Wissenschaftliches Programmieren in Fortran

Meik Hellmund

hellmund@math.uni-leipzig.de

www.math.uni-leipzig.de/∼hellmund/fortran.html

A316, Tel. 9732155

- Gestaltung: anfangs Vorlesungen, später (Juni?) gemeinsame Arbeit am Rechner
- Ziel: Vorbereitung auf das numerische Praktikum
- Scheinvergabe (2 SWS) für das Lösen von Programmieraufgaben
- Vorlesung kann nicht als Prüfungsthema in mündlichen Prüfungen verwendet werden

- Fortran-Grundlagen
- Verwendung der Grafikbibliothek `Vogle`
- Verwendung des Plotprogramms `Gnuplot`
- Fließkommazahlen, Rundungsfehler, Stabilität ...
- Arbeit unter Linux mit Terminalfenster, Editor und Compiler
- keine IDE – Integrated Development Environment
  (z.B. Visual C++/Fortran, Delphi, Eclipse unter Windows;
  Anjuta, Eclipse, Sun Studio, KDevelop unter Linux)

Warum Fortran?

- **Felder (Vektoren, Matrizen, ...)** sind Teil der Sprache
- hochoptimierend (Schleifen, Konzept reiner Funktionen, keine Aliasing-Probleme) $\Rightarrow$ **sehr schneller Code**
- Sauberer Umgang mit Fließkommazahlen, saubere Optimierungen
- Zahlreiche Bibliotheken, insbesondere zur Numerik (Lapack,...)
- **herstellerunabhängige Standardisierung**
- Höchstleistungsrechnen, große numerische Probleme: **"number crunching"**
- Wird in der Industrie nach wie vor eingesetzt
- Flache Lernkurve

Warum nicht Fortran?

- hardwarenahe Programme, Webapplikationen, Datenbanken, ego shooter games,...

Geschichte von Fortran

1957 John Backus von IBM entwickelt eine Sprache zur "**For**mula **tran**slation"

1966 ANSI (American National Standards Institute) veröffentlicht den ersten Standard für eine Programmiersprache überhaupt: **Fortran 66**

1978 ANSI Standard X3.9-1978 definiert **Fortran 77**: strukturierte Programmierung

1991 **Fortran 90** wird von ANSI und ISO als Standard veröffentlicht: Zeiger, dynamische Speicherverwaltung, Modulen und Interfaces, Vektorsprache für Felder,...

1997 Eine leichte Erweiterung und Überarbeitung, **Fortran 95**, wird als Standard verabschiedet.

2004 Revision des Standards **Fortran 2003**: Objektorientierung, Vererbung, parametrisierte Typen, definierte Schnittstelle zu C-Programmen,...

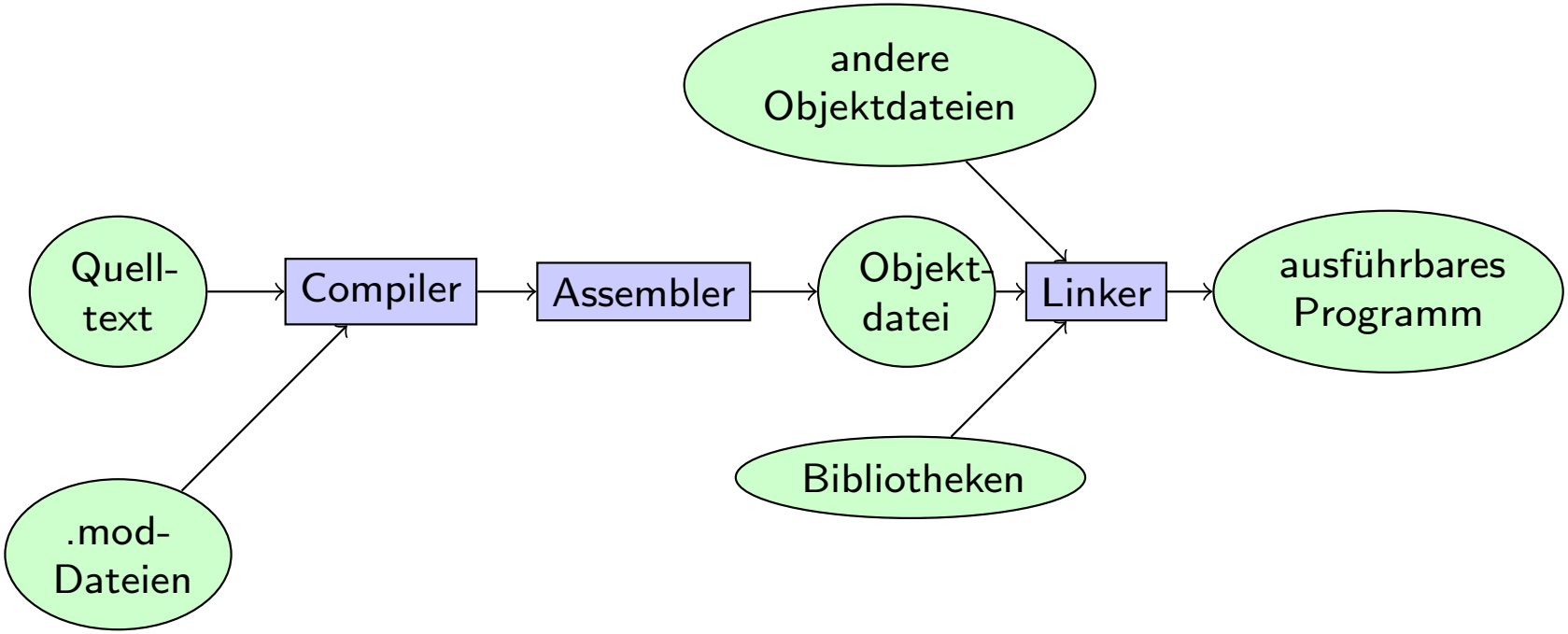
2010 Überarbeitung (**Fortran 2008**) wird von der Fortran working group verabschiedet

```
1  program bsp1
2      implicit none
3      integer :: n
4      logical :: IsPerfect
5      do
6          print *, 'Bitte ganze Zahl eingeben, "0"  fuer Abbruch:'
7          read  *, n
8          if(n == 0) exit
9          if( IsPerfect(n) ) then
10             print *, "Treffer!"
11         else
12             print *, 'Das war leider nicht perfekt. :-('
13         endif
14     end do
15 end program
16
17 logical function IsPerfect(x)
18     implicit none
19     integer :: i, x, m
20     m = 0
21     IsPerfect = .false.
22     do i = 1, x/2
23         if (mod(x, i) == 0) m = m + i
24     end do
25     if(m == x) IsPerfect = .true.
26 end function IsPerfect
```

Fortran ist ...

- **imperativ (prozedural)**
  Programm = Folge von Anweisungen (C, C++, Java, ...)
  ($\Leftrightarrow$ funktional: Haskell, Ocaml, Lisp)
- **stark typisiert:** jede Variable hat einen Typ (C, C++, Java, ...)
  ($\Leftrightarrow$ schwach typisiert: Javascript, PHP, Perl)
- **statisch typisiert:** Typ einer Variablen liegt fest
  ($\Leftrightarrow$ dynamisch typisiert: Javascript, Python)
- **Compilersprache**
  ($\Leftrightarrow$ Interpretersprachen: Javascript, Perl, Python)

Vom Quelltext zum ausführbaren Programm



`gfortran prog1.f90`
erzeugt die ausführbare Datei `a.out`. Programm starten mit `./a.out`

```
gfortran -O2 -Wall -fbounds-check -std=f2003 -pedantic -o prog2 prog2.f90
```

Optimierungsstufe 2, warne bei allen Unsauberkeiten, prüfe zur Laufzeit die Gültigkeit von Indizes, verwende Fortran2003-Standard, sei pedantisch, erzeuge ein ausführbares Programm namens `prog2`.
Programm starten mit `./prog2`

Grundlagen der Syntax

Grobstruktur

- Programm = Menge von Funktionen, die in einer oder mehreren Quelltextdateien (source code files) untergebracht sind
- Zusätzliche Funktionen können durch Programmbibliotheken (libraries) bereitgestellt werden.
- Es gibt eine ausgezeichnete Funktion, die mit der Anweisung `program <name>` eingeleitet wird. Diese wird beim Programmstart vom Betriebssystem aufgerufen.
- Funktionen können in Modulen zusammengefaßt werden.
- Funktionen bestehen aus Deklarationen gefolgt von den (ausführbaren) Anweisungen.
- Funktionen, die keinen Wert zurückliefern, heißen in Fortran `subroutine`.

Grundlagen der Syntax

Namen von Variablen und Funktionen

- erlaubte Zeichen: lateinische Buchstaben `a-z A-Z`, Unterstrich `_`, Ziffern `0-9`
- erstes Zeichen muß Buchstabe sein
- bis zu 63 Zeichen lang
- Groß- und Kleinbuchstaben werden nicht unterschieden!
  `Nmax` und `NMAX` ist dieselbe Variable
- zulässig: `i, x, x23, DieUnbekannteZahl, neuer_Wert`
  unzulässig: `3achsen, zähler, A#b, $this_is_not_Perl`

Fortran ist zeilenorientiert

- Im Normalfall enthält jede Zeile genau eine Anweisung.
- Wenn eine Zeile mit einem & endet, wird die Anweisung auf der nächsten Zeile fortgesetzt. Eine Anweisung kann bis zu 256 Zeilen lang sein.
- Eine Zeile kann bis zu 132 Zeichen lang sein.
- Mehrere Anweisungen pro Zeile sind möglich und müssen mit Semikolon getrennt werden.

Kommentare

Kommentare erstrecken sich von einem ! bis zum Zeilenende:

```
!**************************************************
! Das Folgende deklariert n als ganze Zahl
!**************************************************

  integer :: n       ! jetzt ist n eine ganze Zahl
  n = 0              ! jetzt ist n gleich Null
```

Datentypen

- **Datentyp (Typ) = Menge von Werten und darauf definierten Operationen**
- konkreter: Typ eines Objektes = Vereinbarung, wieviel Speicherplatz es benötigt und wie die Bitmuster in diesem Speicherbereich zu interpretieren sind.
- Beispiel: `integer` auf einer 32bit-CPU:
  Werte: $-2\,147\,483\,648 \ldots +2\,147\,483\,647$ $\quad$ $(-2^{31} \ldots 2^{31}-1)$
  Operationen: Arithmetik ganzer Zahlen **modulo** 2^{32}
  $2\,147\,483\,640 + 10 = -2\,147\,483\,646$
- Neben den **intrinsischen** Typen kann eine Programmiersprache die Definition eigener, **abgeleiteter** Typen erlauben.
- Fortran hat strenges und statisches Typensystem: prinzipiell hat jede Variable und jede Funktion einen Typ und dieser kann während des Programmablaufs nicht verändert werden.

Intrinsische Datentypen

- Ganze Zahlen
- Fließkommazahlen (Gleitkommazahlen)
- komplexe (Fließkomma-) Zahlen
- logische Variablen
- Zeichen und Zeichenketten
- Zeiger

Typ	Bits	Bytes	Bereich
character	8	1	(-128 . . . 127) a-z,A-Z,0-9,!+-_?@()...
integer(kind=2)	16	2	-32 768 . . . 32 767
integer	32	4	-2 147 483 648 . . . 2 147 483 647
integer(kind=8)	64	8	$-2^{63} \ldots 2^{63} - 1$
real	32	4	$\pm 1.1 \times 10^{-38} \ldots \pm 3.4 \times 10^{38}$
real(kind=8)	64	8	$\pm 2.3 \times 10^{-308} \ldots \pm 1.8 \times 10^{308}$

(`kind`-Parameter für gfortran-Compiler auf Intels 32-Bit-Architektur ia32)

Ganze Zahlen

- Positive ganze Zahlen werden in Binärdarstellung dargestellt:

$$\begin{aligned} 0000\cdots0000 &= 0 \\ 0000\cdots0001 &= 1 \\ 0000\cdots0010 &= 2 \\ &\cdots \\ 0111\cdots1111 &= 2^{N-1}-1 \end{aligned}$$

- Für negative ganze Zahlen verwenden zahlreiche Prozessoren, darunter die Intel-Familie, die Zweierkomplement-Darstellung: $-x$ wird dargestellt, indem das Bitmuster von x invertiert und anschließend 1 addiert wird:

$$\begin{aligned} 1111\cdots1111 &= -1 \\ 1111\cdots1110 &= -2 \\ &\cdots \\ 1000\cdots0001 &= -2^{N-1}-1 \\ 1000\cdots0000 &= -2^{N-1} \end{aligned}$$

Damit ist das höchstwertige Bit Vorzeichenbit in dem Sinne, daß seine Anwesenheit negative Zahlen charakterisiert. Sein Setzen oder Löschen entspricht allerdings nicht der Transformation $x \to -x$.

Ganze Zahlen in Fortran

- Datentyp `integer`, genauer spezifiziert durch `kind`-Parameter.
- GNU Fortran Compiler `gfortran` und Intel Compiler `ifort`:

`integer(kind=1)`	1 Byte
`integer(kind=2)`	2 Byte
`integer, integer(kind=4)`	4 Byte
`integer(kind=8)`	8 Byte

- Die `kind`-Werte und ihre Bedeutung sind vom Standard nicht vorgeschrieben, sie sind **implementationsabhängig**.
- Die Funktion `selected_int_kind(p)` liefert kleinsten `kind`, der den Bereich $-10^p < n < 10^p$ umfaßt — oder den Wert `-1`.
  Beispiel:

```
integer, parameter :: long = selected_int_kind(10)
integer(kind=long) :: n,m,l
```

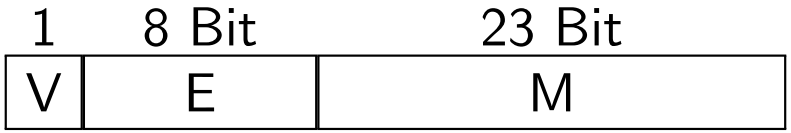
- Ganzzahlige literale Konstanten sind vom Typ `integer`, `kind`-Parameter kann mit Unterstrich angehängt werden:
  `42, 445556_8, 31415976354_long`

Fließkommazahlen

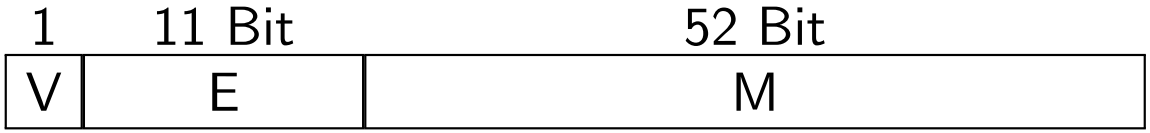
- viel größerer Zahlenbereich
- mit begrenzter Genauigkeit ("Anzahl der gültigen Ziffern")
- $x = (-1)^V \times 2^E \times M$
  E Exponent, M Mantisse $(0 \leq M < 1)$, V Vorzeichenbit
- Standard IEEE 754

Die Intel-32Bit-Architektur (IA32) kennt drei Formate:

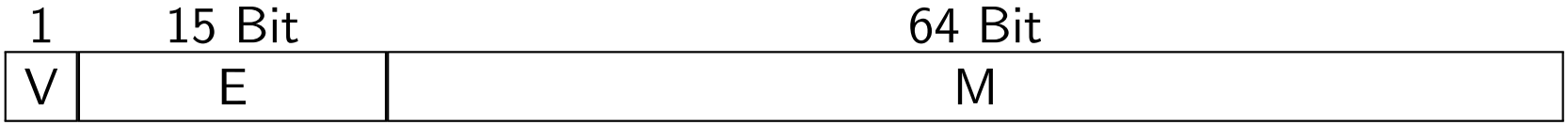
Einfach langes Format: 32 Bit



Doppelt langes Format: 64 Bit



Erweitertes Format: 80 Bit



	einfach	doppelt	erweitert
Exponentenlänge E (Bits)	8	11	15
kleinster Exponent e_{min}	-125	-1021	-16382
größter Exponent e_{max}	128	1024	16384
Mantissenlänge	24	53	64
betragsmäßig kleinste normalisierte Zahl größer Null	1.18×10^{-38}	2.23×10^{-308}	3.36×10^{-4932}
größte darstellbare Zahl	3.40×10^{38}	1.79×10^{308}	1.19×10^{4932}
kleinste Zahl ϵ, für die $1+\epsilon \neq 1$	1.19×10^{-7}	2.22×10^{-16}	1.08×10^{-19}

- Der Exponent wird als vorzeichenlose Zahl kodiert, von der eine feste Zahl (bias) abgezogen wird.
- Die Exponenten überdecken nicht den vollen Wertebereich von E Bits. Das Bitmuster "111...11" ($e_{max}+1$) im Exponentenfeld ist reserviert, um die speziellen Werte `NaN` ("Not a Number", Mantisse ungleich 0) und $\pm\infty$ (Mantisse=0, V=0 oder 1) zu kodieren. Bei einfach- und doppeltgenauen Zahlen ist das Bitmuster "000...00" ($e_{min}-1$) reserviert für denormalisierte Zahlen.
- Denormalisierte Zahlen werden nur dann verwendet, wenn sie betragsmäßig so klein sind, daß eine normalisierte Darstellung nicht mehr möglich ist. Die kleinste normalisierte einfach genaue Zahl hat die Mantisse .1000...0 binär $(=2^{-1}=0.5)$ und den Exponenten $e_{min}=-125$, ist also gleich $2^{-125}\times 2^{-1}=2^{-126}\approx 1.18\times 10^{-38}$. Wird diese Zahl nun z.B. durch 8 geteilt, so ist das Ergebnis nur noch denormalisiert mit der Mantisse .00010...0 binär darstellbar. Aufgrund des impliziten ersten Bits muß man denormalisierte Zahlen durch einen speziellen Wert im Exponentenfeld kennzeichnen. (Interpretiert werden sie natürlich mit dem Exponenten $e=e_{min}$.)

Fließkommazahlen in Fortran

`real, real(kind=4)`	4 Byte
`real(kind=8), double precision`	8 Byte
`real(kind=10)`	10 Byte

- Die `kind`-Werte und ihre Bedeutung sind vom Standard nicht vorgeschrieben, sie sind implementationsabhängig.
- Die Funktion `selected_real_kind(p, r)` liefert kleinsten `kind` mit $\geq p$ gültigen Stellen und Exponentenbereich $\geq r$.
  Beispiel:

  ```
  integer, parameter :: k = selected_real_kind(15)
  real(kind=k) :: x,y,z
  ```

- literale Konstanten mit Dezimalpunkt sind vom Typ `real`:
  `2., 3.1415, 123.0433e-7, -1.e12`
  `kind`-Parameter kann mit Unterstrich angehängt werden: `4._8`
  `1.d12, 234.5555433d0` sind vom Typ `real(kind=8)`

Implizite Deklarationen in Fortran

- Wenn eine Variable nicht deklariert ist, wird ihr vom Fortran-Compiler ein Standardtyp zugeordnet:
  - `integer`, wenn Name mit `i,j,k,l,m,n` anfängt
  - `real`, wenn Name mit `a-h,o-z` anfängt
- Dieses Feature führt z.B. dazu, daß der Compiler keine Fehlermeldung ausgibt, wenn man sich bei einem Variablennamen verschreibt. Man hat dann implizit einen neue Variable eingeführt.
- Es läßt sich durch die Anweisung

  ```
  implicit none
  ```

  am Anfang jeder Funktion/Subroutine abschalten.
  Dann erzeugt jede undeklarierte Variable eine Fehlermeldung.

Komplexe Zahlen

Komplexe Zahlen sind Paare (Re, Im) von Fließkommazahlen

`complex, complex(kind=4)`	8 Byte
`complex(kind=8)`	16 Byte
`complex(kind=10)`	20 Byte

```
complex(kind=8) :: a,b,c
real(kind=8) :: xabs, n=3.14

a = (10, -17)
b = sqrt(a + 2*conjg(a) + sin(a) + cos(a))
                    ! Wurzel, Konjugierte, sin, cos
c = exp( cmplx(2., n) ) + log(a)
                    ! exp, log
xabs = abs(c) + real(a) + aimag(a)
                    ! Betrag, Realteil, Imaginaerteil
```

Konstanten

- Variablen können durch das `parameter`-Atribut als konstant deklariert werden.
- Jeder Versuch, diesen Variablen nach der Initialisierung einen anderen Wert zuzuweisen, führt zu einer Compiler-Fehlermeldung.
- Sie müssen bei der Deklaration initialisiert werden.

```
real(8), parameter :: Pi = 4 * atan(1._8)
```

- Manche Variable müssen als Konstante deklariert sein, z.B. bei der Verwendung einer Variable als `kind`-Parameter:

```
integer, parameter :: D = 8
real(kind = D) :: a,b,x,y
```

(Vorteil: Änderungen nur an einer Stelle nötig)

Intrinsische numerische Funktionen (Auswahl)

- meist generisch: Typ/kind des Arguments bestimmt Typ/kind des Ergebnisses
- Winkel im Bogenmaß
- `sin, cos, tan, asin, acos, atan, atan2(y,x), sinh, cosh, tanh, abs, sqrt, exp, log, log10`
- `max(a1,a2 [,a3,...]), min(a1,a2 [,a3,...])`
- `ceiling(a)` kleinste ganze Zahl $\geq a$.
- `floor(a)` größte ganze Zahl $\leq a$
- `int(a)` zur Null hin gerundete ganze Zahl
- `nint(a)` rundet zur nächsten ganzen Zahl
- `mod(a,b)` Rest von a modulo b; $mod(a,b) = a - int(a/b) * b$
- `modulo(a,b)` $= a - floor(a/b) * b$
- `sign(a,b)` $= |a| \ \frac{b}{|b|}$

Logische Variablen

```
logical :: flag, a, b

a = .true.
b = .false.
flag = x <= y
a = flag .and. (.not. b)
if (flag) then     ! aequivalent zu: if(x<=y) then ...
   ....
```

- Logische Variablen können die Werte `.true.` und `.false.` haben.
- Sie können das Ergebnis von Vergleichoperatoren `==`,`/=`, `<`, `<=`, `>`, `>=` sein.
- Sie können als Argumente von `if()` verwendet werden.
- Sei können durch die logischen Operationen `.not.`, `.and.`, `.or.`, `.eqv.`, `.neqv.` verknüpft werden.
- `.eqv.` logische Gleichheit, `.neqv.` logische Ungleichheit (exklusives oder)

Zeichen und Zeichenketten (Strings)

```
character :: c                  ! ein Byte (Zeichen)
character(len=10) :: name       ! (max) 10 Bytes (Zeichen)

c = 'X'
name = 'Hans'                   ! enthaelt rechts 6 Leerzeichen
name = 'Hans Hägr'              ! '...' und "..."  verwendbar
name = "Hans' Auto"             ! Das andere Zeichen darf vorkommen
name = 'Hans'' Auto'            ! oder es muss verdoppelt werden.
name = "Hans' Autotür"          ! Bei Zuweisungen zu langer Strings
                                ! wird rechts abgeschnitten.
```

- Nicht-ASCII-Zeichen (wie äßø¿) aus dem ISO-8859-1 (Latin-1) Zeichensatz können erlaubt sein. (Die richtige Methode für Unicode: `character(kind=selected_char_kind('ISO_10646'),len=..)` wird später behandelt.)
- Steuerzeichen (wie z.B. ein Zeilenumbruch) sind nicht erlaubt.
- Lange Strings können mit einem & am Zeilenende **und** am Anfang der Fortsetzungszeile eingegeben werden.

```
LString = "Alle meine Entchen schwimmen &
          &auf dem See"
```

Dies wird zu einer Zeile zusammengefügt.

Substrings und Verkettung

```
character(3):: s1='lam'
character(6):: s2='pensch'
character(15) :: x

x = s1 // s2 // 'irm'    ! x = "lampenschirm    "
x = 'M' // s2(2:6)       ! x = "Mensch          "
s2(1:1) = "M"            ! s2= "Mensch"
```

- Eine Zeichenkette `character(len=10)::str` ist etwas anderes als ein Feld von Zeichen `character, dimension(10)::str2`.
- Spezielle Syntax `str(i:j)` zum Ansprechen von Substrings
- Zeichen an `i`-ter Stelle: `str(i:i)`
- Strings in Fortran haben eine feste Länge. Wenn in einer Zuweisung `lhs = rhs` die rechte Seite zu kurz ist, wird die linke Seite mit Leerzeichen aufgefüllt. Wenn die rechte Seite zu lang ist, wird am Ende abgeschnitten.

- Das funktioniert nicht wie gewünscht:

```
character(10) :: s1 = "Das", s2=" ist", s3
s3 = s1 // s2
```

Die rechte Seite bildet den String "das␣␣␣␣␣␣␣ist␣␣␣␣␣␣␣" und bei der Zuweisung an die linke Seite wird er zu "das␣␣␣␣␣␣␣" gekürzt.

```
s3 = trim(s1) // s2
print *, trim(s3), "␣besser."
```

- Vor einer Zuweisung ist ein String, wie jede andere Variable auch, undefiniert und sollte nicht an Funktionen übergeben oder ausgegeben werden. Bei der Zuweisung an einen Teilstring kann der Gesamtstring undefiniert bleiben:

```
character(5):: s          ! Wert undefiniert
s(1:4) = "ab"             ! s noch undef.
```

Jetzt ist **s = "ab␣␣☒"**, das letzte Byte (und damit s) ist noch undefiniert.

Vergleich von Strings

```
character(8) :: s1 = "Auto"      ! "Auto____"
character(4) :: s2 = "Auto"

if(s1 == s2) print *, "Strings sind gleich!"
```

Vor einem Vergleich wird der kürzere Operand mit Leerzeichen bis zur Länge des längeren aufgefüllt. Daher hat `s1 == s2` den Wert `.true.`

Intrinsische Funktionen für Zeichen und Strings (Auswahl)

- `i = iachar(c); c = achar(i)`
  Umwandlung zwischen ganzer Zahl $0 \leq i \leq 127$ und Zeichen c gemäß ASCII-Tabelle: "0"..."9" $\Leftrightarrow$ 48...57; "A"..."Z" $\Leftrightarrow$ 65...90; "a"..."z" $\Leftrightarrow$ 97...122; " " $\Leftrightarrow$ 32
- `i=len(string); str=trim(string); i=len_trim(string)`
  Länge des Strings, String ohne Leerzeichen am Ende; Länge ohne Leerzeichen am Ende
- `i = index(String, Substring)`
  Startposition von Substring in String: `index("Fortran", "ort")` ist 2. Ergebnis 0, wenn Substring nicht in String enthalten ist.
- `i = scan(String1, String2)`
  Position des ersten Zeichens von String1, welches auch in String2 enthalten ist oder 0 wenn String1 keines der Zeichen von String2 enthält.

Ablaufsteuerung

- Verzweigungen: `if ... then ... else ... end if`
- Mehrfachverzweigungen: `select case ...`
- Schleifen: `do ... end do`, `cycle`, `exit`, `do while ...`
- Sprunganweisungen: `goto`, `stop`

Verzweigungen

Eine Anweisung:

```
if( x>3 .or. y /= 0 ) z=3
```

Blockform:

```
if (x > 0) then
    y = sqrt(x)
    z = 0
end if

if (i > 5 .and. j == 0) then
    y = 0
else
    y = 3.3
    z = 0
end if
```

```
if(i>5 .and. j==0) then
    y = 0
else if ( i == 0 ) then
    y = 5
else if ( i == 3 ) then
    y = 7
else
    y = 3.3
    z = 0
end if
```

Mehrfachverzweigungen

```
select case (i)
      case (-3:-1)
                  x = 0
      case (0)
                  x = 1
                  y = 3
      case (1:2)
                  x = 3
      case (-4,3:10,17)
                  x = 9
      case default
                  x = 4
end select
```

```
character :: key
...
key = getkey()
select case (key)
      case ('q')
                  stop
      case ('a')
                  x = 1
                  call brptl(x, y)
      case ('s','w')
                  x = 3
                  call bghr(key, x)
end select
```

Das Argument von `select case ()` kann vom Typ `character`, `integer` oder `logical` sein.
Die Argumente von `case ()` sind Konstanten, Aufzählungen von Konstanten oder Bereiche.
Jeder Wert darf nur in maximal einem `case()` auftreten (keine Überlappung).

Schleifen

```
sum = 0
do i = 1, 10
     sum = sum + i**2
end do
```

```
n = 100
m = 2
do i = n * m, m, -3
     sum = sum + a(i)
end do
```

- Der Schleifenzähler ist eine ganzzahlige Variable.
- Anfangswert, Endwert (und evtl. Schrittweite) sind ganzzahlige Ausdrücke.
- Schleifenzähler darf innerhalb der Schleife nicht modifiziert werden.
- Anzahl der Durchläufe wird vor dem ersten Durchlauf berechnet zu `max( (Endwrt - Anfangswrt + Schrittwte)/Schrittwte, 0)`
- `do`
  `...`
  `end do` ist eine Endlosschleife. Sie muß mit `exit` oder `goto` verlassen werden.

Sonderform do while

```
do while (eps > 1.e-10)
   ....
end do
```

≡

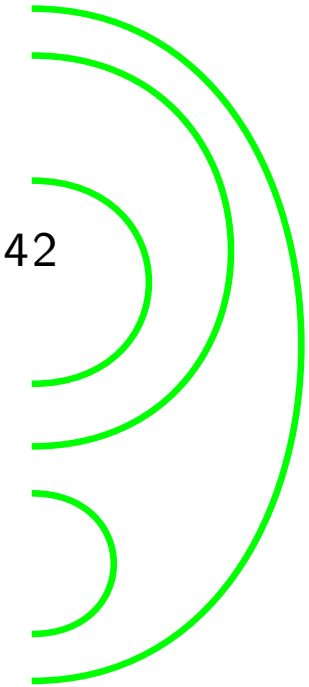
```
do
if (.not. eps > 1.e-10) exit
   ....
end do
```

Do-Blöcke können Namen tragen, auf die eine exit- oder cycle-Anweisung Bezug nimmt. Sonst gilt diese Anweisung für den innersten do-Block.

```
L1: do i = 1, 100
       z = log(i)
       do j = 2, 200, 2
          if (a(i, j) < 0) cycle  !start next iteration
          if ( func1(j) > 20) exit !leave inner loop
          if (a(i,j) = 3 ) exit L1 !leave outer loop
          a(i, j) = i * sin(j)**2
       end do
    end do
```

- Die `stop`-Anweisung führt zum Programmende.
- `goto`-Sprünge in einen `do/if/select`-Block **hinein** sind nicht erlaubt.
- Sprungmarken (labels) sind Nummern aus bis zu 5 Ziffern, die vor einer Anweisung stehen. Nicht mit Blocknamen verwechseln!
- `end (if, do,...)` schließt immer den innersten offenen Block.

```
    if ( x > .3 ) then
       do i = 1, 100
          z = z + log(i)
          do j = 2, 200
             if(a(i,j)<0) goto 42
             a(i,j) = i + j
          end do
       end do
42     if ( z > 180) then
          z = sum(a)
       end if
    end if
```



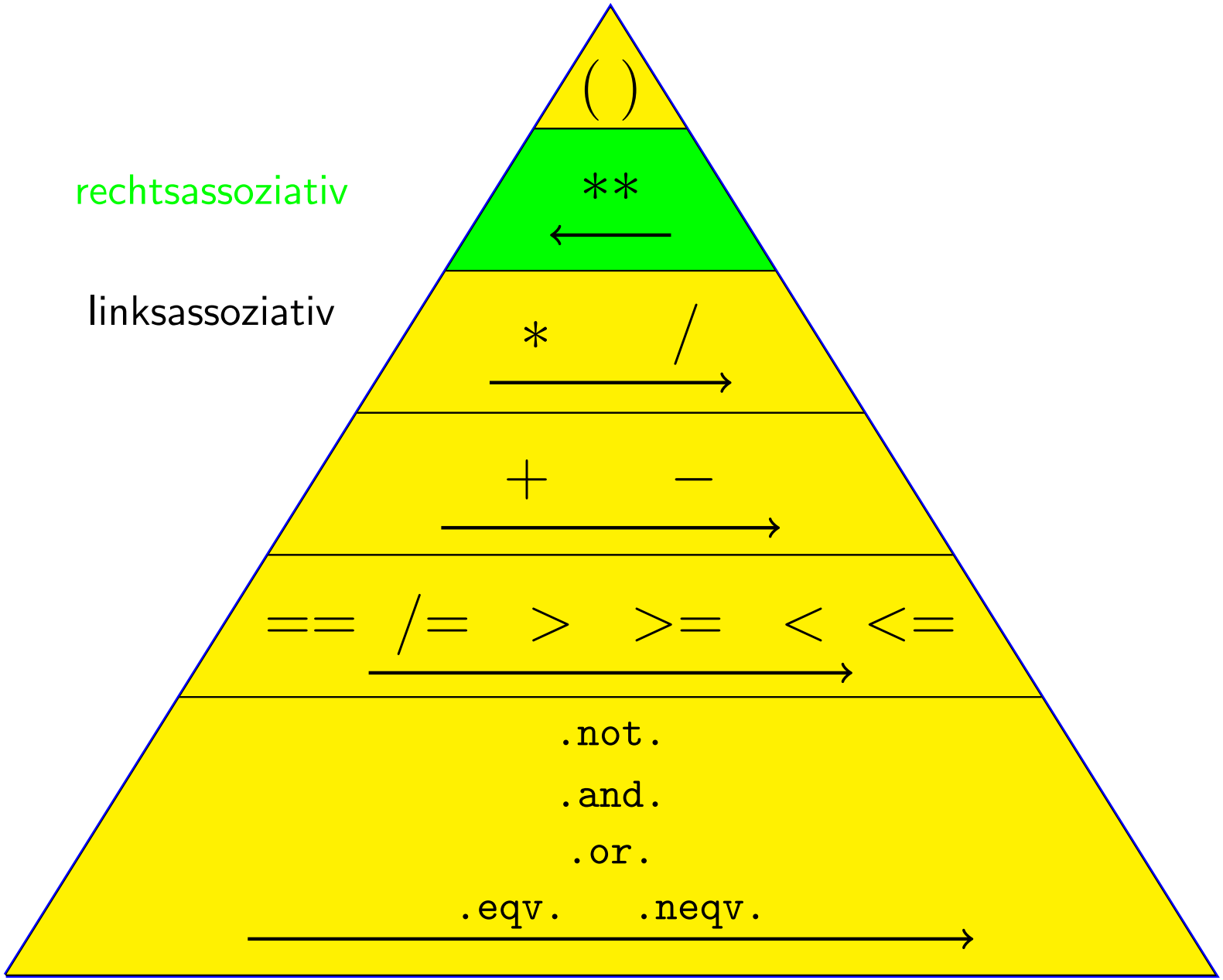
Zuweisungen

- Auf der linken Seite muß eine Variable stehen: ~~`x + y = sin(z)`~~
- Zuweisungen sind keine Werte: ~~`x = y = z = 0`~~ ~~`if( (x=sin(y)) < 0.0 )`~~

Arithmetische Operationen

- Division ist `a/b`, **nicht** `a:b`. Multiplikationsoperator ist nötig: ~~`2(x + 3sqrt(y))`~~ `2*(x + 3*sqrt(y))`
- *,/ bindet stärker als +,– `a + b * c = a + (b * c)`
- *,/,+,– sind linksassoziativ `a / b / c = (a / b) / c`
- ganzzahlige Division rundet zur Null hin `3/4*5.0 = 0*5.0 = 0.0`
- Potenz: `a**b` wird für kleine ganzzahlige Exponenten vom Compiler in Multiplikationen umgewandelt (`a**5`: 3 Multiplikationen), ansonsten ist es äquivalent zu `exp(b*log(a))`

Rangfolge und Assoziativität



`x = a ⊙ b`

- Wenn beide Operanden `a,b` vom gleichen Typ sind, ist auch das Ergebnis `a ⊙ b` von diesem Typ.
- Wenn nicht, wird einer der Operanden "nach oben" zum Typ des anderen konvertiert:
  `integer` ⇒ `real` ⇒ `complex`
  `real(kind=4)` ⇒ `real(kind=8)` ⇒ `real(kind=10)`
- Bei der Zuweisung des Ergebnisses an die linke Seite wird **das Ergebnis** in den Typ der linken Seite umgewandelt. Dabei ist auch eine Abwärtskonversion möglich.
-

```
integer::a; real(4)::b; real(8)::c
x = (a + b) * c
```

  Vorrangregeln und Klammern:
  Addition: `a` ⇒ `real(4)`,
  `tmp=a+b` vom Typ `real(4)`,
  Multiplikation: `tmp` ⇒ `real(8)`

Fallen: Verwendung von literalen Konstanten

- `real(8)::a,b,c; c = 1/2*a*sqrt(b)`: Das Ergebnis ist immer 0!
  Besser: `c = 1./2. * a * sqrt(b)`
  Das klappt, weil die einfach genaue Zahl 0.5 exakt als Maschinenzahl darstellbar ist und daher auch exakt in die doppelt genaue Maschinenzahl 0.5 umwandelbar ist. **Aber:**
- `c = 1./3.` liefert `c= 0.33333334326744070`
  Besser: `c=1./3._8` liefert `c= 0.3333333333333331`
- Analog: `c=sqrt(2.)` liefert c^2=`1.999999315429164`
  Besser: `c=sqrt(2._8)` oder `c=sqrt(2.d0)`
- Die intrinsischen mathematischen Funktionen `abs, sqrt, sin, cos, log, log10, exp, asin, sinh,...` sind sogenannte generische Funktionen:
  `kind` des Ergebnisses = `kind` des Arguments.
- `integer(8):: n; n = 3000000000` liefert Fehler.
  Besser: `n = 3000000000_8`

Zusammengesetzte Typen I: homogene Typen

Felder (arrays): alle Komponenten sind vom selben Typ

Ein Feld von 10 ganzen Zahlen:

```
integer, dimension(10) :: a
```

Die Elemente sind `a(1), a(2),..., a(10)`.

Andere Indexbereiche können definiert werden:

```
real(kind=8), dimension(-2:7) :: a
```

hat die 10 Elemente
`a(-2),a(-1),a(0),a(1),...,a(7)`.

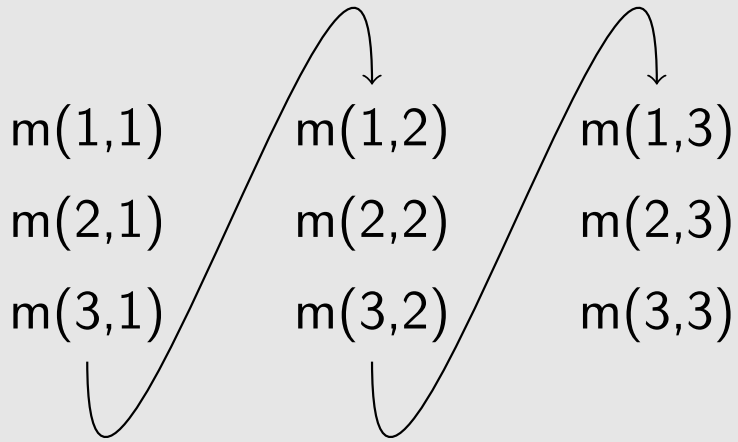
Feld von Zeichenketten:

```
character(len=80), dimension(30) :: page
```

Höherdimensionale Felder

```
integer, dimension(3,3) :: m
```

definiert die 3×3-Matrix `m`.
Die Elemente sind spaltenweise gespeichert, der erste Index variiert am schnellsten:



Andere Indexbereiche:

```
integer, dimension(0:2, 2:4) :: m2
```

ist auch eine 3×3-Matrix

Speicherhierarchie (Cache): Lokalität von Speicherzugriffen wichtig

Langsam!

```
do i = 1, 10000
  do j = 1, 10000
    a(i,j)=b(i,j)+2*c(i,j)
  end do
end do
```

Besser!

```
do j = 1, 10000
  do i = 1, 10000
    a(i,j)=b(i, j)+2*c(i,j)
  end do
end do
```

Dieses Beispiel ist in Fortran 95 natürlich Unsinn: man verwende besser

```
a = b + 2 * c
```

Zusammengesetzte Typen II: Inhomogene Typen (types)

Komponenten könnnen von verschiedenem Typ sein und haben Namen statt Indizes. Als Komponentenselektor wird % verwendet.

```
type Datum
    integer :: Jahr
    character(len=3) :: Monat
    integer :: Tag
end type

type(Datum) :: gestern, heute

heute = Datum(2009,'Mai',31)

heute%Tag = 17

if(heute%Jahr > 2010) ...
```

```
type Point
    real :: x,y
end type Point

type Triangle
    type(Point) :: a,b,c
end type Triangle

type Triangulation
     type(Triangle), dimension(1000) :: triangles
end type Triangulation

type(Triangle) :: t, t2
type(Triangulation) :: tra

t%a%x = 0.2
t%a%y = 0.3
t%b   = Point( 0.6, 1.3)
t2 = Triangle( Point(0.1, 0.2), Point(0.5, 0.6), &
               Point(0.4, 1.) )
tra%triangles(20)%a%x = 0.5
```

Zeiger

Zeiger enthalten die Adresse eines Objekts (sie "zeigen" darauf).

```
character, pointer :: cx
integer, pointer :: ip
real(kind=8), pointer, dimension(:) :: fxx
                                ! Zeiger auf 1-dim Feld
real, pointer, dimension(:,:) :: ax
                                ! Zeiger auf 2-dim Feld
type(Datum), pointer :: dat
                                ! Zeiger auf type
```

Der =>-Operator weist einem Zeiger die Adresse eines Objekts zu. Das ist nur zulässig bei Objekten, die mit dem `target`-Attribut deklariert sind. In einer Anweisung steht der Zeiger für das Objekt, auf das er zeigt.

```
integer, target :: i,j
integer, pointer :: ip
ip => i                 ! jetzt zeigt ip auf i
ip =  3                 ! nun ist i gleich 3
j  =  ip + 2            ! und j gleich 5
ip => null()            ! Nullzeiger
```

Speicherverwaltung

Es gibt zwei grundsätzliche Meinungen zu Speicherverwaltung:

1. C, C++: Speicherverwaltung ist zu wichtig, um sie dem Compiler zu überlassen.
2. Lisp u.a.: Speicherverwaltung ist zu wichtig, um sie dem Programmierer zu überlassen.

Fortran nimmt dem Programmierer viel Speicherverwaltung ab, erlaubt aber auch explizite Eingriffe, wenn dies zweckmäßig ist.

Dynamische Speicherverwaltung

```
integer :: n,i,j
real(8), pointer, dimension(:,:) :: ff

read *, n
allocate(ff(n,n))    ! Speicherplatz anfordern

do i = 1, n
   do j = 1,n
      ff(i,j) = ....   ! Zeiger wie Feld nutzen
      ...
   end do
end do
deallocate(ff)       ! Speicherfreigabe
```

Meist ist es besser, das Feld selbst als `allocatable` zu deklarieren.

Dynamische Speicherverwaltung II – Allozierbare Felder

```
integer :: n,i,j
real(8), allocatable, dimension(:,:) :: ff2

read *, n
allocate(ff2(n , n))    ! Speicherplatz anfordern

do i = 1, n
    do j = 1,n
        ff2(i, j ) = ....   ! Feld nutzen
        ...
    end do
end do
deallocate(ff2)       ! Speicherfreigabe
```

Prozeduren (Unterprogramme)

- `function` und `subroutine`
- intrinsisch (`sin,sqrt,reshape,matmul,..`) oder selbstgeschrieben
- eigener Namensraum (name space), lokale Variablen
- Argumentübergabe durch "call by reference": Argumente können wirksam modifiziert werden
  — Unterschied zu den C-artigen Sprachen: dort "call by value", Modifikationen von Variablen des rufenden Programms nur durch Zeiger möglich

Funktionen

```
integer function timesec(h, m, s)
    integer :: h,m,s
    if(h < 0 .or. m < 0 .or. s < 0) then
        timesec=-1
        return
    endif
    timesec = 60*(60*h + m) + s
end function timesec    !     Name nicht Pflicht
```

- Funktion und Argumente haben einen Typ.
- Ergebnis wird einer Variablen mit dem Namen der Funktion zugewiesen.
- `return` kann zur Ablaufsteuerung verwendet werden.

```
program bla
integer :: asec, hour, min, sec, timesec
...
asec = timesec(hour, min, sec) + 100
...
```

Funktionen II

- Wenn die Funktion einen zusammengesetzten Typ zurückgibt, muß der Typ in einer eigenen Anweisung deklariert werden.
- Bei den Argumenten kann der Zweck (`intent`) angegeben werden. Dies unterstützt Optimierungen und ist in bestimmten Fällen (`pure` oder `elemental function`, selbstdefinierte Operatoren) Pflicht.
  - `intent(in)` Argument wird nur verwendet aber nicht geändert.
  - `intent(out)` Argument wird von Funktion gesetzt aber nicht verwendet.
  - `intent(inout)` Argument wird gelesen und modifiziert.

```
function transpose(n, a)
    integer, intent(in) :: n
    real(8), dimension(n,n), intent(in) :: a
    real(8), dimension(n,n) :: transpose
    ...
end function
```

Funktionen III

- Der Rückgabewert kann mit `result` umbenannt werden

```
function timesec(h, m, s) result (sec)
    integer :: h,m,s,sec
    ...
    sec = ...
end function
```

- Dies ist nötig für rekursive Funktionen

```
recursive function fakultaet(n) result(ff)
    integer :: n, ff
    if(n == 1) then
        ff = 1
    else
        ff = n * fakultaet(n-1)
    end if
end function
```

Subroutinen

Funktionen ohne Rückgabewert heißen subroutinen.
Sie müssen mit call aufgerufen werden.

```
program bsp22
    ...
    read *, hours, mins, secs
    call timesec2(seconds, hours, mins, secs)
    print *, seconds
    ...
end program

subroutine timesec2(x, h, m, s)
    integer, intent(out) :: x
    integer, intent(in) :: h, m, s

    if( s < 0 ) then
        x = -1
        return
    endif
    x = 60*(60*h + m) + s
end subroutine
```

Optimierung I

- Compiler unterstützen verschiedene Optimierungsstufen
  `gfortran, gfortran -O, gfortran -O2`
- Mit Optimierung übersetzter Code kann zum Teil erheblich schneller laufen.
- Dabei sind mathematisch äquivalente Umformungen zulässig.
  Der Compiler kann auf einem Prozessor, der schneller multipliziert als dividiert, `a/b/c` durch `a/(b*c)` ersetzen oder `x+2-x` durch `2`
- Vom Programmierer vorgegebene Klammern verhindern dies:
  `(a/b)/c` oder `x+(2-x)` wird **nicht** umgeformt!
  Dies kann bei Fließkommazahlen sehr wichtig sein.

Optimierung II: Reine Funktionen

- Der optimierende Compiler wird

```
do i =1,1000
   a(i) = b(i) + sin(c)
end do
```

  durch

```
tmp = sin(c)
do i =1,1000
   a(i) = b(i) + tmp
end do
```

  ersetzen.
- Dies ist zulässig, weil der Compiler weiß, daß die intrinsische Funktion `sin()` eine **reine** Funktion ist und daher diese Substitution die Semantik des Programms nicht ändert.
  - Bei gleichem Argument gibt die Funktion das gleiche Ergebnis zurück (im Gegensatz zu z.B. `time()`, `random()`)
  - Die Funktion ändert ihr Argument nicht.
  - Die Funktion ändert keine anderen (globalen oder statischen) Variablen.
  - Die Funktion macht keine Ein/Ausgabeoperationen.

Reine Funktionen II

Bei einer selbstdefinierten Funktion `mysin(c)` kann der Conpiler diese Optimierung nur durchführen, wenn die Funktion als `pure` definiert ist.

```
pure integer function timesec(h, m, s)
    integer, intent(in) :: h,m,s
    if(h < 0 .or. m < 0 .or. s < 0) then
        timesec=-1
        return
    endif
    timesec = 60*(60*h + m) + s
end function timesec
```

Damit eine Funktion als `pure` deklariert werden kann, müssen

- alle Argumente das `intent(in)`-Attribut tragen
- keine Ein/Ausgabe-Operationen durchgeführt werden
- keine globalen oder statischen Variablen verwendet werden
- alle in der Funktion verwendeten Funktionen ebenfalls `pure` sein
- eine `interface`-Definition vorhanden sein.

Felder in Fortran I

Rang und Gestalt

- **Rang:** Anzahl der Indizes $\neq$ Rang in der linearen Algebra
- **Gestalt** (shape): Folge $(n_1, n_2, \ldots, b_{\text{rang}})$ der Anzahl der Elemente in jeder Dimension
  Feld aus 10 Elementen: Gestalt (10)
  8×6-Matrix: Gestalt $(8, 6)$
- Zahl der Elemente, nicht Laufbereich der Indizes bestimmt Gestalt:
  `real, dimension(-10:5, -20:-1, 0:1, -1:0, 2) :: grid`
  definiert ein Feld `"grid"` mit 5 Indizes (Rang 5), mit
  $16 \times 20 \times 2 \times 2 \times 2 = 2560$ Elementen und der Gestalt
  $(16, 20, 2, 2, 2)$.
- Ein Datentyp ohne Index heißt auch **Skalar**.

Felder in Fortran II

- Viele Funktionen und Operationen, die wir bisher für Skalare betrachtet haben, können auf Felder angewendet werden. Dazu müssen die Operanden die gleiche Gestalt wie das Ergebnis haben oder Skalare sein.
- Grundidee: nicht der Laufbereich der Indizes, sondern die Gestalt der Felder muß übereinstimmen:

```
real, dimension(10) :: a
real, dimension(0:9) :: b
real, dimension(2:11) :: c
a = b + c
```

ist zulässig, da alle 3 Vektoren die Gestalt (10) haben. Dies wird also als

```
a(1)  = b(0) + c(2)
a(2)  = b(1) + c(3)
  ...     ...
a(10) = b(9) + c(11)
```

interpretiert.

Felder in Fortran III

- **Alle Operationen werden elementweise durchgeführt.**

```
real(8), dimension(100,100) :: a, b, c, d
c = a * b
d = 1/a
```

Die Matrix c ist nicht das mathematische Matrixprodukt, sondern die Matrix mit den Einträgen $c_{ij} = a_{ij}b_{ij}$. Ebenso ist d nicht die inverse Matrix, sondern die Matrix mit den Einträgen $d_{ij} = 1/a_{ij}$.

- Ein skalares Argument wird behandelt wie Feld der nötigen Gestalt, in dem alle Einträge den gleichen Wert haben.

```
a = 2
b = b + 2
```

setzt alle Einträge von a gleich 2 und addiert 2 zu allen Einträgen von b.

Elementale Funktionen

- Numerische Funktionen lassen sich elementeweise anwenden

```
real, dimension(100) :: a, b, c
...
a = sqrt(b) + sin(c)
```

- Selbstgeschriebene elementale Funktion:
  - Funktion und alle Argumente sind Skalare
  - Funktion ist eine reine Funktion
  - `elemental` impliziert `pure`
  - `interface`-Definition nötig

```
elemental real(kind=8) function hypo(a,b)
    real(kind=8), intent(in) :: a,b
    hypo = sqrt( a**2 + b**2 )
end function
    ...
    real(kind=8), dimension(100) :: a,b,c
    c = hypo(a,b)
```

Feldkonstruktoren

- Die Werte von Feldern vom Rank 1 (Vektoren) kann man mittels `[...]` direkt auflisten:

```
integer :: i
integer, dimension(5) :: x = [12, 13, 14, 15, 16], y
i = 99
y = [0, 0, 2, 2*i, i]
```

- Eine alternative Klammerform ist `(/.../)`: `(/ 1,2,3,4,5 /)`
- implizite Schleifen:
  `[ (i, i=1,100) ]` ist äquivalent zu `[1, 2,..., 100]`
  `[ (i**2, i=1,10,2) ]` ist äquivalent zu `[1, 9, 25, 49, 81]`
  `[ ((i, i=1,3), j=1,3) ]` ist äqu. zu `[1,2,3,1,2,3,1,2,3]`
- Matrizen kann man mit Hilfe der `reshape()` Funktion initialisieren:

```
x = reshape([1,2,3,4,5,6], shape=[2,3])
```
– oder
```
x = reshape([ (i, i=1,6) ], shape=[2,3])
```

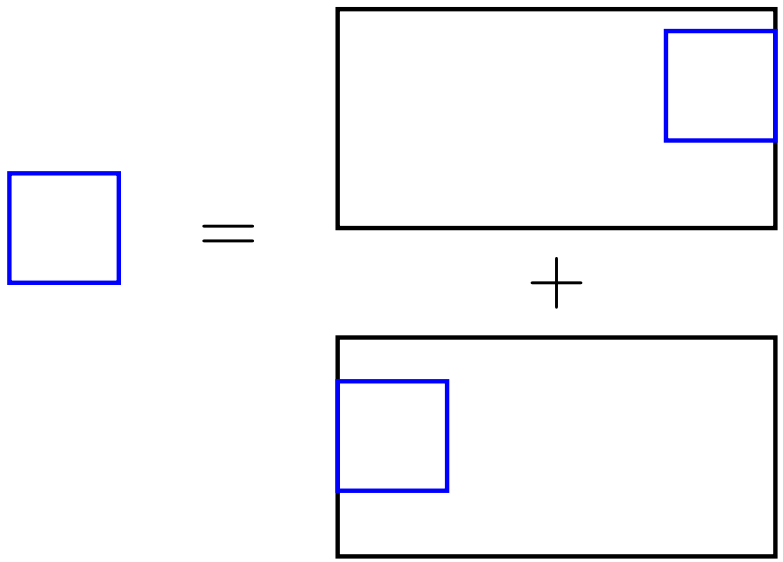
liefert die 2×3-Matrix $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

Teilfelder

- In Deklarationen kann mit der Syntax
  `real, dimension(0:9, 0:9) :: a`
  der Laufbereich der Indizes verändert werden (10×10 Matrix `a`).
- In Anweisungen können mit einer ähnlichen Syntax Teilfelder spezifiziert werden:

```
real, dimension(10, 20) :: a, b
real, dimension(5, 5) :: c

a(2:9, 5:10) = 1.
c = a(2:6, 16:20) + b(3:7, 1:5)
```



Teilfelder II

```
real(8), dimension(10, 20) :: a
real(8), dimension(10) :: c
```

- ohne Doppelpunkt: `a(3, 16:20)` beschreibt ein Teilfeld mit kleinerem Rang, hier also ein Vektor mit den 5 Elementen `a(3,16), a(3,17), a(3,18), a(3,19), a(3,20)`
- weggelassene Grenzen entsprechen der unteren oder oberen Grenze:
  - Statt `a(3, 16:20)` kann man auch `a(3, 16: )` schreiben.
  - `c = a(:, 3)` $\quad\quad$ `c` $\longleftarrow$ `3. Spalte von a`
- Es kann auch eine Schrittweite angegeben werden:
  - `c = a(2, 1:20:2)`
    setzt `c` gleich `[a(2,1), a(2,3), a(2,5),.., a(2,19)]`
  - `c = a(10:1:-1, 5)`
    setzt `c` gleich `[a(10,5), a(9,5), a(8,5), ..., a(1,5)]`

Weitere Beispiele

```
real, dimension (10,20) :: a,b
real, dimension (5)     :: v,w
real, dimension (8, 6)  :: c

w = 5/v + a(1:5, 7)    ! w(i) = 5/v(i) + a(i, 7), i = 1...5

c = a(2:9, 5:10) + b(1:8, 15:20)
 ! c(i,j) = a(i+1, j+4) + b(i, j+14), i = 1...8, j=1...6

a(2, 11:15) = v    ! a(2, i+10) = v(i), i = 1...5

v(2:5) = v(1:4) ! v(2)=v(1);v(3)=v(2);v(4)=v(3);v(5)=v(4)
 ! Hier sind die Werte auf der re. Seite die
 ! Werte vor der Zuweisung

a(2,:) = b(3,:)
 ! Zeile 2 von a wird gleich Zeile 3 von b gesetzt

  a(2, 1:20:4) = v
 ! a(2,1) =v(1); a(2,5)=v(2); a(2,9)=v(3);
 ! a(2,13)=v(4); a(2,17)=v(5)
```

Teilfelder III: Vektorindizes (indirekte Indizierung)

An Stelle eines Indexes oder Indexbereichs darf auch ein `integer`-Feld vom Rank 1 stehen.

```
real(8),dimension(20) :: v = [ (2*i , i=1,20) ]
real(8),dimension(5) :: w
w = v( [1, 7, 3, 7, 17] )
```

Das Feld `w` enthält danach die Werte `v(1),v(7),v(3),v(7),v(17)`, also `w=[2.,14.,6.,14.,34.]`

```
real(8), dimension(5,5) :: m, mp
integer, dimension(5) :: perm
perm = [2,3,1,5,4]
mp = m( : , perm)
```

Die Matrix `mp` entsteh aus der Matrix `m`, indem die Spalten gemäß `perm` permutiert werden.

where-Anweisung

Manchmal soll eine komponentenweise Feldoperation nur auf Komponenten angewandt werden, die einer Bedingung genügen:

```
real, dimension(0:100, 3:10) :: a

where (a /= 0)  a = 1/a
```

Es gibt auch eine Blockform:

```
where (a /= 0)
    a = 1/a
    b = 2 * a
end where
```

```
where (a /= 0)
      a = 1/a
      b = 2 * a
elsewhere
      a = 1000
end where
```

Der Test in der `where`-Anweisung und die Terme in den folgenden Anweisungen müssen alle dieselbe Gestalt haben.
Argument der `where()`-Anweisung ist ein `logical` Feld derselben Gestalt, eine Maske.

forall-Anweisung

- Fortrans Vektorsprache erlaubt die weitgehende Einsparung von Schleifen und eine weitgehend "indexfreie" Programmierung. Manchmal reicht sie aber nicht aus, z.B. für die Zuweisung
  `vector ← Diagonalelemente einer Matrix`
- "paralleles do"

```
forall( i = 1:n )  x(i) = a(i,i)

forall( i = 1:n, j = 1:n, y(i,j) /= 0 .and. i/=j ) &
     x(j,i) = 1./y(i,j)     ! mit Bedingung

forall( i = 1:n )       ! Block-Form
    a(i) = 2*i
    where ( w(i,:) /= w(3,:) ) w(i,:) = 44
end forall
```

Ein `forall`-Block darf keine `if, do,` usw. Kontrollanweisungen enthalten, sondern nur weitere `forall`-Schleifen und `where`-Anweisungen. Die Terme in der `where`-Anweisung müssen alle (inklusive Test) Felder derselben Gestalt sein.

Sequentielle Semantik

```
integer, dimension(5) :: a = [1, 2, 3, 4, 5]
do i = 2,5
   a(i) = a(i-1)
end do
```

Die Zuweisungen `a(2) = a(1); a(3) = a(2); a(4) = a(3); ...` werden **nacheinander** ausgeführt. Am Ende ist

`a = [1, 1, 1, 1, 1]`

Parallele Semantik

```
integer, dimension(5) :: b = [1, 2, 3, 4, 5]
integer, dimension(5) :: c = [1, 2, 3, 4, 5]

forall(i = 2:5) b(i)=b(i-1)
c(2:5) = c(1:4)
```

Das Programm verhält sich so, als ob alle Zuweisungen gleichzeitig (**parallel**) durchgeführt werden. Am Ende ist

`b = c = [1, 1, 2, 3, 4]`

$\Rightarrow$ bessere Ausnutzung von Multiprozessormaschinen

Spezielle Funktionen (Auswahl)

- Skalarprodukt `dot_product(a, b)` $= \sum_i a_i b_i$
- Matrixprodukt `matmul(a, b)`
  - $\sum_j a_{ij}\, b_{jk}$: $(n,m) \times (m,p) \Rightarrow (n,p)$
  - $\sum_j a_{ij}\, b_j$: $(n,m) \times (m) \Rightarrow (n)$
  - $\sum_j a_j\, b_{jk}$: $(m) \times (m,p) \Rightarrow (p)$
- Transponierte `transpose(a)` $(n,m) \Rightarrow (m,n)$
- Gestalt `shape(a)` $\Rightarrow$ ein ganzzahliges Rang-1-Feld mit den Gestaltsparametern $(n, m, \ldots)$ des Feldes `a`.
- Größe `size(a,i)` $\Rightarrow$ Größe in Dimension `i`.

Bsp.:
```
real(8), dimension(10,20) :: a
    shape(a) ⇒ [10,20]
    size(a,1) ⇒ 10
    size(a,2) ⇒ 20
    size(shape(a)) ⇒ 2
```

Reduktionsfunktionen

```
real,   dimension(100) :: x
integer, dimension(1) :: i
a = sum(x)                    !  Summe aller Elemente von x
d = product(x)                !  Produkt aller Elemente
b = maxval(x)                 !  groesstes Element aus x
c = sum(a, mask = a > 0)      !  c ist die Summe aller  a(i),
                              !  die a(i) > 0 erfuellen
i = maxloc(x)                 ! Index des groessten Elements
     ! analog: minval, minloc
```

- optionale Argumente: `mask` und `dim`.

Wird `dim=n` angegeben, wird die Reduktion (Aufsummation etc) nur bezüglich des n-ten Index durchgeführt, Ergebnis ist ein Feld mit um eins kleinerem Rang.

Beispiel: Sei `m` eine Matrix mit 4 Zeilen und 7 Spalten (Gestalt $(4, 7)$). Dann ist `sum(m, dim=1)` ein Feld aus 7 Elementen, den Spaltensummen und `sum(m, dim=2)` ist ein Feld aus 4 Elementen, den Zeilensummen.

$$a = \begin{pmatrix} 1 & 18 & 7 \\ 2 & 13 & 5 \end{pmatrix}$$

$\texttt{sum(a)} = 46$	$\texttt{sum(a, 1)} = [3, 31, 12]$	$\texttt{sum(a, 2)} = [26, 20]$
$\texttt{maxval(a)} = 18$	$\texttt{maxval(a, 1)} = [2, 18, 7]$	$\texttt{maxval(a, 2)} = [18, 13]$
$\texttt{maxloc(a)} = [1, 2]$	$\texttt{maxloc(a, 1)} = [2, 1, 1]$	$\texttt{maxloc(a, 2)} = [2, 2]$

Anwendung: Vektor- und Matrixnormen

```
real(8), dimension(n)     :: x
real(8), dimension(n, n)  :: a

x_norm_1     = sum(abs(x))        ! Betragssummennorm
x_norm_2     = sqrt(sum(x**2))    ! Euklidische Norm
x_norm_max   = maxval(abs(x))     ! Maximumnorm

a_norm_1   = maxval( sum(abs(a), dim=1)) ! Spaltensummennorm
a_norm_max = maxval( sum(abs(a), dim=2)) ! Zeilensummennorm
a_norm_f   = sqrt(sum(a**2))             ! Frobeniusnorm
```

Reduktionsfunktionen II: `all, any, count`

```
logical :: test1, test2
integer:: n
real(8), dimension(100,200):: m
...
test1 = all(m > 20)   ! Sind alle m(i,j) > 20?
test2 = any(m <= 0)   ! Ist mind. ein m(i,j)<=0?
n = count( m > 0)     ! Wieviele sind > 0?
...
if(all(m > 0) .and. count(m > 10) > 10) then
    ...
```

Beispiel

Matrix M enthält die Testergebnisse von s Studenten in t Tests

```
integer, dimension(s, t) :: M
integer, dimension(s) :: GesPunkte, HoechstPunkte
real :: mittel
integer :: zahl
```

Gesamtpunktzahl jedes Studenten (Zeilensumme)

```
GesPunkte = sum(M, dim=2)
```

Höchstpunktzahl jedes Studenten

```
HoechstPunkte = maxval(M, dim=2)
```

Durchschnittl. Punktzahl (gemittelt über alle Studenten und Tests)

```
mittel = (1.*sum(M))/size(M)
```

Wieviele Studenten erreichten in allen Tests mehr als den Durchschnitt?

```
zahl = count(all(M > mittel, dim=2))
```

Weitere Funktionen (Auswahl): `shift, spread`

$$a = \begin{pmatrix} 1 & 18 & 6 & 35 \\ 2 & 17 & 7 & 34 \\ 3 & 16 & 8 & 37 \\ 4 & 15 & 9 & 30 \end{pmatrix}$$

$$\texttt{cshift(a,}-1,2) = \begin{pmatrix} 35 & 1 & 18 & 6 \\ 34 & 2 & 17 & 7 \\ 37 & 3 & 16 & 8 \\ 30 & 4 & 15 & 9 \end{pmatrix} \qquad \texttt{cshift(a,3,1)} = \begin{pmatrix} 4 & 15 & 9 & 30 \\ 1 & 18 & 6 & 35 \\ 2 & 17 & 7 & 34 \\ 3 & 16 & 8 & 37 \end{pmatrix}$$

$$\texttt{eoshift(a,2,99,1)} = \begin{pmatrix} 3 & 16 & 8 & 37 \\ 4 & 15 & 9 & 30 \\ 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 \end{pmatrix}$$

$$\texttt{spread([1,2,3],2,3)} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix} \qquad \texttt{spread([1,2,3],1,2)} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

Sichtbarkeit, Lebensdauer und Initialisierung von Variablen

- Sichtbarkeit (scope): lokale und globale Variablen
- Lebensdauer: automatische und statische Variablen
- Lokale Variablen sind Variablen, die innerhalb von Funktionen definiert werden. Sie sind nur in dieser Funktion sichtbar.
- Globale Variablen sind in mehreren Funktionen sichtbar. Damit sind sie neben der Parameterübergabe durch Funktionsargumente eine Methode des Datenaustausches oder gemeinsamen Datenzugriffs verschiedener Funktionen.
- Automatische Variablen in einer Funktion behalten ihren Wert **nicht** zwischen zwei Funktionsaufrufen. Ihr Anfangswert bei jedem Funktionsaufruf ist undefiniert.
- Statische Variablen behalten ihren Wert zwischen zwei Funktionsaufrufen.

- lokal und automatisch: in einer Funktion deklarierte Variablen:

```
real(8) function f1(y)
      integer :: i, j
      real(8) :: x, y, z
```

  `i,j,x,z` sind bei jedem Aufruf von `f1()` undefiniert.

- statisch a): mit `save`-Attribut

```
       integer, save :: j
```

  wird bei Programmstart mit 0 initialisiert.

- statisch: b) durch Initialisierung bei Deklaration

```
real(8) function f2(y)
      integer :: i = 33, j = 0
      integer, pointer :: ip => null()
      character(len=*), parameter :: str = "hello"
        ! Compiler ermittelt Laenge, nur bei konst.
        ! String (parameter) zulaessig
```

  Diese Initialisierung wird nur einmal, beim Programmstart, ausgeführt!

Globale Variablen

- Variable, die in mehreren Funktionen sichtbar sein sollen, werden durch Module vereinbart

```
 module my_globals
      real(8), dimension(100,100), save :: a,b
      real(8), parameter :: Pi = 4*atan(1._8)
end module
```

- Verwendung in Funktionen durch `use`-Anweisung:

```
integer function f3(y)
      use my_globals
      real(8) :: y
```

- statisch, wenn a) save-Attribut, b) Initialisierung oder c) use-Anweisung im Hauptprogramm

```
program xxx
      use my_globals
      ...
```

Ein- und Ausgabe

- Öffnen einer Datei `open()`
- Lesen aus der Datei/Schreiben in die Datei `read/write`
- Schließen der Datei `close()`

Kanalnummern

Beim Öffnen einer Datei wird ihr eine Kanalnummer (unit) zugewiesen.
`read, write, close` verwenden dann diese Kanalnummer.

```
open(unit = 4, file = "bsp.dat" )
open(file = "bsp2.txt", unit = 5, position="append")
read(4, *) i,j,k
write(5, *) i**2, i+j
close(4)
close(unit = 5)
```

Bemerkung: `open()` ist Beispiel für

- optionale Argumente
- Name=Wert - Argumente

  (`unit` ist 1. Argument oder Argument mit diesem Namen.)

Standardeingabe und Standardausgabe

- stellt das Betriebssystem als geöffnete Kanäle zur Verfügung
- dienen zur interaktiven Ein- und Ausgabe
- `print` schreibt auf Standardausgabe
- `write` mit `unit=*` schreibt auf Standardausgabe
- `read` mit `unit=*` liest von Standardeingabe
- `read` ohne Kanalnummer liest von Standardeingabe

```
print *, "i =", i
write (*,*)  "i =", i
read (*,*) a,b
read *, a,b
```

- Dateiname kann Stringvariable sein

```
character(len=80) :: Name
write (*,*) "Bitte Dateinamen eingeben"
read (*,*) Name
open(3, file=Name)
```

- `open()` öffnet Datei am Dateianfang ⇒ beim Schreiben wird der Inhalt überschrieben
  `open(position="append",...)` öffnet am Ende, Datei wird fortgeschrieben.
- Fehlerbehandlung:

```
integer :: error
open(3, file="test.txt", action="write", iostat=error)
if(error /= 0) then
   print *,"Fehler Nr.",error," aufgetreten!"
   stop
end if
```

Formatierung von Ein- und Ausgabe

- I/O zeilenorientiert, `read`/`write`-Anweisung liest/schreibt eine Zeile.
- Formatstring oder `*`
  1. direkte Angabe des Formatstrings:
     ```
     write (*, '("i= ", I6, A, F10.2 )')    i, 'x=', x
     ```
  2. `format`-Anweisung mit Nummer:
     ```
     write (*, 234) i, 'x=', x
     write(*, 234) j, 'a=', a
     234 format ("i= ", I6, A, F10.2 )
     ```
  3. Name einer String-Variablen:
     ```
     character(len=30) string
     string = '(I6, F10.2 )'
     write (*, string) i,x
     read(*, string) j,y
     ```
     Dieser `string` kann vom Programm dynamisch erzeugt, modifiziert etc. werden.
- Variablenliste `i,x` muß den Formatbeschreibern entsprechen

Formatstrings

- geklammerte Komma-Liste von Formatbeschreibern und Teilstrings:
  `(4X, I6, F10.2, I6, F10.2, / , I8, ' Ex =', G20.8)`
- Teillisten können geklammert und mit einem Wiederholungsfaktor versehen werden:
  `(4X, 2(I6, F10.2), / , I8, ' Ex =', G20.8)`
- Formatbeschreiber:
  - `X` Leerzeichen
  - `In` ganze Zahl rechtsbündig in Breite `n`
  - `/` Zeilenumbruch
  - `A`, `An` Zeichenkette
  - `L`, `Ln` logische Variable
  - `Fn.m` Fließkomazahl, `m` Kommastellen
  - `En.m` Fließkomazahl, `m` Kommastellen, Exponentialschreibweise
  - `Gn.m` Fließkomazahl, `E`- oder `F`-Format, je nach Größe
  - `$` am Ende der Liste: `(I6,G20.3,$)` Zeilenvorschub unterdrückt
    (Nichtstandarderweiterung vieler Compiler,
    ohne `-pedantic -std=f2003` übersetzen)

Fehlerbehandlung

```
integer :: error
read(8,'(3f12.3)', iostat=error,err=110,end=120) a,b,c
   ...
110 print *, "Fehler mit Nummer ", error, "aufgetreten"
   ...
120 print *, "Dateiende erreicht!"
```

Implizite Schleifen in read/write/print

```
print '(5F10.4)', (a(i), i=1,9,2)
print '(10I8)', (x(j), j=1,200)
```

Die letzte Zeile druckt 20 Zeilen mit je 10 Zahlen.

Internal files

- Anstelle einer Kanalnummer (externe Datei) oder * (Standardein/ausgabe) kann bei `read/write` auch ein String angegeben werden.
- Anwendung: Umwandlung Zahl ⇒ Zeichenkette

```
character(len=80) :: buffer
write(buffer, '(G24.6)') x
```

- Datei zeilenweise in Buffer einlesen, dann Buffer analysieren

```
open(unit=8, file="datei")
read(unit=8, '(A)' ) buffer
....
read(buffer, '(2I)' ) n,m
```

Module

- können Funktionen, Variablen und Typdefinitionen enthalten
- `use modulename`

```
module my_globals
    integer :: n1,n2,n3
    real(8), dimension(1000) :: vec1
end module

module mod2
    use  my_globals
    integer :: n4
contains
    integer function f1(x)
    ...
    end function
    function f2(x, y)
    ...
    end function
end module mod2
```

Module II

- `use ...` muß ganz am Anfang einer Funktion/Subroutine stehen.
- `use mod2`
  stellt die Variablen `n1,n2,n3,n4,vec1` und die Funktionen `f1` und `f2` zur Verfügung.
- `use mod2, only: n1, n2, f1 => func1`
  verwende nur `n1` und `n2` sowie `f1` unter dem Namen `func1`
- Die `use`-Anweisung muß in jeder Funktion/Subroutine enthalten sein, welche Daten/Funktionen aus dem Modul verwendet. (Eine Anweisung nur im Hauptprogramm reicht dazu nicht aus.)

Module III

Module können (wie auch Funktionen/Subroutinen) in eigenen Quelltextdateien stehen und separat compiliert werden.

m1.f90

```
program m1
use m1mod
implicit none
integer :: x, y, z

read *, y
k = 3
x = hfunc(y)
k = 5
z = hfunc(y)
print *, "x= ", x, &
    "y= ", y, "z= ", z
end program
```

m1mod.f90

```
module m1mod
   integer :: k
contains
   integer function hfunc(x)
   integer :: x
   integer, save :: i

   i= i + 1
   print *, "hfunc ", &
       i, "mal gerufen"
   hfunc = x * x + k
   x = x + 3
   end function
end module
```

```
gfortran -Wall -std=f2003 -fbounds-check -c m1mod.f90
gfortran -Wall -std=f2003 -fbounds-check -o m1 m1.f90 m1mod.o
```

Interface-Definitionen in Fortran

Deklaration der gerufenen Funktion in der rufenden Programmeinheit

1. Intrinsische Funktionen `sqrt, read, sum, ...` müssen nicht deklariert werden.
2. Bei einfachen Funktionen mit einem nicht zusammengesetzten Rückgabewert reicht die Deklaration des Rückgabewertes:

```
integer function blub(x)
    integer :: x, timesec
    ...
    x = timesec(...)
end function


integer function timesec(h, m, s)
    integer :: h,m,s
...
end function
```

3 expliziter Interface-Block

```
integer function blub(x)
    integer:: x

    interface
        integer function timesec(h, m, s)
            integer, intent(in) :: h,m,s
        end function
        function f1(x)
            real(8) :: f1, x
        end function
    end interface

    ...
    x = timesec(...)
end function blub


integer function timesec(h, m, s)
    integer, intent(in) :: h,m,s
...
end function
```

4. im Allgemeinen beste Methode:
   Alle Funktionen innerhalb von Modulen definieren.

```
module mfunc1
contains
    function func1(x, y)
    ...
    end function
end module

function blub(x)
    use mfunc1
    ...
end function
```

dadurch ist in `blub()` das explizite Interface von `func1()` bekannt.

Ein explizites Interface (also entweder interface-Block oder Modul) ist notwendig z.B. bei Verwendung von `pure`, `elemental` und bei Funktionen, die allocatable-Felder verwenden oder zurückgeben.

Es erlaubt dem Compiler auch eine Überprüfung der Argumente auf korrekten Typ, Anzahl etc.

Felder als Argumente von Funktionen

Felder fester Größe

```
module m_eigenvalues
contains
    function eigenvalues(matrix)
        real(kind=8), dimension(3,3) :: matrix
        real(kind=8), dimension(3) :: eigenvalues
        ...
    end function
end module
```

Felder variabler Größe

- müssen im rufenden Programmteil angelegt werden

```
program blub
    integer :: n
    real(8), dimension(:,:), allocatable :: am
    real(8), dimension(:), allocatable :: v,w
    ...
    allocate(am(n, n+1))
    allocate(v(2*n))
    allocate(w(n))
```

- Variante 1: Größe an Unterprogramm mitgeben

```
subroutine eigenvalues(n, matrix, ev)
    real(8), dimension(n,n) :: matrix
    real(8), dimension(n) :: ev
    real(8), dimension(2*n+1, n) :: workspace
        ...
```

Dabei können im Unterprogramm auch weitere lokale `n`-abhängige Felder angelegt werden.

Felder variabler Größe

- Variante 2: Größe im Unterprogramm mit `size()` ermitteln

```
subroutine eigenvalues(matrix, ev)
    real(8), dimension(:,:) :: matrix
    real(8), dimension(:) :: ev
    real(8), dimension(2*size(matrix,1)) :: wrkspace

        do i = 1, size(matrix,1)
                ....
```

  Dabei können im Unterprogramm auch weitere lokale `size()`-abhängige Felder angelegt werden.

- In beiden Varianten sollte die rufende Funktion das `interface` kennen $\Rightarrow$ Module verwenden.

Funktionen als Argumente: Interface-Block notwendig

```
module m_intgr
contains
  real(8) function intgr(fn, x0, x1)
    real(8) :: x0, x1, delta
    integer :: i

    interface
      real(8) function fn(x)
        real(8) :: x
      end function
    end interface

    intgr = 0
    delta = (x1-x0)/100.
    do  i = 1, 100
      intgr = intgr+delta*fn(x0+delta*i)
    end do
  end function
end module m_intgr
```

```
program intgr_test
  use m_intgr

  interface
    function bsp2(y)
      real(8):: bsp2, y
    end function bsp2
  end interface

  real(8) :: x

  x=intgr(bsp2,0._8,2._8)
  print *, x
end program
```

```
real(8) function bsp2(x)
  real(8) :: x
  bsp2=cos(x)+sqrt(x)
end function
```

Operator overloading

```
module vektorprodukt
    type vec
         real :: x,y,z
    end type

    interface operator (*)
        module procedure vektor_produkt
    end interface

contains
    function vektor_produkt(a, b)
        type(vec)              ::  vektor_produkt
        type(vec), intent(in) :: a, b
        vektor_produkt%x = a%y * b%z - a%z * b%y
        vektor_produkt%y = a%z * b%x - a%x * b%z
        vektor_produkt%z = a%x * b%y - a%y * b%x
    end function
end module

program test
    use vektorprodukt
    type(vec) :: a1,a2,a3

    a1 = vec(2,3,4)
    a2 = vec(7,8,9)

    a3 = a1 * a2     ! vektor_produkt!
    print *, a3
end program
```

- selbstdefinierte Typen → überladen von Operatoren wie `+,-,*,/,=,>,<,...` möglich
- Funktion vom richtigen Typ, Argumente haben Attribut `intent(in)`
- Zuordnung zum Operator mittels `interface operator()`-Block und `module procedure`-Definition

Was fehlt

- Alte F77-Features
  `common blocks, entry, equivalence, block data, statement functions, fixed source form,...`
- weitere Standardfunktionen und Standardmodule, wie z.B.
  `iso_c_binding, ieee_exceptions, ieee_arithmetic`
- Funktionen zur Bitmanipulation
  `bit_size(), btest(), iand(), ior(), ieor(), ishft(),...`
- Verschiedenes
  `internal procedures; keyword, optional arguments, public, private, ...`
- Ein/Ausgabe
  `namelists, direct access I/O, unformatted I/O,...`
- Weitere Features von Fortran2003 oder Fortran2008:
  `procedure pointer, submodules, abstract interfaces, type extensions, select type, parametrized derived types,...`