

Unterprogramme und Modularisierung

Sonderübung

Luise Zieger & Lea Happel

PROG Sonderübung

November 27, 2018

Inhalt

1. Wiederholung - Rekursion
 - Übungsaufgabe
2. Werte- und Referenzparameter
3. Attribute für formale Argumente
 - INTENT(IN)
 - INTENT(OUT)
 - INTENT(INOUT)
 - Aufgabe
4. Modularisierung
 - Prinzip der Modularisierung
 - Übungsaufgabe
 - Vorteile von Modularisierung
5. Schnittstellen (INTERFACE-Blöcke)
 - Generische Schnittstellen
 - Übungsaufgabe

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 faku (n) RESULT (res)
  INTEGER :: n
  INTEGER :: res

  IF (n == 0 ) THEN
    res = 1
  ELSE
    res = n* faku (n-1)
  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 der zu einer ganzen Zahl gehörigen Dreieckszahl ($1 + \dots + n$)
- Berechne Potenzen mit ganzzahligen Exponenten rekursiv
- **Zusatz** Rekursive Berechnung der Binomialkoeffizienten unter Ausnutzung von $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$

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
- garantiert, dass die Variable (a. A.) nach dem UP-Aufruf unverändert ist
- so dürfen auch Ausdrücke übergeben werden

Attribute für formale Argumente

INTENT(OUT)

- f. A. muss zu Beginn des UPs (noch) keinen def. Zustand haben
- daher ist lesender Zugriff auf das f. A. zu Beginn des UPs unzulässig
- f. A. muss am Ende des UPs einen definierten Zustand haben
- garantiert, dass die übergebene Variable (a. A.) nach dem UP einen def. Zustand besitzt
- Ausdrücke dürfen nicht übergeben werden

Attribute für formale Argumente

INTENT(INOUT)

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

Aufgaben

Betrachte noch einmal deine Taschenrechnerfunktionen. Welche Variablen sollte man sinnvollerweise mit `INTENT(IN)`-Attributen versehen? Füge dies ein. Gibt es eine Stelle, bei der `INTENT(OUT)` sinnvoll wäre?

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 :: < Exportlist >  
  ...  
  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

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))=wort(1:len(wort)-anzahl)
    shift(1:anzahl)=wort(len(wort)-anzahl+1:len(wort))
END FUNCTION
...
END MODULE
```

Übungsaufgabe

Überladen die Funktion, die den Binomialkoeffizient berechnet mit dem Operator `.BIN. .`