Elementare Schritte

- Ein **elementarer Berechnungsschritt** eines Algorithmus ändert im Allgemeinen den Wert von Variablen
  - **Zuweisungsoperation** von fundamentaler Bedeutung
  - Zuweisungsoperator
    - In Pascal :=
    - In C, C++, Java =

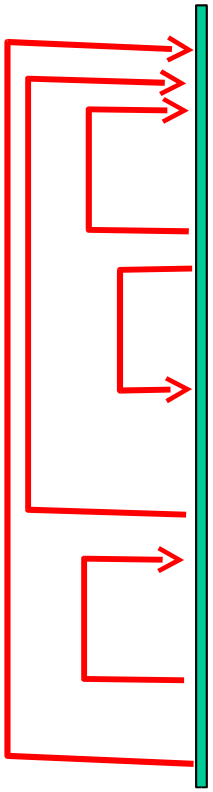
Sprünge - GOTO

- Die Konstruktion „Fahre fort mit Schritt 2“ stellt einen **Sprung** (**GOTO**) im Steuerungsverlauf dar
- Elementarste Form, eine Wiederholung oder sonstige Verzweigung im Ablauf auszudrücken
- Jedoch: Sprünge gelten (im allgemeinen) als äußerst **schlechtes Mittel** der Strukturierung
  - Verlauf wird verworren und unübersichtlich
  - **Klassiker** von Edsger W. Dijkstra (1968): *Go To Statement Considered Harmful*

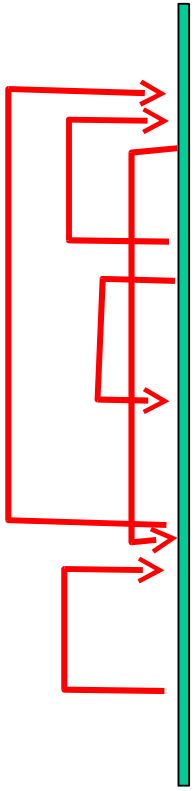
Strukturierte Sprünge

- Darum **Einschränkung**
  - Sprung nur zum Anfang einer Schleife
  - Schleifen der Flussdiagramme sind höchstens ineinander geschachtelt
  - Schleifen überkreuzen sich nicht

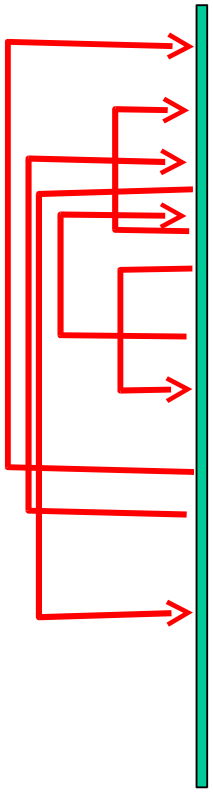
Vergleich strukturierte / beliebige Sprünge



Strukturiert



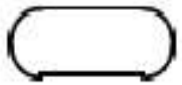
Beliebig



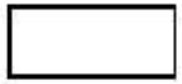
Spaghetti

Flussdiagramme

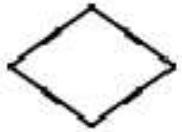
- Steuerungsverlauf von Algorithmen kann mit **Flussdiagrammen** (flow charts) graphisch dargestellt werden
  - **Symbole**



Beginn / Ende des Algorithmus



Anweisungen, Operationen

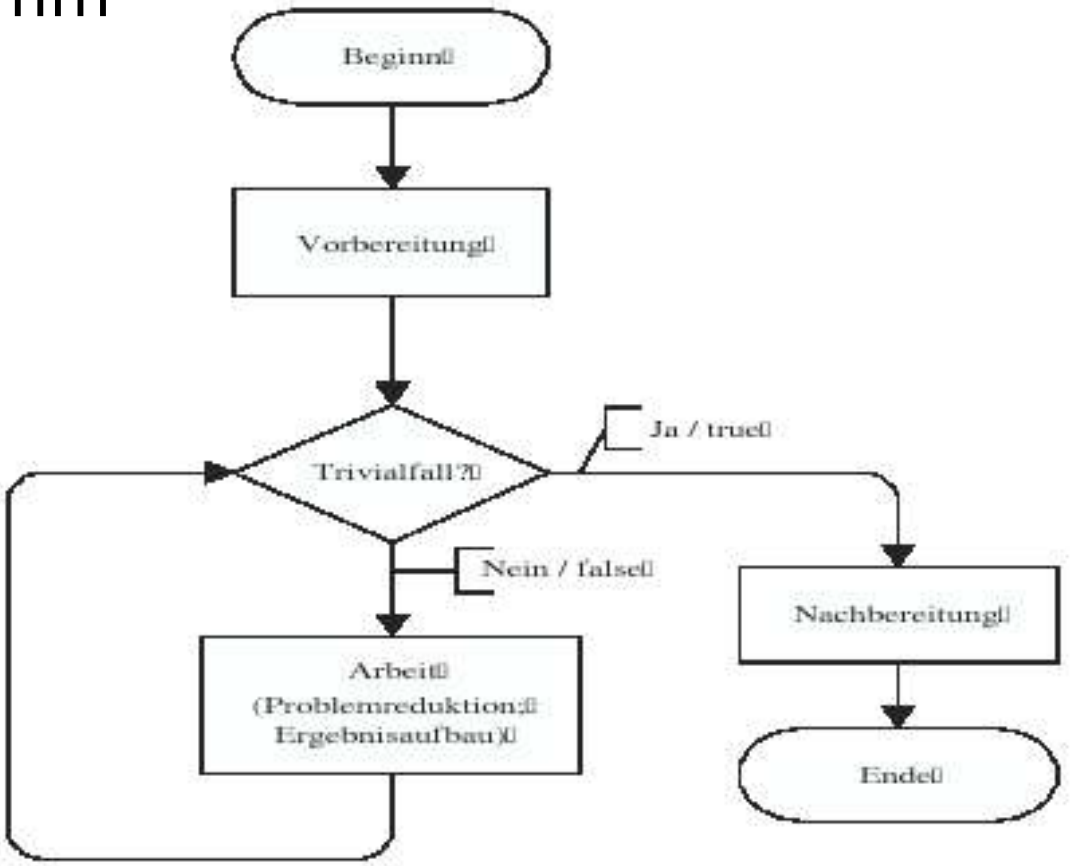


Verzweigungen: Der weitere Verlauf hängt vom Wahrheitswert des Ausdrucks im Inneren ab.

  - Symbole werden mit **Pfeilen** verbunden

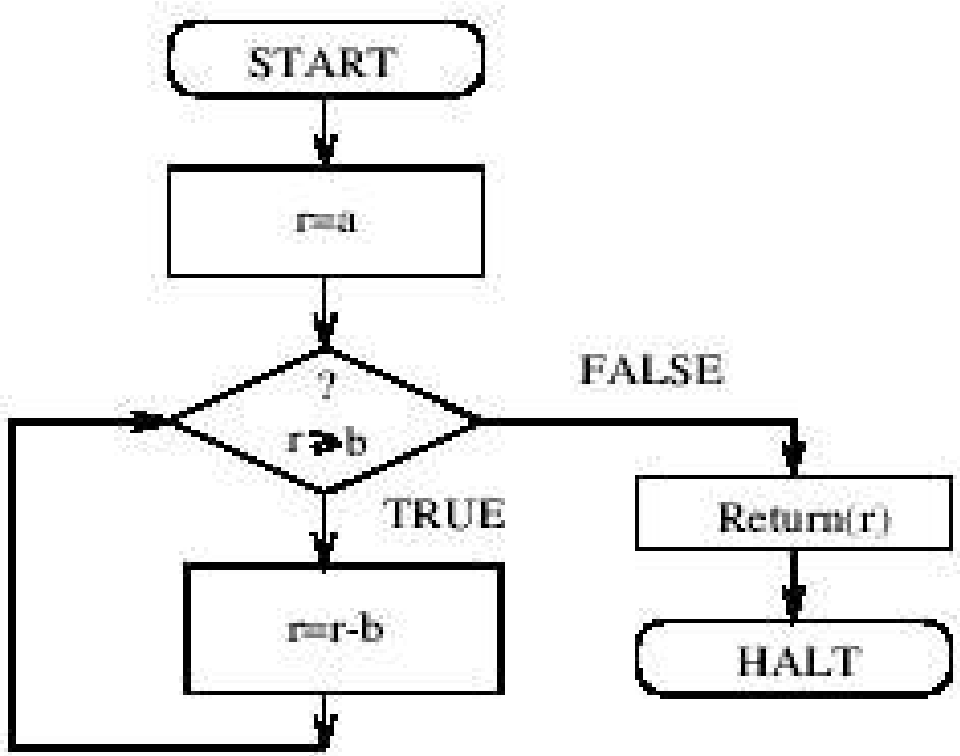
Flussdiagramme

- Grundschema des Algorithmenaufbaus als Flussdiagramm



Modulus-Funktion als Flussdiagramm

- **Beispiel:** Flussdiagramm für Modulus-Funktion



Strukturierte iterative Schleife - WHILE

- Strukturierte Schleife wird mit **WHILE**-Konstrukt beschrieben

  while (*Bedingung*) **do** {*Schleifenrumpf*}

  – Bei Eintritt in die WHILE-Schleife wird zunächst die Bedingung ausgewertet
    - Beim Wert **wahr** (true) wird der Schleifenrumpf einmal ausgeführt und danach zur Bedingung zurückgesprungen
    - Beim Wert **falsch** (false) wird die Schleife (ohne weitere Ausführung des Rumpfes) beendet
  – In Java kein „do"

Strukturierte Verzweigung - IF

- Strukturierte Verzweigung wird mit **IF**-Konstrukt beschrieben

  if (*Bedingung*) **then** {*A*} **else** {*B*}

  – Bei Eintritt in das IF-Konstrukt wird zunächst die Bedingung ausgewertet
    - Beim Wert **wahr** (true) wird *A* ausgeführt und dann zum Ende (über *B* hinweg) gesprungen
    - Beim Wert **falsch** (false) wird über *A* hinweggesprungen und *B* ausgeführt
  – In Java kein „**then**"

Strukturierte iterative Schleife - WHILE

- Die WHILE-Schleife entspricht also der Konstruktion

```
M:   if(Bedingung)
       {Anweisungssequenz;
        goto M;
       }fi
```

Beispiel: Modulus-Funktion mit IF und WHILE

- mod(a,b)

```
r = a;
while (r ≥ b) do {
      r = r-b;
}
return r;
```

.

Schleifenbildung durch Rekursion

- Problem *P* mit Eingabe *X* wird nach folgendem Schema **rekursiv** gelöst
  - **Basisfall**:
    Falls *X* einfacher Natur ist, gibt Lösung direkt an
  - **Rekursionsfall**:
    - Konstruiere aus X einfachere Eingabe *X'*
    - Löse *P* für *X'* **<- Rekursiver Aufruf**
    - Konstruiere Lösung für *X* aus Lösung für *X'*

Rekursive Beschreibungsform: Beispiel

- Berechnung der Modulusfunktion mod(*a*,*b*)

```
if (a < b) then {
      result = a;
} else {
      a'  = a-b;
      result = mod(a',b);
}
return result;
```

Euklidischer Algorithmus zur ggT-Berechung

- Weiteres Beispiel für die rekursive Beschreibungsform:
  Euklidischer Algorithmus zur Berechnung des größten gemeinsamen Teilers (**ggT**) zweier ganzer Zahlen

$$\mathrm{ggT}(a,b) = \begin{cases} a & b = 0 \\ \mathrm{ggT}(b,a) & b > a \\ \mathrm{ggT}(b, a \bmod b) & \text{sonst} \end{cases}$$

Rekursion und funktionale Programmierung

- Rekursive Problemlösung ist die Hauptdenkweise, die **funktionale Programmiersprachen** unterstützen
  - Etwa LISP, Scheme, ML, Haskell
- Vorzug liegt in großer **Eleganz** und **Kompaktheit** (gerade bei kleinen Lehrbuchbeispielen)
- Rekursive und iterative Beschreibungsformen sind **gleich mächtig**

Verifikation iterativer Schleifen

- **Partielle Korrektheit** iterativer Schleifen kann mit Hilfe einer geeigneten **Schleifeninvarianten** bewiesen werden

- **Definition Schleifeninvariante**
  Formel *INV*, die stets (bei jedem Durchlauf) vor und nach Ausführung des Schleifenrumpfes wahr ist

Verifikation iterativer Schleifen

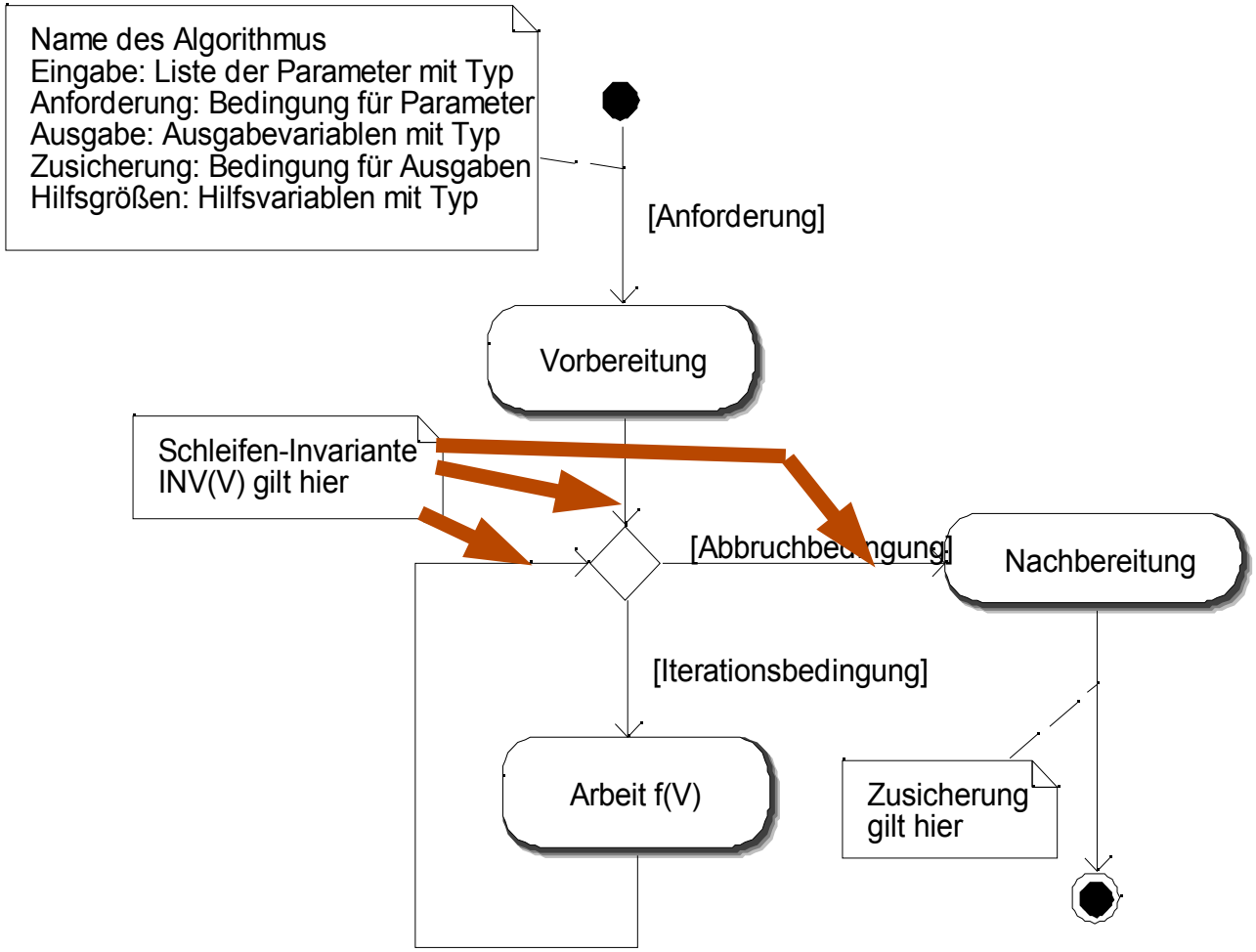
- Zum **Nachweis**, dass *INV* Schleifeninvariante ist, ist **zu zeigen**:
  - *INV* gilt vor dem ersten Durchgang
  - Wenn *INV* vor einer Ausführung des Rumpfes gilt, dann auch hinterher
    (dabei kann man verwenden, dass der Rumpf nur ausgeführt wird, wenn die Schleifenbedingung gilt)

Verifikation iterativer Schleifen

- **Verwendung** einer Invarianten:
  Wenn *INV* Invariante einer Schleife ist, dann folgt, dass direkt nach der letzten Ausführung
  - *INV* wahr ist
  - und sowieso auch:
    die Schleifenbedingung falsch ist
- **Terminierung** ist damit **nicht bewiesen**
  (nur partielle Korrektheisaussage!)

Verifikation iterativer Schleifen

Grundschema der Iteration mit Schleifen-invariante



Verifikation iterativer Schleifen: Beispiel

- mod(*a*,*b*)

  r = *a*;
  while (*r* ≥ *b*) **do** {
  r = *r*-*b*;
  }
  return *r*;

- **Invariante** INV:
  - *a* mod *b* = *r* mod *b* und
  - *r* ≥ 0