Wissenschaftliches Programmieren

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

6. Unterprogramme, Module

Git-Bemerkungen

- Alle Dateien stagen, die sich geändert haben und dann einchecken:

```
git add -u
git commit
```

- Typischer Git-Logeintrag besteht aus zwei Teilen:

  Kurzer Zusammenfassung (weniger als 70 Charektere)
  + Leerzeile und evtl. detailiertere Beschreibun nach einer Leerzeile

```
LU decomposition implemented.
```
← Zusammenfassung

```
Matrix A is first decomposed into L, U and P. System of equations
solved by subsequent forward and backward substitution.
```
↑ Detailierte Beschreibung

- Dateien, die **Git ignorieren** soll, können in der **.gitignore** Datei angegeben werden:

```
*~
a.out
```
} .gitignore ← Sollte auch unter Versionskontrolle gestellt werden

Alle Dateien, die auf „~“ enden und die Datei a.out

Fortran-Bemerkungen

- Wenn bei Subarrayausdrücken der Indexbereich die **leere Menge** ist, hat das resultierende Array die **Größe Null**

`array(1:0)` ← Kein Element mit Index größer gleich 1 und kleiner gleich 0 → `size(array(1:0))` → `0`

- Zuweisung zwischen Arrays der Größe Null ist erlaubt und wird einfach **ignoriert**:

```
! Exchange corresponding rows below the diagonal of L
xx(1:ii-1) = ll(ii,1:ii-1)
ll(ii,1:ii-1) = ll(imax,1:ii-1)
ll(imax,1:ii-1) = xx(1:ii-1)
```

Wird bei ii=1 ignoriert

- Skalarprodukt zweier Vektoren der Größe-Null ist per Definition **Null**:

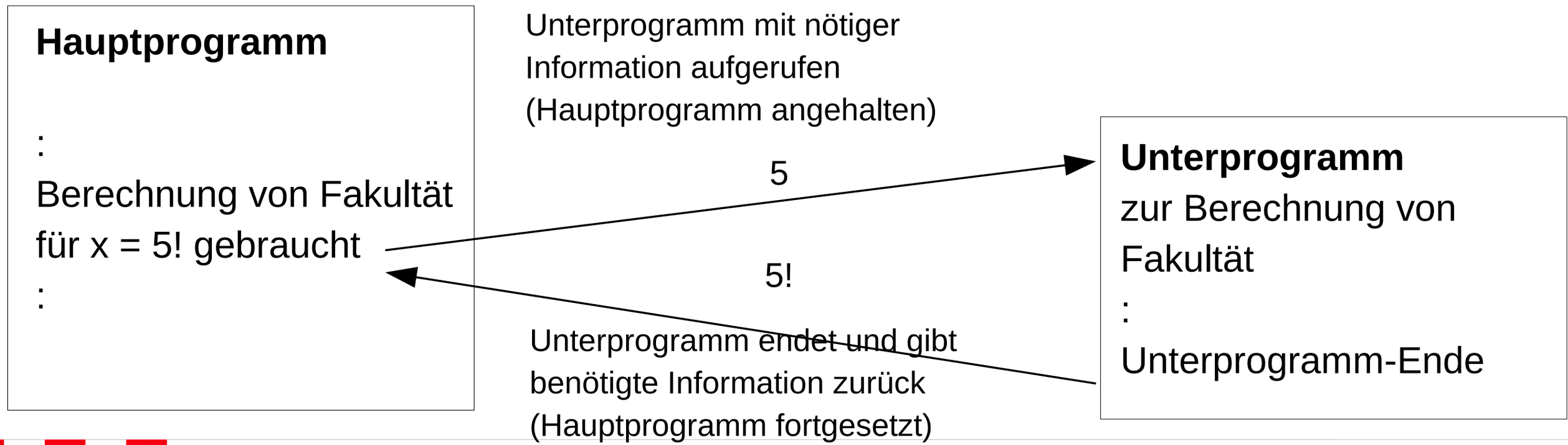
```
! Back substitution
do ii = nvar, 1, -1
  tmp = bb(ii) - dot_product(aa(ii,ii+1:nvar), xx(ii+1:nvar))
  xx(ii) = tmp / aa(ii, ii)
end do
```

Wenn ii = nvar, haben diese Vektoren die Größe Null! → `tmp = bb(1)`

Unterprogramme

Unterprogramm = Separates Programmteil, das mit anderen Teilen des Programmes durch eine festgelegte Schnittstelle kommuniziert

- Aufteilung eines Problems/eines Programmes in kleinere Teilschritte
- Jeder logischer, in sich abgeschlossene Teilschritt kann als eigenes Unterprogramm unabhängig vom restlichen Programmcode entwickelt und getestet werden
- Einzelne Unterprogramme bilden in sich geschlossene, separate Einheiten. Informationsaustausch nur über definierte Schnittstellen.



Vorteile von Unterprogrammen

- **Unabhängiges Entwickeln** und Testen der Teilaufgaben
  - Programmentwicklung und Programmcode übersichtlicher, klarer strukturiert
  - Programm bei Veränderungen weniger Fehleranfällig
- Unterprogrammen sind **wiederverwendbar**
  - Entwicklung- und Testzeit kürzer
- **Klare Programmstruktu**r
  - Schnittstellen zwischen Programmeinheiten klar geregelt
  - (Im Optimalfall) keine Kommunikation zwischen Unterprogrammen außer den definierten Schnittstellen.
  - Variablen in den Unterprogrammen können nicht von draußen modifiziert werden (data hiding)
- Für andere Programmteile sind die **Details der Implementierung** in einem Unterprogramm **versteckt**. (Implementierung kann verbessert werden, ohne die anderen Teile ändern zu müssen.)

Subroutinen

Definition:

```
subroutine Subroutinenname(Liste der Formalparameter)
  ! Datentypdeklaration der formalen Parameter
  :
  ! Datentypdeklaration der lokalen Variablen
  :
  ! Anweisungsteil der Subroutine
  :
end subroutine Subroutinenname
```

Aufruf:

```
call Subroutinenname(Liste der Aktualparameter)
```

Subroutinen – Beispiel

```
program TestFactorial
  implicit none

  integer :: inp, res

  write (*,"(A)", advance="no") "Enter an integer: "
  read (*, *) inp
  call calc_factorial(inp, res)
  write (*, "(A,I0)") "The result is: ", res
end program TestFactorial
```

Aktualparameter
(*actual arguments*)

```
subroutine calc_factorial(num, fact)
  integer, intent(in) :: num
  integer, intent(out) :: fact

  integer :: ii

  fact = 1
  do ii = 2, num
    fact = fact * ii
  end do
end subroutine calc_factorial
```

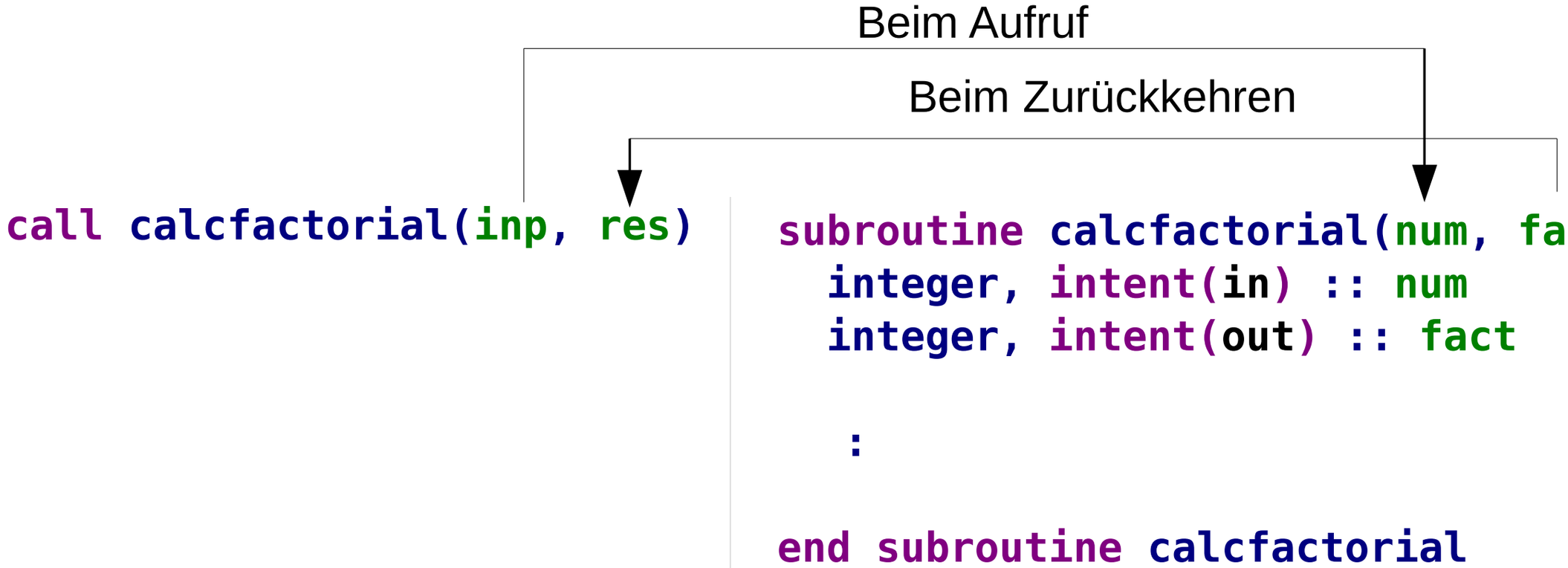
Formalparameter
(*dummy arguments*)

Typdeklaration der Formalparameter

Deklaration der lokalen Variable(n)

Subroutinen-anweisungen

Übergabe von Parametern



- Die **intent(in)** Parameter werden **beim Aufruf** mit den entsprechenden Werten vom aufrufenden Programm initialisiert.
- Die **intent(out)** Parameter, werden **beim Verlassen** des Unterprogrammes mit dem Wert des zugehörigen Parameters aufgefüllt.
- Anzahl und Typ der beim Aufruf übergebenen Parameter müssen exakt der subroutine-Definition entsprechen.
- Die Namen der als Parameter übergebenen Variablen müssen nicht mit den Variablennamen der formalen Parameter identisch sein.

intent(in)-Parameter

- Aufrufendes Programm muss sich darauf verlassen können, dass der Aktualparameter für einen **intent(in)** Formalparameter **nicht** während der Unterprogrammausführung **verändert** wird.

```
real(dp) :: rtmp
rtmp = 2.0_dp       ! rtmp is assigned the value 2.0_dp
call test(rtmp)     ! rtmp corresponds to an intent(in) parameter
write (*,*) rtmp    ! rtmp must be still 2.0_dp
```

- **intent(in)**-Parameter **dürfen** im Unterprogramm **nicht verändert** werden

```
subroutine test(xx)
  real(dp), intent(in) :: xx

  xx = 2.0_dp                ! Compiler error
  :
end subroutine test
```

- Aktualargument für intent(in)-Parameter darf auch eine Konstante oder ein Ausdruck sein:

```
real(dp), parameter :: pi = 3.14_dp
call test(pi)                       ! Passing constant for intent(in)
call test(sin(rtmp) – 3.0_dp)       ! Passing expression for intent(in)
```

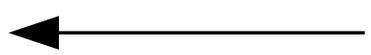
intent(in)-Parameter (#2)

- Unterprogramm muss sich darauf verlassen können, dass den **intent(in)** Formalparametern entsprechende Aktualparameter **vor dem Aufruf** schon **initialisiert** worden sind:

```
program UndeterministicBehaviour
  implicit none

  real(dp) :: alpha

  call test(alpha)
  :
```

Falsch, da der Aktualparameter beim Aufruf noch keinen definierten Wert hat!
(Erzeugt keinen Compilierungsfehler)

```
subroutine test(number)
  real(dp), intent(in) :: number

   write (*,*) number
end subroutine test
```



Nicht-deterministisch, wenn aufrufendes Programm den intent(in) Parameter vorher nicht initialisiert hat

intent(out)-Parameter

- Aufrufendes Programm muss sich darauf verlassen können, dass dem zum **intent(out)**-Formalparameter gehörende Aktualparameter **in der Subroutine** ein **Wert zugewiesen** wird.

```
real(dp) :: xx
:                              ! No value assigned to xx yet
call setvariable(xx)           ! passing xx as intent(out) arg.
write (*, *) xx                ! xx must have a well defined value now
```

- **intent(out)**-Parametern **muss** im Unterprogramm ein **Wert zugewiesen werden**

```
subroutine setvariable(xx)
  real(dp), intent(out) :: xx

  write (*,*) "setvariable called"
end subroutine setvariable
```

Falsch, da xx kein Wert zugewiesen wurde.

- Aktualargument für **intent(out)**-Parameter darf **weder Konstante noch Ausdruck** sein:

```
real(dp), parameter :: pi = 3.14_dp
call setvariable(pi)                    ! Compiler error
call setvariable(sin(rtmp) - 3.0_dp)    ! Compiler error
```

intent(inout)-Parameter

- **intent(inout)**-Parameter verhalten sich
  - **beim Aufruf** des Unterprogrammes wie **intent(in)**-Parameter
  - **beim Verlassen** eines Unterprogrammes, wie **intent(out)**-Parameter

```
real(dp) :: xx

xx = 2.0_dp
call doublevalue(xx)
write (*, *) xx   ! 4.0
```

```
subroutine doublevalue(value)
  real(dp), intent(inout) :: value

  value = 2.0_dp * value

end subroutine doublevalue
```

- Aktualparameter für intent(inout)-Formalparameter müssen vor dem Aufruf schon initialisiert worden sein
- Aktualparameter für intent(inout)-Formalparameter dürfen weder Konstanten noch Ausdrücke sein
- intent(inout)-Formalparametern sollte (muss aber nicht!) im Unterprogramm ein Wert zugewiesen werden.
- Wenn intent(inout)-Formalparameter im Unterprogramm kein neuer Wert zugewiesen wird, behält er den Wert, den er vorm Aufruf hatte.

Übergabe von Arrays

- Arrays können genauso übergeben werden wie skalare Typen:

```
real(dp) :: latvecs(2,2)

latvecs(:,1) = [ 1.0_dp, 0.0_dp ]
latvecs(:,2) = [ 0.0_dp, 1.0_dp ]

call rotatebasis_2d(latvecs, 30.0_dp)
...
```

```
subroutine rotatebasis_2d(basis, degree)
  real(dp), intent(inout) :: basis(2, 2)
  real(dp), intent(in) :: degree

  ! Rotating basis vectors with given amount
  :
end subroutine rotatebasis_2d
```

Übergabe von Arrays (#2)

- Es können auch **Arrays mit unbekannter Größe** übernommen werden.
- Arraygröße für die einzelnen Dimensionen kann im Unterporgramm mit **size()** abgefragt werden.

```
real(dp) :: latvecs(2,2)

latvecs(:,1) = [ 1.0_dp, 0.0_dp ]
latvecs(:,2) = [ 0.0_dp, 1.0_dp ]

call calcinverse(latvecs)
...
```

```
subroutine calcinverse(basis, degree)
  real(dp), intent(inout) :: basis(:,:)
  real(dp), intent(in) :: degree

  ! Sanity check
  if (size(basis, dim=1) /= size(basis, dim=2)) then
    write (*,*) "calcinverse: Invalid parameter (non-square matrix)"
    stop
  end if
  :
end subroutine calcinverse
```

Übergabe von Arrays (#3)

- Die Anzahl der Dimensionen (**Arrayrang**) muss **beim Aktual- und Formalparameter identisch** sein

```
subroutine rotatebasis(basis, degree)
  real(dp), intent(inout) :: basis(:,:)
  real(dp), intent(in) :: degree
  :
end subroutine rotatebasis
```

← Formalparameter **2D**

```
real(dp) :: vec(2)
vec = [ 2.0_dp, -1.0_dp ]
call rotatebasis(vec, 30.0_dp)
```

Falsch, da Aktualparameter **1D**

Übergabe von Arrays (#4)

- Es können auch **Subarrays als Aktualparameter** verwendet werden, sofern sie die richtige Anzahl von Dimensionen besitzen
- Subarrays können sowohl für alle Parametertypen (intent(in), intent(out) und intent(inout)) verwendet werden.

```
subroutine vectorproduct(vec1, vec2, res)
  real(dp), intent(in) :: vec1(:), vec2(:)
  real(dp), intent(out) :: res(:)

  ! Calculates vec1 x vec2 and stores it in res
   :
end subroutine vectorproduct
```

```
real(dp) :: latvecs(3,3)

latvecs(:,1) = [ 1.0_dp, 0.0_dp, 0.0_dp ]
latvecs(:,2) = [ -0.5_dp, sqrt(3.0_dp) / 2.0_dp, 0.0_dp ]
call vectorproduct(latvecs(:,1), latvecs(:,2), latvecs(:,3))
! latVecs(:,3) is now 0.0_dp 0.0_dp 1.0_dp
```

Übergabe allokierbarer Arrays

- Wenn ein allokierbares Array des Hauptprogramms **im Unterprogramm allokiert/deallokiert** werden soll (**und nur dann!**), muss das Formalparameter das Attribut **allocatable** haben.

```
program Test
  implicit none

  real(dp), allocatable :: array1(:,:), array2(:)

  allocate(array1(10, 10))
  call testarrays(array1, array2)
  write (*,*) allocated(array2)                    ! True
  :
```

```
subroutine testarrays(test1, test2)
  real(dp), allocatable, intent(in) :: test1(:,:)
  real(dp), allocatable, intent(inout) :: test2(:)

  write (*,*) "Allocated 1: ", allocated(test1)    ! True
  write (*,*) "Allocated 2: ", allocated(test2)    ! False
  allocate(test2(100))
  :
```

Array mit Allokierungsstatus übergeben

Übergabe allokierbarer Arrays (#2)

- Allokierungstatus von **intent(in) allokierbaren** Arrays kann in der Routine **nicht verändert** werden (intent(in) Arrays können weder allokiert noch deallokiert werden)

```
program Test
  implicit none

  integer, allocatable :: aa(:,:)

  call testarray(aa)
  :
```

```
subroutine testarray(array)
  integer, allocatable,&
    & intent(in) :: array(:,:)

  allocate(array(10,10))  ! Comp.Error

end subroutine testarray
```

- **intent(out) allokierbare** Arrays werden beim Routinenaufruf **automatisch deallokiert**

```
program Test
  implicit none

  integer, allocatable :: aa(:,:)

  allocate(aa(10, 10))
  write (*,*) allocated(aa) ! T
  call testarray(aa)
  write (*,*) allocated(aa) ! F
```

```
subroutine testarray(array)
  integer, allocatable,&
    & intent(out) :: array(:,:)

  write(*,*) "allocated(array) ! F

end subroutine testarray
```

Funktionen

- Funktionen verhalten sich äquivalent zu Subroutinen, **geben** aber zusätzlich einen **Wert** (Skalar oder Feld) **zurück**
- In der Funktion wird der Rückgabewert durch eine Variable repräsentiert, die den selben Namen und Typ hat, wie die Funktion selbst:

Definition:

```
Rückgabewert-Typ function Funktionenname(Liste der Formalparameter)
  ! Datentypdeklaration der formalen Parameter
  :
  ! Datentypdeklaration der lokalen Variablen
  :
  ! Anweisungsteil der Funktion
  :
  Funktionenname = ...
  :
end function Funktionenname
```

Aufruf:

```
Funktionenname(Liste der Aktualparameter)
```

Funktionen

- Der **Rückgabewert** einer Funktion muss **in einem Ausdruck** verwendet werden:

```
integer :: num

num = factorial(5) + 12
```

```
integer function factorial(num)
  integer, intent(in) :: num

  integer :: ii

  factorial = 1
  do ii = 2, num
    factorial = factorial * ii
  end do
end function factorial
```

- **Funktionen** sollten grundsätzlich **nur intent(in)**-Parameter besitzen

```
integer function BAD_STYLE(arg1, arg2)
  integer, intent(inout) :: arg1
  integer, intent(out) :: arg2
```

Sollte vermieden werden!

- Wenn ein Unterprogramm mehrere Variablen modifizieren/zurückgeben soll, sollte es als Subroutine implementiert werden.

Alternative Deklarationen des Rückgabewertes

- Deklaration als **Variable mit dem selben Name wie die Funktion**:
  (Spezifikation der Rückgabewerttypes vor dem Funktionnamen entfällt)

```
function factorial(num)
  integer, intent(in) :: num
  integer :: factorial
```

- Als Variable mit beliebigem Namen via **result()**-Anweisung.
  In der Funktion muss für die Zuweisung des Rückgabewertes statt des Funktionnamens dieser Variablenname verwendet werden:

```
function factorial(num) result(fact)
  integer, intent(in) :: num
  integer :: fact

  integer :: ii

  fact = 1
  do ii = 2, num
    fact = fact * ii
  end do
end function factorial
```

Rekursive Unterprogramme

- Wenn ein Unterprogramm direkt oder indirekt (über andere Unterprogramme) sich selbst aufruft, muss es als **rekursiv** deklariert werden:
- Bei rekursiven Funktionen muss Typ und Variablenname für den Rückgabewert via result() deklariert werden:

```
recursive subroutine test(arg1, ...)
:
end subroutine test
```

```
recursive function factorial(number) result(res)
  integer, intent(in) :: number
  integer :: res

  if (number > 1) then
    res = number * factorial(number – 1)
  else
    res = 1
  end if
end function factorial
```

Nicht besonders effizient!

Verlassen eines Unterprogrammes

- Ein Unterprogramm (Funktion oder Subroutine) wird beendet, wenn die **end** Anweisung erreicht wird
- Vorzeitiger Ausstieg aus dem Unterprogramm kann mit **return** erzwungen werden:

```
recursive function factorial(number) result(res)
  integer, intent(in) :: number
  integer :: res

  if (number == 1) then
    res = 1
    return
  end if
  res = number * factorial(number – 1)

end function factorial
```

Unterprogramm wird verlassen, es werden keine weitere Anweisungen im Unterprogramm ausgeführt

Module

- Logisch zusammengehörende Unterprogramme sollten in größere Einheiten (sog. „Modulen“) zusammengefasst werden.

```
module Modulname

Allgemeiner Deklarationsteil
(Variablen, Konstante, Intefaces)

contains

Definition der Unterprogramme

end module Modulname
```

Wenn das Modul keine Unterprogrammdefinitionen enthält, fällt die „contains“-Anweisung weg.

```
module mymath
  implicit none

  integer :: modvar  ! Modulvariable

contains

  function factorial(num) result(res)
    integer, intent(in) :: num
    integer :: res

    :
  end function factorial


  subroutine transform(...)
  :
  end subroutine transform

end module mymath
```

Module

- Inhalt eines Moduls kann mit dem **use**-Befehl importiert werden
- Das importierende Programm bekommt Zugriff auf alle Variablen und Unterprogamme des Moduls. (Nicht aber zu den lokalen Variablen der Unterprograme im Modul!)
- Es dürfen beliebig viele Module importiert werden (via entsprechender Anzahl von „use“-Befehlen)
- Die use-Anweisungen müssen vor der **implicit none** Anweisung stehen.

```
program TestFactorial
  use mymath              ! Imports factorial, transform etc. von mymath
  implicit none

  integer :: num

  read (*,*, advance="no") num
  ! Calling module function
  write (*,"(I0,A,I0)") num, "! = ", factorial(num)
  ! Manipulating module variable
  modvar = 12

end program TestFactorial
```

Module

- Module können nicht nur Unterprogramme, sondern auch Konstanten und Variablen enthalten und diese für andere Programme bereitstellen

```
! Defining extra module for accuracy (precision) settings to make
! sure, that the same accuracy is used throughout the entire code.
module accuracy
  implicit none

  integer, parameter :: dp = selected_real_kind(15,300)

end module accuracy
```

contains-Anweisung fehlt, da keine Unterprogramme im Modul

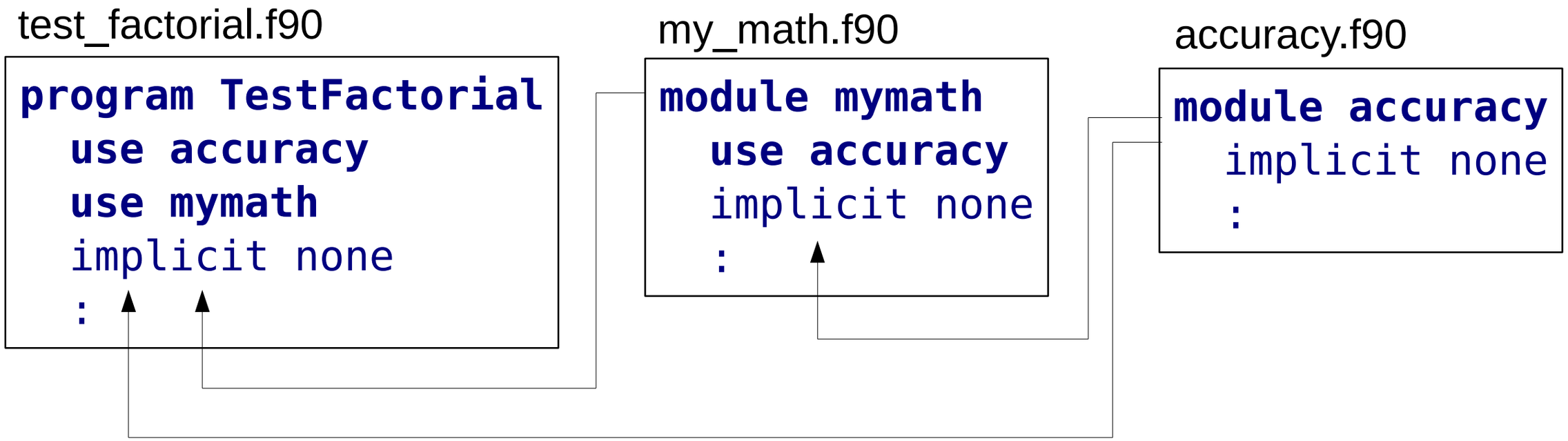
```
program TestFactorial
  use accuracy
  use mymath
  implicit none

  real(dp) :: rTmp
```

```
module mymath
  use accuracy
  implicit none

contains
  subroutine test(xx)
    real(dp), intent(in) :: xx
    :
```

Ein Modul kann gleichzeitig in mehrere Module/Programmteile importiert werden.

Compilierung von Modulen

- Jedes Modul sollte in einer separaten Datei gespeichert werden
  (nicht zwingend, aber gute Praxis)
- Es müssen alle Abhängigkeiten eines Modules compiliert werden, **bevor** das Modul selbst compiliert werden kann:



- Compilierungsreihenfolge: accuracy.f90, my_math.f90, test_factorial.f90:

```
gfortran -o test_factorial accuracy.f90 my_math.f90 test_factorial.f90
```

Bemerkungen

- Ein Unterprogramm soll exakt nur eine Aufgabe machen
- Die Aufgabe soll aus Namen des Unterprogrammes eindeutig sein.
- Wenn das Unterprogramm nicht treffend benannt werden kann, ist diese wohl zu komplex.
- Unterprogramme sollten nach Möglichkeit nicht verschiedene Code-Ebenen (high level Aufrufe und low-level Berechnungen) miteinander mischen.

Bemerkungen

- Nützliche Abkürzungen in Emacs (wenn .emacs_abbrevs verwendet wird):

`subroutine`	**`sub$`**
`function`	**`fu$`**
`intent(in)`	**`ini$`**
`intent(out)`	**`ino$`**
`intent(inout)`	**`inio$`**
`module`	**`mo$`**
`contains`	**`cn$`**
`end subroutine`	**`end`** ***`[Enter]`***
`end function`	**`end`** ***`[Enter]`***
`end module`	**`end`** ***`[Enter]`***
`allocate`	**`al$`**
`allocatable`	***`ab$`***

Aufgabe

Spalten Sie dass LR-Zerlegungsprogramm in folgende Module bzw. Subroutinen auf:

- **Modul accuracy**
  - Konstante zur Festlegung der Darstellungsgröße für Fließkommazahlen (dp)
- **Module eqsolver:**
  - Subroutine ludecompose()

    In: Quadratische Matrix (A)

    Out: Obere/Untere Dreiecksmatrix L/R, Permutationsmatrix P
  - Subroutine substituteback()

    In: Obere/Untere Dreiecksmatrix L/R, Permutationsmatrix P, Vektor b

    Out: Lösungsvektor (x)
- **Module io:**
  - Subroutine readinput()

    In: Unallokierte Matrix (A) und Vektor (B), Dateiname

    Out: Allokierte Matrix (A) und Vektor (B) eingelesen aus der Datei gauss.inp.
  - Subroutine writeoutput()

    Input: Lösunsgvektor b
- **Hauptprogramm**

  Ruft readinput() auf, dann ludecompose() und substituteback() und schließlich writeoutput()

Aufgabe (#2)

- Ergänzen Sie die README-Datei mit den nötigen Compilierungsinformationen!
- Checken Sie den neuen Quellcode (eventuell auch sinvolle Zwischenschritte bei der Entwicklung) ins Repository ein.
  (Vergessen Sie nicht die neu zugekommenen Dateien unter Versionsverwaltung zu stellen!)

Bonusaufgabe (Optimierung)

- Überlegen Sie sich, wie Sie mit möglich wenig Speicher auskommen können. Insbesonders sollten Sie die Anzahl der quadratischen Matrizen reduzieren, indem Sie den Inhalt der drei Matrizen A, L und R in nur einer Matrix speichern und statt der Matrix P nur einen Vektor (deren Länge der Anzahl der Variablen entspricht) verwenden können.

LR-Zerlegung

- In der Numerik wird die Gauss-Elimination meistens als LR-Zerlegung realisiert. Dabei wird die Matrix A als Produkt **P** * **A** = **L** * **R** zerlegt, wobei **P** eine Permutationsmatrix, **R** eine obere Dreiecksmatrix (gauss-eliminierte Form des Gleichungssystems) und **L** eine untere Dreiecksmatrix mit 1-en in der Diagonale ist, die die Koeffizienten für die Linearkombinationen der Zeilen enthält.

```
0.0  0.0  1.0       2.0  4.0  4.0       1.0  0.0  0.0     5.0  4.0  2.0
1.0  0.0  0.0   *   1.0  2.0 -1.0   =   0.4  1.0  0.0   * 0.0  2.4  3.2
0.0  1.0  0.0       5.0  4.0  2.0       0.2  0.5  1.0     0.0  0.0 -3.0
```

- Gleichungssystem **A** x = **b** wird mit Hilfe von **P**, **L** und **R** in 2 Schritten gelöst:

  L y = **P** **b** durch „vorwärtseinsetzen“ nach y aufgelöst.

  R **x** = y durch „rückwärtseinsetzen“ nach x aufgelöst.

Beispiel für die LR-Zerlegung

P			L			R		
1.0	0.0	0.0	1.0	0.0	0.0	2.0	4.0	4.0
0.0	1.0	0.0	0.0	1.0	0.0	1.0	2.0	-1.0
0.0	0.0	1.0	0.0	0.0	1.0	5.0	4.0	2.0

Tausche Zeilen 1,3 (Pivot)

0.0	**0.0**	**1.0**	1.0	0.0	0.0	**5.0**	**4.0**	**2.0**
0.0	1.0	0.0	0.0	1.0	0.0	1.0	2.0	-1.0
1.0	**0.0**	**0.0**	0.0	0.0	1.0	**2.0**	**4.0**	**4.0**

- 0.2 * Z.1

- 0.4 * Z.1

Bilde Linearkomb. um aktuelle Spalte auszunullen, speichere Koeffs in L

0.0	0.0	1.0	1.0	0.0	0.0	5.0	4.0	2.0
0.0	1.0	0.0	**0.2**	1.0	0.0	**0.0**	**1.2**	**-1.4**
1.0	0.0	0.0	**0.4**	0.0	1.0	**0.0**	**2.4**	**3.2**

Zeilentausch (Pivot)
Tausche auch **in allen entsprechenden Spalten unterhalb der Diagonale in L**

0.0	0.0	1.0	1.0	0.0	0.0	5.0	4.0	2.0
1.0	**0.0**	**0.0**	**0.4**	1.0	0.0	**0.0**	**2.4**	**3.2**
0.0	**1.0**	**0.0**	**0.2**	0.0	1.0	**0.0**	**1.2**	**-1.4**

- 0.5 * Z.2

Bilde Linearkomb. um aktuelle Spalte auszunullen

0.0	0.0	1.0	1.0	0.0	0.0	5.0	4.0	2.0
1.0	0.0	0.0	0.4	1.0	0.0	0.0	2.4	3.2
0.0	1.0	0.0	0.2	**0.5**	1.0	**0.0**	**0.0**	**-3.0**

Beispiel für Vorwärts- und Rückwärtseinsetzen

b		**P * b**
1.0	→	4.0
2.0		1.0
4.0		2.0

Vorwärtseinsetzen:

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.4 & 1.0 & 0.0 \\ 0.2 & \mathbf{0.5} & 1.0 \end{pmatrix} * \begin{pmatrix} Y1 \\ Y2 \\ Y3 \end{pmatrix} = \begin{pmatrix} 4.0 \\ 1.0 \\ 2.0 \end{pmatrix}$$

→

```
Y1 = 4.0
Y2 = 1.0 – 0.4 * Y1 = -0.6
Y3 = 2.0 – 0.2 * Y1 – 0.5 * Y2 = 1.5
```

Rückwärtseinsetzen:

$$\begin{pmatrix} 5.0 & 4.0 & 2.0 \\ 0.0 & 2.4 & 3.2 \\ 0.0 & 0.0 & -3.0 \end{pmatrix} * \begin{pmatrix} \mathbf{X1} \\ \mathbf{X2} \\ \mathbf{X3} \end{pmatrix} = \begin{pmatrix} 4.0 \\ -0.6 \\ 1.5 \end{pmatrix}$$

→

```
X3 = 1.5 / -3.0 = 0.5
X2 = (-0.6 – 3.2 * X3) / 2.4 = 0.41666...
X1 = (4.0 – 4.0 * X2 – 2.0 * X3) / 5.0 = 0.66666...
```