Kontrollstrukturen Kap 3

strukturierte Anweisung

- zusammengesetzte Anweisung
- bedingte Anweisung
- wiederholende Anweisung

Prototyp ist if Anweisung

in Java if (Bedingung) Anweisung_1 else Anweisung_2

können wieder strukturiert sein
also auch wieder if Anweisungen

Fallunterscheidungen mit geschachtelten if Anweisung

in Java

```
        if ( Bed 1 ) {
einrücken →   Anweisung_1
        } else if ( Bed_2 ) {
         →    Anweisung_2
        } else if (Bed_3) {
         →    Anweisung_3
        } else {
         →    Anweisung_4
        }
```

↑ Klammern von Java Codeconvention so vorgeschlagen

Für Fallunterscheidungen eigene Kontrollstrukturen

Pseudocode <u>case</u>

case B of

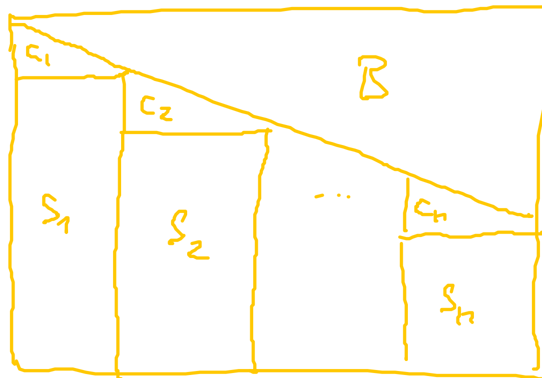
c_1 : S_1

c_2 : S_2

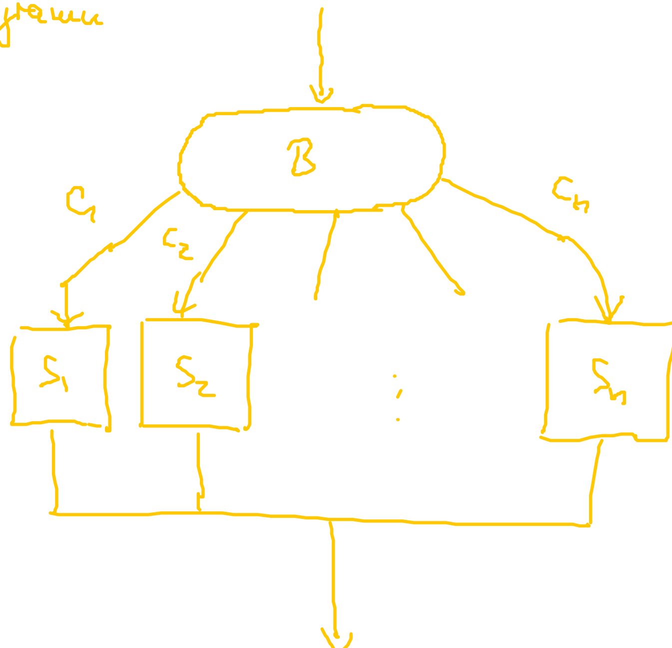
⋮

end

Struktogramm



Flussdiagramm



Java: switch statement (braucht dafür break, später)

3.4.3 Wiederholung

Prototyp ist while Anweisung

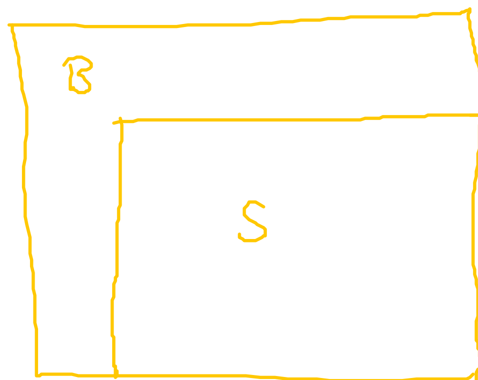
Pseudocode while B do S

B: Boolescher Ausdruck ergibt true oder false

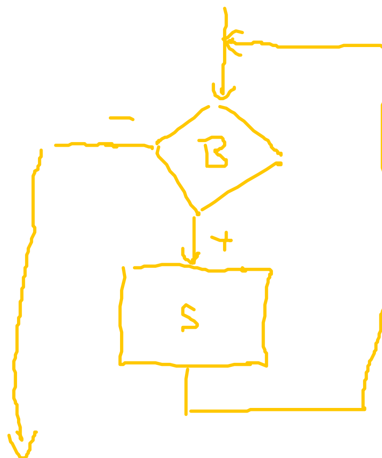
S: wird bei true ausgeführt, dann wird B wieder überprüft

bei false wird Schleife verlassen

Struktogramm



Flussdiagramm



zeigt Schleifencharakter deutlich

```
Java    while ( B ) {
            Anweisungen S
        }
```

Beispiel: Berechnung ggT von zwei Zahlen $x, y \in \mathbb{N}$, $x, y > 0$

Bsp: $ggT(12, 16) = 4$

verwenden Euklidischen Algorithmus
beruht auf mathematischen Aussage

LEMMA:

> Sind $a, b \in \mathbb{N}$ und $a > b > 0$ so gilt
> $$ggT(a, b) = ggT(a-b, b)$$

liefert mit Wiederholung eines einfachen Prinzip zur Berechnung des ggT

Zahlenbeispiel

28	→	16	→	4	→	4	→	4
12		12		12		8		4

↑ hier klar $ggT(4,4) = 4$

$\Rightarrow ggT(28, 12) = 4$

Beweis des Lemmas:

zeige: a, b und $a-b, b$ haben dieselben Teiler

Menge gemeinsamer Teiler $\overset{a)}{\subseteq}$ $\supseteq^{b)}$ Menge gemeinsamer Teiler ← diese sind gleich

$\Rightarrow$ dann haben a, b und $a-b, b$ auch denselben größten gemeinsamen Teiler

Zeige a) Ist k ein Teiler von a und von b, so ist k auch ein Teiler von $a-b$ und b

b) Ist k ein Teiler von $a-b$ und von b, so ist k auch ein Teiler von a und von b

zu a) Sei k ein Teiler von a und von b

$\Rightarrow$ $\exists$ natürliche Zahlen r, s mit $a = r \cdot k$, $b = s \cdot k$ und $r > s$

$\Rightarrow$ $a - b = r \cdot k - s \cdot k = \underbrace{(r-s)}_{\text{natürliche Zahl } > 0} k$

$\Rightarrow$ k ist Teiler von $a-b$

$\Rightarrow$ a) gezeigt

zu b) Sei k ein Teiler von $a-b$ und b ← klar nach Voraussetzung

(zu zeigen: k ist Teiler von a und b)

$\Rightarrow$ $\exists$ Zahlen $\underbrace{r', s'}_{\in \mathbb{N}, > 0}$ mit $a-b = r' \cdot k$, $b = s' \cdot k$

$\Rightarrow$ $a = a - b + b = r' \cdot k + s' \cdot k = \underbrace{(r' + s')}\cdot k$

natürl. Zahl

$\Rightarrow$ k ist Teiler von a

Algorithmus 3.2 (ggT)

Input: natürliche Zahlen $x, y > 0$

Output: $ggT(x,y)$

```
a := x;
b := y;
while a ≠ b do
    if a > b then a := a - b;      ⇐ Anwendung von □
             else b := b - a
    endif
    { ggT(a,b) = ggT(x,y) }
endwhile
{ ggT(x,y) = a }      ⇐ nach Beenden der while Schleife

return a;
```

SATZ Algo 3.2 berechnet für zwei beliebige positive natürliche Zahlen x, y ihren größten gemeinsamen Teiler

zeigt Korrektheit des Algorithmus

Beweis: □ garantiert, dass nach jedem Durchlauf der while Schleife $ggT(a,b) = ggT(x,y)$ gilt

Am Ende der while Schleife ist $a = b$

und damit $ggT(x,y) = ggT(a,a) = a$

bei Halt des Algo wird also das korrekte Ergebnis berechnet

Frage: kommt es immer (d.h. bei beliebigem Input x, y) zum Halt?

Ja: in jeder Iteration wird entweder a oder b um mindestens um 1 kleiner und a und b bleiben positive natürliche Zahlen
⇒ man kann while Schleife nur endlich oft durchlaufen

maximale Anzahl Durchläufe der Schleife $\leq \max\{x, y\}$

APPLET
JAVA CODE

Beachte: unzulässige Eingaben (negative Zahlen) werden abgefangen
hier notwendig, da bei negativen Zahlen eine Endlosschleife möglich wäre.

<u>Varianten der while Anweisung</u>

bei while wird immer zunächst die Bedingung geprüft und statement nur ausgeführt wenn Bedingung gilt

oft möchte man statement auf jeden Fall einmal ausführen und erst dann die Bedingung prüfen

Postchecking statt Prechecking

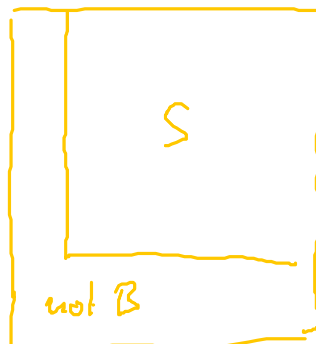
Pseudocode repeat Anweisung

repeat S while B

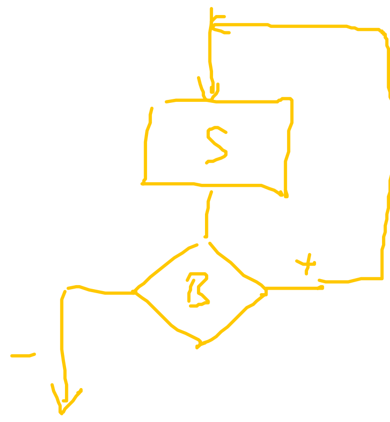
oder äquivalent dazu

repeat S until not B

Struktogramm



Flussdiagramm



Java: do while Schleife

```
do {
    Anweisungen
} while (B);
```

Manchmal möchte man eine Wiederholung nur eine bestimmte Anzahl von Malen durchführen

Bsp n mal

könnte man so realisieren:

```
z := n
while z > 0 do
    Anweisung
    z := z-1;
endwhile
```

dafür for statement

↑ ausführlich in Übung

3.4.4. Zur Mächtigkeit von Kontrollstrukturen

↑ Frage aus theoretischer Informatik und mathematischer Logik

Was kann ich berechnen wenn ich (nur) diese Kontrollstrukturen zur Verfügung habe

Sind solche Prog. Sprachen schwächer als andere?

Antwort: mit den hier behandelten Kontrollstrukturen kann man alles berechnen was überhaupt (im Sinne der Theorie der Berechenbarkeit) berechenbar ist.

Es reichen bereits Verkettung + Wiederholung

if kann dadurch simuliert werden

Andere Prog. Sprachen kennen GOTO Anweisung

if B goto Programmzeile   (frühe Fortran, ..)

aus modernen Prog. Sprachen fast völlig verdrängt (Spaghetti - Code)

allerdings eingeschränkt durchaus sinnvoll

in Java 2 solche Anweisungen

break, continue

Beispiele

```
while (true) {
    ...
    ...
    if (!Fortsetzungsbed.) break;
    ...
    ..
}
```

Sprungen an das Ende der Schleife

```
while (Fortsetzungsbed) {
    ...
    if (!Bedingung) continue;
    weitere Anweisungen
}
```

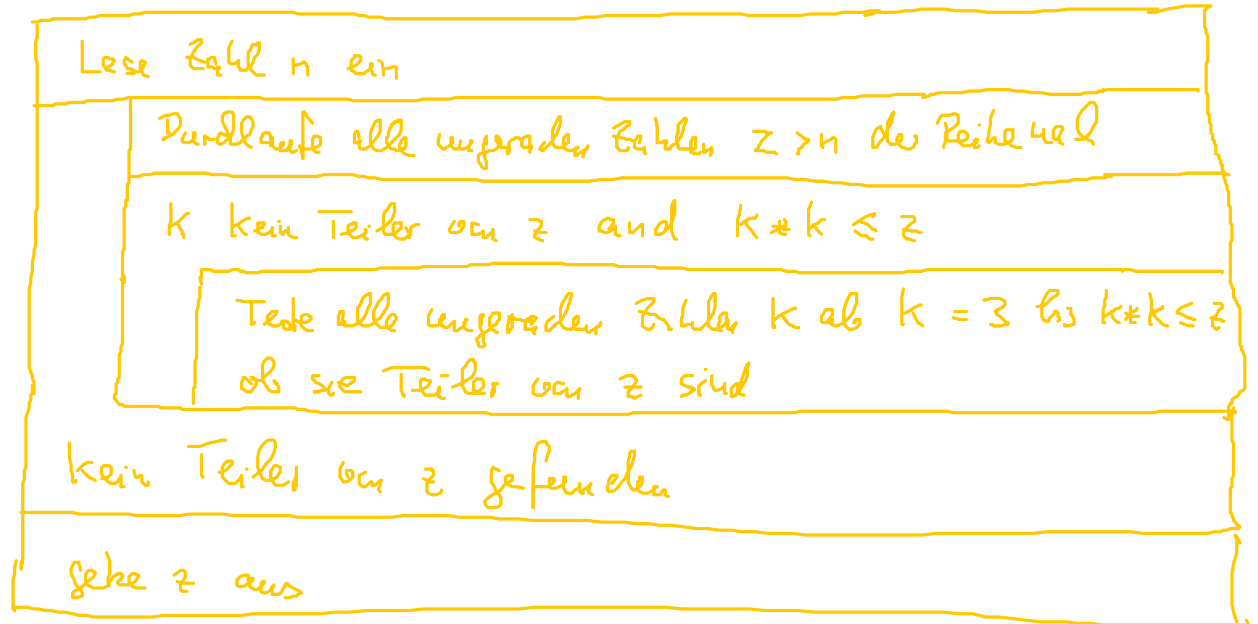
Sprung an Anfang

weitere Verwendung von break in switch statement (Übung)

Übung an Kontrollstrukturen:

Struktogramm des Primzahl-Algorithmus

Grobversion



Feinversion aus Skript vorgeführt