Wissenschaftliches Programmieren

Bálint Aradi
http://www.bccms.uni-bremen.de/cms/people/b-aradi/

2. Grundlegende Datentypen, Operatoren, Verzweigungen, Schleifen

Das Ziel

- Intrinsische Datentypen in Fortran95/2003 kennen
- Konversion zwischen Datentypen (insbes. zwischen reellen und ganzen Zahlen!)
- Grundlegende Kontrollstrukturen verstehen und anwenden
- Anwendung:
  - Erzeugung von Fibonacci-Zahlen
  - Sortierung von Charakteren

Struktur von Fortran 2003 Programmen

Programmkopf

Vereinbarungs- oder Deklarationsteil

Eigentlicher Anweisungsteil

Programmende

```
! Personalized version of hello world
program hello_person
  implicit none

  character(len=10) :: name

  write(*,*) "Please enter your name:"
  read(*,*) name
  write(*,*) "Hel&
    &lo ", trim(name) // "!"

end program hello_person
```

- Feinstrukturierung größerer Programme mit Unterprogrammen, Modulen möglich (diese haben ähnliche Struktur)

Zeichen, Zeilen etc.

- Alphanumerische Zeichen: A-Z (groß), a-z (klein), Ziffern (0-9), Unterstrich „_“
- Weitere Symbole: + - * / ( ) . = , ' $! " % & ; < > : ? [Leerzeichen]

Fortran unterscheidet nicht zwischen Groß- und Kleinschreibung! (außer in Strings)

```
character(len=10) ::
name
read(*,*) NAME
WRITE(*,*) nAmE
```

aber

```
write (*,*) "Hello!"  → Hello!
write (*,*) "HELLO!"  → HELLO!
```

Namen (Variablen-, Funktionen-, Modulname, etc.)

- Nur alphanumerische Charakter und '_' (max. 31)
- Erster Charakter muss eine Buchstabe sein

Zeilen:

- Maximale Länge 132 Charakter
- Fortsetzung mit „&“

äquivalent:

```
read (*,*) na&
    &me
read (*,*) na&
    me
```

nicht äquivalent:

```
write (*,*) "Hel&
    &lo!"        → Hello!
write (*,*) "Hel&
       lo!"      → Hel    lo!
```

Datentypen

- **Intrinsiche Datentypen** (vom Compiler/Sprache bereitgestellt)

`integer`	ganze Zahlen	`1  -2`
`real`	reele Fließkomazahlen	`12.3  3.2e-7  3.2E-7  322.8E-7`
`complex`	komplexe Fließkomazahlen	`(1.0, -3e-12)  (-.12, 4.9)`
`logical`	logische Werte	`.true.  .TRUE.  .false.  .FALSE.`
`character`	Charakter/Zeichenkette	`'a'  'test'  "Test"  "a"`

- **Abgeleitete Datentypen (derived types)** (benutzerdefiniert)

```
type Coordinates
  real :: xx, yy
end type Coordinates

type Pixel
  type(Coordinates) :: coords
  integer :: color
end type Pixel
```

Kombination von intrinsichen Datentypen und/oder abgeleiteten Datentypen

Variablendeklaration

- **Explizite Variablendeklaration:**

```
program Test
  implicit none

  real :: xx, yy, area
  ! Alternativ:  real xx, yy, area
  real :: surface
  integer :: nPolygon
  integer :: i1, i2, i3

  ! Hier beginnt der Anweisungsteil
  xx = 12.0
  :
```

- Variablennamen sollen treffend sein, nicht zu kurz, nicht zu lang
- **Keine Variablennamen mit nur einer Buchstabe wählen** (erschwert die Suche nach der Variable)

Es sollte **auschließlich** die **explizite Variablendeklaration** (mit **implicit none**) verwendet werden!

- **Implizite Variablendekaration:**
  - Wenn implicit none nicht spezifiziert, müssen Variablen nicht explizit deklariert werden
  - Variablentyp hängt von der Anfangsbuchstabe ab (z.B. a-h, x, y, etc. real)

```
program NEVERDoThis
  xx = 12.0
  :
```

Konstante

Konstante = Variable, deren Wert während der Laufzeit nicht verändert werden darf

→ Wert einer Konstante **muss** bei der Deklaration festgelegt werden

```
program parameter_test
  implicit none

  real, parameter :: speedOfLight = 300000000.0
  complex, parameter :: sqrt1 = (0.0, 1.0)
  integer, parameter :: nDim = 2
  logical, parameter :: ja = .true.
  character(4), parameter :: name = "test"
  integer :: keineKonstante = 5                ! DON'T DO THIS

  write (*,*) speedOfLight
  speedOfLight = 1.0          ! Fehler: nicht veränderbar
  write (*,*) keineKonstante  ! Ergibt "5"
  keineKonstante = 12         ! Verursacht keinen Fehler
```

Variablenattribut

Wertzuweisung für normale Variablen im Deklarationsteil vermeiden!

→ Bei Unterprogrammen kann das zu unerwünschten Nebeneffekten führen (implizites SAVE-Attribut)

Wertebereiche

Wertebereich von integer, real und complex variablen ist darstellungsgrößenabhängig:

• real	4 byte	6 Stellen	+/-1.18E-38 .. +/-3.40E+38	(single precision)
	8 byte	15 Stellen	+/-2.23E-308 .. +/-1.80E+308	(double precision)
• integer	4 byte		-2147483647 .. 2147483647	(-2**31 .. 2**31)
	8 byte		-2**63 .. 2**63	

Alle reellen Variablen eines Programms sollten die selbe Genauigkeit haben.

Festlegung der globalen Genauigkeit

Genaugkeit sollte durch einen Parameter bestimmt werden:

```
integer, parameter :: dp = kind(1.0d0)
real(dp) :: xx, yy, zz

! Kleinste, größte, Genauigkeit
write (*,*) tiny(xx), huge(xx), precision(xx)
```

dp repräsentiert doppelte Genauigkeit

Alternativ man bestimmt den rellen Zahlentyp (kind-Parameter) anhand der Anzahl der Stellen und des Exponentenbereiches

```
! Real variablen mit mind. 12 Stellen
! Exponenten mind. zw. -99 und 99
integer, parameter :: dp = selected_real_kind(12, 99)

real(dp) :: xx, yy, zz
```

Arithmetische Operatoren

• Addition	+	a + b
• Subtraktion	-	a – b
• Multiplikation	*	a * b

• Division	/	a / b
• Exponentiation	**	a**b
• Negation	-	-a

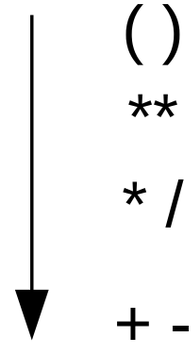
Regeln für arithmetische Operatoren:

- Zwei Operatoren dürfen nicht unmittelbar aufeinander folgen:
  falsch: a * -b richtig: a * (-b)
- Der Multiplikationsoperator darf nicht weggelassen werden:
  falsch: a (x + y) richtig: a * (x + y)
- Klammern werden von innen nach außen ausgewertet:

  *2**((7 * 4) / 14) = 2**(28 / 14) = 2**(2) = 2**2 = 4*

- Operatoren werden nach Priorität und von links nach rechts ausgewertet (außer Exponentiation)

Operatoren-priorität:



Datentypkonversion (#1)

Zwei Variablen, Konstanten oder Werte mit gleichen oder unterschiedlichen Datentypen mit +, -, * oder / verknüpft, Datentyp des resultierenden Ausdrucks:

a <op> b	**integer ::b**	**real(dp) :: b**
integer :: a	integer	real(dp)
real(dp) :: b	real(dp)	real(dp)

Datentypkonversion findet auch bei Operation mit Fließkommazahlen unterschiedlicher Genauigkeit statt:

a <op> b	**real(4)::b**	**real(8) :: b**
real(4) :: a	real(4)	real(8)
real(8) :: b	real(8)	real(8)

- Man sollte anstelle von impliziter Konversion explizite Typkonversion verwenden!
- Die Genauigkeit sollte bei Fließkommazahlen immer angegeben werden!
- Wird **keine Genauigkeit angegeben**, wird **einfache Genauigkeit** angenommen!

Datentypkonversion (#2)

Angabe der Darstellung bei Zahlenkonstanten: `aa = 13.5_dp`

Darstellungstyp der voranstehenden Zahl

Was ist die Ausgabe des folgenden Progammes? Wieso?

```
program conversion_demo                                    ! pm$ ConversionDemo
  implicit none                                            ! im$

  integer, parameter :: dp = selected_real_kind(8,10) !i$, pa$ ...
  integer :: ergebnis                                      !i$ ...

  ergebnis = 1.25 + 9 / 7
  write(*,*) '1.25 + 9 / 7 = ', ergebnis          ! wrs$ '1.25...'
  write(*,*) '1.25 + 9 / 7 = ', 1.25 + 9 / 7
  Write(*,*) '1.25 + 9 / 7 = ', 1.25 + 9 / real(7, dp)
  write(*,*) '1.25 +  9.0 / 7 = ', 1.25 + 9.0 / 7
  write(*,*) '1.25 + 9.0 / 7.0 = ', 1.25 + 9.0 / 7.0
  write(*,*) '1.25_dp + 9.0 / 7.0 =', 1.25_dp + 9.0 / 7.0
  write(*,*) '1.25_dp + 9.0_dp / 7.0_dp =', &
      & 1.25_dp + 9.0_dp / 7.0_dp
end program conversion_demo
```

Datentypkonversion – Exponentiation

Bei der Exponentiation sollte keine Typumwandlung vorgenommen werden:

```
real(dp) :: xx
write (*,*) xx**2           ! In Ordnung: xx * xx
write (*,*) xx**2.0_dp      ! Falsch: exp(2.0 * log(xx))
```

Es sind keine nicht ganzzahlige Exponenten von negativen Zahlen erlaubt:

```
program exp_error
  implicit none

  integer, parameter :: dp = selected_real_kind(9, 99)
  real(dp) :: xx, yy

  xx = -8.0_dp
  yy = 1.0_dp / 3.0_dp
  write(*,*)
  write(*,*) 'x = ', xx, '  y = ', yy
  write(*,*) 'x**y = ', xx**yy            ! Was passiert hier?

end program exp_error
```

Logische Operatoren – Vergleichsoperatoren

Vergleichsoperatoren:

• gleich	==	.EQ.
• ungleich	/=	.NE.
• größer als	>	.GT.
• größer oder gleich	>=	.GE.
• kleiner als	<	.LT.
• kleiner oder gleich	<=	.LE.

Beispiele:

3 == 4	.FALSE.
3 /= 4	.TRUE.
3 > 4	.FALSE.
3 >= 4	.FALSE.
3 < 4	.TRUE.
3 <= 4	.TRUE.

- Logische Operatoren haben niedrigere Priorität als arithmetische Operatoren:

```
7 + 3 < 2 + 11        äquivalent mit    (7 + 3) < (2 + 11)
```

- Bei logischen Operatoren gibt es auch implizite Typkonversion (bitte meiden!)

```
4.2_dp > 4        äquivalent mit    4.2_dp > real(4, dp)
```

explizite Konversion von integer nach real(dp)

Logische Operatoren – Verknüfpungsoperatoren

Logisches Und	.and.
• Logisches Oder	.or.
• Selbe Werte	.eqv.
• Entgegengesetzte Werte	.neqv.
• Negation	.not.

l1	.not. l1
.true.	.false.
.false.	.true.

l1	l2	l1 .and. l2	l1 .or. l2	l1 .eqv. l2	l1 .neqv. l2
.true.	.true.	.true.	.true.	.true.	.false.
.true.	.false.	.false.	.true.	.false.	.true.
.false.	.true.	.false.	.true.	.false.	.true.
.false.	.false.	.false.	.false.	.true.	.false.

- Verknüpfungsoperatoren haben eine niedrigere Priorität als Vergleichsoperatoren:

```
x > 0 .and. x < 3
```
äquivalent
```
(x > 0) .and. (x < 3)
```

- Logische Werte immer mit Verknüfpungsoperatoren vergleichen:

falsch: `(x1 > 0) == (x2 > 0)` richtig: `(x1 > 0) .eqv. (x2 > 0)`

Zeichenketten

Eine (statische) **Zeichenkette** enthält eine vorgegebene Anzahl von Charaktern:

```
charater(len=10) :: string   äquiv.   character(10) :: string
```

- **Verkettung** von Zeichenketten: ***//***

```
character(4) :: char1, char2
character(8) :: result
char1 = "any "
char2 = "book"
result = char1 // char2     ! "any book"
```

- **Substrings**

```
result = char1(1:3) // char2(1:)                    "anybook_"
write(*,*) len(result), len_trim(result), trim(result)   8 7 any book
result = char1(:3) // char1(:3) // char2            "anyanybo"
write (*,*) len(result), len_trim(result), trim(result) 8 8 anyanybo
result(1:2) = "BB"                                  "BByanybo"
```

- Bei Zuweisungen zwischen Zeichenketten verschiedener Länge wird abgeschnitten oder mit „Leercharakteren“ aufgefüllt.

- `write` schreibt immer die ganze Zeichenkette aus (inkl. Leercharakter)
  `trim()` reduziert die Zeichenkette aufs Wesentliche (ohne Leercharakter)

Verzweigungen: if

```
if (logischer_Ausdruck)
then
  Anweisung1
  :
  AnweisungN
end if
```

Wird nur dann ausgeführt, wenn ***logischer_Ausdruck*** `.true.` ergibt

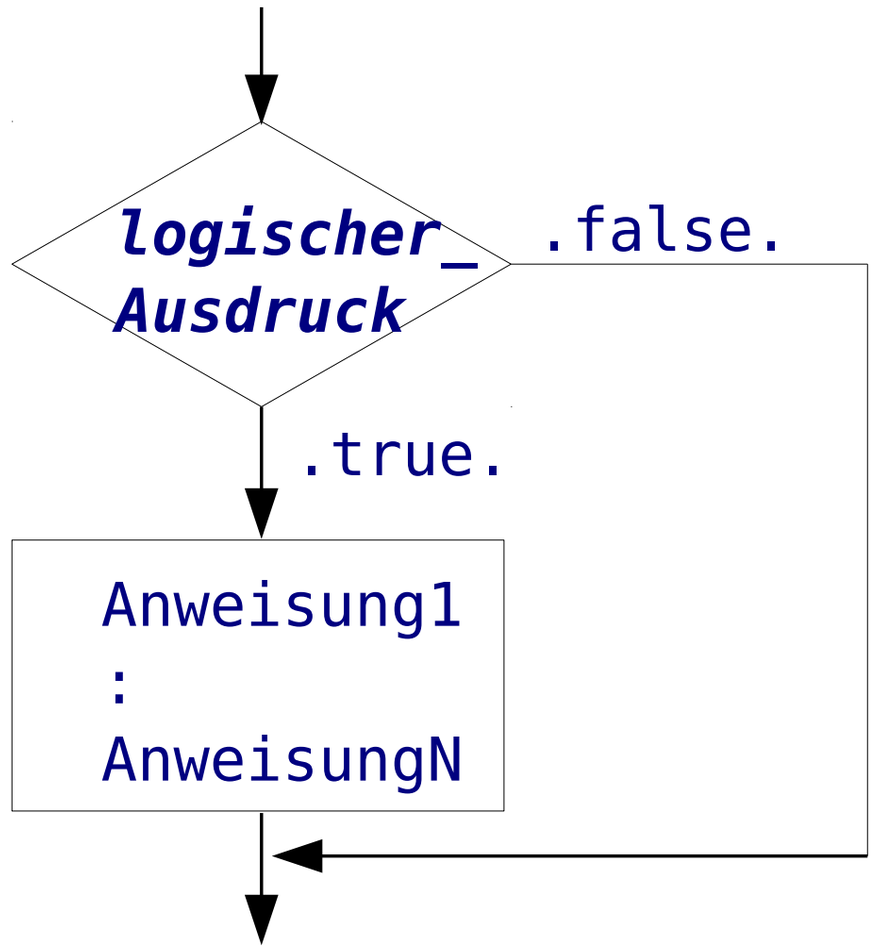
Einfacher Block:

```
if (xx < yy) then
  temp = xx
  xx = yy
  yy = temp
end if
```

Block mit Name:

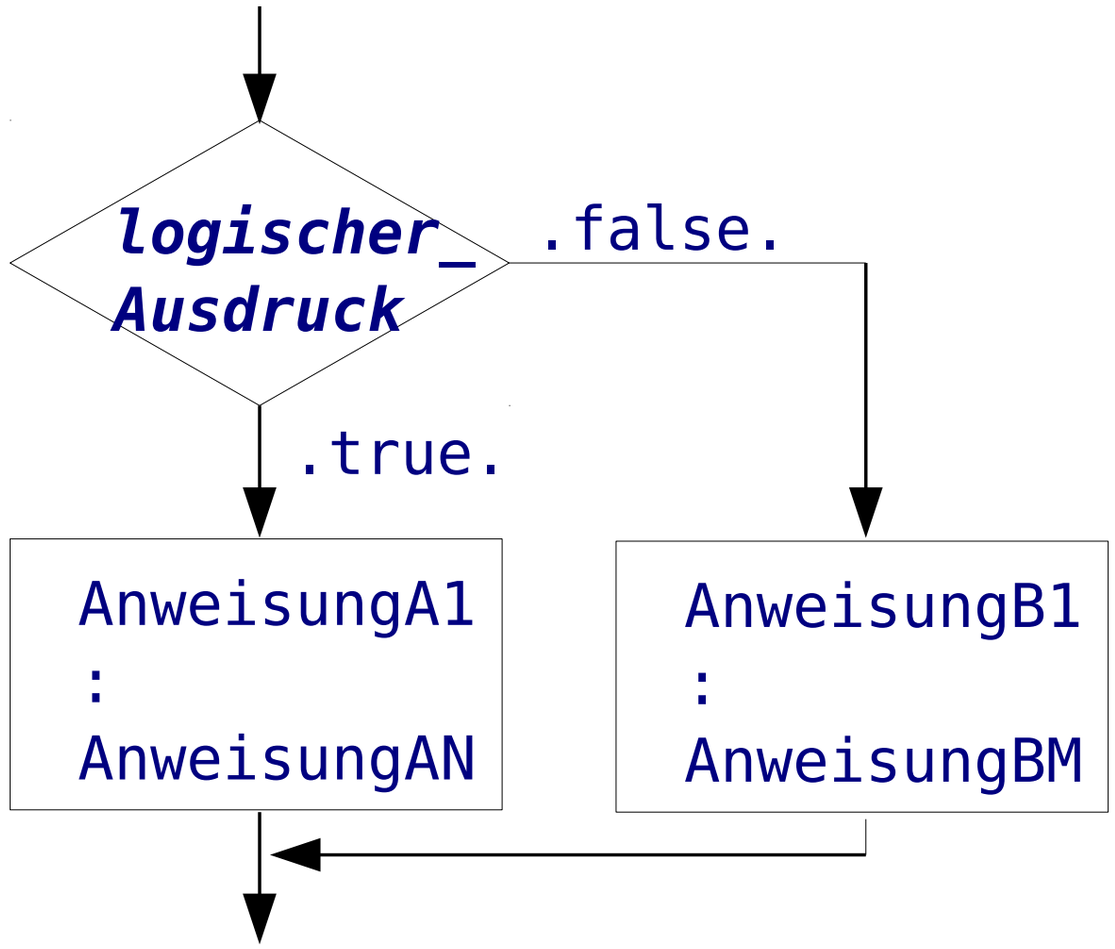
```
swap: if (xx < yy) then
  temp = xx
  xx = yy
  yy = temp
end if swap
```

- Blockname muss eindeutig sein.
- Alle Kontrollstrukturen (`if`, `case`, `do`) können mit Name versehen werden



Verzweigungen: if – else

```
if (logischer_Ausdruck) then
  AnweisungA1
  :
  AnweisungAN
else
  AnweisungB1
  :
  AnweisungBM
end if
```

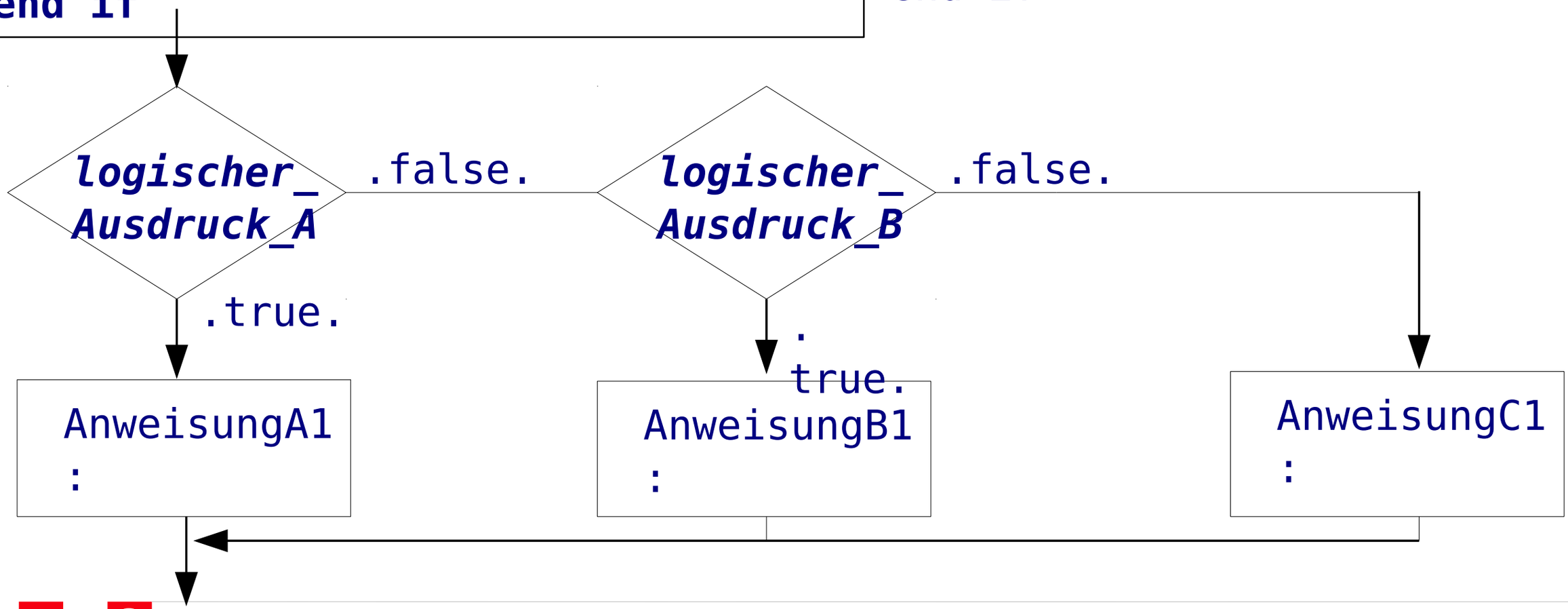



```
if (xx >= 0) then
  yy = sqrt(xx)
else
  write(*,*) "Wurzel nicht reell"
  stop
end if
```

```
check: if (xx >= 0) then
  yy = sqrt(xx)
else
  write(*,*) "Wurzel nicht reell"
  stop
end if check
```

Verzweigungen: if – else if – else if – ... – end if

```
if (logischer_Ausdruck_A) then
  AnweisungA1
  :
else if (logischer_Ausdruck_B) then
  AnweisungB1
  :
else
  AnweisungC1
  :
end if
```



- Es können beliebig viele else if Abfragen eingefügt werden.

```
if (mysign < 0) then
  write(*,*) "Negative"
else if (mysign == 0) then
  write(*,*) "Exact zero"
else
  write(*,*) "Positive"
end if
```

Verzweigungen: case

- Kann in gewissen Fällen das if – else if – ... – else – end if Konstrukt ersetzen:

```
select case (Ausdruck)
case (Werte1)
  Anweisungsblock1
case (Werte2)
  Anweisungsblock2
:
case default
  DefaultAnweisungsblock
end select
```

- ***Ausdruck*** muss von Typ **`character`**, **`logical`**, oder **`integer`** sein
- Im Gegensatz zu if – else if wird **nur ein Ausdruck ausgewertet**.
- Wert kann auch eine Aufzählung oder ein Intervall sein
- Es wird **maximal ein** (der erste passende) case Block ausgeführt

```
select case (number)
case (1)
  write (*,*) "Nur eines"
case (2,3,4,5)
  write (*,*) "Ein Paar"
case (6:)
  write (*,*) "Viele"
end select
```

```
select case (word(1:1))
case ("a":"z")
  write (*,*) "kleine Anfangsbuchstabe"
case ("A":"Z")
  write (*,*) "große Anfangsbuchstabe"
case default
  write (*,*) "Anfangscharakter keine &
      &Buchstabe"
end select
```

Schleifen: do

Iteration durch ein Intervall:

```
do Variable = Ausdr1, Ausdr2 [, Ausdr3]
  Anweisungsblock
end do
```

```
! Addition der geraden Zahlen 1-100
mysum = 0                    ! mysum is integer
do ii = 2, 100, 2
  mysum = mysum + ii
end do
```

- Schleifenvariable muss von Typ **`integer`** sein
- Schleifenvariable startet von ***Ausdr1***
- Schleifenvariable läuft bis ***Ausdr2***
- Schleifenvariable wird nach jedem Durchlaufen um ***Ausdr3*** erhöht
- Wenn ***Ausdr3*** fehlt, wird 1 angenommen
- ***Ausdr3*** kann auch negativ sein, dann muss jedoch ***Ausdr1*** **`>=`** ***Ausdr2***
- ***Ausdr1***, ***Ausdr2***, ***Ausdr3*** **dürfen nicht von der Schleifenvariable abhängen**

Schleifen: do

Iteration durch ein reelles Intervall:

```
! Durchlaufen eines reellen Intervalles
real(dp) :: xStart, xEnd, dx
integer :: nInter

xStart = 0.0_dp
xEnd = 100.0_dp
dx = 2.0_dp
nInter = ceiling((xEnd – xStart) / dx)
do ii = 0, nInter
  xx = xStart + real(ii, dp) * dx
  Andere Anweisungen mit xx
end do
```

- `ceiling(xx)` gibt den kleinsten **integer**größer als xx (xx ist **real**)

Explizite Konversion von Integer nach real(dp)

Schleifen: do while

```
do while (logischer_Ausdruck)
  Anweisungen
end do
```

Wird solange ausgeführt, bis ***logisher_Ausdruck*** `.true.` ergibt.

```
integer :: mysum, input
mysum = 0
do while (mysum <= 21)
  write(*,*) "Geben Sie eine Zahl ein: "
  read(*,*) input
  mysum = mysum + input
end do
```

Addiert die eingegebenen Zahlen bis 21 erreicht ist

```
do
  Anweisungen
end do
```

- Wird unendlich lange ausgeführt
- Es muss ein Abbruchkriterium innerhalb des Blockes implementiert werden!

Schleifen: Austritt

- **exit** Verlassen der Schleife, Programmausführung nach *end do*
- **cycle** Zurück zum Schleifenkopf (bei Iterationen Schleifenvariable erhöht/erniedrigt)

```
! Generator für gerade Zahlen (ineff.)
do ii = 1, 100
  if (mod(ii,2) == 1) then
    cycle
  end if
  write(*,*) "Naechste gerade Zahl:", ii
end do
```

```
! Summe solange Eingabe pos.
mysum = 0
do
  write(*,*) "Naechste Zahl:"
  read(*,*) input
  if (input < 0) then
    exit
  end if
  mysum = mysum + input
end do
```

Bei verschachtelten Schleifen kann der **Sprungziel durch den Blocknamen** ausgewählt werden:

```
do ii = 1, 100
  do jj = 1, ii
    :
  end do
end do
```

effizientere Version

Elemente der unteren Dreiecksmatrix

```
schleife1: do ii = 1, 100
  schleife2: do jj = 1, 100
    if (jj > ii) then
      cycle schleife1
    end if
    :
  end do schleife2
end do schleife1
```

Aufgabe 1

- Programm conversion_demo ausprobieren und verstehen, warum die Ergebnisse der einzelnen Ausdrücke unterschiedlich sind!

Aufgabe 2

- Ein Programm, das integer Variable *limit* einliest und die ersten *limit* Glieder einer Fibonacci-Reihe ausschreibt.
  (Die ersten beiden Glieder einer Fibonacci-Reihe sind 1. Alle weitere Glieder ergeben sich als die Summe der vorangehenden zwei Glieder.)
- Es sollen so lange neue Fibonacci-Reihen mit neuem Wert für *limit* berechnet und ausgeschrieben werden bis eine negative Zahl oder Null als *limit* eingegeben wird.
- Was passiert, wenn man für *limit* 100 eingibt?
- Wie könnte man den Überlauf verhindern, sodass es unabhängig von der Repräsentationsgröße eines Integers funktioniert?
  (Hinweis: mit `huge(xx)` kann die größte darstellbare Zahl für den Datentyp von `xx` abgefragt werden.)

Aufgabe 3 (Bonusaufgabe)

- Strings mit Länge 20 oder kürzer einlesen, Charaktere der Strings sortieren und wieder ausgeben.
- Vorgang sollte solange wiederholt werden, bis das Wort `STOP` eingegeben wird.
- Möglicher Input/Output

```
hgzelomndz      ->    deghlmnozz
 test           ->     estt
 STOP
```

- Sortierung mittels Bubblesort-Algorithmus (siehe z.B. Wikipedia: http://de.wikipedia.org/wiki/Bubblesort)