

Einführung in die Informatik I
Funktionale Programmierung

Prof. Bernd Brügge, Ph.D
Dr. Christian Herzog
Technische Universität München

Wintersemester 2000/2001

27. November 2000

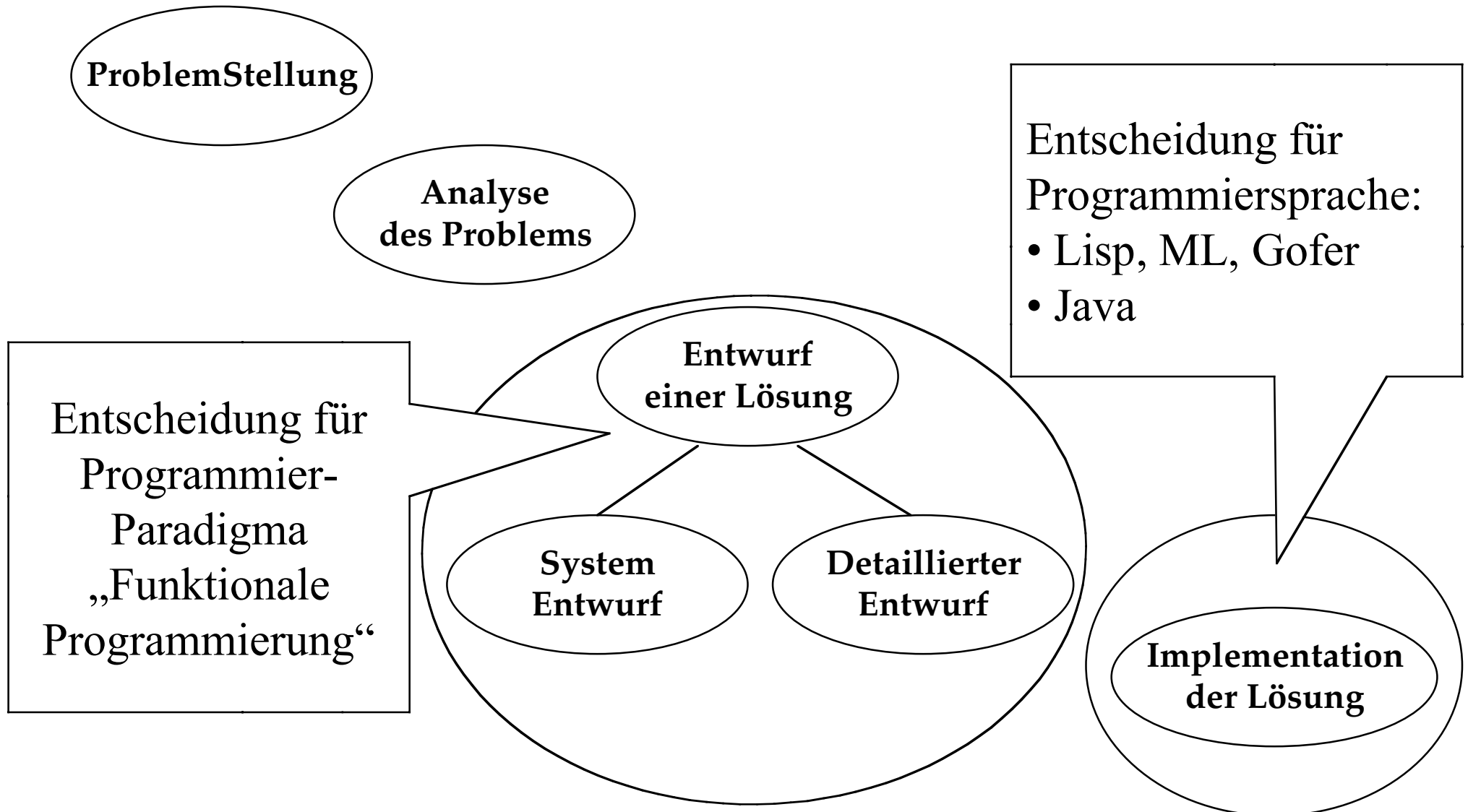
Überblick über den Vorlesungsblock

- ❖ Elementare Sprachkonstrukte
- ❖ Programmieretechniken:
 - Rekursion
 - Einbettung
- ❖ Terminierung und partielle Korrektheit
- ❖ Beweistechnik: Induktion
- ❖ Rekursive Datenstrukturen
- ❖ Semantik funktionaler Programme
 - als Termersetzungssysteme
 - als Funktionen
- ❖ **Wichtigstes Ziel dieses Vorlesungsblockes:**
 - Vertrautheit mit den beiden Techniken Rekursion und Induktion

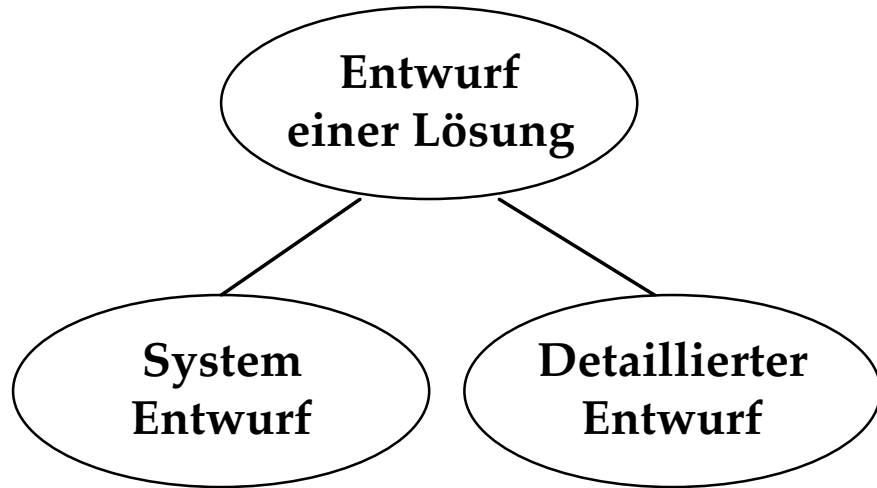
Wo stehen wir?

- ❖ **Oktober/November:** Modellierung, Informatik Systeme, Textersetzungs-system, Markov-Algorithmen, Algebra, Boolesche Algebra, Aussagenlogik, Termersetzungs-systeme
- ❖ **Rest des Semesters:** Programmierparadigmen
 - **jetzt:** Funktionale Programmierung
 - Imperative Programmierung (Dezember)
 - Objekt-Orientierte Programmierung (Januar)
- ❖ **Ergänzungen im Sommersemester:**
 - Ereignis-basierte Programmierung
 - Regel-basierte Programmierung

Aktivitäten bei der Entwicklung eines Informatik-Systems



Entscheidung für Funktionale Programmierung



- ❖ In der Regel wird die Entscheidung nicht für das Gesamtsystem sondern nur für einzelne Komponenten getroffen.
- ❖ Erinnerung: Kategorisierung von Systemen bzw. Komponenten

1. Berechnung von Funktionen

2. Interaktive Systeme

3. Prozessüberwachung

4. Eingebettete Systeme

5. Adaptive Systeme

Offene Systeme eignen sich im Allgemeinen nicht für funktionale Programmierung

Funktionale Programmierung kann verwendet werden, wenn beim System-Entwurf Komponenten entstehen, die Funktionen berechnen.

Definition: Funktionales Programm

Definition: Ein funktionales Programm besteht aus Funktionsvereinbarungen (Funktionsdeklarationen) und einem Ausdruck, der die deklarierten Funktionen aufruft.

❖ **Beispiel** (mit nur einer Funktion) in mathematischer Notation:

– Funktionsvereinbarung:

$$\text{abs}: \mathbb{Z} \rightarrow \mathbb{N}_0$$

$$\text{abs}(x) = \begin{cases} -x, & \text{falls } x < 0 \\ x, & \text{sonst} \end{cases}$$

– Ausdruck:

$$25 + \text{abs}(1000 - 27000)$$

❖ Bei der Ausführung eines funktionalen Programms wird der **Ausdruck ausgewertet** (der Wert des Ausdrucks **berechnet**).

Weitere Beispiele in mathematischer Notation

- ❖ Beispiel mit einer Funktionsvereinbarung:

$$\text{fahrenheitToCelsius: } \mathbb{R} \rightarrow \mathbb{R}$$

$$\text{fahrenheitToCelsius}(f) = 5.0 * (f - 32.0) / 9.0$$

Ausdruck: `fahrenheitToCelsius(68.0)`

- ❖ Beispiel mit zwei Funktionsvereinbarungen:

$$\text{ggT: } \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}, \quad \text{kgV: } \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\text{ggT}(a,b) = \begin{cases} a, & \text{falls } a=b \\ \text{ggT}(a - b, b), & \text{falls } a>b \\ \text{ggT}(a, b - a), & \text{falls } a<b \end{cases}$$

$$\text{kgV}(a,b) = a * b / \text{ggT}(a, b)$$

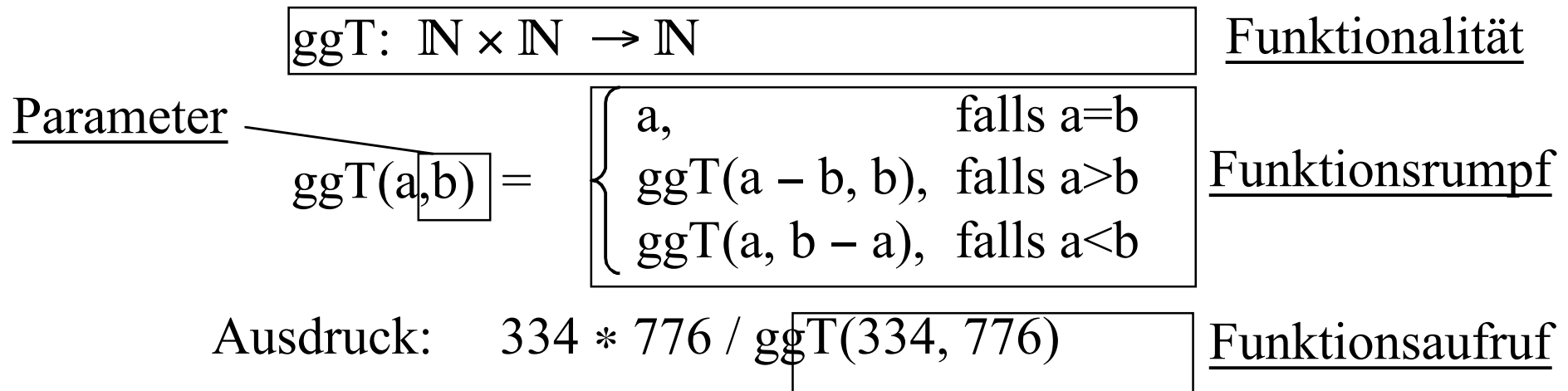
Ausdruck:

$$334 * 776 / \text{ggT}(334, \text{kgV}(334, 776))$$

Sprachkonzepte funktionaler Programme

- ❖ Die wesentlichen Sprachkonzepte am Beispiel:

Funktionsvereinbarung:



- ❖ Der Funktionsrumpf ist selbst wieder ein Ausdruck (in unserem Beispiel ein sog. **bedingter Ausdruck**)
- ❖ Im Funktionsrumpf kommen die Parameter als **freie Identifikatoren** vor.

Die Beispiele in einer funktionalen Programmiersprache (hier Gofer):

❖ abs: $\mathbb{Z} \rightarrow \mathbb{N}_0$

$$\text{abs}(x) = \begin{cases} -x, & \text{falls } x < 0 \\ x, & \text{sonst} \end{cases}$$

25 + abs(1000-27000)

abs :: Int -> Int

abs x | x < 0 = -x
| otherwise = x

25 + abs(1000-27000)

❖ celsius: $\mathbb{R} \rightarrow \mathbb{R}$

$$\text{celsius}(f) = 5.0 * (f - 32.0) / 9.0$$

celsius(68.0)

celsius :: Float -> Float

celsius f = 5.0*(f-32.0)/9.0

celsius 68.0

❖ ggT: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$$\text{ggT}(a,b) = \begin{cases} a, & \text{falls } a=b \\ \text{ggT}(a-b, b), & \text{falls } a>b \\ \text{ggT}(a, b-a), & \text{falls } a<b \end{cases}$$

334 * 776 / ggT(334, 776)

ggT :: Int -> Int -> Int

ggT a b | a==b = a
| a>b = ggT (a-b) b
| a<b = ggT a (b-a)

334 * 776 / ggT 334 776

Die Sprachkonzepte in Gofer:

`ggT :: Int -> Int -> Int`

Funktionalität

`ggT a b`

`a==b = a`

Parameter

`a>b = ggT (a-b) b`

`a<b = ggT a (b-a)`

Funktionsrumpf

`334 * 776 / ggT 334 776`

Funktionsaufruf

Die Funktionale Programmiersprache Gofer

- ❖ Die Schreibweise (Syntax) von Gofer lehnt sich eng an die mathematische Notation an.
- ❖ Gofer enthält nur funktionale Sprachkonzepte. Gofer ist also ausschließlich für das funktionale Programmierparadigma geeignet.
- ❖ Gofer wurde von M.P. Jones entwickelt (1991) und ist frei verfügbar.
- ❖ Zu Gofer verwandte Sprachen sind z.B. ML (Meta Language), SML (standard ML) oder Haskell.
- ❖ Eine andere, weit verbreitete funktionale Sprache ist Lisp.
 - Lisp (List Processor) wurde von John McCarthy 1959 vorgestellt.
 - Grundlegender Datentyp ist die Liste. Auch Programme sind in Listenform und können als Daten aufgefasst werden.

Dieselben Beispiele in der Programmiersprache Java

❖ $\text{abs}: \mathbb{Z} \rightarrow \mathbb{N}_0$

$$\text{abs}(x) = \begin{cases} -x, & \text{falls } x < 0 \\ x, & \text{sonst} \end{cases}$$

$25 + \text{abs}(1000 - 27000)$

```
int abs(int x) {
    return x < 0 ? -x : x;
}

... 25 + abs(1000-27000) ...
```

❖ $\text{celsius}: \mathbb{R} \rightarrow \mathbb{R}$

$$\text{celsius}(f) = 5.0 * (f - 32.0) / 9.0$$

$\text{celsius}(68.0)$

```
double celsius (double f) {
    return 5.0*(f-32.0)/9.0;
}

... celsius(68.0) ...
```

❖ $\text{ggT}: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$$\text{ggT}(a,b) = \begin{cases} a, & \text{falls } a=b \\ \text{ggT}(a-b, b), & \text{falls } a>b \\ \text{ggT}(a, b-a), & \text{falls } a<b \end{cases}$$

$334 * 776 / \text{ggT}(334, 776)$

```
int ggT (int a, int b) {
    return
        a==b ? a
            : a>b ? ggT(a-b, b)
            : ggT(a, b-a);
}

... 334 * 776 / ggT(334,776) ...
```

Die Sprachkonzepte in Java

Funktionalität

Parameter

```
int ggT (int a, int b) {
```

```
    return a==b ? a  
           : a>b ? ggT(a-b, b)  
           : ggT(a, b-a);
```

```
}
```

```
... 334 * 776 / ggT(334, 776) ...
```

Funktionsrumpf

Funktionsaufruf

Zur Programmiersprache Java

- ❖ Java ist keine (reine) funktionale Programmiersprache.
- ❖ Sondern: Java ist eine objekt-orientierte Sprache.
- ❖ Dennoch: das funktionale Programmierparadigma lässt sich auch in Java gut ausdrücken.
- ❖ Die Syntax von Java ist stark an C bzw. C++ angelehnt.
- ❖ Wichtige Unterschiede zu Gofer und der mathematischen Notation:
 - Funktionsrümpfe (in Java spricht man von Operations- oder Methoden-rümpfen) sind in Java grundsätzlich Anweisungen und nicht Ausdrücke.
Für die Programmierung im funktionalen Programmierparadigma verwenden wir in Java `return <Ausdruck>`.
 - Ausdrücke können nicht isoliert (als „Hauptprogramm“) auftreten sondern nur innerhalb von Methodenrümpfen. (Auf den vorigen Folien wurde das durch ... `<Ausdruck>` ... angedeutet.) Das Ausführen eines Programms bedeutet deshalb auch das Ausführen eines vorgegebenen Funktions-aufrufes (meist der Methode `main`).

„Historische“ Notizen zu Java

- ❖ Ursprünglich (ab 1991) wurde Java (unter dem Namen Oak) für interaktives Fernsehen (TV SetTop-Boxen) bei Sun Microsystems entwickelt (P. Naughton, J. Gosling u.a.).
- ❖ Diese Produktlinie konnte sich nicht durchsetzen.
- ❖ Im World Wide Web wurde ein neuer Anwendungsbereich gefunden: 1994 konnte die Gruppe um P. Naughton mit dem WWW-Browser WebRunner (später HotJava) erstmals kleine Java-Programme (Applets) aus dem WWW laden und ausführen.
- ❖ Der Durchbruch gelang, als Netscape die Java-Technologie übernahm (1995).
- ❖ 1996: JDK 1.0, erste Version des Java Development Kit
- ❖ 1997: JDK 1.1 (wesentlich verbessert, in einigen Teilen nicht mehr kompatibel mit JDK 1.0)
- ❖ zurzeit: JDK 1.2

Definition: Ausdruck

❖ Vorbemerkungen:

- Wie am Beispiel arithmetischer oder Boolescher Ausdrücke bereits gesehen, wollen wir auch die Ausdrücke, die als Funktionsrümpfe zugelassen sind, rekursiv über ihre Grundelemente definieren.
- Wir verwenden dabei die Syntax von Java.
- Uns ist dabei aber mehr am prinzipiellen Aufbau als an einer vollständigen Definition gelegen. Deshalb vereinfachen wir.

❖ Der Typ eines Ausdruckes:

- Jeder Ausdruck hat einen Typ, z.B. int, double, boolean oder char, der dem Typ des Wertes entspricht, der aus dem Ausdruck berechnet wird.
- In den Beispielen:
 - $5.0 * (f - 32.0) / 9.0$ ist vom Typ double
 - $334 * 776 / \text{ggT}(334, 776)$ ist vom Typ int

Definition Ausdruck (Fortsetzung)

❖ **Grundelemente:**

- Jede Konstante eines Typs in ihrer Standardbezeichnung ist ein Ausdruck des entsprechenden Typs:
 - 1 2 -298 sind drei Ausdrücke vom Typ int;
 - true und false sind zwei Ausdrücke vom Typ boolean;
 - 0.5 3.14 1.0 sind drei Ausdrücke vom Typ double;
 - 'a' 'A' '1' '@' ';' sind fünf Ausdrücke vom Typ char.
- Jeder Parameter, der als freier Identifikator im Funktions-rumpf auftritt, ist Ausdruck seines im Funktionskopf definierten Typs.

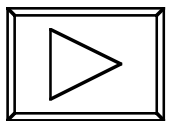
```
– In double celsius (double f) {  
    return 5.0*(f-32.0)/9.0;  
}
```

ist das markierte `f` ein Ausdruck vom Typ double.

Definition Ausdruck (Fortsetzung)

❖ **Arithmetische Operatoren:**

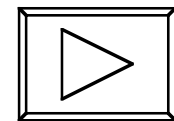
- Einstellige arithmetische Operatoren sind $+$ $-$
- Zweistellige arithmetische Operatoren sind $+$ $-$ $*$ $/$ $\%$
- Sind A und B zwei Ausdrücke eines Typs, auf dem die entsprechende Operation definiert ist, so sind
 $+A$, $-A$, $(A+B)$, $(A-B)$, $(A*B)$, (A/B) , $(A\%B)$
jeweils Ausdrücke desselben Typs.
- Beispiele:
 - $(7.5 / (9.3 - 9.1))$ ist vom Typ double
 - $(7 / (9 - 8))$ ist vom Typ int
- zur näheren Erklärung vgl. etwa
<http://medoc.informatik.tu-muenchen.de/Java/krueger/>



Definition Ausdruck (Fortsetzung)

❖ **Vergleichsoperatoren:**

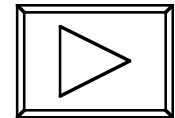
- Vergleichsoperatoren sind
== (gleich) != (ungleich) < <= >= >
- Sind A und B zwei Ausdrücke eines Typs, auf dem die entsprechende Vergleichsoperation definiert ist, so sind
(A==B), (A!=B), (A<B), (A<=B), (A>=B), (A>B)
jeweils Ausdrücke vom Typ boolean.



Definition Ausdruck (Fortsetzung)

❖ **Boolesche Operatoren:**

- ! (nicht) ist ein einstelliger boolescher Operator.
- & (und), | (oder) und ^ (exklusives oder) sind zweistellige boolesche Operatoren.
- Sind A und B zwei Ausdrücke vom Typ boolean, so sind
 ! A , ($A \& B$), ($A | B$), ($A \wedge B$)
jeweils Ausdrücke vom Typ boolean.
- Beispiele:
 - false | !false
 - ((1 <= x) & (x <= n))



Definition Ausdruck (Fortsetzung)

❖ **Funktionsaufruf:**

– Ist f die folgendermaßen vereinbarte Funktion:

$T \ f(T_1 \ x_1, T_2 \ x_2, \dots, T_n \ x_n) \ \{ \text{return } A; \}$

– mit dem Ergebnistyp T

– mit Parametern x_1 vom Typ T_1 , x_2 vom Typ T_2 , ...
und x_n vom Typ T_n ,

– und einem Ausdruck A vom Typ T ,

– und sind A_1, A_2, \dots, A_n Ausdrücke von den Typen T_1, T_2, \dots, T_n ,

– dann ist der Funktionsaufruf

$f(A_1, A_2, \dots, A_n)$

ein Ausdruck vom Typ T .

– Beispiel: $ggT(334+9, 667-5)$ ist ein Ausdruck vom
Typ int .

Definition Ausdruck (Fortsetzung)

❖ **Bedingter Ausdruck:**

- Sind A_1 und A_2 zwei Ausdrücke vom selben Typ T , und ist B ein Ausdruck vom Typ boolean (eine Bedingung), so ist

$(B ? A_1 : A_2)$ -- sprich: falls B dann A_1 sonst A_2

ebenfalls ein Ausdruck vom Typ T .

- Beispiele:

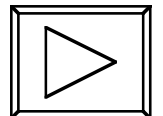
- $(7 > 9 ? 10 : true)$ ist kein Ausdruck, da 10 und true nicht vom selben Typ sind;

- $(A == B ? true : false)$ ist ein Ausdruck vom Typ boolean, falls A und B Ausdrücke gleichen Typs sind. Er ist übrigens semantisch äquivalent zum Ausdruck $(A == B)$.

Definition Ausdruck (Fortsetzung)

❖ **Weglassen von Klammern:**

- Klammern machen die Zuordnung von Operanden zu Operatoren eindeutig.
- „Lange“ Ausdrücke sind dann aber oft mühsam zu lesen und umständlich zu schreiben.
- Deshalb gibt es in Java Regeln, die es erlauben, Klammern in Ausdrücken wegzulassen und die Zuordnung dennoch eindeutig zu belassen.
 - z.B. dürfen die äußersten Klammern weggelassen werden
- Aus der Schule ist die *Punkt-vor-Strich-Regel* bekannt, die besagt, dass im Ausdruck $7+(8*4)$ die Klammern weggelassen werden können, da $*$ stärker bindet als $+$ (oder $*$ Vorrang hat vor $+$).
- Auch für die Java-Operatoren gibt es zahlreiche Vorrangregeln:
 - z.B. ist $7+4*5 < 10 \ \& \ 20 > 5 \ | \ 100-7==93$
dasselbe wie $((7+(4*5)) < 10) \ \& \ (20 > 5) \ | \ ((100-7)==93)$
- bei gleichen Operatoren oder Operatoren gleichen Vorrangs (wie $+$ und $-$) wird assoziativ verknüpft (meist von links):
 - $7+4-3+9$ ist dasselbe wie $((7+4) -3) +9$



Auswerten von Ausdrücken

- ❖ **Erinnerung** (Folie 6): Bei der Ausführung eines funktionalen Programms wird der **Ausdruck ausgewertet**.
- ❖ **Auswerten der arithmetischen, booleschen und Vergleichs-Operationen:**
 - Zuerst werden die Operanden ausgewertet.
 - (Die Operanden sind selbst i.A. wieder Ausdrücke. Die Auswertung von Ausdrücken ist also ein rekursives Verfahren.)
 - Danach wird die Operation auf die ermittelten Werte angewendet.
 - Ist einer der ermittelten Werte der Operanden undefiniert (in Zeichen \perp), z.B. weil eine Division durch 0 auftritt, so ist auch das Ergebnis der Operation undefiniert.
 - Diese Art der Auswertung (Ergebnis \perp , falls einer der Operanden = \perp) nennt man **strikte Auswertung**.

Nicht strikte Operatoren

- ❖ Beispiel: Auswertung von $(7+4*5 < 10) \ \& \ (20 > 5)$
 - Der linke Operand liefert false, der rechte true, insgesamt liefert der Ausdruck also false.
 - Nach Auswertung des linken Operanden steht das Ergebnis bereits fest.
- ❖ Idee: **verkürzte Auswertung** (short circuit evaluation):
 - Auswertung beenden, falls durch einen Operanden der Wert bereits feststeht.
 - In Java: zusätzliche Operatoren $\&\&$ (und) und $\|\|$ (oder)
 - Beispiele:
 - $(7+4*5 < 10) \ \&\& \ (20 > 5)$
 - $(20 > 5) \ \|\| \ (7+4*5 < 10)$
 - der rechte Operand wird jeweils nicht mehr ausgewertet
- ❖ verkürzte Auswertung ist **nicht strikt**:
 - $(20 > 5) \ \|\| \ (10/0 == 1)$ liefert true und nicht \perp

Auswertung bedingter Ausdrücke

- ❖ Auswertung von $(B ? A_1 : A_2)$
 - Zuerst wird die Bedingung, also der Ausdruck B ausgewertet.
 - Liefert die Auswertung von B $true$, dann wird A_1 ausgewertet und der Wert des bedingten Ausdrucks ist der Wert von A_1 .
 - Liefert die Auswertung von B $false$, dann wird A_2 ausgewertet und der Wert des bedingten Ausdrucks ist der Wert von A_2 .
 - Liefert die Auswertung von B \perp , dann ist der Wert des bedingten Ausdrucks ebenfalls \perp .
- ❖ Fasst man den bedingten Ausdruck als dreistelligen Operator $(. ? . : .)$ auf, dann ist die Auswertung dieses Operators nicht strikt:
 - $(true ? 9999 : \langle \text{Ausdruck} \rangle)$ liefert 9999 , auch wenn $\langle \text{Ausdruck} \rangle$ den Wert \perp hat.

Auswertung von Funktionsaufrufen

- ❖ Ist f wieder die folgendermaßen deklarierte Funktion:

$T \ f(T_1 \ x_1, T_2 \ x_2, \dots, T_n \ x_n) \{ \text{return } A; \}$

- mit dem Ergebnistyp T

- mit Parametern x_1 vom Typ T_1 , x_2 vom Typ T_2 , ...
und x_n vom Typ T_n ,

- und einem Ausdruck A vom Typ T ,

- ❖ Dann wird der Aufruf $f(A_1, A_2, \dots, A_n)$ folgendermaßen ausgewertet:

- zuerst werden **alle** Ausdrücke A_i ausgewertet;

- liefert ein A_i den Wert \perp , dann liefert der Funktionsaufruf \perp ;

- ansonsten ist der Wert des Funktionsaufrufes der Wert des Ausdruckes A , wenn dort jedes Auftreten eines Parameters x_i durch den Wert von A_i ersetzt (substituiert) wird.

- ❖ Die Auswertung eines Funktionsaufrufes ist wieder **strikt**.

Beispiel für die Auswertung eines Funktionsaufrufes

```
double celsius (double f) {  
    return 5.0*(f-32.0)/9.0;  
}  
boolean istKalt (double f) {  
    return celsius(f) <= 10.0 ;  
}
```

- ❖ Aufruf: `istKalt(10.0 + 49.0)`
- ❖ Auswertung des Ausdruckes auf Parameterposition liefert `59.0`
- ❖ Substitution von `f` durch `59.0` im Rumpf von `istKalt` liefert:
`celsius(59.0) <= 10.0 (*)`
- ❖ Zunächst nun Auswertung von `celsius(59.0)`
 - Substitution von `f` durch `59.0` im Rumpf von `celsius` liefert:
`5.0*(59.0-32.0)/9.0`
 - Dies liefert nacheinander `5.0*27.0/9.0`, `5.0*3.0`, `15.0`
- ❖ Eingesetzt in (*) ergibt sich also: `15.0 <= 10.0`
- ❖ Und dies liefert das Endergebnis: `false`

Funktionsaufruf: Call-by-Value vs. Call-by-Name

- ❖ In Java werden Funktionsaufrufe ausgewertet, indem die Parameter (auch *formale Parameter* genannt) im Funktionsrumpf durch die **Werte** der auf Parameterposition stehenden Ausdrücke (*aktuelle Parameter*) substituiert werden.
- ❖ Diese Art der Auswertung wird Wertaufruf (Call-by-Value) genannt.
- ❖ Eine andere Art der Auswertung ist Call-by-Name:
 - Die formalen Parameter im Funktionsrumpf werden durch die nicht ausgewerteten aktuellen Parameter substituiert.
 - Vorteil: Parameter, deren Wert nicht benötigt wird (weil sie z.B. in einem bedingten Ausdruck nur im nicht ausgewerteten Ausdruck auftreten), werden auch nicht ausgewertet; sog. faule Auswertung (lazy evaluation).
 - Nachteil: Parameter, die öfter im Rumpf auftreten, werden auch öfter ausgewertet.
- ❖ Die Auswertung nach Call-by-Name ist nicht strikt.

Beispiel für Auswertung Call-by-Value

```
int auswahl (int t, int x, int y, int z){  
    return t==0 ? x*x*x : t==1 ? x*x*y : x*y*z ;  
}
```

Aufruf: ... `auswahl(1*1, 12-8+100, 12+8-100, 12+8+100)` ...

- ❖ Auswerten der 4 aktuellen Parameter liefert: 1, 104, -80, 120
- ❖ Substitution der Werte der aktuellen Parameter liefert:
`1==0 ? 104*104*104 : 1==1 ? 104*104*(-80) : 104*(-80)*120`
- ❖ Auswerten der äußeren Bedingung `1==0` liefert:
`false ? 104*104*104 : 1==1 ? 104*104*(-80) : 104*(-80)*120`
- ❖ Auswertungsregel für bedingte Ausdrücke liefert:
`1==1 ? 104*104*(-80) : 104*(-80)*120`
- ❖ Auswerten der (ehemals inneren) Bedingung `1==1` liefert:
`true ? 104*104*(-80) : 104*(-80)*120`
- ❖ Auswertungsregel für bedingte Ausdrücke liefert: `104*104*(-80)`
- ❖ Ergebnis: -865280

- ❖ Anzahl der Auswertungen für `t`, `x`, `y` und `z`: je ein Mal

Beispiel für Auswertung Call-by-Name

```
int auswahl (int t, int x, int y, int z){  
    return t==0 ? x*x*x: t==1 ? x*x*y : x*y*z ;  
}
```

Aufruf: ... `auswahl(1*1, 12-8+100, 12+8-100, 12+8+100)` ...

❖ Substitution der aktuellen Parameter (als Ausdrücke) liefert:

$$\begin{aligned} & 1*1==0 ? (12-8+100)*(12-8+100)*(12-8+1000) \\ & \quad : 1*1==1 ? (12-8+100)*(12-8+100)*(12+8-100) \\ & \quad \quad : (12-8+100)*(12+8-100)*(12+8+100) \end{aligned}$$

❖ Auswerten der äußeren Bedingung `1*1==0` liefert:

$$\begin{aligned} & \text{false} ? (12-8+100)*(12-8+100)*(12-8+1000) \\ & \quad : 1*1==1 ? (12-8+100)*(12-8+100)*(12+8-100) \\ & \quad \quad : (12-8+100)*(12+8-100)*(12+8+100) \end{aligned}$$

❖ Auswertungsregel für bedingte Ausdrücke liefert:

$$\begin{aligned} & 1*1==1 ? (12-8+100)*(12-8+100)*(12+8-100) \\ & \quad : (12-8+100)*(12+8-100)*(12+8+100) \end{aligned}$$

Beispiel für Auswertung Call-by-Name (Fortsetzung)

❖ Substitution der aktuellen Parameter (als Ausdrücke) liefert:

$$\begin{aligned} 1*1==0 & ? (12-8+100) * (12-8+100) * (12-8+1000) \\ & : 1*1==1 ? (12-8+100) * (12-8+100) * (12+8-100) \\ & : (12-8+100) * (12+8-100) * (12+8+100) \end{aligned}$$

❖ Auswerten der äußeren Bedingung $1*1==0$ liefert:

$$\begin{aligned} \text{false} & ? (12-8+100) * (12-8+100) * (12-8+1000) \\ & : 1*1==1 ? (12-8+100) * (12-8+100) * (12+8-100) \\ & : (12-8+100) * (12+8-100) * (12+8+100) \end{aligned}$$

❖ Auswertungsregel für bedingte Ausdrücke liefert:

$$\begin{aligned} 1*1==1 & ? (12-8+100) * (12-8+100) * (12+8-100) \\ & : (12-8+100) * (12+8-100) * (12+8+100) \end{aligned}$$

❖ Auswerten der (ehemals inneren) Bedingung $1*1==1$ liefert:

$$\begin{aligned} \text{true} & ? (12-8+100) * (12-8+100) * (12+8-100) \\ & : (12-8+100) * (12+8-100) * (12+8+100) \end{aligned}$$

❖ Auswertungsregel für bedingte Ausdrücke liefert:

$$(12-8+100) * (12-8+100) * (12+8-100)$$

❖ Ergebnis: -865280

Beispiel für Auswertung Call-by-Name (Fortsetzung)

```
int auswahl (int t, int x, int y, int z){  
    return t==0 ? x*x*x: t==1 ? x*x*y : x*y*z ;  
}
```

Aufruf: ... `auswahl(1*1, 12-8+100, 12+8-100, 12+8+100)` ...

❖ Substitution der aktuellen Parameter (als Ausdrücke) liefert:

```
1*1==0 ? (12-8+100)*(12-8+100)*(12-8+1000)  
        : 1*1==1 ? (12-8+100)*(12-8+100)*(12+8-100)  
                : (12-8+100)*(12+8-100)*(12+8+100)
```

❖ Auswerten der äußeren Bedingung `1*1==0` liefert:

```
false ? (12-8+100)*(12-8+100)*(12-8+1000)  
       : 1*1==1 ? (12-8+100)*(12-8+100)*(12+8-100)  
               : (12-8+100)*(12+8-100)*(12+8+100)
```

❖ Auswertungsregel für bedingte Ausdrücke liefert:

```
1*1==1 ? (12-8+100)*(12-8+100)*(12+8-100)  
        : (12-8+100)*(12+8-100)*(12+8+100)
```

❖ Auswerten der (ehemals inneren) Bedingung `1*1==1` liefert:

```
true ? (12-8+100)*(12-8+100)*(12+8-100)  
      : (12-8+100)*(12+8-100)*(12+8+100)
```

❖ Auswertungsregel für bedingte Ausdrücke liefert:

```
(12-8+100)*(12-8+100)*(12+8-100)
```

❖ Ergebnis: -865280

Anzahl der
Auswertungen

für t: 2

für x: 2

für y: 1

für z: 0

Bemerkungen zum Funktionsaufruf

- ❖ Klarstellung zur Art der Auswertung in Java:
 - In Java erfolgt die Auswertung von Funktionsaufrufen durch Call-by-Value.
 - Die formalen Parameter im Funktionsrumpf werden ersetzt durch die **Werte** der aktuellen Parameter.
 - Diese Auswertung ist strikt.
- ❖ Begriffliches:
 - Der Aufruf einer Funktion wird oft auch als Funktions-*anwendung* bzw. Funktions*applikation* bezeichnet.
 - Funktionales Programmieren wird oft auch als *Applikatives Programmieren* (z.B. bei Broy) bezeichnet, weil die Funktionsanwendung wichtiges Sprachkonzept ist.

Rekursive Funktionen bzw. Funktionsdeklarationen

❖ Beispiel: ggT

```
int ggT (int a, int b) {  
    return a==b ? a : a>b ? ggT(a-b, b)  
                        : ggT(a, b-a);  
}
```

❖ Im Rumpf der Funktion ggT treten Aufrufe von ggT auf (Selbstaufufe).

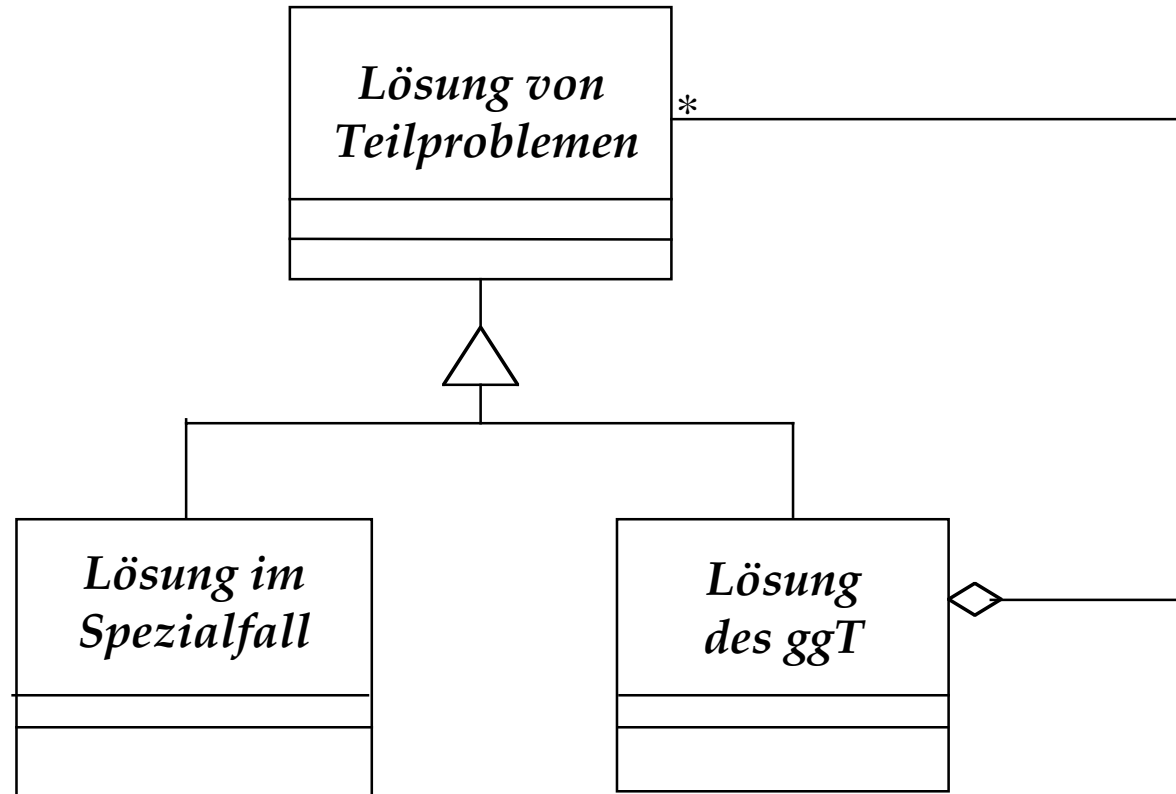
❖ Definition: Eine Funktion bzw. eine Funktionsdeklaration heißt **rekursiv**, falls der Ausdruck im Rumpf der Funktion einen Aufruf derselben Funktion enthält.

❖ Rekursive Funktionen entsprechen rekursiven Problemlösetechniken:

- die Lösung des ggT zerfällt in die Behandlung eines Spezialfalls und die Lösung von ähnlichen, aber kleineren Problemen.

Rekursive Problemlösung und Kompositionsmuster

❖ Beispiel: ggT



Funktionsauswertung bei rekursiven Funktionen

❖ Beispiel: ggT

```
int ggT (int a, int b) {  
    return a==b ? a : a>b ? ggT(a-b, b) : ggT(a, b-a);  
}
```

❖ Aufruf: ggT (9, 12)

❖ Parameter im Rumpf substituieren:

$9==12 ? 9 : 9>12 ? \text{ggT}(9-12, 12) : \text{ggT}(9, 12-9)$

❖ Auswertung des bedingten Ausdruckes:

$9>12 ? \text{ggT}(9-12, 12) : \text{ggT}(9, 12-9)$

❖ Auswertung des bedingten Ausdruckes:

$\text{ggT}(9, 12-9)$

❖ Auswertung der aktuellen Parameter:

$\text{ggT}(9, 3)$

❖ Parameter im Rumpf substituieren:

$9==3 ? 9 : 9>3 ? \text{ggT}(9-3, 3) : \text{ggT}(9, 3-9)$

Funktionsauswertung bei rek. Funktionen (Forts.)

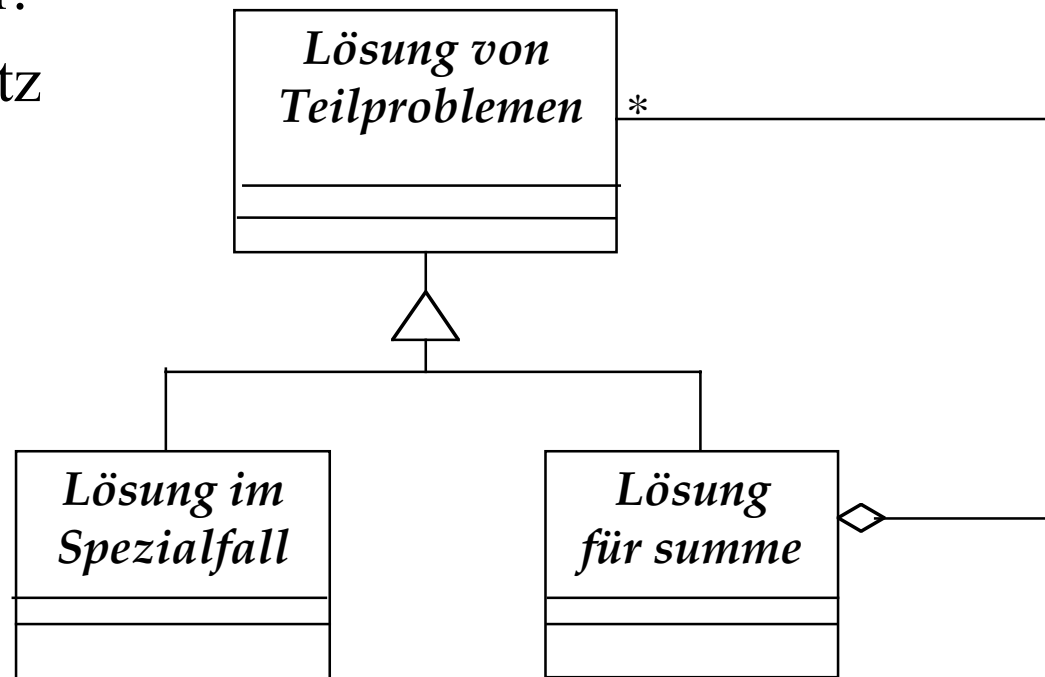
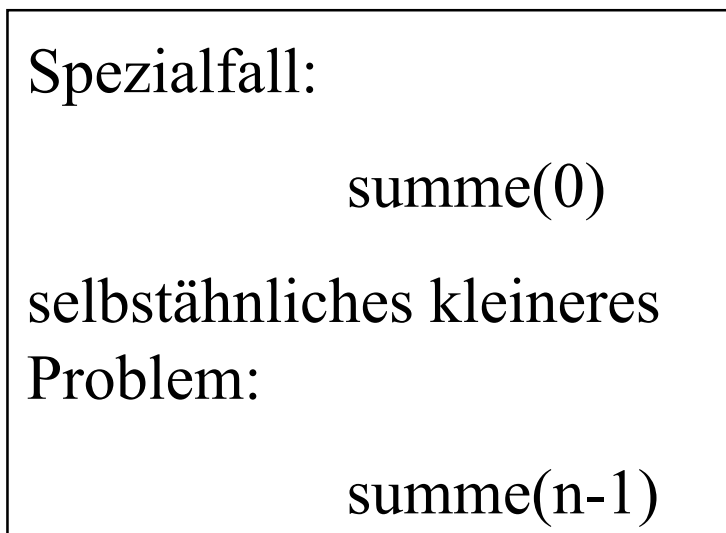
❖ Beispiel: ggT

```
int ggT (int a, int b) {  
    return a==b ? a : a>b ? ggT(a-b, b) : ggT(a, b-a);  
}
```

- ❖ ggT (9, 12)
- ❖ 9==12 ? 9 : 9>12 ? ggT(9-12, 12) : ggT(9, 12-9) -- ParSub
- ❖ 9>12 ? ggT(9-12, 12) : ggT(9, 12-9) -- AnwBed
- ❖ ggT(9, 12-9) -- AnwBed
- ❖ ggT(9, 3) -- aktPar
- ❖ 9==3 ? 9 : 9>3 ? ggT(9-3, 3) : ggT(9, 3-9) -- ParSub
- ❖ 9>3 ? ggT(9-3, 3) : ggT(9, 3-9) -- Bed
- ❖ ggT(9-3, 3) -- Bed
- ❖ ggT(6, 3) -- aktPar
- ❖ 6==3 ? 6 : 6>3 ? ggT(6-3, 3) : ggT(6, 3-6) -- ParSub
- ❖ 6>3 ? ggT(6-3, 3) : ggT(6, 3-6) -- AnwBed
- ❖ ggT(6-3, 3) -- AnwBed
- ❖ ggT(3, 3) -- aktPar
- ❖ 3==3 ? 3 : 3>3 ? ggT(3-3, 3) : ggT(3, 3-3) -- ParSub
- ❖ 3 -- AnwBed

Berechnung der Summe der ersten n Zahlen

- ❖ Gesucht: eine Funktion `summe`, die die Summe der ersten n natürlichen Zahlen berechnet:
 - `summe: $\mathbb{N} \rightarrow \mathbb{N}$`
 - `summe(n) = 1 + ... + n`
- ❖ Problem: Wie setze ich die Pünktchen „...“ in eine präzise Funktionsdeklaration um?
- ❖ Lösung: rekursiver Ansatz



Umsetzung in die Sprache Java

- ❖ Funktionalität: $\mathbb{N} \rightarrow \mathbb{N}$
- ❖ Parameter: n
- ❖ Funktionsrumpf
- ❖ Spezialfall: $\text{summe}(0) = 0$
- ❖ Rekursiver Fall: $\text{summe}(n) = \text{summe}(n-1) + n$

```
int summe (int n) {  
    return n==0 ? 0 : summe (n-1) + n ;  
}
```

- ❖ Bemerkung:
 - Die Schnittstelle der Java-Funktion entspricht nicht der Funktionalität $\mathbb{N} \rightarrow \mathbb{N}$, da in Java nur `int` zur Verfügung steht.
 - Es muss bei der Anwendung der Funktion sicher gestellt werden, dass der Wert des aktuellen Parameters nicht negativ ist!

Auswerten der Funktion Summe

```
int summe (int n) {  
    return n==0 ? 0 : summe (n-1) + n ;  
}
```

- ❖ `summe (3)`
- ❖ `3==0 ? 0 : summe (3-1) + 3` -- ParSub
- ❖ `summe (3-1) + 3` -- AnwBed
- ❖ `summe (2) + 3` -- AktPar
- ❖ `(2==0 ? 0 : summe (2-1) + 2) + 3` -- ParSub
- ❖ `(summe (2-1) + 2) + 3` -- AnwBed
- ❖ `(summe (1) + 2) + 3` -- AktPar
- ❖ `((1==0 ? 0 : summe (1-1) + 1) + 2) + 3` -- ParSub
- ❖ `((summe (1-1) + 1) + 2) + 3` -- AnwBed
- ❖ `((summe (0) + 1) + 2) + 3` -- AktPar
- ❖ `((0==0 ? 0 : summe (0-1) + 0) + 1) + 2) + 3` -- ParSub
- ❖ `((0) + 1) + 2) + 3` -- AnwBed
- ❖ `6` -- Arithm

Berechnung der Fakultätsfunktion

❖ Gesucht: eine Funktion `fakultät`, die das **Produkt** der ersten n natürlichen Zahlen berechnet:

– `fakultät`: $\mathbb{N} \rightarrow \mathbb{N}$

– `fakultät`(n) = $1 * \dots * n$

– `fakultät`(0) = 1 -- Erweiterung auf 0

❖ Mathematische Notation für `fakultät`(n): $n!$

❖ Lösung völlig analog zur Summe:

```
int summe (int n) {  
    return n == 0 ? 0 : summe(n-1) + n ;  
}
```

❖ Dasselbe Rekursionsschema benutzen wir nun für die Fakultät:

```
int fakultaet (int n) {  
    return n == 0 ? 1 : fakultaet(n-1) * n ;  
}
```

Bemerkungen zur Fakultätsfunktion

- ❖ Sie ist **das** Standardbeispiel für rekursive Funktionen.
- ❖ Einige Werte:
 - $0! = 1! = 1$
 - $2! = 2$
 - $3! = 6$
 - $4! = 24$
 - $5! = 120$
 - $10! = 3\,628\,800$
 - $12! = 479\,001\,600$
- ❖ Die Fakultätsfunktion ist also eine sehr rasch anwachsende Funktion.
- ❖ Achtung: In Java lassen sich mit `int` nur die Zahlen zwischen -2^{31} und $2^{31}-1$ darstellen: $2^{31}-1 = 2\,147\,483\,647$
- ❖ $13!$ lässt sich mit `int` also nicht mehr darstellen.
- ❖ In Java gibt es einen weiteren Typ `long` für Zahlen zwischen -2^{63} und $2^{63}-1$

Multiplikation durch Addition und Subtraktion

❖ Gesucht: eine Funktion `mult`, die die Multiplikation zweier ganzer Zahlen auf Addition und Subtraktion zurückführt:

– `mult: $\mathbb{Z} \rightarrow \mathbb{Z}$`

– `mult(x, y) = x * y` -- allerdings ohne Verwendung von `*`

❖ **Idee** (ähnlicher Algorithmus im Kapitel Termersetzung):

– `mult(x, 0) = 0`

– `mult(x, y) = mult(x, y-1) + x`

❖ Umsetzung nach Java:

```
Int mult (int x, int y) {  
    return y < 0 ? -mult(x, -y)  
           : y == 0 ? 0 : mult(x, y-1) + x;  
}
```

❖ Problem: `y` kann negativ sein!

❖ Lösung: zusätzliche Bedingung

Auswerten der Funktion mult

```
int mult (int x, int y) {  
    return y<0 ? -mult(x,-y) : y==0 ? 0 : mult(x,y-1) + x; }
```

```
❖ mult(5,-2)  
❖ -2<0 ? -mult(5,-2) : -2==0 ? 0 : mult(5,-2-1)+5 -- ParSub  
❖ -mult(5,-2) -- AnwBed  
❖ -mult(5,2) -- AktPar  
❖ -(2<0 ? -mult(5,-2) : 2==0 ? 0 : mult(5,2-1)+5) -- ParSub  
❖ -(2==0 ? 0 : mult(5,2-1) + 5) -- AnwBed  
❖ -(mult(5,2-1) + 5) -- AnwBed  
❖ -(mult(5,1) + 5) -- AktPar  
❖ -((1<0 ? -mult(5,-1) : 1==0 ? 0 : mult(5,1-1) + 5) + 5) -- ParSub  
❖ -((1==0 ? 0 : mult(5,1-1) + 5) + 5) -- AnwBed  
❖ -(mult(5,1-1) + 5) + 5) -- AnwBed  
❖ -(mult(5,0) + 5) + 5) -- AktPar  
❖ -((0<0 ? -mult(5,-0) : 0==0 ? 0 : mult(5,0-1) + 5)+5)+5) -- ParSub  
❖ -(((0==0 ? 0 : mult(5,0-1) + 5) + 5) + 5) -- AnwBed  
❖ -(((0) + 5) + 5) -- AnwBed  
❖ -10 -- Arithm
```