über Attribute.....	15
4.5. KIND Typparameter.....	16
4.6. Ein Beispiel für die Definition von Typparametern.....	17
4.7. Zahlenmodelle.....	17
4.8. Vordefinierte Abfragefunktionen für Modellparameter.....	18
5. Feldverarbeitung.....	19
5.1. Felder und Zeichenketten der Länge 0.....	19
5.2. Felder mit übernommener Gestalt.....	19
5.3. Parametrisierte lokale Felder.....	20
5.4. Dynamische Felder.....	21
5.5. Vordefinierte Abfragefunktionen für Felder.....	22
5.6. Teilfeld-Selektoren.....	23
5.7. Operationen auf Feldern.....	24
5.8. Zeiger-Variablen.....	25
5.9. Zusammenfassung.....	27
6. Kontrollstrukturen.....	28
6.1. DO Anweisung.....	28
6.2. SELECT CASE Anweisung.....	29
7. Prozeduren.....	31
7.1. Attribute für Formalparameter.....	31
7.2. Rekursive Prozeduren.....	33
7.3. Schnittstellenbeschreibung für Prozeduren.....	34
7.4. Generische Prozeduren.....	34
7.5. Benutzerdefinierte Operatoren und Zuweisungen.....	34
7.6. Interne Prozeduren.....	35
8. Module.....	36
8.1. Ersatz der COMMON Anweisung.....	36
8.2. Modul als Prozedurbibliothek.....	37
8.3. Öffentliche und private Daten.....	39

9. Ein-/Ausgabe.....	40
9.1. NAMELIST Anweisung.....	40
9.2. Nicht-vorrückende Ein- und Ausgabe.....	40
10. Beschreibung des NAG Fortran 90 Compilers.....	42
10.1. Einführendes Beispiel.....	42
10.2. Optionen.....	42
10.3. Warnungen und Fehlermeldungen.....	43
10.4. Module	44
10.5. Erweiterungen für make.....	45
10.6. Debugging	46
11. Anhang.....	48
11.1. Veraltete Sprachelemente.....	48
11.2. Schlüsselwörter und Symbole.....	49
11.3. Vordefinierte Prozeduren (Unterprogramme und Funktionen).....	50
11.4. Beispielprogramme.....	51
11.5. Literaturverzeichnis.....	52

1. Einleitung

Diese Einführung in die neuen Konzepte der Programmiersprache **Fortran 90** wendet sich an Programmierer, die bislang mit **FORTRAN 77** gearbeitet haben und sich nun einen Überblick über Fortran 90 verschaffen wollen. Auf bereits in FORTRAN 77 enthaltene Konzepte wird deshalb nicht eingegangen, obwohl sie ebenfalls in Fortran 90 enthalten sind. (Fortran 90 enthält FORTRAN 77 als Teilmenge!)

Die neuen Sprachkonzepte werden anhand einer stichwortartigen Beschreibung schrittweise und mit kurzen erläuternden Beispielen vorgestellt. Im Anschluß zeigt ein kurzes Programm ein vollständiges Beispiel. Bezüglich einer genauen Sprachbeschreibung sei auf die Referenzhandbücher [2] und [3] verwiesen (siehe Abschnitt **Literaturhinweise**).

Folgende typographischen und Namenskonventionen werden verwendet:

- Im Fließtext sind FORTRAN Schlüsselwörter und Funktionen durch **Fettschrift** gekennzeichnet.
- Variablen des gleichen Datentyps sind durch ein **einheitliches Präfix** gekennzeichnet:

```
i = INTEGER      (i1, i2, iResult, ...)  
r = REAL        (r1, r2, rTemp1, ...)  
c = COMPLEX     (c1, c2, cEigenValue, ...)  
ch = CHARACTER  (chMsg1, ...)  
l = LOGICAL     (l1, l2, ...)
```

- Für Felder, Zeiger und Strukturen gibt es ein zusätzliches Präfix:

```
a = array      (ai10, ar, ac, ...)  
v = vector     (vil, vr, ...)  
p = pointer    (pil, pr, ...)  
s = struct     (sPerson, sDate, ...)
```

- In Beispielen und Programmbeispielen wird **freies Eingabeformat** verwendet, FORTRAN Schlüsselwörter werden klein geschrieben.
- Eingaben sind in **Courier** dargestellt, Ausgaben in *Courier*. Das Zeichen \$ symbolisiert die Eingabeaufforderung (*UNIX prompt*).

Beispiel: \$ **cat sample3.f90**

- Vollständige Programme sind zusätzlich durch einen Rahmen hervorgehoben.
- Die erste Dimension eines Feldes wird in Anlehnung an die mathematische Notation für Matrizen als Zeile, die zweite als Spalte bezeichnet. (Felder werden in FORTRAN allerdings in Spalten-Zeilen-Ordnung abgespeichert.)

2. Überblick

In diesem Kapitel werden folgende Themen behandelt:

- Entwicklungsziele für Fortran 90
- Wesentliche Neuheiten in Fortran 90
- Migration von FORTRAN 77 nach Fortran 90

2.1. Entwicklungsziele für Fortran 90

- Die Abwärtskompatibilität zu FORTRAN 77 bleibt erhalten.
- Wesentliche Mängel von FORTRAN 77 werden behoben.
- Die Sprache wird modernisiert.
- Die besondere Eignung für wissenschaftlich-technische Anwendungen wird verbessert.
- Die numerische Portabilität wird verbessert.
- Der Sprachumfang wird nicht zu komplex.

2.2. Wesentliche Neuheiten in Fortran 90

Quelltextformat:

- freies Eingabeformat
- bis zu 31 Zeichen lange Namen
- Kommentare innerhalb einer Zeile

Datentypen:

- benutzerdefinierte Datentypen (Strukturen)
- Typparameter für numerische Datentypen
- Abfragefunktionen zu Eigenschaften numerischer Datentypen
- Zeiger-Variablen

Operatoren und Zuweisungen:

- benutzerdefinierte Operatoren und Zuweisungen
- Operatoren und Zuweisungen für Felder
- überladene Operatoren und Zuweisungen

Dynamische Speicherverwaltung:

- dynamische Felder variabler Größe
- Übergabe von Feldern variabler Größe an Prozeduren

- lokale Felder variabler Größe

Feldverarbeitung:

- Bearbeiten von Feldern als Ganzes
- verbesserte Auswahl von Teilfeldern
- vordefinierte Prozeduren für Felder

Prozeduren:

- Ein- und Ausgabe- sowie optionale Parameter
- Schlüsselwort-Parameter
- Schnittstellenbeschreibung für Prozeduren
- viele neue vordefinierte Prozeduren
- rekursive Prozeduren

Module:

- neue Programmeinheit MODULE als Verbesserung gegenüber COMMON
- integrierte Schnittstellenbeschreibungen
- private und öffentliche Daten
- Bibliotheken in Modulen

Programmsteuerung:

- erweiterte DO Anweisung
- EXIT und CYCLE Anweisung
- neue Anweisung SELECT CASE

Sonstiges:

- Verbesserungen bei der Ein- und Ausgabe
- Kennzeichnung veralteter Sprachelemente durch den Compiler

2.3. Migration von FORTRAN 77 nach Fortran 90

Der Migrationspfad von **FORTRAN 77** nach **Fortran 90** kann über folgende Stufen verlaufen:

- Bisherige FORTRAN 77 Programme und Bibliotheken werden mit einem Fortran 90 Compiler übersetzt und weiterhin verwendet. Die Übersetzung ist möglich, da FORTRAN 77 als **Teilmenge** in Fortran 90 enthalten ist.
- FORTRAN 77 Programme mit veralteten (*obsolescent*) Sprachelementen werden (manuell oder werkzeug-unterstützt) modernisiert, wobei die Diagnosemöglichkeiten eines Fortran 90 Compilers bei der Kennzeichnung derartiger Stellen ausgenutzt werden können.

- Moderne Programmiertechniken wie z.B. dynamische Speicherallokierung, problem-orientierte Datentypen, optionale Parameter und Kapselung in Modulen werden in alten und neuen Programmen integriert.
- Portable Programme im wissenschaftlich-technischen Bereich werden unter Zuhilfenahme systemunabhängiger numerischer Datentypen erstellt.
- Felder werden als Ganzes bearbeitet. Die optimale Nutzung von modernen Architekturen (u.a. Optimierung, Vektorisierung, Parallelisierung) wird nicht durch den Programmierer allein erreicht, sondern durch einen Fortran 90 Compiler oder andere Tools weitgehend automatisiert.

3. Lexikalische Erweiterungen

In diesem Kapitel werden folgende lexikalischen Erweiterungen behandelt:

- erweiterter Zeichensatz
- längere Namen
- Symbole für Relationsoperatoren
- formatfreier Quelltext
- Kommentare in Anweisungszeilen
- mehrere Anweisungen in einer Zeile
- **INCLUDE** Anweisung

3.1. Zeichensatz und Namen

- Namen beginnen mit einem Buchstaben und bestehen aus **bis zu 31 Zeichen**.

```
FUNCTION TransposeMatrix(aMatrix1)
REAL :: rLeftIntervalBound=0.0, rRightIntervalBound=1.0
```

- **Unterstriche** in Namen sind erlaubt.

```
REAL a_Transposed_Matrix(100,100)
LOGICAL l_EOF_Reached
```

- **Kleinbuchstaben** sind erlaubt, wobei Klein- und Großbuchstaben die selbe Bedeutung haben.

```
PRINT *, a, b, c
print *, A, B, C
```

- Der Zeichensatz wird durch folgende neue **Sonderzeichen** erweitert:

Sonderzeichen	Bedeutung
! <i>comment</i>	leitet einen Kommentar <i>comment</i> ein.
" <i>string</i> "	begrenzt eine Zeichenkette <i>string</i> .
%	selektiert eine Komponente einer Struktur.
&	kennzeichnet eine Fortsetzungszeile.
;	trennt Anweisungen.
<	symbolisiert den Relationsoperator .LT.
>	symbolisiert den Relationsoperator .GT.
?	-

\$	ist als Bestandteil eines Namen erlaubt.
----	--

- **Relationsoperatoren** können symbolisch durch `>`, `<`, `<=`, `>=`, `==` und `/=` dargestellt werden (Achtung: `/=` und nicht `!=` wie in C):

Operator	Symbol	Bedeutung
.LT.	<	kleiner
.LE.	<=	kleiner oder gleich
.GT.	>	größer
.GE.	>=	größer oder gleich
.EQ.	==	gleich
.NE.	/=	ungleich

```
if ( r1 > 0 ) ...
if ( (i1 /= 0) .and. (i2 == 2) ) ...
```

3.2. Formatfreie Form des Quelltextes

- Eine Zeile enthält maximal 132 Zeichen.
- Das Zeichen `!` leitet einen Kommentar ein, der bis zum Ende der Zeile reicht.

```
real function MaxEV(ar) ! computes maximal Eigenvalue
                        ! from matrix ar
real ar(:, :)          ! declares an assumed-shape matrix ar
```

- Das Zeichen `;` trennt Anweisungen voneinander.

```
r1=5; r2=3; r3=5
do i=1,100; read *, ar(i); end do
```

- Das Zeichen `&` als letztes Zeichen einer Zeile kennzeichnet eine Fortsetzung.

```
print *, "This is a very long line that is splitted here &
because it does not fit on a 80 column screen editor."

print *, "This string is continued &
&with no space"

--> This string is continued with no space
```

- **Leerzeichen** sind bei der lexikalischen Analyse im freien Eingabeformat signifikant.

```
doi=1,100           ! generates a compiler error.
do i=1, 100         ! correct input format
```

Programmbeispiel:

```
$ cat sample1.f90
```

```
! file: sample1.f90
program hello_world
! prints the text Hello world to standard output
implicit none
integer :: i1, i2=3      ! i2 defines the number of iterations
do i1=1, i2
  print "Hello World"
end do
end program
```

```
$ f90 -o sample1 sample1.f90
```

```
$ sample1
```

```
Hello World
Hello World
Hello World
```

3.3. Einfügen von Quelltext

- Eine Datei kann durch eine **INCLUDE** Anweisung in den Quelltext eingefügt werden. Dies entspricht der aus C bekannten Praxis und kann z.B. zum Einfügen von systemabhängigen Parametern verwendet werden.

```
include "constants.h"
include "system.h"
```

- **INCLUDE** Anweisungen können geschachtelt werden, d.h. eine eingefügte Datei darf ihrerseits **INCLUDE** Anweisungen besitzen. Ein bedingtes Einfügen kann allerdings nur über einen C Präprozessor realisiert werden.

Programmbeispiel:

```
$ cat boolean.h
```

```
! file: boolean.h
logical, parameter :: t=.TRUE., f=.FALSE.
! end file: boolean.h
```

```
$ cat sample2.f90
```

```
program test
include "boolean.h"
print *, "t .or. f = ", t .or. f
end program test
```

```
$ # Compile and link step are omitted from now on ...
```

```
$ sample2  
t .or. f = t
```

3.4. Gegenüberstellung

Die formatfreie und die formatgebundene Form besitzen folgende Merkmale:

Merkmal	formatgebundene Form	formatfreie Form
Zeilenlänge	1-80	1-132
Marken	1-5	Position frei
Trennzeichen zwischen Anweisungen	;	;
Leerzeichen	nicht signifikant	signifikant
Trennzeichen zwischen lexikalischen Einheiten	nicht notwendig	notwendig
Fortsetzung	Zeichen in Spalte 6	& als letztes Zeichen
Kommentar	C oder * in Spalte 1, ! an beliebiger Position	! an beliebiger Position

4. Typvereinbarung

In diesem Kapitel werden folgende Erweiterungen bei der Typvereinbarung erläutert:

- **IMPLICIT NONE**
- **TYPE** Anweisung für benutzerdefinierte Datentypen
- Attribute für Variablen
- **KIND** Typparameter für Datentypen

4.1. IMPLICIT NONE

- Die Anweisung **IMPLICIT NONE** erzwingt eine Typvereinbarung aller verwendeten Variablen. Nicht vereinbarte Variablen werden vom Compiler gemeldet.

```
implicit none
```

4.2. Benutzerdefinierte Datentypen

- Neben den vordefinierten (*intrinsic*) Datentypen **REAL**, **DOUBLE PRECISION**, **INTEGER**, **COMPLEX**, **CHARACTER** und **LOGICAL** können benutzerdefinierte (*derived*) Datentypen verwendet werden. Im folgenden werden sie kurz als **Strukturen** bezeichnet.
- Die Definition einer Struktur erfolgt in einer **TYPE** Anweisung. Eine Struktur setzt sich aus **Komponenten** zusammen. (Im folgenden wird zur Unterscheidung bei Feldern von **Elementen** gesprochen.)

```
type Person
  character :: chName(30)           ! component 1
  character :: chSurname(30)       ! component 2
  integer   :: iAge                 ! component 3
end type Person
```

- Die Verschachtelung von Strukturen ist erlaubt, d.h. Komponenten von Strukturen können selbst wieder Strukturen sein.

```
type Point                               ! defines structure point
  real :: rX, rY                          ! with 2 components of type real
end type Point
type Circle                               ! defines structure circle
  type(Point): sPointMP                   ! defines midpoint with type point
  real :: rRad
end type Circle
```

- Die Typvereinbarung einer Variablen mit einem benutzerdefiniertem Datentyp *struct* erfolgt mit der **TYPE(struct)** Anweisung.

```
type(Person) :: sPersonMe               ! defines a variable
type(Person) :: asPersonList(30)        ! defines an array
```

- Eine gleichzeitige **Anfangswertzuweisung** ist möglich, wobei verschachtelte Strukturen aufzulösen sind (siehe auch nächsten Abschnitt).

```
type(Point)  :: sPoint1=Point(5.0,2.0)
type(Circle) :: sCircle1=Circle(Point(0.0,0.0),1.0)
type(Person) :: sPerson1=Person("Mustermann", "H.", 34)
```

- Der **Zugriff** auf einzelne Komponenten einer Struktur erfolgt über den **Selektor %**.

```
sPoint1%rX=4.0           ! assigns a value to level-1 component
sPoint1%rY=2.0
sCircle%sPointMP%rX=0.0 ! assigns a value to level-2 component
print *, sPoint1%rY      ! prints a component of type real
```

- Die **Zuweisung** ist für Variablen und Ausdrücke gleichen Datentyps immer definiert und erfolgt durch Kopieren aller Komponenten. Ein Struktur-Konstruktor ist wie in der Anfangswertzuweisung ebenfalls zulässig.

```
sPoint2=sPoint1
sPerson2=sPerson1
sPerson3=Person("Mueller", "F.", 37)
```

- Für Strukturen können benutzerdefinierte Prozeduren, Operatoren und Zuweisungen eingeführt werden.

```
subroutine PrintPerson(sPerson)      ! prints 2 comp. of Person
type(Person) :: sPerson
print *, "Name:      ", sPerson%chName
print *, "Surname:  ", sPerson%chSurname
end subroutine PrintPerson
```

Programmbeispiel:

\$ cat sample3

```
program test
! gives and uses definitions for user-defined types.
type Point
  real rX,rY
end type Point
type Circle
  type(Point) :: sPointMP      ! midpoint
  real        :: rRad          ! radius
end type circle

type(Point)  :: sPoint1, sPoint2(10)
type(Circle) :: sCircle1, sCircle2=Circle(Point(1.0,1.0),5.0)
sPoint1=Point(1.0,2.0)
sCircle1%sPointMP=sPoint1
print *, "sCircle1%sPointMP%rX = ", sCircle1%sPointMP%rX
print *, "sCircle1%sPointMP%rY = ", sCircle1%sPointMP%rY
end program test
```

\$ sample3

```
sCircle1%sPointMP%rX = 1.0000000
```

```
sCircle1%sPointMP%rY = 2.0000000
```

4.3. Attribute und Anfangswerte in einer Typvereinbarung

- Gleichzeitig mit der Vereinbarung können skalaren und vektorwertigen Variablen **Anfangswerte** zugewiesen werden. Typbezeichnung und Variablen sind durch 2 Doppelpunkte **::** voneinander zu trennen.

```
real :: rA=0.0, rB=0.5, rC=1.0
character (len=80) :: chErrMsg1="Division by zero."
```

- Bei Strukturen erfolgt die Anfangswertzuweisung komponentenweise. Verschachtelungen sind durch entsprechende Klammerung nachzubilden.

```
type(Point) :: sPoint1=Point(0.0,0.0)
type(Circle) :: sCircle1=Circle(Point(1.0,1.0),5.0)
```

- Für eine vektorwertige Variable (Feld mit Rang 1) kann eine Wertzuweisung mit einem **Feldkonstrukt** der Form (*/value1, value2, .../*) erfolgen.

```
integer :: i(5) = (/ 1,2,3,4,5 /)
```

- Ergänzend zum Typ können weitere Eigenschaften von Variablen durch **Attribute** festgelegt werden. Neben den bekannten Attributen wie **PARAMETER**, **DIMENSION** und **SAVE** sind weitere Attribute möglich (siehe folgende Kapitel).

```
real, parameter :: rPi=3.141.. ! defines symb. constant
integer, parameter :: iMAX=100
real, dimension(iMAX) :: vrVector1
real, dimension(:,,:), allocatable :: arMatrix
```

4.4. Überblick über Attribute

Die folgende Tabelle listet alle verfügbaren Attribute für Variablen auf:

Attribut	Bedeutung
([KIND=] ...)	wählt eine Genauigkeit oder einen Exponentenbereich für eine numerische Variable aus.
([LEN[GTH]=] ...)	legt eine maximale Länge für eine Zeichenkette fest.
ALLOCATABLE	ermöglicht die dynamische Zuweisung und Freigabe von Speicher für ein Feld zur Laufzeit eines Programmes.
DIMENSION	legt den Rang und die Indexgrenzen eines Feldes fest.
EXTERNAL	kennzeichnet eine externe Prozedur.

INTENT(IN) INTENT(OUT) INTENT(INOUT)	kennzeichnet einen Ein- und/oder Ausgabe-Parameter (nur für Prozeduren)
INTRINSIC	kennzeichnet eine vordefinierte (<i>intrinsic</i>) Prozedur.
OPTIONAL	kennzeichnet einen optionalen Parameter (nur für Prozeduren).
PARAMETER	kennzeichnet eine symbolische Konstante.
POINTER	kennzeichnet eine Zeiger-Variable (<i>Pointer</i>).
PRIVATE	kennzeichnet eine nur in einem Modul sichtbare Variable (nur für Module).
PUBLIC	kennzeichnet eine auch außerhalb eines Moduls sichtbare Variable (nur für Module).
SAVE	kennzeichnet eine statische Variable einer Prozedur (nur für Prozeduren sinnvoll).
TARGET	kennzeichnet eine statische Variable, auf die eine Zeiger-Variable zeigen darf.

Bemerkungen:

- Einige Attribute sind nur in einem bestimmten Kontext erlaubt wie z.B. **OPTIONAL** nur in Prozeduren oder **TARGET** nur für Felder.
- Einige Attribute schließen einander aus wie z.B. **PARAMETER** und **ALLOCATABLE**.

4.5. KIND Typparameter

- **Typparameter** ermöglichen in Zusammenarbeit mit der vordefinierten Funktion **selected_real_kind** bzw. **selected_int_kind** eine systemunabhängige Auswahl von Typen mit garantierten Genauigkeiten und Wertebereichen.
- Ein Typparameter (*KIND value*), der vom Benutzer definierte Anforderungen erfüllen soll, kann systemunabhängig durch die vordefinierte Funktion **selected_real_kind** ermittelt werden.

```
! REAL_6_150 is a symbolic constant
integer, parameter :: REAL_6_150=selected_real_kind(6,150)
```

- Variablen der Datentypen **REAL** und **COMPLEX** können durch einen **Typparameter** modifiziert werden, der eine bestimmte Genauigkeit (*precision*) und einen bestimmten Exponentenbereich (*range*) festlegt.

```
! Variables of kind REAL_6_150 have at least 6 significant
! digits and hold values in the range from 10**(-150)
! to 10**(150).
```

```
REAL, (KIND=REAL_6_150) :: r1
```

- Der NAG Fortran 90 Compiler (**f90**) enthält im Lieferumfang eine Moduldatei `f90_kind.mod`, in der alle verfügbaren Typparameter als symbolische Konstanten definiert sind (siehe unten). Die Definitionen können nach einer **USE** Anweisung verwendet werden.

```
use f90_kind          ! includes type definitions, e.g. double
complex (double), parameter :: cImagUnit=(0.0,1.0)
complex (double) :: c3, c4, c5=(1.0,1.0)
```

Zusammengefaßt:

Anweisung	Bedeutung
<code>kind_value=selected_real_kind(prec, exp_range)</code>	berechnet einen KIND Typparameter <code>kind_value</code> , der mindestens der Genauigkeit <code>prec</code> und dem Exponentenbereiches <code>exp_range</code> entspricht.
<code>REAL (KIND=kind_value) :: var1, ...</code>	vereinbart eine REAL Variable <code>var1</code> mit einem KIND Typparameter, der z.B. vorher mit selected_real_kind ermittelt wurde.

4.6. Ein Beispiel für die Definition von Typparametern

```
$ cat f90_kind.f90 # f90 V2.0 for IBM RS/6000
```

```
...
module f90_kind
!
! Indicator that the KIND= is not available
! for this compiler/host:
!   integer, parameter :: not_available = -1
!
! Real and Complex numbers
!   Single precision
!   integer, parameter :: single = 1      # defines single
!   Double precision
!   integer, parameter :: double = 2
!   Quadruple precision
!   integer, parameter :: quad = not_available
...

```

4.7. Zahlenmodelle

- Reelle Zahlen und ganze Zahlen werden auf Konstanten oder Variablen der Datentypen **REAL** und **INTEGER** abgebildet. Die Zahlenmodelle für reelle Zahlen (*floating point numbers*) und ganze Zahlen (*integer numbers*) sind unterschiedlich.

- Jedes Zahlenmodell ist durch **Modellparameter** eindeutig charakterisiert.
- Das reelle Zahlenmodell besitzt die Modellparameter **b**, **p**, **E_{min}** und **E_{max}**, wobei eine auf p Ziffern abgeschnittene reelle Zahl x folgendermaßen dargestellt wird:

$$x = \pm (\sum a_k \cdot b^{-k}) b^e = \pm m \cdot b^e$$

$$a_k \in \{0, 1, \dots, b-1\}, k=1, \dots, p$$

$$E_{\max} \geq e \geq E_{\min}$$

- Das ganzzahlige Zahlenmodell besitzt die Modellparameter **E_{min}** und **E_{max}**:

$$x = \pm (\sum a_k \cdot 2^k)$$

$$a_k \in \{0, 1\}; \quad k=0, \dots, n$$

$$E_{\max} \geq n \geq E_{\min}$$

- Die Modellparameter und hieraus abgeleitete Größen können über vordefinierte Prozeduren bestimmt werden.

4.8. Vordefinierte Abfragefunktionen für Modellparameter

Die folgenden intrinsischen Prozeduren liefern systemspezifische Informationen:

Name	Funktion
digits(x)	berechnet die Anzahl signifikanter Ziffern p im Zahlenmodell.
epsilon(x)	berechnet die kleinste Modellzahl mit Exponent 0 (b^{1-p}).
exponent(x)	berechnet den Exponenten e von x im Zahlenmodell.
fraction(x)	berechnet die Mantisse m von x im Zahlenmodell.
huge(x)	berechnet die größte Zahl im Zahlenmodell $(1-b^{-p}) \cdot b^{E_{\max}}$
kind(x)	berechnet den KIND Parameter von x.
maxexponent(x)	berechnet den maximalen Exponenten E_{\max} im Zahlenmodell.
minexponent(x)	berechnet den minimalen Exponenten E_{\min} im Zahlenmodell.
nearest(x,direction)	berechnet die nächste darstellbare Zahl im Zahlenmodell.
precision(x)	berechnet die dezimale Genauigkeit im Zahlenmodell.
radix(x)	berechnet die Basis b des Zahlenmodells.
range(x)	berechnet das Minimum von E_{\min} und E_{\max} .

selected_int_kind (<i>exp_range</i>)	berechnet den kleinsten zugehörigen KIND Parameter.
selected_real_kind (<i>prec[,exp_range]</i>)	berechnet den kleinsten zugehörigen KIND Parameter.
set_exponent(x,i)	berechnet $x*b^{i-e}$.
tiny(x)	berechnet die kleinste Zahl im Zahlenmodell ($b^{E_{\min}-1}$).

5. Feldverarbeitung

In diesem Kapitel werden folgende Klassen von Feldern behandelt:

- Felder und Zeichenketten der Länge 0
- Felder mit übernommener Gestalt
- parametrisierte lokale Felder
- dynamische Felder

5.1. Felder und Zeichenketten der Länge 0

- Die Länge Null für Zeichenketten und die Ausdehnung Null (*extent*) für Felder sind zulässig.

```
character (0) :: chEmptyString
print *, "len(chEmptyString = ", len(chEmptyString)
```

5.2. Felder mit übernommener Gestalt

- In einer Prozedur kann als Formalparameter ein **Feld mit übernommener Gestalt** (*assumed-shape array*) verwendet werden. Der Rang (Anzahl Dimensionen) ist allerdings nicht variabel.

```
subroutine x(arM)
real arM(:, :) ! formal parameter is assumed-shape array
```

- Die Gestalt des Feldes wird automatisch aus dem übergebenen Feld bestimmt. Die unteren Indexgrenzen liegen standardmäßig bei 1 und können bei der Typvereinbarung verändert werden.

```
subroutine x(arM)
real arM(10:, 100:) ! index starts with 10 (1D) and 100 (2D)
```

- Die zunächst unbekannt **Ausdehnungen** des übergebenen Feldes sind über vordefinierte Abfragefunktionen zu ermitteln.

```
subroutine x(arM)
real arM(:, :)
iNumberOfRows=size(arM,1) ! yields 1st dimension
```

Programmbeispiel:

```
$ cat sample4.f90
```

```
program test
interface
  subroutine x(arM)
    real arM(:, :)
  end subroutine x
end interface
```

```

real arM(5,2)
call x(arM)
end program test

subroutine x(arM)      ! uses assumed-shape array.
real arM(:,:)        ! defines formal parameter(s)
integer i1, i2
do i1=1, size(arM,1)  ! loops over 1st dimension (row)
  do i2=1, size(arM,2) ! loops over 2nd dimension (column)
    arM(i1,i2)=10*i1+i2
    print *, 'arM(',i1,',',i2,') = ', arM(i1,i2)
  end do
end do
end subroutine x

```

\$ sample4

```

arM( 1 , 1 ) = 11
...
arM( 5 , 1 ) = 51
arM( 5 , 2 ) = 52

```

5.3. Parametrisierte lokale Felder

- In einer Prozedur kann ein **lokales Feld** (*workspace*) festgelegten Ranges mit parametrisierten Grenzen verwendet werden.

```

subroutine x(n,m)
real, dimension(1:n,1:m) :: arWorkspace      ! defines local array

```

- Die **Parametrisierung** kann durch globale Variablen (**COMMON** oder **MODULE**) oder, indirekt oder direkt, durch Formalparameter erfolgen.

```

subroutine x(arM)
real, dimension(:,:) :: arM                ! assumed-shape array
real, dimension(size(arM,1)) :: vrWS      ! local array
...
end subroutine x

```

- Das lokale Feld wird bei Eintritt in die Prozedur **automatisch** erzeugt und bei Verlassen der Prozedur wieder freigegeben.
- Die Schnittstelle einer Prozedur, die ein lokales Feld verwendet, muß vor dem Aufruf in einer **INTERFACE** Anweisung definiert werden (siehe auch Abschnitt: **Modulprozedur**).

```

interface
  subroutine x(n,m)
    real, dimension(1:n,1:m) :: arWorkspace
  end subroutine x
end interface

```

Programmbeispiel:

\$ cat sample5.f90

```

program test
interface
  subroutine x(n)
    real :: arM(n,n)
  end subroutine x
end interface
call x(2)
end program test

subroutine x(n)
integer :: n
real    :: arM(n,n**3)      ! defines parameterized local array
integer i1, i2, n
do i1=1, n
  do i2=1, n**3
    arM(i1,i2)=10*i1+i2
    print *, 'arM(',i1,',',i2,') = ', arM(i1,i2)
  end do
end do
end subroutine x

```

\$ sample5

```

arM( 1 , 1 ) = 11
...
arM( 2 , 8 ) = 28

```

5.4. Dynamische Felder

- Für ein Feld festen Rangs, das mit dem Attribut **ALLOCATABLE** vereinbart worden ist, kann zur Laufzeit Speicher in zur Laufzeit definierter Größe angefordert werden (*dynamic memory allocation*). Die maximale Größe ist abhängig vom Computersystem.

```
real, dimension(:, :, :), allocatable :: arT1
```

- Das Anfordern und Zuordnen von Speicher für ein dynamisches Feld erfolgt durch die **ALLOCATE** Anweisung.

```
read *, i1, n1, i2, n2, i3, n3      ! user input
allocate(arT1(i1:n1,i2:n2,i3:n3), stat=iAllocStatus)
```

- Der Status eines dynamischen Feldes kann über die **ALLOCATED** Anweisung überprüft werden; d.h. während der Ausführung eines Programm kann abgefragt werden, ob für ein dynamisches Feld bereits Speicher zugewiesen wurde.

```
if (allocated(arT1)) print *, "Array arT1 is allocated."
```

- Nicht mehr benötigter Speicher kann (und sollte) durch die **DEALLOCATE** Anweisung freigegeben werden.

```
deallocate(arT1, stat=iDeallocStatus)
```

Programmbeispiel:

\$ cat sample6

```

program test
  ! uses allocatable array of rang 2

  real, dimension(:, :), allocatable :: arM
  integer :: i1, i2, iAllocStatus, iDeallocStatus, n

  read *, n                                ! interactive input
  allocate(arM(n,n), stat=iAllocStatus)
  if (iAllocStatus /= 0) then
    print *, "Error: Alloc failed: ", iAllocStatus
    return
  end if

  do i1=1, n
    do i2=1, n
      arM(i1,i2)=10*i1+i2
      print *, 'arM(', i1, ', ', i2, ') = ', arM(i1,i2)
    end do
  end do
  deallocate(arM, stat=iDeallocStatus)
  if (iDeallocStatus /= 0) then
    print *, "Dealloc failed: ", iDeallocStatus
    return
  end if
end program test

```

\$ sample6 < echo 2

```

a(1,1)=11
a(1,2)=12
a(2,1)=21
a(2,2)=22

```

\$ sample6 < echo 16384 # You should compute how big the array would be!

Error: Alloc failed: 1 # This error is due to insufficient memory.

5.5. Vordefinierte Abfragefunktionen für Felder

Die folgenden Prozeduren liefern Informationen über Felder:

Name	Bedeutung
allocated(a)	liefert den Status eines dynamischen Feldes.
lbound(a [, dim])	liefert die untere(n) Indexgrenze(n) (<i>lower bound</i>).
len(string)	liefert die Länge einer Zeichenkette <i>string</i> .
maxval(a[, dim])	liefert das größte Element (in einer Dimension).

<code>minval(a[, dim])</code>	liefert das kleinste Element (in einer Dimension).
<code>shape(a)</code>	liefert die Gestalt von <i>a</i> , d.h. einen Vektor, der die Ausdehnungen in den einzelnen Dimensionen von <i>a</i> enthält.
<code>size(a [, dim])</code>	liefert die Gesamtanzahl aller Elemente (bzw. die Ausdehnung in einer Dimension) von <i>a</i> .
<code>ubound(a [, dim])</code>	liefert die obere(n) Indexgrenze(n) (<i>upper bound</i>).

5.6. Teilfeld-Selektoren

- Die Auswahl eines rechteckigen Teilfeldes aus einem Feld kann durch ein **Triplex** oder durch einen **Vektorindex** erfolgen.
- Ein rechteckiges Teilfeld von *a* kann durch ein **Triplex** der Form *a(i1:i2:i3, ...)*, *i1*=Anfangswert, *i2*=Endewert und *i3*=Schrittweite, ausgewählt werden.

```
integer aiM1(3,3)      =>  12 13 14
                        22 23 24
                        32 33 34

aiM2=aiM1(1:2,1:3)    =>  12 13 14
                        22 23 24
                        -  -  -
                        Zeile 1 und 2
```

- Fehlende Werte im Triplex werden durch **Standardwerte** ergänzt (untere Indexgrenzen, obere Indexgrenze, 1).

```
ai(1:2:1,1:3:1)      ! specifies all 3 values
ai(1:2, 1:3 )        ! uses default step width 1
ai( :2, : )          ! uses defaults
```

- Negative Schrittweiten** (Spiegelung) sind ähnlich wie in einer DO Anweisung erlaubt.

```
aiM2=aiM1(:,3:1:-1) =>  13 12 11
                        23 22 21
                        33 32 31
                        (Spiegelung)
```

- Insbesondere können die Zeile *i* und die Spalte *j* einer Matrix durch **a(i,:)** und **a(:,j)** ausgewählt werden.

- Ein Teilfeld von *a* kann über einen **Vektor von Indizes** *v* ausgewählt werden.

```
integer :: vi=(/2,4/)      ! yields an index vector
integer :: aiM(5,5)
```

```
aiM2=aiM1(vi,:) ! extracts 1D (row) 2 and 4
```

Programmbeispiel:

```
$ cat sample7.f90
```

```
program test
integer :: i1, i2
integer :: aiM(3,2)
integer :: vi(size(aiM,2))
do i1=1, ubound(aiM,1)
  do i2=1, ubound(aiM,2)
    aiM(i1,i2)=10*i1+i2
  end do
end do
vi=aiM(:,2) ! extracts column 2
do i1=1, ubound(vi)
  print *, 'vi(', i1, ') = ', vi(i1)
end do
end program test
```

```
$ sample7
```

```
vi(1) = 21
vi(2) = 22
```

5.7. Operationen auf Feldern

- Arithmetische und boolesche Operationen wie z.B. Addition und Multiplikation können **elementweise** auf komplette Felder oder Teilfelder angewendet werden (*array processing*).

```
a3 = a1 + a2 ! a3(i,j) = a1(i,j) + a2(i,j)
a3 = a1 * a2 ! a3(i,j) = a1(i,j) * a2(i,j)

a13 = a11 .and. a1 ! a13(i,j) = a11(i,j) .and. a12(i,j)
```

- Die beteiligten Felder müssen in ihrer **Gestalt**, (*shape*) nicht aber in ihren Indexgrenzen, übereinstimmen (*shape conformance*). Die Operationen werden mit korrespondierenden Feldelementen durchgeführt.

```
real ar1(100) ! extent = 100 - 1 + 1 = 100
real ar2(201:300) ! extent = 300 - 201 + 1 = 100
ar1 = ar1+ar2 ! ar1(i) = ar1(i)+ar2(200+i)
```

- Skalare passen zu allen Feldern.

```
ar2 = ar1 + r1 ! ar2(i,j) = ar1(i,j) + r1
```

- Die vordefinierten elementaren Funktionen können zur **elementweisen** Manipulation ganzer Felder verwendet werden.

```
ar2=sin(ar1)+r1 ! ar2(i,j) = sin(ar1(i,j))
```

- Einige neue vordefinierte Funktionen dienen zur Feldbearbeitung wie z.B. **matmul** zur Matrixmultiplikation, **maxval** zur Ermittlung des größten Elementes, **sum** zur Berechnung von Summen und **transpose** zum Vertauschen von Zeilen und Spalten.

```
a3=matmul(a1,a2)    ! a3(i,j) = Σ a1(i,k)*a2(k,j)
a2=transpose(a1)   ! a2(i,j) = a1(j,i)
```

- **Benutzerdefinierte** Funktionen, Operatoren und Zuweisungen können auf ganze Felder wirken bzw. ein Feld als Ergebnis liefern.

```
function arInvertMatrix(n,arM)
real, dimension(n,n) :: arInvertMatrix ! Result is a matrix.
```

- Eine Zuweisung kann durch eine **WHERE** Anweisung "maskiert" werden. Die Zuweisung wird nur dann durchgeführt, wenn eine bestimmte Bedingung erfüllt ist (*masked array assignment*).

```
WHERE (ar1 > 0.0)
    ar1=log(ar1)
ELSE WHERE
    ar1=0.0
END WHERE
```

- In einer **WHERE** Anweisung müssen die in der Bedingung und der Zuweisung verwendeten Felder gleiche Gestalt haben.

```
real ar1(5), ar2(101:105)
WHERE (abs(ar2) > rEPSILON)
    ar1=ar1/ar2
...
END WHERE
```

- Die rechte Seite darf während einer feldwertigen Zuweisung nicht verändert werden, da aus Optimierungsgründen die gesamte rechte Seite während der Zuweisung als konstant angesehen wird.

```
! Attention: Different results

integer :: ai1=(/1,1,1/), ai2=(/1,2,3/)
ai1(2:3)=ai1(1:2)+ai2(1:2)

integer :: ai1=(/1,1,1/), ai2=(/1,2,3/)
do i=2, 3
    ai1(i)=ai1(i-1)+ai2(i-1)
end do
```

5.8. Zeiger-Variablen

- Eine Variable, die mit dem Attribut **POINTER** vereinbart wurde, kann auf ein Feld festgelegten Typs und Rangs oder auf eine andere Zeiger-Variable zeigen (verweisen).

```
real, dimension(:,:), pointer : pr ! may point to real rank-2 array
```

- Eine Zeiger-Variable kann nur auf eine andere Zeiger-Variable gleichen Typs oder ein Feld zeigen, die mit dem Attribut **TARGET** vereinbart wurden.

```
real, dimension(5,5), target: ar           ! real array of rank 2
```

- Die Assoziation (Verknüpfung) einer Zeiger-Variablen mit einem Feld erfolgt durch eine Zeiger-Zuweisung der Form **Pointer => object**.

```
pr => ar
```

- Der Status einer Zeiger-Variablen kann mit der **ASSOCIATED** Anweisung abgefragt werden.

```
if (associated(pr [,target])) then ...
```

- Die Zuordnung einer Zeiger-Variablen wird mit der **NULLIFY** Anweisung aufgehoben.

```
nullify(pr)
```

- Mit Hilfe von Zeiger-Variablen sind z.B. **verkettete Listen** realisierbar. Unverkettete Zeiger-Variablen müssen mit einer **NULLIFY** Anweisung explizit auf den Nullzeiger gesetzt werden, damit eine Endbedingung überprüft werden kann.

```
type list
  type(list), pointer :: psListP ! points to prev. element
  type(list), pointer :: psListN ! points to next element
  real :: rContent
end type list
```

Programmbeispiel:

```
$ cat sample8.f90
```

```
program test
  real, dimension(100,100), target :: ar
  real, dimension(:,,:), pointer :: pr
  ar = 1.0 ! ar(i,j) = 1.0
  print *, "Status of association (before =>): ", associated(pr)
  pr => ar(1:20,1:20)
  print *, "Status of association (after =>): ", associated(pr)
  print *, "pr(1,1) = ", pr(1,1)
  print *, "pr(20,20) = ", pr(20,20)
  nullify(pr)
  print *, "Status of association (after nullify): ", associated(pr)
  pr => ar(1:50,1:50)
end program test
```

```
$ sample8
```

```
... F
... T
... F
```

5.9. Zusammenfassung

Folgende Klassen von Feldern existieren:

Beschreibung	Festlegung der Indexgrenzen
Ein Feld wird für die gesamte Laufzeit des Programmes erzeugt. Die Indexgrenzen werden bereits zur Übersetzungszeit festgelegt.	Typvereinbarung
Ein vorhandenes Feld wird an eine Prozedur übergeben. Die Indexgrenzen werden parametrisiert, wobei die letzte Dimension ggf. mit * vereinbart werden kann.	Formalparameter
Ein vorhandenes Feld wird an eine Prozedur übergeben. Die Indexgrenzen werden implizit übergeben (<i>assumed shape array</i>).	Aktualparameter (übernommene Gestalt)
Ein lokales Feld wird erzeugt, wobei die Indexgrenzen parametrisiert sind (<i>parameterized local array</i>).	Aktualparameter oder globale Variablen
Ein Feld wird dynamisch erzeugt und freigegeben.	ALLOCATE Anweisung
Ein Feld kann synonym über eine Zeiger-Variable referenziert werden.	Zeiger-Zuweisung (übernommene Gestalt)
Ein Teilfeld wird durch einen Teilfeld-Selektor (Triplex-Notation oder Vektor-Index) ausgewählt.	Werte in der Triplex-Notation oder Werte der Vektor-Indizes.

6. Kontrollstrukturen

In diesem Kapitel werden folgende Erweiterungen beschrieben:

- erweiterte **DO** Anweisung
- Namen für Kontrollstrukturen
- **EXIT** und **CYCLE** Anweisung
- Fallunterscheidung (**SELECT CASE** Anweisung)

6.1. DO Anweisung

- Die Anweisungen **DO** und **END DO** begrenzen einen DO Block. (Achtung: Marken und **CONTINUE** Anweisungen sind nicht mehr notwendig und werden von einem Fortran 90 Compiler als veraltet gemeldet!)

```
do i1=1, 100
  ...
end do
```

- Jede **DO** Anweisung kann durch einen **Namen** benannt werden (siehe auch **EXIT** und **CYCLE** Anweisung).

```
outer_loop: do i1=1, 100
  inner_loop: do i2=1, 200
    ...
  end do inner_loop
end do outer_loop
```

- Eine abweisende Schleife kann durch eine **DO WHILE** Anweisung realisiert werden.

```
do while (i1 > 10 .and. i2 < 100)
  ...
end do
```

- Eine Schleifenblock wird durch eine **EXIT** Anweisung verlassen. Der Rücksprung zu einer umgebenden Schleifenebene kann über den zugehörigen Namen erreicht werden.

```
loop1: do ...
  loop2: do ...
    if (...) exit loop2
  ...end do loop 2
end do loop1
```

- Eine **CYCLE** Anweisung startet einen neuen Schleifenzyklus. Eine umgebende Schleifenebene kann über den zugehörigen Namen erreicht werden.

```
do ...
  if (...) cycle
end do
```

Programmbeispiel:

```
$ cat sample9
```

```
program test
integer, parameter :: n=100
integer :: il
integer :: vi(n)

do il=1, n
    read *, vi(il)
    print *, 'vi(',il,') = ', vi(il)
    if (vi(il) == 0) exit
end do

end program test
```

```
$ sample9 < echo 0
```

```
0
```

6.2. SELECT CASE Anweisung

- In einer **SELECT CASE** Anweisung wird eine Fallunterscheidung nach einem Ausdruck durchgeführt.

```
iReturnValue=DoSomething(...)
select case (iReturnValue)
case(0):
    print *, "DoSomething: Successful."
case(-1)
    print *, "DoSomething: Failed."
case default
    print *, "DoSomething: iReturnValue should be 0 or -1."
end select
```

- Der untersuchte Ausdruck kann vom Datentyp **INTEGER**, **CHARACTER** oder **LOGICAL** sein.

```
select case (il)
case (1) ...

select case (ch1)
case ("A") ...

select case (l1)
case (.TRUE.) ...
```

- Mögliche Fälle sind durch **CASE Selektoren** charakterisiert. Ein CASE Selektor besteht aus einer Konstanten bzw. einer Menge oder einem Bereich von Konstanten. Eine Verzweigung erfolgt, wenn ein Vergleich des Ausdrucks mit dem CASE Selektor erfolgreich ist.

```
case (0)                ! 0
case (-3:1)             ! -3 to 1
```

```
case (:100)                ! -infinity to 100
case ("0":"9")            ! "0" to "9"
case ("Y", "y", "J", "j") ! all variants of Y[ES]
case ("a":"z", "A":"Z")   ! all letters
```

- Überschneidungen sind nicht zulässig, d.h. es darf höchstens ein Fall zutreffen.

```
case (5:10)
case (7:11)          ! ERROR: Overlapping ranges
```

- Bei Angabe eines **DEFAULT** Falles wird dieser ausgeführt, falls kein anderer zutrifft.

```
select case (...)
...
case default: print *, "Default case."
end select
```

Programmbeispiel:

```
$ cat sample10.f90
```

```
program test
character(1) :: ch1
print *, 'Enter a character:'
read *, ch1
select case (ch1)
  case("0":"9")           print *, "Digit."
  case("a":"z","A":"Z")   print *, "Letter."
  case default           print *, "Neither digit nor letter."
end select
end program test
```

```
$ sample10 < echo 9
```

```
Digit.
```

7. Prozeduren

In diesem Kapitel werden folgende Erweiterungen bei Prozeduren behandelt:

- Attribute **INTENT** und **OPTIONAL**
- Schlüsselwort-Parameter
- Abprüfen der Existenz von optionalen Parametern
- Rekursive Prozeduren
- Feld, Struktur oder Zeiger als Funktionsergebnis
- Interne Prozeduren
- Schnittstellenbeschreibungen von Prozeduren
- Generische Schnittstellen
- Benutzerdefinierte Operatoren und Zuweisungen

7.1. Attribute für Formalparameter

- Bei der Vereinbarung von Formalparametern kennzeichnen die Attribute **INTENT** und **OPTIONAL** Ein- und/oder Ausgabeparameter und optionale Parameter.

```
subroutine x(r1, r2, r3, r4, iOpt1)
  real, intent(in)      :: r1, r2           ! for input only
  real, intent(out)    :: r3               ! for output only
  real, intent(inout)  :: r4               ! input and output
  integer, optional    :: iOpt1           ! may be omitted
```

- Ein Fortran 90 Compiler überprüft, ob die übergebenen Aktualparameter zulässig sind. Z.B. ist eine Konstante als Aktualparameter für einen Formalparameter mit Attribut **INTENT(OUT)** unzulässig.

```
integer function x(i1)
  integer, intent(out) :: i1

  iRC=x(3)           ! ERROR: i1 must be a variable
```

- Eine Prozedur kann mit **Schlüsselwort-Parametern** aufgerufen werden, wobei in diesem Fall die Reihenfolge (Position) der Aktualparameter beliebig ist und optionale Parameter weggelassen werden können.

```
integer function x(i1, i2, iOpt1, iOpt2, iOpt3)
  integer :: i1, i2
  integer, optional :: iOpt1, iOpt2, iOpt3
  ...
end function x

iRC = x(iOpt3=4, i2=4, i1=5)
```

- Bei einem gemischten Aufruf mit Positions-Parametern und Schlüsselwort-Parametern müssen **zu Beginn** Positions- und **am Ende** Schlüsselwort-Parameter aufgelistet werden.

```
iRC = x(5, 4, iOpt3=4)
```

- Die Existenz von optionalen Parametern kann über die vordefinierte Funktion **present** abgeprüft werden.

```
real, optional :: rOpt
if (present(rOpt)) print *, "Procedure received arg for rOpt."
```

Zusammengefaßt:

Attribut	Bedeutung	Beispiel
INTENT(IN)	kennzeichnet Eingabeparameter.	integer, intent(in) :: i1, i2
INTENT(OUT)	kennzeichnet Ausgabeparameter.	real, intent(out) :: r1
INTENT(INOUT)	kennzeichnet Parameter, die sowohl Ein- als auch Ausgabeparameter sind.	real dimension(:,:), & intent(inout):: a1, a2
OPTIONAL	kennzeichnet optionale Parameter (günstig am Ende der Parameterliste).	integer, optional :: iMaxIt

Programmbeispiel:

\$ cat sample11.f90

```

program test
interface
  real function rSquareRoot(r1, rOptPrec)
  real, intent(in) :: r1
  real, optional   :: rOptPrec           ! yields desired prec.
  end function rSquareRoot
end interface
print *, 'rSquareRoot(9.0)          = ', rSquareRoot(9.0)
print *, 'rSquareRoot(9.0,1.E-3) = ', rSquareRoot(9.0,1.E-3)
end program test

real function rSquareRoot(r1, rOptPrec)
real, intent(in) :: r1
real, optional   :: rOptPrec
real             :: r2=1.0E-6           ! defines default precision
real             :: rTemp
if (present(rOptPrecision)) then
  r2=rPrecision                          ! takes user-defined precision
end if
...! TO DO: Calculate rSquareRoot using Newton's formula
...! using the given precision as stop condition.
end function rSquareRoot

```

\$ sample11

```
rSquareRoot(9.0)          = 3.0000000
rSquareRoot(9.0,1.E-1) = 3.0000916
```

7.2. Rekursive Prozeduren

- Prozeduren, die mit dem Schlüsselwort **RECURSIVE** vereinbart sind, können sich selbst aufrufen.

```
recursive subroutine x()
```

- Bei rekursiven Funktionen muß der Rückgabewert über das Schlüsselwort **RESULT** benannt werden, da sonst keine Unterscheidung zwischen Funktionsaufruf und Funktionsergebnis möglich ist.

```
recursive integer function iGCD(i1,i2) result(iOut)
```

Programmbeispiele:

```
$ cat sample12.f90
```

```
program test
integer i1=20, i2=15
print *, "GCD(", i1, ",", i2, ") = ", iGCD(i1,i2)
end program test

recursive integer function iGCD(i1,i2) result(iOut)
! computes greatest common divisor of i1 and i2.
integer intent(in) :: i1, i2
integer iTemp=mod(i1,i2)
if (iTemp == 0)
    iOut=abs(i2)
else
    iGCD(i2,iTemp)
end function GCD
```

```
$ sample12
```

```
5
```

```
$ cat sample13.f90
```

```
program test
call TOH(3,1,2,3)
end program test

recursive subroutine TOH(n, iStart, iTarget, iSwap)
! tells how to move disks in the Tower of Hanoi problem.
integer n, iStart, iTarget, iSwap
if (n > 1) call TOH(n-1, iStart, iSwap, iTarget)
print *, "Move disk ", n, " from ", iStart, " to ", iTarget, "."
if (n > 1) call TOH(n-1, iSwap, iTarget, iStart)
end subroutine TOH
```

```
$ sample13
```

```
Move disk 1 from 1 to 2 .
...
```

```
Move disk 2 from 3 to 2 .
Move disk 1 from 1 to 2 .
```

7.3. Schnittstellenbeschreibung für Prozeduren

- Die **INTERFACE** Anweisung erweitert die **EXTERNAL** Anweisung. Vor dem Aufrufen einer Prozedur sollte (und muß in einigen Fällen) die Schnittstelle zur Prozedur in einer **INTERFACE** Anweisung beschrieben werden, damit der Compiler die Aufrufparameter auf Gültigkeit überprüfen kann.

```
interface
  real function rSquareRoot(r1)
  real, intent(in) :: r1
end interface
```

- Eine **INTERFACE** Anweisung enthält Typ und Namen einer **Prozedur** und Typen, Eigenschaften und Namen der **Formalparameter**.

```
interface
  real function x1(ar1)
  real, intent(inout), dimension(:, :) :: ar1
  end real function x1
end interface
```

- Eine Schnittstellenbeschreibung kann einfacher in einem Modul durchgeführt werden (siehe Kapitel: **Module**). Die Vereinbarungen werden in diesem Fall durch eine **USE** Anweisung eingefügt.

```
use special_functions ! includes interface for SinusHyp from module
r2=SinusHyp(r1)
```

7.4. Generische Prozeduren

- Eine **INTERFACE** Anweisung kann für die Definition einer **generischen Prozedur** verwendet werden. Die **Signaturen** der spezifischen Prozeduren müssen unterschiedlich sein, damit anhand der Aktualparameter eindeutig eine spezifische Prozedur bestimmt werden kann. (**Signatur**: Anzahl, Typ, Typparameter und Rang der Formalparameter)

```
interface Swap ! generic interface
  real function Swap_Real(r1)
  ...
  end function
  complex function Swap_Complex(c1)
  ...
  end function
end interface

call Swap(...) ! callable with real or complex argss
```

7.5. Benutzerdefinierte Operatoren und Zuweisungen

- In einer **INTERFACE** Anweisung können benutzerdefinierte Operatoren (**OPERATOR**) und Zuweisungen (**ASSIGNMENT**) definiert werden, die auch vordefinierte Operatoren und Zuweisungen überladen können.

```
interface operator (+)          ! overloads + for type Interval
function AddInterval(sIntervall1, sIntervall2)
type(Interval) :: sIntervall1, sIntervall2
...
end interface
```

7.6. Interne Prozeduren

- **Interne Prozeduren** sind ein möglicher Ersatz für Formelfunktionen. Sie werden am Ende einer Programmeinheit definiert und können nur von der umgebenden Programmeinheit aufgerufen werden. Die Definition interner Prozeduren wird mit dem Schlüsselwort **CONTAINS** eingeleitet.

```
subroutine x
call x1
...
contains          ! internal procedure(s)
subroutine x1
...
end subroutine x2
end subroutine x
```

- Interne Prozeduren können auf Variablen der umgebenden Programmeinheit zugreifen. Variablen in internen Prozeduren überdecken Variablen gleichen Namens der umgebenden Programmeinheit.
- Interne Prozeduren dürfen selbst keine internen Prozeduren enthalten, d.h. die Schachtelungstiefe beträgt nur 1.

Programmbeispiel:

\$ cat sample14.f90

```
program test
call x
end program test

subroutine x
integer :: i1=1
call x1
call x2
contains
subroutine x1
print *, 'Internal subroutine x1: i1 = ', i1
end subroutine x1
subroutine x2
integer :: i1=2
print *, 'Internal subroutine x2: i1 = ', i1
end subroutine x2
end subroutine x
```

\$ sample14

```
Internal subroutine x1: i1 = 1  
Internal subroutine x2: i1 = 2
```

8. Module

In diesem Kapitel wird das Modulkonzept in folgenden Abschnitten erläutert:

- Ersatz der **COMMON** Anweisung
- Vereinbarungen
- Anfangswertzuweisungen
- **USE** Anweisung
- Generische Schnittstellen
- benutzerdefinierte Operatoren und Zuweisungen
- Attribute **PRIVATE** und **PUBLIC**

8.1. Ersatz der COMMON Anweisung

- Die Programmeinheit Modul (*module*) stellt eine Erweiterung der **COMMON** Anweisung bzw. der Programmeinheit **BLOCK DATA** dar (und sollte diese langfristig ersetzen).
- In einem Modul können **globale Konstanten** und **Variablen** vereinbart und Anfangswertzuweisungen durchgeführt werden.

```

module global_data
  integer :: iError=0                ! defines global variable
  integer, parameter :: iMaxIt=1000 ! defines global parameter
end module global_data

```

- Eine Programmeinheit erhält über eine **USE** Anweisung Zugriff auf die in einem Modul verfügbaren Vereinbarungen und Definitionen.

```

program test
  use global_data                ! includes module global_data
  ...
  do i1=1, iMaxIt                ! uses global variable iMaxIt
  ...
end program test

```

- Die Liste der durch eine **USE** Anweisung prinzipiell verfügbaren Variablen kann durch den Parameter **ONLY** eingeschränkt werden.

```

program test
  use global_data, only: iError    ! iMaxIt is not available.
  ...

```

- Zur Vermeidung von Namenskollisionen können Variablen eines Moduls in einer Programmeinheit unter einem anderen Namen angesprochen werden. Die Zuordnung eines lokalen Namens zu dem Namen im Modul erfolgt in der Form: *local_name => module_name*.

```

use workspace, vrVector_local => vrVector    ! local => module

```

Programmbeispiel:

\$ cat sample15.f90

```

module physical_constants
  real :: rSpeedOfLight = 2.99792458E8      ! Meter/Second
  real :: rPlanckConstant = 6.6260755E-34   ! Joule*Second
end module physical_constants

program test
use physical_constants
...
rFrequency=rSpeedOfLight/rWaveLength
...
end program test

```

8.2. Modul als Prozedurbibliothek

- Eine Schnittstellenbeschreibung einer Prozedur, die **innerhalb** eines Moduls nach einer **CONTAINS** Anweisung definiert wird, braucht nicht mehr explizit wiederholt zu werden. Sie muß lediglich als Modulprozedur (*module procedure*) gekennzeichnet werden. (Hiermit entfällt die fehleranfällige Wiederholung von Schnittstellenbeschreibungen.)

```

interface
  module procedure rSinh
end interface

contains
real function rSinh
...

```

- Schnittstellenbeschreibungen von Prozeduren (*interface*) und die zugehörigen Definitionen (*implementation*) sollten in einem separat übersetzbaren Modul zusammengefaßt werden.

```

module special_functions

interface          ! procedure interface(s)
  module procedure rSinh, ...
end interface

contains          ! procedure implementation(s)
real function rSinh(r1)
...
end real function rSinh
...
end module special_functions

```

- In einem Modul können **generische Schnittstellen** vereinbart und definiert werden, d.h. eine Prozedur ist für unterschiedliche Typen von Aktualparametern verwendbar (*generic procedure*).

```

module ...

```

```

interface generic_xx      ! interface for generic procedure
  module procedure real_xx, complex_xx, int_xx, ...
end interface
contains                  ! implementation of specific procedures
real function real_xx(...)
...
end module

```

- Beim Aufrufen einer generischen Prozedur wird anhand der **Aktualparameter** die passende spezifische Prozedur ausgewählt.

```

generic_xx(3)      ⇒    int_xx(3)
generic_xx(3.0)   ⇒    real_xx(3.0)

```

- In einem Modul können **benutzerdefinierte Operatoren** und **Zuweisungen** vereinbart und definiert werden.

```

interface operator (+)      ! overloads operator + for type Point
  module procedure AddPoints
end interface

contains                    ! implementation(s) for operator
type(Point) function AddPoints(...)
...
end function AddPoints

```

Programmbeispiel:

\$ cat module1.f90

```

module generic_swap
interface Swap
  module procedure int_Swap, real_Swap, complex_Swap
end interface
contains
subroutine int_Swap(i1,i2)
integer, intent(inout) :: i1, i2
integer :: iTemp; iTemp=i1; i1=i2; i2=iTemp
end subroutine int_Swap
subroutine real_Swap(r1,r2)
...
end module generic_Swap

```

\$ cat sample16.f90

```

program test
use generic_Swap
integer :: i1=1, i2=2
complex :: c1=(1.0,1.0), c2=(2.0,2.0)
print *, 'Before calling Swap ...'
print *, 'i1 = ', i1, 'i2 = ', i2
call Swap(i1,i2)
print *, 'After calling Swap ...'
print *, 'i1 = ', i1, 'i2 = ', i2
...
call Swap(c1,c2)

```

```
...  
end program test
```

\$ sample16

```
Before calling Swap ...  
i1 = 1 i2 = 2  
After calling Swap ...  
i1 = 2 i2 = 1
```

```
Before calling Swap ...  
c1 = ( 1.0000, 1.0000) c2 = ( 2.0000, 2.0000)  
After calling Swap ...  
c1 = ( 2.0000, 2.0000) c2 = ( 1.0000, 1.0000)
```

8.3. Öffentliche und private Daten

- Objekte, die nur innerhalb eines Moduls bekannt sein sollen, werden durch das Attribut **PRIVATE** gekennzeichnet, solche, die auch außerhalb bekannt sein sollen, durch **PUBLIC**. Die Voreinstellung ist **PUBLIC**.

```
module ...  
real, public :: r1, r2  
real, private : r3, r4  
private :: assignment (=), operator (+)  
...  
end module
```

9. Ein-/Ausgabe

In diesem Kapitel werden folgende Erweiterungen bei der Ein- und Ausgabe behandelt:

- **NAMELIST** Anweisung
- Nicht-vorrückende Ein- und Ausgabe

9.1. NAMELIST Anweisung

- Eine **NAMELIST** Anweisung ermöglicht eine für den Benutzer besser lesbare Aufbereitung von Ein- und Ausgabedaten.

```
real :: r1, r2, r3
namelist /iolist1/ r1, r2, r3
```

- Die **Eingabedaten** für eine Namensliste beginnen mit deren Namen und enthalten dann die Namen der Variablen mit ihrem Wert in beliebiger Reihenfolge. Die Liste wird mit dem Zeichen / beendet.

```
read(*, nml=iolist)
--> &iolist1 r1=5.0, r3=0.0, r2=18.0 /
```

- Die **Ausgabedaten** beginnen ebenfalls mit dem Namen der Namensliste und enthalten die Namen der Variablen gefolgt von ihrem Wert.

```
write (*, nml=iolist)
--> &iolist1 r1=5.0, r3=0.0, r2=18.0 /
```

Programmbeispiel:

```
$ cat sample17.f90
```

```
program test
real :: r1, r2, r3
namelist /iolist1/ r1, r2, r3      ! defines namelist
read(*, nml=iolist1)              ! reads data from stdin
write(*, nml=iolist1)             ! writes data to stdout
end program test
```

```
$ sample17 < echo '&iolist1 r1=5.0, r3=0.0, r2=18.0 /'
```

```
&iolist1 r1=5.0, r2.=18.0, r3=0.0 /
```

9.2. Nicht-vorrückende Ein- und Ausgabe

- Statt eines ganzen Datensatzes können auch einzelne Zeichen gelesen und geschrieben werden. Hierzu ist in der **READ** bzw. **WRITE** Anweisung der Parameter **ADVANCE** auf **NO** zu setzen.

```
WRITE(*,*, ADVANCE='NO') 'Enter an integer: '  
READ (*,*) iInput
```

Die Anzahl der übertragenen Zeichen kann über einen zusätzlichen Parameter **SIZE** ermittelt werden. Ferner kann über eine Marke für den Parameter **EOR** das Erreichen des Datensatzendes behandelt werden.

```
READ (*,'(I3)', ADVANCE='NO', SIZE=iSize, EOR=1000) iInput  
...  
1000 print *, 'End of record reached. SIZE = ', iSize
```

10. Beschreibung des NAG Fortran 90 Compilers

In diesem Kapitel wird als Beispiel für einen Fortran 90 Compiler der Compiler **f90** der Firma NAG in folgenden Abschnitten beschrieben:

- Einführendes Beispiel
- Optionen
- Meldungen
- Module
- Erweiterungen für **make**
- Fehlersuche mit **dbx**

10.1. Einführendes Beispiel

Das Einrichten der Umgebung kann auf Ihrem System ggf. anders verlaufen. Erkundigen Sie sich in diesem Fall bei einem lokalen Experten.

```
$ setup f90                # sets environment variables
$ man f90                  # lists manual page for f90

...(man page)...

$ vi hellow.f90
```

```
program hellow
print *, "Hello World!"
end program
```

```
$ f90 -o hellow hellow.f90
```

```
$ hellow
```

```
Hello World!
```

10.2. Optionen

Übersetzen und Binden können getrennt durchgeführt werden:

```
$ f90 -c pgm.f90          # compiles source file(s)
$ f90 -o pgm pgm.o       # links object file(s)
```

Folgende Optionen stehen u.a. zur Verfügung (siehe **man f90**):

Option	Bedeutung
-c	übersetzt, ohne zu binden.

-C	fügt zusätzlichen Code ein, um Feldgrenzenüberschreitungen zur Laufzeit zu entdecken.
-g	fügt zusätzlichen Code für den Debugger dbx ein.
-fixed	erwartet festes Eingabeformat.
-free	erwartet freies Eingabeformat.
-I <i>pathname</i>	sucht Moduldateien * .mod zusätzlich im Verzeichnis <i>pathname</i> .
-mismatch	gibt im Falle nicht übereinstimmender Formal- und Aktualparameter statt einer Fehlermeldung nur eine Warnung aus.
-o <i>pgm</i>	benennt das ausführbare Programm mit dem Namen <i>pgm</i> .
-O	optimiert den Code.
-P	fügt zusätzlichen Code ein, um die Verwendung von nicht-assoziierten Zeigern aufzudecken.
-pg	fügt zusätzlichen Code für den Profiler gprof ein.
-S	erzeugt C-Quellcode.
-temp= <i>tempdir</i>	erzeugt temporäre Dateien im Verzeichnis <i>tempdir</i> .
-v	erzeugt während der Übersetzung informative Meldungen.

10.3. Warnungen und Fehlermeldungen

Während der Übersetzung erzeugt **£90** Meldungen zum Quellcode. Besonders bei FORTRAN 77 Quelldateien mit veralteten Sprachelementen gibt es i.d.R. eine Flut von Meldungen, die sich auf zulässige, jedoch fragwürdige Sprachkonstrukte beziehen (z.B. nicht explizit definierte Variablen oder im Datentyp nicht übereinstimmende Formal- und Aktualparameter). Jede Meldung wird durch ein Schlüsselwort eingeleitet, die den Typ der Meldung beschreibt:

Schlüsselwort	Typ	Bedeutung
Info	informative Meldung	beschreibt einen Aspekt des Quellcodes, der für den Benutzer interessant sein könnte.
Warning	Warnung	weist auf "verdächtigen" Quellcode hin, der ggf. einen Programmierfehler darstellt.
Extension	Hinweis auf Erweiterung	weist auf Quellcode hin, der nicht Standard FORTRAN 77 konform ist, aber trotzdem übersetzt werden konnte.

Obsolescent	Hinweis auf veraltetes Sprachelement	weist auf veraltete Sprach-elemente im Quellcode hin (siehe Anhang).
Error	Fehler	meldet einen Syntaxfehler (siehe auch Option -mismatch).

Programmbeispiel:

```

REAL FUNCTION COMPAV(SCORE,COUNT)           ! 1st arg: INTEGER
  INTEGER SUM,COUNT,J,SCORE(5)           ! J is never used!
  DO 30 I = 1,COUNT
    SUM = SUM + SCORE(I)
30  CONTINUE
    COMPAV = SUM/COUNT
  END

PROGRAM AVENUM
  PARAMETER(MAXNOS=5)
  INTEGER I, COUNT
  REAL NUMS(MAXNOS), AVG
  COUNT = 0
  DO 80 I = 1,MAXNOS
    READ (5,*,END=100) NUMS(I)
    COUNT = COUNT + 1
80  CONTINUE
100  AVG = COMPAV(NUMS, COUNT)           ! 1st arg: REAL
  END

```

\$ f90 -o avenum avenum.f

Warning: Unused symbol **J** at line ...
detected at END@<end-of-statement>
[f90 continuing despite warning messages]

Error: Argument **SCORE** (no. 1) in reference
to COMPAV from AVENUM has the wrong data type
...(further warnings)...
[f90 error termination]

10.4. Module

- **f90** erzeugt bei Übersetzung einer **Modulquelldatei** *.f90 eine **Moduldatei** *.mod und (nur für Modulprozeduren) eine **Objektdatei** *.o.
- Die Moduldatei *.mod muß während der **Übersetzungsphase**, die Objektdatei mit Modulprozeduren *.o während der **Bindephase** angegeben werden.
- Die Namensgebung ist unterschiedlich, da der Name der Moduldatei *.mod vom **logischen Namen** des Moduls abgeleitet wird, während der Name der Objektdatei vom **Dateinamen** abgeleitet wird.

- Der Suchpfad nach Moduldateien kann über die **Option** `-I` gesetzt werden.

Programmbeispiel:

\$ cat mod_phy1.f90

```

module physics
...
end module physics
    
```

\$ f90 -c mod_phy1.f90

\$ ls

```

mod_phy1.f90          # module source file
physics.mod           # module definition file
mod_phy1.o           # module object file
    
```

```

$ f90 -c pgm.c physics.mod      # compiles pgm.c
$ f90 -o pgm pgm.o mod_phy1.o  # links pgm.o and mod_phy.o
    
```

```

$ pgm                          # starts executable program pgm
    
```

Zusammengefaßt:

Modulquelldatei *.c ↓				
⇒	Moduldatei *.mod	+	Quelldatei pgm.c ↓	Übersetzen
⇒	Objektdatei *.o	+	Objektdatei pgm.o ↓	Binden
			Programm pgm	

10.5. Erweiterungen für make

Das Programm **make** sorgt für eine regelbasierte Erzeugung von Programmen, die sich aus mehreren Quelldateien zusammensetzen.

Das Suffix `.f90` für Fortran 90 Quelldateien ist **make** standardmäßig nicht bekannt, so daß in einer MAKE Beschreibungsdatei (*makefile*) zunächst die **Transformationsregeln** für Fortran 90 Programme definiert werden müssen.

\$ cat makefile.template

```
# Template makefile for f90

# At first we introduce symbolic names for
# the f90 compiler and the default compiler option(s):
F90=f90
FFLAGS=-0

# These rules transform .f90 source files
# to object files or executable files:
.SUFFIXES: .f90
.f90.o:
    $(F90) $(FFLAGS) -c $<
.f90:
    $(F90) $(FFLAGS) -o $@ $<
```

Das folgende Beispiel zeigt eine *makefile* Datei für ein Fortran 90 Programm, das sich aus den Quelldateien `sample.f90`, `sub1.f90` und `sub2.f90` zusammensetzt.

\$ cat makefile

```
F90=f90
FFLAGS=-0

.SUFFIXES: .f90
.f90.o:
    $(F90) $(FFLAGS) -c $<
.f90:
    $(F90) $(FFLAGS) -o $@ $<

OBJS=sub1.o sub2.o

sample: $(OBJS)
    $(F90) $(FFLAGS) -o $@ $@.o $(OBJS)
```

make creates objects first, then links objects to executable

\$ make

```
f90 -O -c sample.f90
f90 -O -c sub1.f90
f90 -O -c sub2.f90
f90 -o sample sample.o sub1.o sub2.o
```

10.6. Debugging

Fortran 90 Programme, die mit der Option `-g` übersetzt worden sind, können mit dem Debugger **dbx** analysiert werden.

Folgende Punkte sind zu beachten:

- Die Prozedurnamen sind im Debugger **dbx** in Kleinbuchstaben und mit einem angehängtes Zeichen **_** einzugeben, z.B. entspricht dem Fortran 90 Namen **AddPoints** der Name **add-points_**.

```
$ dbx pgm
(dbx) stop in addPoints_
(dbx) run
...
(dbx) quit
```

- **Modulprozeduren** erhalten einen Namen, der sich aus dem Modulnamen, der Zeichenkette **_MP_** und dem Prozedurnamen zusammensetzt, hier z.B. der Modulname **module1** und der Prozedurname **rsinh**.

```
module1_MP_rsinh_
```

- **f90** erzeugt als temporären Zwischencode C, der im Anschluß automatisch vom C Compiler übersetzt wird. Der Debugger **dbx** bezieht sich in seiner Diagnose also auf eine C Quelle. Falls die C-Quelle benötigt wird, erzeugt **f90** über die Option **-s** explizit ein Listing der C-Quelle.

```
$ f90 -S pgm.f90          # creates C-source permanently
$ vi pgm.c                # loads C-source
```

- Die **dbx** Funktionen sind über das **help** Kommando abrufbar.

```
(dbx) help
```

11. Anhang

In diesem Anhang werden folgende Auflistungen und Tabellen zur Verfügung gestellt:

- Auflistung der veralteten Sprachelemente
- Tabelle der Schlüsselwörter und Symbole
- Tabelle der vordefinierten Prozeduren

11.1. Veraltete Sprachelemente

Folgende Sprachelemente von FORTRAN 77 gelten für Fortran 90 Compiler als **veraltet** (*obsolescent*):

- arithmetisches IF
- Variablen vom Typ **REAL** oder **DOUBLE PRECISION** als Schleifenindex
- **DO** Schleifen, die von einer gemeinsamen Anweisung beendet werden
- **DO** Schleifen, die nicht auf **END DO** oder **CONTINUE** enden
- Sprünge von außerhalb auf eine **END IF** Anweisung
- Rücksprung aus einer Prozedur auf eine Marke (**RETURN i**)
- **PAUSE** Anweisung
- **ASSIGN** Anweisung, *assigned* **GOTO** und *assigned* **FORMAT**
- Hollerith Format (H Format)

Bemerkungen:

- Fortran 90 Compiler kennzeichnen die genannten Sprachelemente als veraltet (*obsolescent*).
- Im nächsten Fortran Standard werden sie voraussichtlich nicht mehr enthalten sein.

11.2. Schlüsselwörter und Symbole

(/.../)	do	open
.and.	double	opened
.eq., ==	e	operator
.eqv.	else	optional
.false.	elsewhere	out
.ge., >=	en	p
.gt., >	end	pad
.le., <=	endfile	parameter
.lt., <	entry	pause
.ne., /=	eor	Pointer
.neqv.	equivalence	position
.not.	err	precision
.or.	es	print
.true.	exist	private
!	exit	procedure
%	external	program
+, -, /, *, **	f	public
//	file	read
:	fmt	readwrite
::	form	real
=	format	rec
=>	formatted	recl
a	function	recursive
access	g	result
action	go	return
advance	h	rewind
allocatable	i	s
allocate	if	save
assign	implicit	select
assignment	in	sequence
b	inout	sequential
backspace	inquire	size
blank	integer	sp
block	intent	ss
bn	interface	stat
bz	intrinsic	status
c	iolength	stop
call	iostat	subroutine
case	kind	t
character	l	target
close	len	then
common	logical	tl
complex	module	to
contains	name	tr
continue	named	type
cycle	namelist	unformatted
d	nextrec	unit
data	nml	use
deallocate	none	where
default	nullify	while
delim	number	write
dimension	o	x
direct	only	z

11.3. Vordefinierte Prozeduren (Unterprogramme und Funktionen)

abs	len_trim	transfer
achar	lge	transpose
acos	lgt	trim
adjustl	lle	ubound
adjustr	llt	unpack
aimag	log	verify
aint	log10	alog*
all	logical	alog10*
allocated	macval	amax0*
anint	matmul	amax1*
any	max	amin0*
asin	maxexponent	amin1*
associated	maxloc	amod*
atan	merge	cabs*
atan2	min	ccos*
bit_size	minexponent	cexp*
btest	minloc	clog*
ceiling	minval	csin*
char	mod	csqrt*
cmplx	modulo	dabs*
conj	mvbitrs	dacos*
cos	nearest	dasin*
cosh	nint	datan*
count	not	datan2*
cshift	pack	dcos*
date_and_time	precision	dcosh*
dbble	product	ddim*
digits	radix	dexp*
dim	random	dint*
dot_product	random_seed	dlog*
dprod	range	dloglo*
eoshift	real	omax1*
epsilon	repeat	omin1*
exp	reshape	omod*
exponent	rrspacing	dnint*
floor	scale	dsign*
fraction	scan	dsinh*
huge	selected_int_kind	dsqrt*
iachar	selected_real_kind	dtan*
iand	set_exponent	dtanh*
ibclr	shape	float*
ibits	sign	iabs*
ibset	sin	idim*
ichar	sinh	idint*
ieor	size	idnint*
index	spacing	ifix
int	spread	isign
ior	sqrt	max0*
ishft	sum	max1*
ishftc	system_clock	min0*
kind	tan	min1*
lbound	tanh	sngl*
len	tiny	* = wegen Abwärtskompatibilität

11.4. Beispielprogramme

Fortran 90 Beispielprogramme befinden sich u.a. auf unserem *anonymous FTP Server*:

Name: ftp.rz.Uni-Osnabrueck.DE

Datei: pub/reports/rz/f90_samples.tar.z

Nach dem Dekomprimieren und Entpacken (`uncompress ...`, `tar xvf ...`) gibt die Datei `README` einen Überblick über die einzelnen Dateien.

Eine Diskette mit den Beispielprogrammen ist auf Wunsch auch beim Autor erhältlich.

Für Umsteiger von FORTRAN 77 auf Fortran 90 empfehle ich das Buch *Fortran 90 Explained* [1], das zahlreiche Programmbeispiele enthält. Als Referenz empfehle ich das preiswerte Referenzhandbuch des RRZN [6], ebenfalls mit kurzen Code-Fragmenten.

11.5. Literaturverzeichnis

Bücher:

- [1] **Fortran 90 Explained**,
Metcalf. M.; Reid, J.,
Oxford Univ. Press,
Oxford (1990)

- [2] **Fortran 90 Referenzhandbuch**,
Gehrke, Wilhelm,
Hanser-Verlag, ISBN 3-446-16321-2
München (1991)

- [3] **Information technology - Programming languages - Fortran**,
ISO/IEC 1539:1991(E),
Beuth-Verlag,
Berlin (1991)

- [4] **Fortran 90 - eine informelle Einführung**
Heisterkamp. M, Rothhäuser K.-H.,
B.I. Wissenschaftsverlag,
Mannheim, Wien, Zürich (1991)

- [5] **Programmers Guide to Fortran 90**
Brainer W., Goldber C., Adams J.,
ISBN 0-07-000248-7
Mc Graw Hill

- [6] **Einführung in Fortran 90**
Gehrke, Wilhelm,
RRZN Hannover (1992)

- [7] **NAG f90 Compiler (Unix)**,
NAG Ltd.
Oxford (1992)

Artikel:

- [1] **Fortran 90 - der neue Standard,**
Rotthäuser, K.-H.,
PIK 14 (1991) 1, S.35ff,
K. G. Saur Verlag,
München (1991)

- [2] **Summary of Revised Fortran,**
Meissner, L. P.,
ACM SIGPLAN Fortran Forum,
Vol. 8, Mai 90, S15 ff und
Vol. 9, März 91, S31 ff

- [3] **Unterlagen zum Kurs "Fortran 90 Präsentation" der GMD,**
Rotthäuser, K.-H.,
St. Augustin, 04.11.1991

- [4] **Fortran 90: Neuer Entwurf zur Diskussion freigegeben,**
Rotthäuser, K.-H.,
COMPUTERWOCHE, Nr. 37-39,
München (1990)