

# Unterprogramme und Modularisierung

## Sonderübung

Nadine Schärmann & Thomas Klütz

PR10

15. Dezember 2020

# Inhalt

# Rekursive Unterprogramme

Ein rekursives UP ruft sich selbst auf (direkt oder indirekt, d. h. über andere UPe). Ein **RECURSIVE** - Attribut im UP-Kopf ist notwendig. Funktionen brauchen zusätzlich noch eine **RESULT (res)** -Klausel.

## Beispiel

```
RECURSIVE FUNCTION fibo (n) RESULT (res)
  INTEGER :: n
  INTEGER :: res

  IF (n == 0 .OR. n == 1) THEN
    res = 1
  ELSE
    res = fibo (n-1) + fibo (n-2)
  END IF
END FUNCTION
```

**ACHTUNG:** Abbruchbedingung nicht vergessen!

# Übungsaufgabe

## Aufgabe

Unser Taschenrechner soll noch mehr Funktionalität erhalten.  
Schreibe Unterprogramme, die die folgenden Berechnungen durchführen:

- Rekursive Berechnung von Potenzen mit ganzzahligen Exponenten
- **ZUSATZ:** Die Primzahleigenschaft einer natürlichen Zahl  $p$  kann durch Ausprobieren aller potentiellen Teiler von 2 bis  $z - 1$  überprüft werden. Es geht aber auch rekursiv. Die Funktion  $\text{PrimzahlTest}(p)$  sei wie folgt mit Hilfe der rekursiven Funktion  $\text{istPrimzahl}(p, z)$  definiert:

$$\text{PrimzahlTest}(p) := \text{istPrimzahl}(p, p - 1)$$

$$\text{istPrimzahl}(p, z) := \begin{cases} .\text{true}., & \text{falls } z = 1, \\ .\text{false}., & \text{falls } p \text{ durch } z \text{ teilbar ist,} \\ \text{istPrimzahl}(p, z - 1), & \text{falls } p \text{ nicht durch } z \text{ teilbar ist.} \end{cases}$$

# Werte- und Referenzparameter

## Werteparameter (= value parameter)

- Ergebniswert des aktuellen-Argument-Ausdrucks wird dem formalen Argument zugewiesen
- verhält sich anschließend wie lokale Variable im UP

## Referenzparameter (= reference parameter)

- f. A. wird mit a. A. (Variable) assoziiert
- d. h. wird der Wert des f. A. im UP verändert, so ändert sich auch der Wert des a. A.

## WICHTIG: In Fortran gibt es nur Referenzparameter.

Das heißt, wenn ihr eine Variable in einem Unterprogramm verändert, ändert sich ihr Wert dauerhaft (d.h. auch nach Beendigung des UP ist der Wert anders).

# Attribute für formale Argumente

## INTENT(IN)

- das assoziierte a. A. muss einen Wert haben
- Wert des a. A. darf im UP nicht verändert werden
- so dürfen auch Ausdrücke übergeben werden

## INTENT(OUT)

- f. A. muss zu Beginn des UPs (noch) keinen def. Zustand haben
- f. A. muss am Ende des UPs einen definierten Zustand haben
- Ausdrücke dürfen nicht übergeben werden

## INTENT(INOUT)

- Alles erlaubt (lesen und schreiben aus der/ in die Variable)

# Was ist Modularisierung?

- Idee: komplexes System nach dem Baukastenprinzip aus Einzelbausteinen zusammensetzen
- findet im alltäglichen Leben Anwendung, aber auch in vielen Bereichen der Informatik (insbesondere bei der Entwicklung von Programmen)
- UPe entsprechen einfacher Umsetzungen des Modularisierungsprinzips
- UPe als Bausteine bei der Entwicklung von Programmen verwendbar, falls die UPe als eigenständige Verarbeitungseinheiten konzipiert sind

# Was ist Modularisierung?

## Grundlegende Struktur eines Moduls

```
MODULE < Modulname >  
  IMPLICIT NONE  
  PRIVATE          ! anything that is not declared PUBLIC is PRIVATE  
  PUBLIC :: < Exportliste >  
  ...  
  CONTAINS  
    < UPe >  
END MODULE [< Modulname >]
```



# Übungsaufgabe

Erstelle ein Modul mit allen zu dem Taschenrechner gehörigen Funktionen.

# Vorteile von Modularisierung



# Vorteile von Modularisierung

- **Strukturierung** von Programmen: strukturierte Programme können schneller verstanden und leichter überarbeitet werden
- 
- 
-

# Vorteile von Modularisierung

- **Strukturierung** von Programmen: strukturierte Programme können schneller verstanden und leichter überarbeitet werden
- **Vermeidung von Coderedundanz**: gleicher Programmcode muss nicht mehrfach geschrieben werden
- 
-

# Vorteile von Modularisierung

- **Strukturierung** von Programmen: strukturierte Programme können schneller verstanden und leichter überarbeitet werden
- **Vermeidung von Coderedundanz**: gleicher Programmcode muss nicht mehrfach geschrieben werden
- **Wiederverwendung** von Programmteilen: unabhängige Programmteile in verschiedenen Programmen verwendbar, somit schnellere und zuverlässigere Entwicklung von Programmen
-

# Vorteile von Modularisierung

- **Strukturierung** von Programmen: strukturierte Programme können schneller verstanden und leichter überarbeitet werden
- **Vermeidung von Coderedundanz**: gleicher Programmcode muss nicht mehrfach geschrieben werden
- **Wiederverwendung** von Programmteilen: unabhängige Programmteile in verschiedenen Programmen verwendbar, somit schnellere und zuverlässigere Entwicklung von Programmen
- **Kollaboration**: mehrere Programmierer können unabhängig voneinander an verschiedenen Programmteilen arbeiten

# Schnittstellen (INTERFACE-Blöcke)

## Spezifische Schnittstellen

eindeutig definiert durch:

- Information, ob Subroutine oder Funktion
- Prozedurnamen/ Operatornamen bzw. -symbol/ Zuweisungssymbol (=)
- Liste der Argument- bzw. Operandentypen
- nur bei Funktionen: Ergebnistyp

## Generische Schnittstellen und Überladung

- Überladung = mehrfache Verwendung von Prozedurnamen bzw. Operatornamen/ -symbolen (+, −, \*, ..., auch: =) für verschiedene Prozeduren
- jede Überladung muss eindeutige **Schnittstelle** haben

# Generische Schnittstellen und Überladung

Bei der Überladung eines Operators, müssen die Funktionsargumente immer das Attribut **INTENT(IN)** haben!

```
INTERFACE < generi_name >  
    MODULE PROCEDURE <function_1> [, ... , <function_i>]  
                        bzw. <subroutine_1> [, ... , <subroutine_i>]  
END INTERFACE
```

```
INTERFACE OPERATOR (+)                                oder: -, *, /, <, .NOT., ... , .MYOPE.  
    MODULE PROCEDURE <function_1> [, ... , <function_i>]  
  
END INTERFACE
```

```
INTERFACE ASSIGNMENT (=)  
    MODULE PROCEDURE <subroutine_1> [, ... , <subroutine_i>]  
END INTERFACE
```



# Generische Schnittstellen und Überladung - ein Beispiel

```
MODULE shifted
  IMPLICIT NONE
  ...

  INTERFACE OPERATOR (.S.)
    MODULE PROCEDURE SHIFT
  END INTERFACE
  ...

CONTAINS

  FUNCTION shift(word,Anzahl)
    CHARACTER(*), INTENT(IN) :: word
    INTEGER, INTENT(IN) :: anzahl
    CHARACTER(LEN(word)):: shift

    shift=' '
    shift(anzahl+1 : len(word)) = word(1 : len(word) - anzahl)
    shift(1 : anzahl) = word(len(word) - anzahl + 1 : len(word))
  END FUNCTION
  ...

END MODULE
```

# Übungsaufgabe

Überlade die Funktion, die den Binomialkoeffizient berechnet, mit dem Operator *.UEBER.* und die Funktion zur Berechnung der Potenz mit dem Operator *.HOCH.*

Nun müssen nicht mehr die Funktionen selbst, sondern ihre Operatorüberladungen als *PUBLIC* deklariert werden.