Grundlagen der Programmierung

Dr. Christian Herzog
Technische Universität München

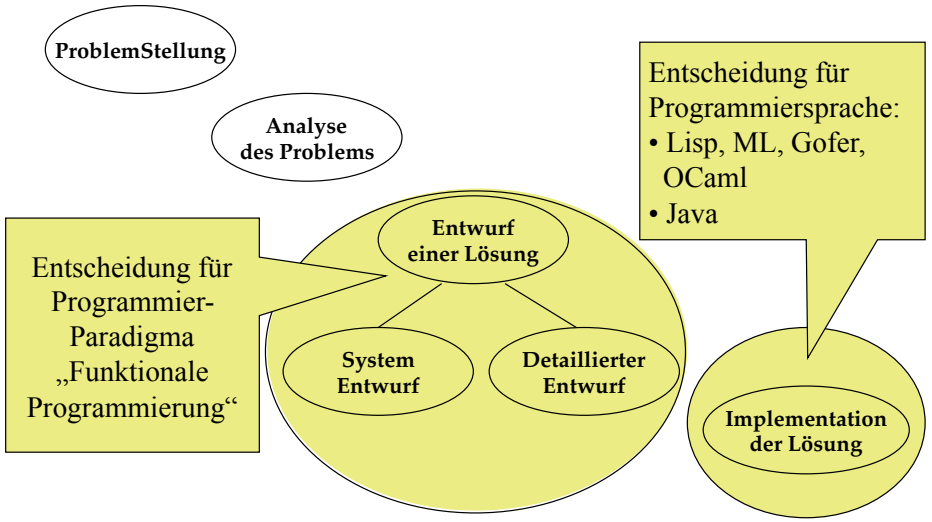
Wintersemester 2018/2019

Kapitel 5: Funktionaler Programmierstil und Rekursion

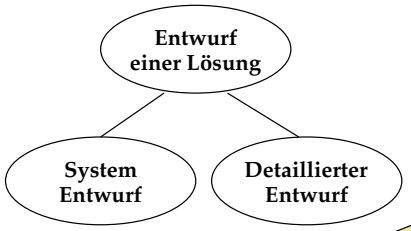
Überblick über dieses Kapitel

- Sprachkonstrukte für den funktionalen Programmierstil in Java
  - Ausdrücke
  - Methodenaufrufe (hier aufgefasst als Funktionsaufrufe)
- Programmiertechniken:
  - Rekursion
  - Einbettung
- Terminierung und partielle Korrektheit
- Beweistechnik: Induktion
- Rekursive Datenstrukturen
- **Wichtigstes Ziel dieses Vorlesungsblockes**:
  - Vertrautheit mit den beiden Techniken ***Rekursion*** und ***Induktion***

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. Prozessüberwachung
  3. Eingebettete Systeme
  4. Adaptive Systeme

Funktionaler Programmierstil kann verwendet werden, wenn beim System-Entwurf Komponenten entstehen, die Funktionen berechnen.

Offene Systeme eignen sich im Allgemeinen nicht für funktionale Programmierung

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:

    abs: $Z \rightarrow N_0$

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

  - Ausdruck:

    25 + 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:

  fahrenheitToCelsius: $R \rightarrow R$

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

  Ausdruck: fahrenheitToCelsius(68.0)

- Beispiel mit zwei Funktionsvereinbarungen:

  ggT: $N \times N \rightarrow N$, kgV: $N \times N \rightarrow N$

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

  kgV(a,b) = a * b / ggT(a, b)

  Ausdruck:

  334 * 776 / ggT(334, kgV(334, 776))

Sprachkonzepte funktionaler Programme

- Die wesentlichen Sprachkonzepte am Beispiel:

  Funktionsvereinbarung:

  ggT: $N \times N \rightarrow N$ — Funktionalität

  Parameter: a,b

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

  Funktionsrumpf

  Ausdruck: 334 * 776 / ggT(334, 776) — Funktionsaufruf

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

Die Beispiele in der funktionalen Programmiersprache Gofer:

- abs: $Z \rightarrow N_0$

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

  25 + abs(1000-27000)

```
abs :: Int -> Int
abs x | x < 0     = -x
      | otherwise = x

25 + abs(1000-27000)
```

Funktionalität; Parameter; Funktionsrumpf; Funktionsaufruf

- celsius: $R \rightarrow R$

  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: $N \times N \rightarrow N$

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

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

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

Funktionalität

Parameter

Funktionsrumpf

Funktionsaufruf

Currying

```
ggT :: (Int, Int)-> Int
ggT (a,b) | a==b = a
          | a>b  = ggT (a-b,b)
          | a<b  = ggT (a,b-a)
... ggt (334,776) ...
```

Parameter ist ein Paar von Zahlen

- Statt der Funktionalität `ggt::(Int,Int)->Int` wird in funktionalen Sprachen für Funktionen mit 2 Parametern oft lieber die Funktionalität `ggt::Int->Int->Int` gewählt. Dieses Vorgehen wird nach dem Logiker Haskell Curry auch ***currying*** genannt.
- In der ersten Variante hat `ggt` ein Paar von Zahlen als (einzigen) Parameter.
- In der zweiten Variante (currying) hat `ggt` eine Zahl als Parameter und liefert eine Funktion als Ergebnis. `ggt 334` ist z.B. die Funktion, die berechnet, welchen größten gemeinsamen Teiler eine Zahl mit der Zahl 334 hat. `ggt 334 776` berechnet dann wie oben den größten gemeinsamen Teiler von 334 und 776.

Ergebnis ist selbst wieder eine Funktion.

```
ggT :: Int -> Int -> Int
ggT a b | a==b = a
        | a>b  = ggT (a-b) b
        | a<b  = ggT a (b-a)
... ggt 334 776 ...
```


Die vorigen Beispiele in der Programmiersprache OCaml:

- abs: $Z \rightarrow N_0$

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

25 + abs(1000-27000)

```
let abs x =
   if x < 0 then -x
            else x;;

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

- celsius: $R \rightarrow R$

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

celsius(68.0)

```
let celsius f =
  5.0 *. (f -. 32.0) /. 9.0;;

celsius 68.0;;
```

Eigene Operationen für `float`.

- ggT: $N \times N \rightarrow N$

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

334 * 776 / ggT(334, 776)

```
let rec ggT a b =
  if a=b then a
  else if a>b then ggT (a-b) b
       else ggT a (b-a);;

334 * 776 / ggT 334 776;;
```


Die Sprachkonzepte in OCaml:

```
let rec ggT a b =
  if a=b then a
  else if a>b then ggT (a-b) b
       else ggT a (b-a);;

334 * 776 / ggT 333 776;;
```

Parameter

Funktionsrumpf

Funktionsaufruf

Die Funktionalität wird in OCaml nicht explizit angegeben. Sie ergibt sich jedoch eindeutig aus der Art, wie die Parameter verwendet werden, und aus dem Typ des Ausdrucks im Funktionsrumpf.

Die Programmiersprache OCaml

- Ursprung ist die funktionale Programmiersprache ML (*MetaLanguage*), die von Robin Milner in Edinburgh um 1973 für den Theorembeweiser LCF entwickelt wurde.
- Am INRIA (Frankreich) wurde unter Gérard Hue 1984-1985 ML zu Caml (*Categorical Abstract Machine + ML*) weiter entwickelt.
- Wiederum am INRIA wurde schließlich 1990 unter Xavier Leroy Caml zu OCaml (Objective Caml) erweitert. OCaml vereinigt funktionale, imperative und objektorientierte Konzepte.
- Zu ML/Caml verwandte Sprachen sind u.a. SML (Standard ML), Haskell und Gofer.
- 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.

Noch einmal dieselben Beispiele, diesmal in Java

- abs: $Z \rightarrow N_0$

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

25 + abs(1000-27000)

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

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

- celsius: $R \rightarrow R$

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

celsius(68.0)

```
double celsius (double f) {
  return 5.0*(f-32.0)/9.0;
}
... celsius(68.0) ...
```

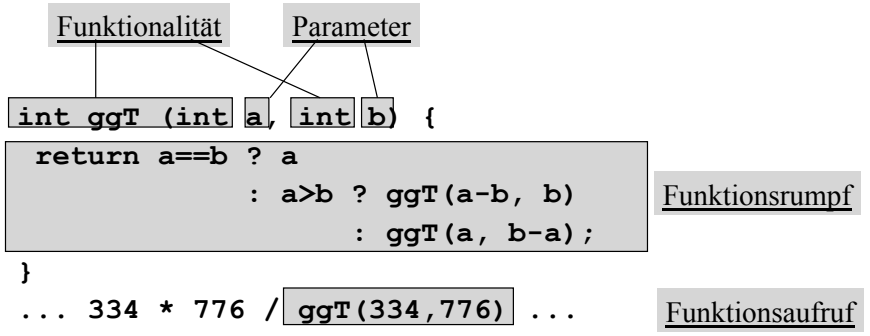
- ggT: $N \times N \rightarrow N$

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

334 * 776 / 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 funktionalen Sprachkonzepte in Java



Funktionaler Programmierstil in Java

- Java ist keine funktionale Programmiersprache.
- Sondern: Java ist eine objektorientierte Sprache.
- Dennoch: der funktionale Programmierstil lässt sich auch in Java gut ausdrücken.
- Wichtige Unterschiede zwischen Java und Gofer bzw. OCaml bzw. der mathematischen Notation:
  - Funktionen werden in Java durch Methoden implementiert. Methodenrümpfe sind in Java grundsätzlich Anweisungen und nicht Ausdrücke.
    Für die Programmierung im funktionalen Programmierstil 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 Funktionsaufrufes (meist der Methode **`main()`**).
- Im folgenden werden wir auch in Java oft von Funktionen statt von Methoden sprechen. Wir meinen damit die Verwendung von Methoden im Sinne des funktionalen Programmierstils.

„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)
- 1998: JDK 1.2 (Java 2)
- Verbreitet als Java ME, Java SE, Java EE (Micro/Standard/Enterprise Edition)
- Seit September 2018 aktuelle Version für Java SE: 1.11 (Java SE 11)
- Kostenlos verfügbar unter http://www.oracle.com/technetwork/java/index.html

Syntax von Ausdrücken

- **Vorbemerkungen**:
  - Wir werden die Syntax von Ausdrücken, die als Funktionsrümpfe zugelassen sind, **induktiv** 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 betrachten wir nur die wichtigsten Sprachelemente für Ausdrücke.
- **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/ggT(334,776)` ist vom Typ `int`

Syntax von Ausdrücken: Grundelemente

- **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 im Funktionsrumpf 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`.

Syntax von Ausdrücken: Arithmetische Operatoren

- **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

Syntax von Ausdrücken: Vergleichsoperatoren

- **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`.

Syntax von Ausdrücken: Boolesche Operatoren

- **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^B)`
    jeweils Ausdrücke vom Typ `boolean`.
  - Beispiele:
    - `false | !false`
    - `((1<=x) & (x<=n))`

Syntax von Ausdrücken: Funktionsaufruf

- **Funktionsaufruf**:
  - Ist `f` die folgendermaßen vereinbarte Funktion:
    $\texttt{T f (T}_1\ \texttt{x}_1\texttt{, T}_2\ \texttt{x}_2\texttt{,…, T}_n\ \texttt{x}_n\texttt{)\{return A;\}}$
    - mit dem Ergebnistyp `T`
    - mit Parametern $\texttt{x}_1$ vom Typ $\texttt{T}_1$, $\texttt{x}_2$ vom Typ $\texttt{T}_2$, … und $\texttt{x}_n$ vom Typ $\texttt{T}_n$,
    - und einem Ausdruck `A` vomTyp `T`,
  - und sind $\texttt{A}_1\texttt{,A}_2\texttt{,…,A}_n$ Ausdrücke von den Typen $\texttt{T}_1\texttt{,T}_2\texttt{,…,T}_n$,
  - dann ist der Funktionsaufruf $\texttt{f(A}_1\texttt{,A}_2\texttt{,…,A}_n\texttt{)}$ ein Ausdruck vom Typ `T`.
  - Beispiel: `ggT(334+9,667-5)` ist ein Ausdruck vom Typ `int`.
  - $\texttt{x}_1\texttt{,x}_2\texttt{,…,x}_n$ heißen *formale Parameter* von `f`
  - $\texttt{A}_1\texttt{,A}_2\texttt{,…,A}_n$ heißen *aktuelle Parameter* beim Aufruf von `f`

Syntax von Ausdrücken: Bedingter Ausdruck

- **Bedingter Ausdruck**:
  - Sind $\texttt{A}_1$ und $\texttt{A}_2$ zwei Ausdrücke vom selben Typ `T`, und ist `B` ein Ausdruck vom Typ `boolean` (eine Bedingung), so ist
    $\texttt{(B ? A}_1\texttt{ : A}_2\texttt{)}$ -- sprich: falls `B` dann $\texttt{A}_1$ sonst $\texttt{A}_2$
    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 äquivalent zum Ausdruck `(A==B)`.

Bedingter Ausdruck vs. if-then-else-Konstrukt

- Leider ist in Java die Syntax des bedingten Ausdrucks nicht besonders gut lesbar: **`( B ? A1 : A2 )`**
  - Deshalb wird sie in der Regel nicht oft verwendet.
- In Gofer oder OCaml ist die Syntax intuitiver:
  `if B then A1 else A2`
- In Java wird der bedingte Ausdruck meist durch die bedingte Anweisung ersetzt:
  - Bei der Zuweisung schreibt man z.B. statt
    ```
    abs = (x<0 ? -x : x)
    ```
    meist
    ```
    if (x<0) abs = -x; else abs = x;
    ```
  - Bei der **`return`**-Anweisung schreibt man z.B. statt
    ```
    return (x < 0 ? -x : x)
    ```
    meist
    ```
    if (x<0) return -x; else return x;
    ```
- Wir werden den bedingten Ausdruck in Java wegen der schlechten Lesbarkeit nur in diesem Vorlesungskapitel verwenden.

Weglassen von Klammern in Ausdrücken

- 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`**

Was werden wir in Kapitel 5 noch alles besprechen?

- ✓ Syntax von Ausdrücken
- Auswertung von Ausdrücken
- Beispiele von rekursiven Funktionen
- Arten von Rekursionen
- Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Was werden wir in Kapitel 5 noch alles besprechen?

- ✓ Syntax von Ausdrücken
- ➢ Auswertung von Ausdrücken
- Beispiele von rekursiven Funktionen
- Arten von Rekursionen
- Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Auswerten von undefinierten Ausdrücken

- **Erinnerung** (Folie 5): Bei der Ausführung eines funktionalen Programms wird der **Ausdruck ausgewertet.**
- **Undefinierter Wert** eines Ausdrucks: Manchmal ist der Wert eines Ausdrucks nicht definiert,
  - z.B. bei Division durch 0, bei Zugriff auf unzulässigen Reihungsindex, bei endloser Folge von Berechnungsschritten.
- Systeme reagieren darauf unterschiedlich,
  - z.B. Abbruch, Fehlermeldung, Ausnahme, Ignorieren, zufällige Fortführung.
- Wir führen einen expliziten Wert **`undefiniert`** ein (in Zeichen ⊥), der andeuten soll, dass das Verhalten des Systems nach Auswerten dieses Ausdrucks nicht näher festgelegt ist.
  - Dabei unterscheiden wir nicht nach dem Typ des Ausdrucks: ⊥ ist Wert eines jeden undefinierten Ausdrucks.

Auswerten von arithmetischen, booleschen und Vergleichs-Operationen

- bei arithmetischen, booleschen und Vergleichs-Operationen werden im ersten Schritt 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 der Operanden angewendet.
- Ist einer der ermittelten Werte der Operanden undefiniert (in Zeichen ⊥), z.B. weil eine Division durch 0 auftritt, so ist auch das Ergebnis der Operation undefiniert.
- Diese Art der Auswertung (Ergebnis ⊥, falls einer der Operanden = ⊥) nennt man **strikte Auswertung**. Die entsprechenden Operatoren nennt man **strikte Operatoren**.

Nicht strikte Operatoren (nicht strikte Auswertung)

- 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 ⊥.

Auswertung bedingter Ausdrücke

- Auswertung von **`( B ? A1 : A2 )`**
  - Zuerst wird die Bedingung, also der Ausdruck **`B`** ausgewertet.
  - Liefert die Auswertung von **`B`** den Wert **`true`**, dann wird $\mathbf{A_1}$ ausgewertet und der Wert des bedingten Ausdrucks ist der Wert von $\mathbf{A_1}$.
  - Liefert die Auswertung von **`B`** false, dann wird $\mathbf{A_2}$ ausgewertet und der Wert des bedingten Ausdrucks ist der Wert von $\mathbf{A_2}$.
  - Liefert die Auswertung von **`B`** den Wert ⊥, dann ist der Wert des bedingten Ausdrucks ebenfalls ⊥.
- Fasst man den bedingten Ausdruck als dreistelligen Operator **`(.?.:.)`** auf, dann ist die Auswertung dieses Operators nicht strikt:
  - **`(true ? 9999 : <Ausdruck>)`** liefert **`9999`**, auch wenn **`<Ausdruck>`** den Wert ⊥ hat.

Auswertung von Funktionsaufrufen

- Ist f die deklarierte Funktion:

  `T f (T1 x1, T2 x2,..., Tn xn) {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 ***Funktionsaufruf*** **`f(A1,A2,...,An)`** folgendermaßen ***ausgewertet***:

  – zuerst werden **alle** aktuellen Parameter A_v ausgewertet;

  – liefert ein A_v den Wert $\perp$, dann liefert der Funktionsaufruf $\perp$;

  – ansonsten ist der Wert des Funktionsaufrufes der Wert des Ausdruckes **`A`**, wenn dort jedes Auftreten eines formalen Parameters x_v durch den Wert des aktuellen Parameters A_v ersetzt wird (auch Parametersubstitution genannt).

- 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 ;
}
```

Schauen wir uns jetzt einmal den Aufruf **`istKalt(10.0 + 49.0)`** an:

– A_i ist **`10.0 + 49.0`**

– Auswertung des Ausdruckes A_i liefert **`59.0`** (also nicht $\perp$)

– Substitution von **`f`** durch **`59.0`** im Rumpf von **`istKalt`**:

```
return celsius(59.0) <= 10.0;          (XX)
```

– Auswertung des Funktionsaufrufes **`celsius(59.0)`**

– Substitution von **`f`** durch **`59.0`** im Rumpf von **`celsius`**:

```
return 5.0*(59.0-32.0)/9.0;
```

– Dies liefert nacheinander **`5.0*27.0/9.0`**, **`135.0/9.0`**, **`15.0`**

– **`15.0`** eingesetzt in (XX) ergibt: **`return 15.0 <= 10.0;`**

– Und dies liefert das Endergebnis: **`false`**

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

- **Wertaufruf (Call-by-Value):** Eine Form der Auswertung von Funktionsaufrufen, in der die Parameter im Funktionsrumpf (formale Parameter) durch die ***Werte*** der auf Parameterposition stehenden Ausdrücke (aktuelle Parameter) ersetzt werden.
- Eine andere Art der Auswertung ist **Namensaufruf (Call-by-Name):**
  - Die formalen Parameter im Funktionsrumpf werden ***textuell*** durch die nicht ausgewerteten aktuellen Parameter ersetzt.
    - **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 (wie in Java)

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

Pseudocode:
Wenn t gleich 0 ist, dann x^3.
Sonst, wenn t gleich 1 ist, dann x^2y.
Sonst x*y*z

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

- Auswerten des Funktionsaufrufes:
  – Call-by-Value Auswertung der 4 aktuellen Parameter liefert: **`1, 104, -80, 120`**
  – Substitution der Werte der aktuellen Parameter im Rumpf von **`auswahl`** liefert:
  `return 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 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 (nicht in Java)

```
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)...`

- Textersetzung (Substitution) der aktuellen Parameter liefert:

```
1*1==0 ? (12-8+100)*(12-8+100)*(12-8+100)
       : 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+100)
      : 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 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 (Methodenaufrufen) durch Call-by-Value.
  - Die formalen Parameter im Funktionsrumpf werden ersetzt durch die **Werte** der aktuellen Parameter im Funktionsaufruf.
  - 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 und rekursives Problemlösen

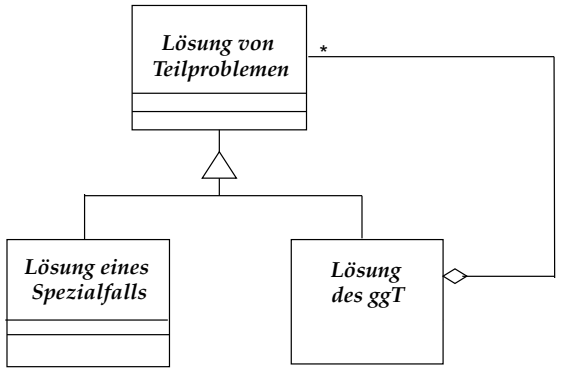
- Beispiel: ggT

```
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 zwei Aufrufe von ggT auf (Selbstaufrufe).
- Definition: Eine Funktion bzw. eine Funktionsdeklaration heißt **rekursiv**, falls der Ausdruck im Rumpf der Funktion einen Aufruf derselben Funktion enthält.
- Rekursive Funktionen entsprechen rekursiven Problemlöse-techniken:
  - die Berechnung des ggT erfordert eine Terminierungsbedingung und die Berechnung von 2 rekursiven Funktionsaufrufen
  - die Lösung des ggT zerfällt in die Behandlung eines Spezialfalls und die Lösung von ähnlichen, aber kleineren Problemen.

Rekursives Problemlösen 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);
}
```

- Funktionsaufruf: `ggT (9, 12)`
- Parameter im Rumpf substituieren:

```
9==12 ? 9 : 9>12 ? ggT(9-12,12) : ggT(9,12-9)
```

- Auswertung des ersten bedingten Ausdruckes ergibt `false` :

```
9>12 ? ggT(9-12,12) : ggT(9,12-9)
```

- Auswertung des zweiten bedingten Ausdruckes ebenfalls `false` :

```
ggT(9,12-9)
```

- Auswertung der aktuellen Parameter (Call-by-Value):

```
ggT(9,3)
```

- Parameter im Rumpf substituieren:

```
9==3 ? 9 : 9>3 ? ggT(9-3,3) : 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)               -- AuswBed
✓ ggT(9,12-9)                                     -- AuswBed
✓ 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)         -- AuswBed
                  ggT(9-3,3)                      -- AuswBed
                  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)         -- AuswBed
                  ggT(6-3,3)                      -- AuswBed
                  ggT(3,3)                        -- aktPar
  3==3 ? 3 : 3>3 ? ggT(3-3,3) : ggT(3,3-3)        -- ParSub
    3                                             -- AuswBed
```


Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- Beispiele von rekursiven Funktionen
- Arten von Rekursionen
- Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ➢ Beispiele von rekursiven Funktionen
- Arten von Rekursionen
- Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Überblick über das Thema Rekursion

Drei Beispiele für rekursive Funktionen:

- Berechnung der Summe der ersten n natürlichen Zahlen
- Berechnung der Fakultätsfunktion
- Multiplikation durch Addition und Subtraktion

Vier Rekursionsarten:

- Lineare Rekursion
- Repetetive Rekursion
- Kaskadenartige Rekursion
  - Fibonacci Zahlen als Beispiel für kaskadenartige Rekursion
- Verschränkte Rekursion

Beispiel 1: Berechnung der Summe der ersten n Zahlen

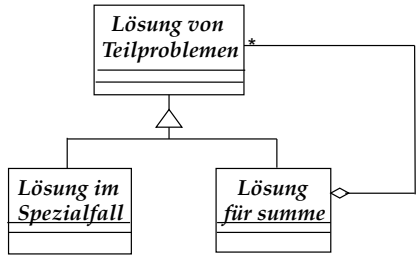
- Gesucht: eine Funktion summe, die die Summe der ersten n natürlichen Zahlen berechnet:
  - summe: $N_0 \rightarrow N_0$
  - summe(n) = 1 + ... + n
- Problem: Wie setze ich die Pünktchen „ ... “ in eine präzise Funktionsdeklaration um?
- Lösung: rekursiver Ansatz summe(n) = summe(n-1) + n

Der Spezialfall:

summe(0)

Das selbstähnliche kleinere Problem:

summe(n-1)



Umsetzung in die Sprache Java

- Funktionalität: $N_0 \rightarrow N_0$
- Parameter: n
- Funktionsrumpf
- Spezialfall: summe(0) = 0
- Rekursiver Fall: summe(n) = summe(n-1) + n

Das selbstähnliche Problem

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

- Bemerkung:
  - Die Schnittstelle der Java-Funktion entspricht nicht der Funktionalität $N_0 \rightarrow N_0$, 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 (Beispiel)

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

Funktionsaufruf summe(3)

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

Beispiel 2: Berechnung der Fakultätsfunktion

- Gesucht: eine Funktion fakultät, die das **Produkt** der ersten n natürlichen Zahlen berechnet:
  - fakultät: $\mathbb{N}_0 \rightarrow \mathbb{N}$
  - fakultät(n) = 1 * ... * 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` bereits nicht mehr darstellen.
- In Java gibt es einen weiteren Typ `long` für Zahlen zwischen -2^{63} und $2^{63}-1$

Beispiel 3: Multiplikation durch Addition und Subtraktion

- Gesucht: eine Funktion mult, die die Multiplikation zweier ganzer Zahlen auf Addition und Subtraktion zurückführt:
  - mult: $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
  - mult(x, y) = x * y -- allerdings ohne Verwendung von *
- **Idee:**
  - mult(x, 0) = 0
  - mult(x, y) = mult(x, y-1) + x, falls $y \neq 0$
- 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 (Beispiel)

```
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)                                  -- AuswBed
       -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)    -- AuswBed
                                 -(mult(5,2-1) + 5)  -- AuswBed
                                 -(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)
                                  -(mult(5,1-1) + 5) + 5)
                                  -(mult(5,0) + 5) + 5)
-((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)
                            -(((0) + 5) + 5)
-10                                                  -- Arithm
```

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ❖ Arten von Rekursionen
- ❖ Terminierung von Funktionen
- ❖ Korrektheit von Funktionen
- ❖ Das Verhältnis zwischen Induktion und Rekursion
- ❖ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Wo stehen wir?

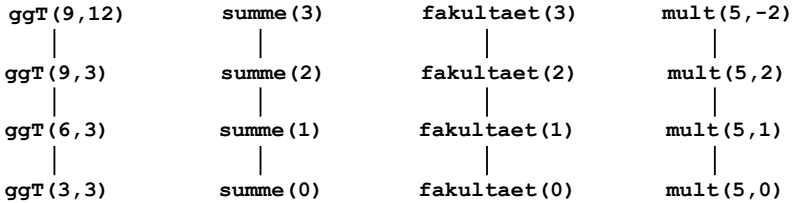
- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ➢ Arten von Rekursionen
- ❖ Terminierung von Funktionen
- ❖ Korrektheit von Funktionen
- ❖ Das Verhältnis zwischen Induktion und Rekursion
- ❖ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ➢ Arten von Rekursionen
  - ➢ Lineare Rekursion
  - ➢ Repetitive Rekursion
  - ➢ Kaskadenartige Rekursion
  - ➢ Verschränkte Rekursion
- ❖ Terminierung von Funktionen
- ❖ Korrektheit von Funktionen
- ❖ Das Verhältnis zwischen Induktion und Rekursion
- ❖ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Rekursionsarten: Lineare Rekursion

- ❖ Definition **Lineare Rekursion**:
  - Eine rekursive Funktion bzw. Funktionsdeklaration heißt linear rekursiv, wenn in jedem Zweig des bedingten Ausdrucks höchstens **ein** Selbstaufruf der Funktion auftritt.
- ❖ **Bemerkung**: Alle bisher betrachteteten rekursiven Funktionen (ggT, summe, fakultaet, mult) sind linear rekursiv.
- ❖ Eine Funktion ist genau dann linear rekursiv, wenn ihre *Aufrufstruktur* linear ist:



Rekursionsarten: Repetitive Rekursion

- Definition **Repetitive Rekursion**:
  - Eine linear rekursive Funktion bzw. Funktionsdeklaration heißt repetitiv rekursiv (*tail recursion*), wenn jeder Selbstaufruf dieser Funktion der letzte auszuwertende (Teil-)Ausdruck ist.

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

Nach den rekursiven Aufrufen muss jeweils keine weitere Operation mehr ausgewertet werden. **ggT** ist also repetitiv rekursiv.

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

Nach einem der rekursiven Aufrufe muss noch eine Negation, nach dem anderen eine Addition ausgeführt werden. mult ist also **nicht** repetitiv rekursiv.

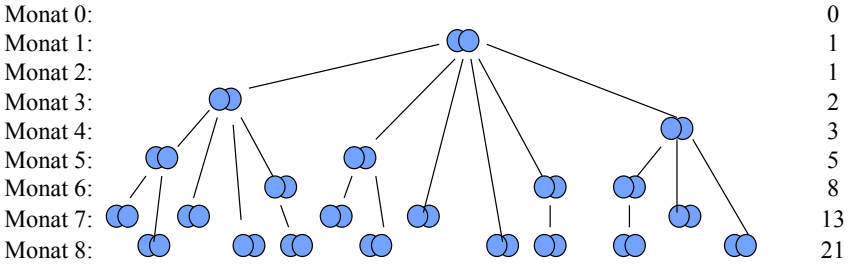
- Das nachträgliche Ausführen einer Operation wird auch „Nach-klappern“ genannt.

Rekursionsarten: Kaskadenartige Rekursion

- Definition **Kaskadenartige Rekursion**:
  - Eine rekursive Funktion bzw. Funktionsdeklaration, die **nicht** linear rekursiv ist, heißt kaskadenartig rekursiv.
  - Eine Funktion ist also kaskadenartig rekursiv, wenn in mindestens einem Zweig des bedingten Ausdrucks mehr als ein Selbstaufruf der Funktion auftritt.

Die Fibonacci-Zahlen

- Gegeben sei ein neu geborenes Kaninchenpaar. Jedes Kaninchenweibchen, das mindestens zwei Monate alt ist, bringt jeden Monat ein weiteres Paar zur Welt. Kaninchen leben ewig. Wie viele Kaninchenpaare gibt es nach n Monaten?



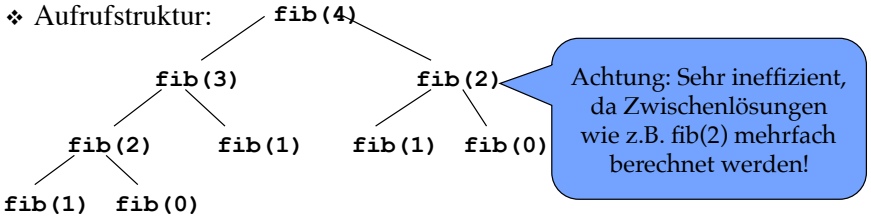
- Diese Aufgabe stellte Fibonacci im Jahre 1202.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... heißen Fibonacci-Zahlen.
- Für uns sind die Fibonacci-Zahlen interessant, weil jede Fibonacci-Zahl die Summe ihrer beiden Vorgänger ist: $fib_n = fib_{n-1} + fib_{n-2}$ für n>1

Berechnung der Fibonacci-Zahlen

- $fib_n = fib_{n-1} + fib_{n-2}$ für n>1

```
int fib (int n) {
  return n==0 ? 0
              : n==1 ? 1
                     : fib(n-1) + fib(n-2);
}
```

- Aufrufstruktur:



- fib ist kaskadenartig rekursiv.

Rekursionsarten: Verschränkte Rekursion

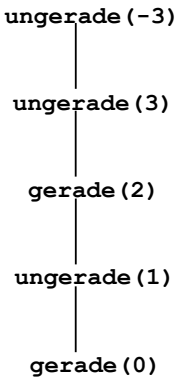
- Gesucht: zwei Funktionen **gerade** und **ungerade**, die feststellen, ob eine ganze Zahl gerade ist bzw. ob sie ungerade ist.

```
boolean gerade (int x) {
  return x<0 ? gerade(-x)
             : x==0 ? true
                    : ungerade(x-1) ;
}

boolean ungerade (int x) {
  return x<0 ? ungerade(-x)
             : x==0 ? false
                    : gerade(x-1) ;
}
```

- Definition **Verschränkte Rekursion**
  - Zwei oder mehr Funktionen, die sich gegenseitig aufrufen, heißen **verschränkt** rekursiv.

Aufrufstruktur der Funktionen gerade bzw. ungerade

```
boolean gerade (int x) {
  return x<0 ? gerade(-x)
             : x==0 ? true
                    : ungerade(x-1) ;
}

boolean ungerade (int x) {
  return x<0 ? ungerade(-x)
             : x==0 ? false
                    : gerade(x-1) ;
}
```



Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ➢ Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ➢ Terminierung von Funktionen
  - ➢ Nichtterminierende Funktionen
  - ➢ Nachweis der Terminierung
- ❖ Korrektheit von Funktionen
- ❖ Das Verhältnis zwischen Induktion und Rekursion
- ❖ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Nicht terminierende Auswertung einer Funktion

- ❖ Betrachten wir noch einmal die Funktion summe:

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

- ❖ Was geschieht beim Aufruf von **`summe(-2)`** ?

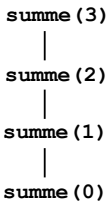
- ❖ `summe(-2)`
- ❖ `-2==0 ? 0 : summe(-2-1) + -2` -- ParSub
- ❖ `summe(-2-1) + -2` -- AuswBed
- ❖ `summe(-3) + -2` -- AktPar
- ❖ `(-3==0 ? 0 : summe(-3-1) + -3) + -2` -- ParSub
- ❖ `(summe(-3-1) + -3) + -2` -- AuswBed
- ❖ `(summe(-4) + -3) + -2` -- AktPar
- ❖ `((-4==0 ? 0 : summe(-4-1) + -4) + -3) + -2` -- ParSub
- ❖ `((summe(-4-1) + -4) + -3) + -2` -- AuswBed
- ❖ `((summe(-5) + -4) + -3) + -2` -- AktPar
- ❖ `...` Die Abbruchbedingung `n==0` wird offensichtlich niemals erreicht!

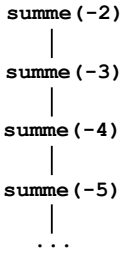
Nicht terminierende Funktion

- ❖ Der Auswertung von **`summe(-2)`** führt zu den Funktionsaufrufen **`summe(-3)`** , **`summe(-4)`** , **`summe(-5)`** , ...
- ❖ Allgemein:
  - – Wenn die Auswertung einer Funktion **`f`** für einen Parameterwert **`w`** zu einer unendlichen Folge von rekursiven Aufrufen von **`f`** führt, so sagen wir: **`f`** **terminiert für** **`w`** **nicht**.
- ❖ Wichtige Fragestellungen bei rekursiven Funktionen:
  - – Für welche Parameterwerte terminiert eine Funktion?
  - – Wie kann die Terminierung nachgewiesen werden?

Beobachtungen zu Terminierung und Nicht-Terminierung

- ❖ Betrachten wir die Aufrufe von **`summe(3)`** und **`summe(-2)`**:





Der Abstand der Parameterwerte zum Terminierungsfall **`(n==0)`** wird mit jedem Aufruf kleiner, bis der Terminierungsfall erreicht wird.

Der Abstand der Parameterwerte zum Terminierungsfall **`(n==0)`** wird mit jedem Aufruf größer: der Terminierungsfall wird niemals erreicht.

Weitere Beobachtungen zur (Nicht-)Terminierung

- Eine Variante der Funktion Summe, die nur die geraden Zahlen bis n addiert:

```
int summeGerade (int n) {
  return n == 0 ? 0 : summeGerade(n-2) + n;
}
```

- Der Aufruf `summeGerade(6)`

```
summeGerade(6)
      |
summeGerade(4)
      |
summeGerade(2)
      |
summeGerade(0)
```

Der Abstand der Parameterwerte zum Terminierungsfall `(n==0)` wird wieder kleiner, bis der Terminierungsfall erreicht ist.

- Der Aufruf `summeGerade(5)`

```
summeGerade(5)
      |
summeGerade(3)
      |
summeGerade(1)
      |
summeGerade(-1)
      |
     ...
```

Der Abstand wird zwar zunächst kleiner, der Terminierungsfall wird aber „übersprungen“.

Folgerungen aus den Beobachtungen zur Terminierung

- Die Beobachtungen lassen zwei Kriterien erkennen, aus denen man offensichtlich auf die Terminierung einer Funktion schließen kann:
  - der Abstand der Parameterwerte vom Terminierungsfall wird bei jedem rekursiven Funktionsaufruf kleiner **und**
  - der Terminierungsfall wird auch tatsächlich erreicht und nicht „übersprungen“.
- Wie misst man aber den Abstand zum Terminierungsfall, wenn es mehrere Parameter gibt (z.B. beim ggT) oder wenn die Parameter keine Zahlen sind?
- Idee, die auf der nächsten Folie formalisiert wird:
  - Wir bilden die Parameter mittels einer „geschickt“ gewählten Funktion h so auf die natürlichen Zahlen N_0 ab, dass die h-Bilder der Parameter von Aufruf zu Aufruf kleiner werden.
  - Da wir fordern, dass als Bild der Funktion h immer eine natürliche Zahl aus N_0 (und nicht eine ganze Zahl) auftritt, kann es nur endlich viele Aufrufe geben, da die 0 nicht „übersprungen“ werden kann.

Nachweis der Terminierung (vereinfachte Version)

- Gegeben sei folgende rekursive Funktionsdeklaration:
  `T f (T1 x1, T2 x2,… , Tn xn) {return A;}`
  wobei der Ausdruck `A` rekursive Funktionsaufrufe der Form `f(A1,A2,…,An)` enthalte.
- Gegeben sei weiterhin eine Funktion
  h: $T_1 \times T_2 \times \ldots \times T_n \rightarrow N_0$
  die jeder Kombination von Parameterwerten eine natürliche Zahl zuordnet.
- Gilt dann für jeden rekursiven Funktionsaufruf
  $h(a_1, a_2, \ldots, a_n) < h(t_1, t_2, \ldots, t_n)$
  wobei $a_1, a_2, \ldots, a_n$ die Werte sind, die sich aus den Ausdrücken $A_1, A_2, \ldots, A_n$ berechnen, und $t_1, t_2, \ldots, t_3$ die Parameterwerte des ursprünglichen Aufrufes sind,

dann **terminiert** `f` für alle Parameterwerte aus $T_1, T_2, \ldots, T_n$

Hier ist das Kriterium „der Terminierungsfall wird nicht übersprungen“ formalisiert.

Hier ist das Kriterium „der Abstand wird kleiner“ formalisiert.

Nachweis der Terminierung von mult für alle int-Werte

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

- `T`, T_1 und T_2 sind in diesem Fall alle int
- Wähle für h folgende Funktion:

$$h(x,y) = \begin{cases} y, & \text{falls } y \geq 0 \\ |y| + 1, & \text{sonst} \end{cases} = \begin{cases} y, & \text{falls } y \geq 0 \\ -y+1, & \text{sonst} \end{cases}$$

- Nun betrachten wir jeden rekursiven Funktionsaufruf:
  - Der rekursive Aufruf `mult(x,-y)` wird ausgewertet, falls $y < 0$ gilt. In diesem Fall ist $-y > 0$ und damit gilt für h:
    - $h(x, -y) = -y < -y+1 = h(x, y)$
  - Der rekursive Aufruf `mult(x,y-1)` wird nur ausgewertet, falls $y > 0$ gilt. In diesem Fall ist $y-1 \geq 0$ und damit gilt für h:
    - $h(x, y-1) = y-1 < y = h(x, y)$
- Also terminiert `mult` für alle Werte aus int.

Das Bild von h liegt in beiden Fällen in N_0.

„Geschickte“ Wahl von h für den Sonderfall y<0:

Also „sinkt“ h in diesem Fall

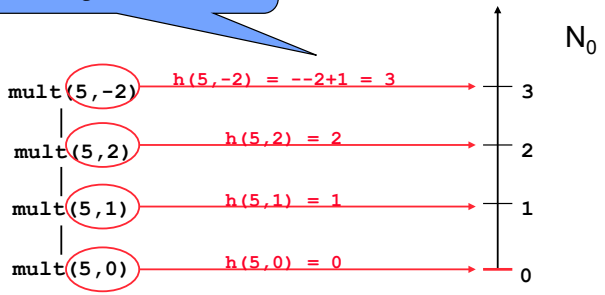
Also „sinkt“ h auch in diesem Fall

Die Abstiegsfunktion h am Beispiel mult(x,y)

- Wir haben für h folgende Funktion gewählt:

$$h(x,y) = \begin{cases} y, & \text{falls } y \geq 0 \\ |y| + 1, & \text{sonst} \end{cases} = \begin{cases} y, & \text{falls } y \geq 0 \\ -y+1, & \text{sonst} \end{cases}$$

„Geschickte“ Wahl von h:
h(5,-2) ist um 1 größer als h(5,2)



Bemerkungen zum Terminierungsnachweis

- Der Terminierungsnachweis ist korrekt, da in N_0 jede monoton fallende Folge endlich ist. Eine unendliche Folge rekursiver Aufrufe ist also durch die Voraussetzungen ausgeschlossen.
- Die Funktion h wird *Abstiegsfunktion* genannt.
- Eine *Ordnung*, in der jede monotone Folge endlich ist, wird auch *fundiert* oder *noethersch* genannt.
  - In dem Terminierungsnachweis kann man statt N_0 mit $\leq$ und $<$ auch eine andere Grundmenge mit einer anderen noetherschen Ordnung verwenden.

Terminierungsnachweis für eingeschränkte Bereiche von Parameterwerten

- In der bisherigen Fassung kann die Terminierung nur nachgewiesen werden, wenn sie für alle Parameterwerte aus dem Definitionsbereich gilt.
- In der Regel terminieren Funktionen jedoch nur dann, wenn die Parameter aus bestimmten Teilmengen des Definitionsbereichs stammen.
  - Beispiel: Die Funktion **`summe`** terminiert für Parameter aus N_0, nicht jedoch für negative Parameter.
- Wir verallgemeinern unseren Formalismus nun so, dass die Terminierung nicht für Parameter aus dem gesamten Definitionsbereich nachgewiesen werden muss, sondern nur für eingeschränkte Bereiche.
  - Dabei muss sicher gestellt werden, dass auch die rekursiven Aufrufe nur Parameterwerte aus den eingeschränkten Bereichen verwenden.

Nachweis der Terminierung (allgemeinere Fassung für eingeschränkte Parameterbereiche)

- Gegeben sei wieder folgende rekursive Funktionsdeklaration:
  `T f (T1 x1, T2 x2,... , Tn xn) {return A;}`
  wobei der Ausdruck **`A`** rekursive Funktionsaufrufe der Form **`f(A1,A2,...,An)`** enthalte.
  Einschränkung
- Sei **`E`** $\subseteq$ **`T1`** $\times$ **`T2`** $\times \ldots \times$ **`Tn`** eine Teilmenge des Parameterraumes derart,
  - dass sich die Ausdrücke **`A1`**, **`A2`**, ... , **`An`** eines jeden rekursiven Aufrufes zu Werten $(a_1, a_2, \ldots, a_n)$ aus **`E`** berechnen,

    falls die Parameter $t_1, t_2, \ldots, t_n$ des ursprünglichen Aufrufes ebenfalls Werte dieses Teilraumes sind, d.h. $(t_1, t_1, \ldots, t_n) \in$ **`E`** ,

  und gelte mit einer Funktion h: **`E`** $\rightarrow N_0$, die jeder Kombination von Parameterwerten aus dem eingeschränkten Bereich **`E`** eine natürliche Zahl zuordnet,
  - $h(a_1, a_2, \ldots, a_n) < h(t_1, t_2, \ldots, t_n)$,
- dann **terminiert `f`** für Parameterwerte aus **`E`**.

Nachweis der Terminierung von summe für int-Werte, die nicht negativ sind

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

- $\mathbf{T}_1$ (der Typ des ersten Parameters) ist in diesem Fall also int.
- Wähle für **E** die Werte aus int, die nicht negativ sind.
- Wähle als Abstiegsfunktion h: $\mathbf{E} \to \mathbb{N}_0$ mit h(n) = n.
- Dann gilt:
  - Der rekursive Aufruf **summe(n-1)** wird nur in dem Fall ausgewertet, wenn **n** nicht **0** ist;
  - Damit hat **summe(n-1)** einen nichtnegativen Parameterwert (der also ebenfalls im eingeschränkten Parameterbereich **E** liegt), falls **summe** mit einem nichtnegativen Wert **n** aufgerufen wurde.
  - Außerdem gilt dann auch: h(n-1) = n-1 < n = h(n)
- Also terminiert **summe** für nichtnegative Werte aus int.

Nachweis der Terminierung von fib für int-Werte, die nicht negativ sind

```
int fib (int n) {
  return n==0 ? 0 : n==1 ? 1 : fib(n-1) + fib(n-2); }
```

- $\mathbf{T}_1$ ist in diesem Fall wieder int.
- Wähle für **E** wiederum die Werte aus int, die nicht negativ sind.
- Wähle als Abstiegsfunktion wieder h: $\mathbf{E} \to \mathbb{N}_0$ mit h(n) = n.
- Dann gilt:
  - Die rekursiven Aufrufe **fib(n-1)** und **fib(n-2)** werden nur in dem Fall ausgewertet, wenn **n** weder **0** noch **1** ist;
  - Damit haben **fib(n-1)** und **fib(n-2)** einen nichtnegativen Parameterwert, falls **n** mit einem nichtnegativen Wert **n** aufgerufen wurde.
  - Außerdem gilt dann auch: h(n-1) = n-1 < n = h(n)
  - und analog: h(n-2) = n-2 < n = h(n)
- Also terminiert **fib** für nichtnegative Werte aus int.

Nachweis der Terminierung von ggT für int-Werte, die größer als 0 sind

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

- $\mathbf{T}_1$ und $\mathbf{T}_2$ sind in diesem Fall beide int.
- Wähle $\mathbf{E} = \mathbf{T}' \times \mathbf{T}'$, wobei **T'** die Werte aus int enthält, die größer als **0** sind.
- Wähle folgende Abstiegsfunktion:
  - h: $\mathbf{E} \to \mathbb{N}$ mit h(a, b) = a + b
- Dann gilt:
  - Der rekursive Aufruf **ggT(a-b,b)** wird nur ausgewertet, falls **a>b**
  - Der rekursive Aufruf **ggT(a,b-a)** wird nur ausgewertet, falls **a<b**
  - Damit haben **ggT(a-b,b)** und **ggT(a,b-a)** echt positive Parameterwerte, falls **ggT** mit echt positiven Werten **a** und **b** aufgerufen wurde.
  - Außerdem gilt dann auch: h(a-b, b) = a-b+b = a < a+b = h(a, b)
  - und: h(a, b-a) = a+b-a = b < a+b = h(a, b)
- Also terminiert **ggT** für echt positive Werte aus int.

„Geschickte“ Wahl von h: die Summe wird kleiner, wenn einer der Summanden kleiner wird.

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ➢ Korrektheit von Funktionen
- ❖ Das Verhältnis zwischen Induktion und Rekursion
- ❖ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ➢ Korrektheit von Funktionen
  - ➢ Partielle Korrektheit
  - ➢ Totale Korrektheit
  - ➢ Vollständige Induktion
- ❖ Das Verhältnis zwischen Induktion und Rekursion
- ❖ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Binomialkoeffizienten

- ❖ Betrachten wir folgende Funktion über den natürlichen Zahlen:
  - binom: $\mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$
  - $$\text{binom}(n, k) = \begin{cases} \dfrac{n!}{k! * (n-k)!} & \text{falls } 0 \le k \le n \\ 0 & \text{sonst} \end{cases} \qquad (X)$$
- ❖ Die binom(n, k) werden Binomialkoeffizienten genannt, denn es gilt die Binomische Formel:
  - $(a+b)^n = \text{binom}(n,0) * a^{n-0}b^0 + \text{binom}(n,1) * a^{n-1}b^1 + \ldots + \text{binom}(n,n) * a^{n-n}b^n$
- ❖ Eine andere Schreibweise für binom(n, k) ist $\binom{n}{k}$, gesprochen: „n über k“.
- ❖ Zur Berechnung von binom(n, k) ist die Formel in (X) ungeeignet, da schon für kleine n die Fakultät n! nicht mehr in Java darstellbar ist.
- ❖ Wir verwenden deshalb eine rekursive Funktionsdefinition, die ohne Multiplikation auskommt.

Rekursive Definition der Binomialkoeffizienten

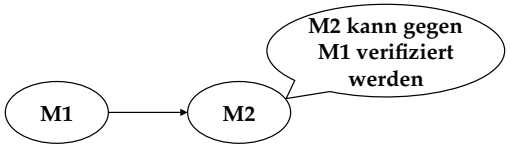
- ❖ Rekursive Definition der Binomialkoeffizienten:
  - binomRek(n, k) = 0, falls k > n
  - binomRek(n, k) = 1, falls k = 0 oder k = n (XX)
  - binomRek(n, k) = binomRek(n-1, k-1) + binomRek(n-1, k), sonst
- ❖ Umgesetzt in Java ergibt dies:

```
int binomRek (int n, int k) {
   return k>n ? 0
              : k==0 || k==n ? 1
              : binomRek(n-1,k-1) + binomRek(n-1, k); }
```

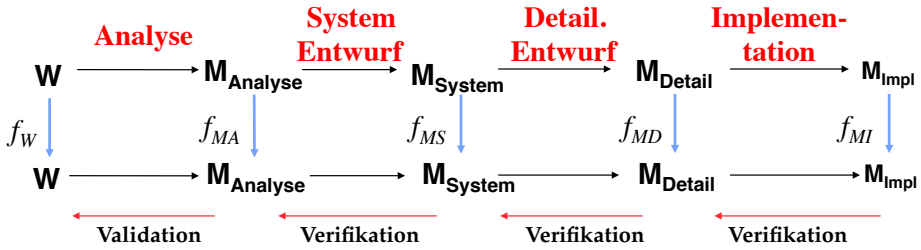
- ❖ Frage zur Terminierung:
  - Terminiert **`binomRek`** für alle natürlichen Zahlen n und k?
    - Antwort JA; Nachweis mit h(n, k) = n
- ❖ Frage zur **Korrektheit**:
  - Berechnet **`binomRek`** wirklich die auf der vorausgegangenen Folie in (X) definierte Funktion?

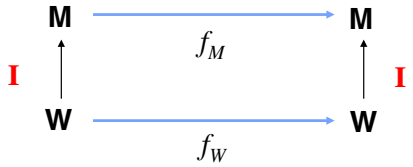
Wiederholung aus Kapitel 2 (Folie 26):

- Definition Spezifikation: **Eine Wirklichkeit, die unserer Gedankenwelt entstammt, oder ein Modell.**
- Die Übereinstimmung einer Spezifikation mit einem Modell, d.h. seine Korrektheit, lässt sich mit mathematischen und logischen Schritten prüfen.
  - **Definition Verifikation:** Die Überprüfung des Wahrheitsgehaltes eines Modells M2 gegen eine Spezifikation M1.

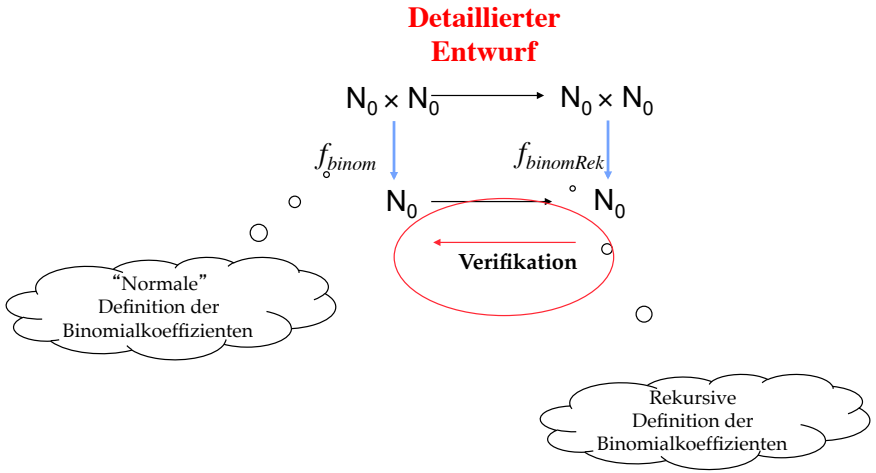


Verifikation und Validation von Modellen





Verifikation der rekursiven Definition von Binomialkoeffizienten



Korrektheit und partielle Korrektheit von Funktionen

- Eine (rekursive) Funktion f heißt für die Parameterwerte $t_1, t_2, \ldots, t_n$ **partiell korrekt**,
  - wenn das berechnete Ergebnis der Spezifikation entspricht, falls die Funktion f für $t_1, t_2, \ldots, t_n$ terminiert (Terminierung von f wird bei partieller Korrektheit also gar nicht vorausgesetzt).
- Eine (rekursive) Funktion f heißt für Parameterwerte $t_1, t_2, \ldots, t_n$ **korrekt**,
  - falls f für die Parameterwerte $t_1, t_2, \ldots, t_n$ **partiell korrekt** ist
  - und falls f für die Parameterwerte $t_1, t_2, \ldots, t_n$ **terminiert**.
- Zum Nachweis der Terminierung eignet sich die Methode mit der Abstiegsfunktion h.
- Zum Nachweis der partiellen Korrektheit eignet sich die Methode der vollständigen **Induktion**.

Partielle Korrektheit und vollständige Induktion

- Ist A(n) eine Aussage über eine natürliche Zahl n (zum Beispiel die Aussage summe(n) = n * (n+1) / 2), dann bedeutet die Methode der vollständigen Induktion folgendes:
  - Gilt für ein $n_{start} \in \mathbb{N}_0$ die Aussage $A(n_{start})$ (sog. **Induktionsanfang**)
  - **und** folgt aus der Gültigkeit der Aussage A(n) die Gültigkeit von A(n+1) (sog. **Induktionsschritt**)
  - **dann** gilt A(n) für alle $n \in \mathbb{N}_0$ mit $n \geq n_{start}$
- Beispiel anhand der summe:
  - **Induktionsanfang:** summe(0) = 0 und 0 * (0+1) / 2 = 0, also gilt A(0)
  - **Induktionsschritt:** summe(n+1) = summe(n) + n+1
    Gilt nun A(n), so ist summe(n) = n * (n+1) / 2 und damit
    summe(n+1) = n * (n+1) / 2 + n+1
    = (n * (n+1) + 2 * (n+1)) / 2
    = (n+1) * (n+2) / 2
    also folgt aus A(n) auch A(n+1)
  - Nach der Methode der vollständigen Induktion gilt A(n) also für alle $n \in \mathbb{N}_0$

Zurück zu den Binomialkoeffizienten

- Direkte Definition über die Fakultätsfunktion:
  - binom: $\mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$
  - $$\text{binom}(n, k) = \begin{cases} \dfrac{n!}{k! * (n-k)!} & \text{falls } 0 \leq k \leq n \\ 0 & \text{sonst} \end{cases} \quad (X)$$
- Rekursive Definition von binomRek(n, k):
  - binomRek(n, k) = 0, falls k > n
  - binomRek(n, k) = 1, falls k = 0 oder k = n (XX)
  - binomRek(n, k) = binomRek(n-1, k-1) + binomRek(n-1, k), sonst
- Wir zeigen die Übereinstimmung von (X) und (XX) mittels vollständiger Induktion über n:
- Dabei lautet die Aussage A(n):
  - für alle natürlichen Zahlen k mit $0 \leq k \leq n$ gilt:
  - $$\text{binomRek}(n, k) = \frac{n!}{k! * (n-k)!}$$

Weiter mit den Binomialkoeffizienten

- Zu beweisen ist die Aussage A(n):

  für alle natürlichen Zahlen k mit $0 \leq k \leq n$ gilt:
  $$\text{binomRek}(n, k) = \frac{n!}{k! * (n-k)!}$$

- Induktionsanfang (A(0)), zu untersuchen ist nur k = 0:
  - binomRek(0, 0) = 1 und 0! / (0! * 0!) = 1 / 1 * 1 = 1
  - also gilt A(0)

- Induktionsschritt ($A(n) \rightarrow A(n+1)$), es gelte A(n):
  - 1. Fall: k = n+1:
    binomRek(n+1, n+1) = 1 und (n+1)! / ((n+1)! * 0!) = 1
  - 2. Fall: k = 0:
    binomRek(n+1, 0) = 1 und (n+1)! / (0! * (n+1)!) = 1

Weiter mit dem Induktionsbeweis:

- 3. Fall: $0 < k < n+1$:

  binomRek(n+1, k) = binomRek(n, k-1) + binomRek(n, k)
  Wegen A(n) gilt damit:
  binomRek(n+1, k) = n! / ((k-1)! * (n-k+1)!) + n! / (k! * (n-k)!)
  Auf gemeinsamen Nenner bringen:
  binomRek(n+1, k) = (n! * k + n! * (n-k+1)) / (k! * (n-k+1)!)
  Zähler ausrechnen und Nenner umformen:
  binomRek(n+1, k) = n! * (k+n-k+1) / (k! * (n-k+1)!)
  = (n+1)! / (k! * ((n+1) - k)!)
  und dies ist genau die Aussage A(n+1)

- In jedem Fall folgt also aus A(n) auch A(n+1) q.e.d.

Varianten zur vollständigen Induktion

- Die bisherige Variante der vollständigen Induktion:
  - gilt $A(n_0)$
  - und folgt aus $A(n)$ auch $A(n+1)$
  - dann gilt $A(n)$ für alle $n \in \mathbb{N}_0$ mit $n \geq n_0$
- Eine Variante mit zwei Spezialfällen als Induktionsanfang:
  - gelten $A(n_0)$ und $A(n_0+1)$
  - und folgt aus $A(n)$ und $A(n+1)$ auch die Gültigkeit von $A(n+2)$
  - dann gilt $A(n)$ für alle $n \in \mathbb{N}_0$ mit $n \geq n_0$
- Die Varianten mit drei oder mehr Spezialfällen als Induktionsanfang sind analog.
- Eine andere Variante des Induktionsschlusses:
  - gilt $A(n_0)$
  - und folgt aus $A(n_0), A(n_0+1), A(n_0+2), \ldots, A(n)$ auch $A(n+1)$
  - dann gilt $A(n)$ für alle $n \in \mathbb{N}_0$ mit $n \geq n_0$

Wo stehen wir?

- ✓ Auswertung von Funktionen
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Auswertung von Funktionen
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ➢ Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Induktion und Rekursion

- **Induktion**: Vom Einfachen zum Komplizierten:
  - Induktion startet im Spezialfall. Die Annahme kann für alle natürlichen Zahlen ab dem Startwert gezeigt werden, da der Induktionsschritt „nur“ endlich oft angewendet werden muss.
- **Rekursion**: Vom Komplizierten zum Einfachen:
  - Die Rekursion führt ein komplexes Problem durch endlich-malige Anwendung des Rekursionsschrittes auf einen Spezialfall zurück.
- Induktion und Rekursion sind endliche Beschreibungsmittel für Zusammenhänge, die beliebig groß sein können. Sie erlauben *Skalierbarkeit*.
  - **Skalierbarkeit**: Datenstrukturen und Programme sind skalierbar, wenn sie auf Probleme angewandt werden können, deren Größe nicht von vornherein feststeht.
- Zum Beweis der partiellen Korrektheit rekursiver Funktionen ist die Induktion das passende Beweisprinzip.

Wann soll Rekursion in der Programmierung verwendet werden?

- In der funktionalen Programmierung **müssen** rekursive Funktionen immer dann verwendet werden, wenn die Anzahl der auszuführenden Operationen nicht von vornherein beschränkt ist,
  - denn dafür ist Rekursion das einzige zur Verfügung stehende Sprachkonzept (es gibt keine Schleifen).
- Aus den vorangegangenen Beispielen konnte man aber erkennen, bei welchen Problemen die Verwendung rekursiver Funktionen besonders vorteilhaft ist, auch wenn andere Sprachkonzepte zur Verfügung stehen:
  - falls sich das Problem auf einige Spezialfälle und die Lösung eines oder mehreren kleineren aber ähnlichen Problemen zurückführen lässt (vgl. ggT oder binom).
- Nun werden wir noch eine neue Problemklasse kennen lernen, für die sich rekursive Funktionen besonders eignen:
  - falls die Datentypen, auf denen die Funktionen arbeiten, selbst rekursiv sind.

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- ➢ Rekursive Datentypen
- Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- ➢ Rekursive Datentypen
  - ➢ Definition und Beispiele
  - ➢ strukturelle Induktion
- Einbettung
- Pattern Matching

Rekursive Datentypen

- Beispiel: Die natürlichen Zahlen können (vereinfacht) folgendermaßen charakterisiert werden:
  - 0 ist eine natürliche Zahl
  - Ist n eine natürliche Zahl, dann ist auch n+1 eine natürliche Zahl.
  - Jede natürliche Zahl lässt sich durch endlich viele der beiden oben genannten Schritte erzeugen.
- Diese Art der Definition wird **induktiv** genannt: ausgehend von Grundelementen wird beschrieben, wie man von vorhandenen Elementen zu weiteren kommt.
- Der entstehende Datentyp wird **rekursiv** genannt: ein Element des Datentyps ist entweder ein Grundelement oder baut auf andere Elemente desselben Typs auf.
- Beispiel: Die Definition der Syntax von Ausdrücken am Anfang des Kapitels war induktiv, aufgefasst als Datentyp sind Ausdrücke rekursiv.

Sequenzen ganzer Zahlen als rekursiver Datentyp

- Betrachten wir Sequenzen ganzer Zahlen:
  - (3,6,4) (3,6,3,4,7) (3) ()
- Wir können auf Sequenzen ganzer Zahlen folgende Operationen definieren:
  - **`create`** ist eine nullstellige (parameterlose, konstante) Funktion, die die leere Sequenz () liefert.
  - **`stock`** ist eine zweistellige Funktion, die eine ganze Zahl vorn an eine Sequenz anfügt:
    - **`stock(3,(6,4)) = (3,6,4)`**
    - **`stock(3,()) = stock(3,create) = (3)`**
    - **`stock(3,stock(6,stock(4,create))) = (3,6,4)`**
- Damit können wir Sequenzen ganzer Zahlen induktiv über **`create`** und **`stock`** definieren:
  - **`create`** ist eine Sequenz ganzer Zahlen.
  - Ist **`z`** eine ganze Zahl und **`s`** eine Sequenz ganzer Zahlen, dann ist auch **`stock(z, s)`** eine Sequenz ganzer Zahlen.
  - Jede Sequenz ganzer Zahlen lässt sich durch endlich viele der beiden oben genannten Schritte erzeugen.

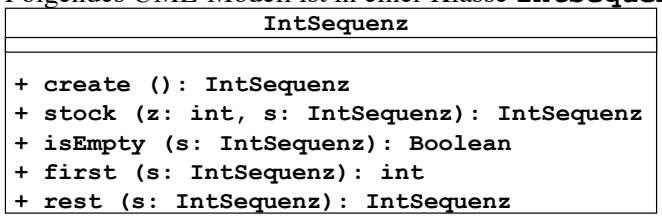
Rekursive Datentypen in funktionalen Programmiersprachen

- In OCaml lässt sich die angegebene Definition für Sequenzen ganzer Zahlen direkt umsetzen:
  - `type intSequenz = Create | Stock of (int * intSequenz);;`
  - damit sind in OCaml
    - `Create`
    - `Stock(1, Create)`
    - `Stock(5, Stock(2, Create))`

    drei Elemente des Typs `intSequenz`
- Auch in Java ist die Definition rekursiver Datentypen möglich.
- Allerdings werden dazu nichtfunktionale Sprachelemente (Referenzvariablen) verwendet.
- Wir werden deshalb für Vorlesung und Übung auch in Java eine Programmierschnittstelle vorgeben, die uns einen funktionalen Zugang zu rekursiven Datentypen erlauben.

Eine Schnittstelle für Sequenzen ganzer Zahlen in Java im Rahmen dieser Vorlesung

- Folgendes UML-Modell ist in einer Klasse **`IntSequenz`** implementiert:



`isEmpty` gibt an, ob eine Sequenz leer ist, **`first`** liefert das erste Element einer Sequenz, **`rest`** liefert die um das erste Element verkürzte Sequenz.

- Beispiel für die Verwendung in einer **`main`**-Funktion:

```
public static void main(String[] args) {
  IntSequenz s = new IntSequenz();
  System.out.println(
       s.first(s.rest(s.stock(1, s.stock(2, s.create())))));
}
```

Beispiel: Berechnung der Länge einer Sequenz

- Gesucht: eine Funktion **`laenge`**, die die Länge einer Sequenz ganzer Zahlen bestimmt. Dabei wollen wir die Länge einer Sequenz induktiv folgendermaßen definieren:
  - die Länge von **`create`** ist 0
  - die Länge von **`stock(z,s)`** ist um 1 größer als die von **`s`**
- Wir ergänzen die Java-Klasse **`IntSequenz`** um folgende Funktion:

```
public int laenge (IntSequenz s) {
   return isEmpty(s) ? 0 : laenge(rest(s)) + 1 ;
}
```

- Frage: terminiert die Funktion **`laenge`** für alle Parameterwerte?
  - Antwort: JA
  - Benutze zum Nachweis die Abstiegsfunktion h(s) = Anzahl der **`stock`** -Aufrufe, die zur Erzeugung von **`s`** aus **`create`** nötig sind

Partielle Korrektheit der Funktion laenge

- die Länge von **`create`** ist 0
- die Länge von **`stock(z,s)`** ist um 1 größer als die von **`s`**

```
public int laenge (IntSequenz s) {
   return isEmpty(s) ? 0 : laenge(rest(s)) + 1 ;}
```

- Frage: ist die Java-Funktion **`laenge`** partiell korrekt, berechnet sie wirklich die Länge einer Sequenz so, wie es oben definiert ist?
- Wir müssen also beweisen, dass die Aussage
  A(**`s`**) = „**`laenge(s)`** berechnet die Länge von **`s`**“
  für alle Sequenzen **`s`** gilt. Dafür benutzen wir eine neue Beweis-Methode, nämlich Induktion über den Aufbau von **`s`**:
- **1. Fall**: **`s = create`**,
  - Dann ist die Länge von **`s`** nach Definition 0.
  - Auch die Java-Funktion **`laenge`** liefert 0, da die Bedingung **`istEmpty(s)`** erfüllt ist.

Partielle Korrektheit der Funktion laenge

- die Länge von **`create`** ist 0
- die Länge von **`stock(z,s)`** ist um 1 größer als die von **`s`**

```
public int laenge (IntSequenz s) {
   return isEmpty(s) ? 0 : laenge(rest(s)) + 1 ;}
```

- Wir beweisen gerade die Aussage
  A(**`s`**) = „**`laenge(s)`** berechnet die Länge von **`s`**“
  durch Induktion über den Aufbau von **`s`**:
- **2. Fall**: **`s = stock(z,r)`**
  Dann sei unsere Induktionsannahme, dass A(**`r`**) gilt, d.h **`laenge(r)`** berechnet die Länge von **`r`**.
  - Nach Definition ist die Länge von **`s`** um 1 größer als die von **`r`**.
  - Und da **`rest(s)`** gleich **`r`** ist, ist wegen der Annahme A(**`r`**) auch das Ergebnis der Funktion **`laenge`** um 1 größer als die Länge von **`r`**.
- Damit gilt: A(**`s`**) für alle Sequenzen **`s`**.

Strukturelle Induktion

- Wir haben zum Beweis der partiellen Korrektheit eine weitere Beweismethode benutzt:
- Strukturelle Induktion:
  - Ist S ein induktiv definierter Datentyp und ist A(s) eine Aussage über ein Element s von S.
  - Gilt für jedes s_{start} aus einer Teilmenge S_{start} von S die Aussage $A(s_{start})$ (**Induktionsanfang**)
  - und folgt aus der Gültigkeit von A(s') für alle s' aus einer Teilmenge S' von S auch die Gültigkeit von A(s'') für alle s'', die aus Elementen von S' in einem Schritt erzeugt werden können (**Induktionsschritt**),
  - dann gilt die Aussage für alle Elemente von S, die aus Elementen von S_{start} in endlich vielen Schritten erzeugt werden können.

Strukturelle Induktion (Fortsetzung)

- Bemerkung:
  - Auf Folie 101 haben wir gezeigt, wie die natürlichen Zahlen als rekursiver Datentyp aufgefasst werden können.
  - Die strukturelle Induktion über den natürlichen Zahlen (als rekursiven Datentypen) ist dann gerade die vollständige Induktion.
  - Die vollständige Induktion ist also ein Spezialfall der strukturellen Induktion.
  - oft orientiert sich eine rekursive Funktion am strukturellen Aufbau eines Datentyps (vgl. **`laenge`**)
  - zum Nachweis der partiellen Korrektheit mittels struktureller Induktion ist deshalb (wie bei **`laenge`**) oft „fast nichts zu zeigen“.
  - Dies bedeutet aber auch, dass es dem Programmierer hier besonders leicht fällt, ein korrektes Programm zu schreiben.

Beispiel: Ist eine Sequenz Teilsequenz einer anderen?

- Gesucht: eine Funktion istTeilsequenz, die feststellt, ob eine Sequenz t in einer Sequenz s als Teilsequenz enthalten ist. Beispiele:
  - ( ) ist in jeder Sequenz enthalten.
  - (1,4) ist nicht in (1,2,3,4) enthalten.
  - (1,4) ist in (0,1,4,2,3,9) enthalten.
  - (1,4) ist in (1,4) enthalten.
- Als Hilfsfunktion suchen wir eine Funktion istPraefix, die feststellt, ob eine Sequenz a mit dem Anfang einer anderen Sequenz s übereinstimmt.
  - ( ) ist Praefix jeder Sequenz.
  - (1,2) ist Praefix von (1,2,3,1,4)
  - (1,4) ist Praefix von (1,4)
  - (1,4) ist nicht Praefix von (1,2,3,1,4)

Beispiel: die Funktion istPraefix in Java

- Die Funktion istPraefix in Java:

```
boolean istPraefix (IntSequenz a, IntSequenz s) {
  return isEmpty(a) ? true
         : isEmpty(s) ? false
           : first(a) != first(s) ? false
             : istPraefix (rest(a), rest(s)) ;
}
```

- terminiert die Funktion istPraefix?
  - Antwort: Ja
  - Abstiegsfunktion: h(a, s) = laenge(a)
- ist die Funktion istPraefix partiell korrekt?
  - Wir beweisen die partielle Korrektheit mittels struktureller Induktion über den ersten Parameter.

Partielle Korrektheit der Funktion istPraefix

```
boolean istPraefix (IntSequenz a, IntSequenz s) {
  return isEmpty(a) ? true
         : isEmpty(s) ? false
           : first(a) != first(s) ? false
             : istPraefix (rest(a), rest(s)) ;  }
```

- **1. Fall**: der Parameter **`a`** ist von der Form **`create`** (Induktionsanfang).
  - Dann ist **`a`** als leere Sequenz Präfix jeder anderen und auch die Funktion **`istPraefix`** liefert immer **`true`**.

Partielle Korrektheit der Funktion istPraefix

```
boolean istPraefix (IntSequenz a, IntSequenz s) {
  return isEmpty(a) ? true
         : isEmpty(s) ? false
           : first(a) != first(s) ? false
             : istPraefix (rest(a), rest(s)) ;  }
```

- **2. Fall**: der Parameter **a** ist von der Form **stock(z,r)** (Induktionsschritt).
  Dann gilt die Induktionsannahme: **istPraefix(r,t)** liefert für alle Sequenzen **t** das korrekte Ergebnis.

  Nun unterscheiden wir für den zweite Parameter **s** wieder zwei Fälle:
  - **Fall 2.1**: **s** ist von der Form **create**:
    dann ist **a** nicht leer und **s** ist leer, das Ergebnis **false** der Funktion ist also korrekt.

Partielle Korrektheit der Funktion istPraefix

```
boolean istPraefix (IntSequenz a, IntSequenz s) {
  return isEmpty(a) ? true
         : isEmpty(s) ? false
           : first(a) != first(s) ? false
             : istPraefix (rest(a), rest(s)) ;  }
```

- **2. Fall**: der Parameter **a** ist von der Form **stock(z,r)** (Induktionsschritt).
  Dann gilt die Induktionsannahme: **istPraefix(r,t)** liefert für alle Sequenzen **t** das korrekte Ergebnis.

  Nun unterscheiden wir für den zweite Parameter **s** wieder zwei Fälle:
  - **Fall 2.2**: **s** ist von der Form **stock(x,y)**:
    - Dann ist **a** sicher kein Präfix von **s**, falls sich die ersten Elemente **z** und **x** unterscheiden, das Ergebnis **false** der Funktion ist also korrekt.

Partielle Korrektheit der Funktion istPraefix

```
boolean istPraefix (IntSequenz a, IntSequenz s) {
  return isEmpty(a) ? true
         : isEmpty(s) ? false
           : first(a) != first(s) ? false
             : istPraefix (rest(a), rest(s)) ;  }
```

- **2. Fall**: der Parameter **a** ist von der Form **stock(z,r)** (Induktionsschritt).
  Dann gilt die Induktionsannahme: **istPraefix(r,t)** liefert für alle Sequenzen **t** das korrekte Ergebnis.

  Nun unterscheiden wir für den zweite Parameter **s** wieder zwei Fälle:
  - **Fall 2.2**: **s** ist von der Form **stock(x,y)**:
    - Dann ist **a** sicher kein Präfix von **s**, falls sich die ersten Elemente **z** und **x** unterscheiden, das Ergebnis **false** der Funktion ist also korrekt.
    - Falls jedoch **z** und **x** identisch sind, dann ist **stock(z,r)** genau dann ein Präfix von **stock(x,y)**, falls **r** ein Präfix von **y** ist.

      Nach der Induktionsannahme liefert für diesen Fall die Funktion **istPraefix**, das korrekte Ergebnis.
- Die Funktion **istPraefix** ist also auch in diesem Fall partiell korrekt!

Beispiel: die Funktion istTeilsequenz in Java

- Die Funktion istTeilsequenz in Java:

```
boolean istTeilsequenz (IntSequenz t, IntSequenz s) {
  return isEmpty(t) ? true
         : isEmpty(s) ? false
           : istPraefix(t,s) ? true
             : istTeilsequenz(t, rest(s)) ;
}
```

- terminiert die Funktion istTeilsequenz?
  - Antwort: Ja
  - Abstiegsfunktion: h(t, s) = laenge(s)
- ist die Funktion istTeilsequenz partiell korrekt?
  - Offensichtlich ja, wegen Entwicklung analog zur Struktur von s.
    (Beweis analog zu vorher mittels struktureller Induktion über s)

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- ✓ Rekursive Datentypen
- ❖ Einbettung
- ❖ Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- ✓ Rekursive Datentypen
- ➢ Einbettung
- ❖ Pattern Matching

Beispiel: Suche nach der minimalen Zahl in einer Sequenz

- ❖ Gesucht: eine Funktion
  `int minimum(IntSequenz s)`
  die das Minimum der Zahlen in einer nicht leeren Sequenz s berechnet.
- ❖ Idee:
  - Wir merken uns das erste Element als „vorläufiges“ Minimum;
  - dann untersuchen wir rekursiv den Rest der Sequenz: falls wir eine kleinere Zahl finden, machen wir mit dieser weiter.
- ❖ Problem:
  - Wie können wir uns eine Zahl „merken“?
    (Die funktionale Programmierung kennt keine Variablen!)
- ❖ Lösung:
  - Wir „betten“ die Funktion in eine allgemeinere Funktion ein, die einen zusätzlichen Parameter bekommt.
  - In diesem zusätzlichen Parameter „merken“ wir uns das bisher gefundene Minimum.
  - Der zusätzliche Parameter dient also als Ersatz für eine Variable.

Das Prinzip der Einbettung

- ❖ Die Funktion `int minimum(IntSequenz s)`
  wird in eine allgemeinere Funktion (d.h. in eine Funktion mit einem zusätzlichen Parameter) eingebettet:
  `int minimumBett(IntSequenz s, int altesMin)`
- ❖ Die allgemeinere Funktion **`minimumBett`** hat einen zusätzlichen Parameter **`altesMin`**, der das bisher gefundenen Minimum enthält.
- ❖ Die Funktion **`minimumBett`** liefert als Ergebnis das Minimum von **`s`**, falls dieses kleiner ist als das bereits gefundene Minimum (**`altesMin`**), sonst liefert sie das bereits gefundene Minimum.
- ❖ Der Rumpf von **`minimum`** besteht dann nur noch aus einem Aufruf von **`minimumBett`** mit **`first(s)`** als vorläufigem Minimum:

```
public int minimum (IntSequenz s) {
  return minimumBett(rest(s), first(s));
}
```

Das Prinzip der Einbettung: die Funktion minimumBett

- Die Funktion minimumBett in Java:

`minimumBett` ist nur Hilfsfunktion für `minimum`.

```
private int minimumBett (IntSequenz s, int altesMin) {
   return  isEmpty(s) ? altesMin
           : first(s) < altesMin
             ? minimumBett (rest(s), first(s))
             : minimumBett (rest(s), altesMin);
}
```

Zusätzlicher Parameter „merkt“ sich vorläufiges Minimum.

- terminiert die Funktion minimumBett?
  - Antwort: Ja; Abstiegsfunktion: h(s, altesMin) = laenge(s)
- ist die Funktion minimumBett partiell korrekt?
  - Offensichtlich ja, wegen Entwicklung analog zur Struktur von s (Beweis über strukturelle Induktion über s).

Die Funktion minimum ohne Einbettung

- Minimum kann auch direkt ohne Einbettung berechnet werden:

```
int minimum(IntSequenz s) {
  return isEmpty(rest(s)) ? first(s)
         : first(s) < minimum(rest(s))? first(s)
           : minimum(rest(s)) ;
}
```

- terminiert die Funktion minimum?
  - Antwort: Ja, Abstiegsfunktion: h(s) = laenge(s)
- ist die Funktion minimum partiell korrekt?
  - Zur eigenen Übung
- Hinweis: den doppelten Aufruf von `minimum(rest(s))` kann man vermeiden, wenn man eine Hilfsfunktion `min` verwendet:

```
public int min (int a, int b) { return a<b ? a : b ; }
public int minimum(IntSequenz s) {
  return isEmpty(rest(s))
           ? first(s)
           : min(first(s), minimum(rest(s))); }
```


Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- ✓ Rekursive Datentypen
- ✓ Einbettung
- Pattern Matching

Wo stehen wir?

- ✓ Syntax von Ausdrücken
- ✓ Auswertung von Ausdrücken
- ✓ Beispiele von rekursiven Funktionen
- ✓ Arten von Rekursionen
- ✓ Terminierung von Funktionen
- ✓ Korrektheit von Funktionen
- ✓ Das Verhältnis zwischen Induktion und Rekursion
- ✓ Rekursive Datentypen
- ✓ Einbettung
- ➢ Pattern Matching

Parameterübergabe mittels „Pattern Matching“

- Bei vielen funktionalen Programmiersprachen gibt es zusätzlich zur „Call by Value“-Parameterübergabe auch die Parameterübergabe mittels „Pattern Matching“.
- Der Funktionsrumpf besteht dann nicht aus einem einzigen Ausdruck sondern aus mehreren möglichen Varianten von Ausdrücken.
- Vor jedem der Ausdrucksvarianten steht ein weiterer Ausdruck als „Muster“:
  - Wenn die aktuellen Parameter (d.h. die Ausdrücke, mit denen die Funktion aufgerufen wird) „auf das Muster passen“, wird die entsprechende Ausdrucksvariante ausgewählt und ausgewertet.
  - Der Muster-Ausdruck (*pattern*) kann selbst neue formale Parameter enthalten: Bei der Parameterübergabe (*matching*) werden diese mit den „passenden“ Teilausdrücken des aktuellen Parameters besetzt (jetzt per *call by value*).
- In Java gibt es die Parameterübergabe mittels „Pattern Matching“ nicht. Wir verwenden für die folgenden Beispiele deshalb OCaml.

„Pattern Matching“ in OCaml

- Eine rekursive Funktion g mit Parameter n bei „normaler“ Parameterübergabe in OCaml:

```
let rec g n =
   <Ausdruck für Funktionsrumpf> ;;
```

- Eine rekursive Funktion f mit Parameter n bei Parameterübergabe mittels „Pattern Matching“ mit 4 Mustern in OCaml:

```
let rec f n =
   match n with
     <Muster1> -> <AusdrucksVariante1>
   | <Muster2> -> <AusdrucksVariante2>
   | <Muster3> -> <AusdrucksVariante3>
   | <Muster4> -> <AusdrucksVariante4> ;;
```

Der Auswahl-Strich | trennt die 4 Varianten.

Die Muster werden von oben nach unten untersucht; das erste passende wird ausgewählt.

Die Fibonacci-Zahlen mit „Pattern Matching“ in OCaml

- Eine Version der Funktion fib mit „normaler“ Parameterübergabe in OCaml:

```
let rec fib n =
   if n=0 then 0
          else if n=1 then 1
                      else fib (n-1) + fib (n-2) ;;
```

- Die Funktion fib mit „Pattern Matching“in OCaml:

```
let rec fib n =
   match n with
     0 -> 0
   | 1 -> 1
   | m -> fib (m-1) + fib (m-2) ;;
```

Beim Aufruf **fib 0** wird das „passende“ Muster gefunden und die erste Variante gewählt

Analog beim Aufruf **fib 1**

Beim Aufruf **fib 5** wird der neue formale Parameter **m** mit **5** besetzt.

„Pattern Matching“ bei Sequenzen als Parameter

- Den rekursiven Typ für Sequenzen ganzer Zahlen kann man in Ocaml folgendermaßen definieren:

```
type intSequenz = Create | Stock of (int * intSequenz)
```

- Die Funktion laenge mit „Pattern Matching“in OCaml:

```
let rec laenge s =
   match s with
     Create -> 0
   | Stock(z,r) -> 1 + laenge r ;;
```

Die neuen formalen Parameter z und r werden mit dem ersten Element bzw. mit dem Rest der Sequenz besetzt.

z wird mit 1 „gematched“.

r wird mit Stock(2, Stock(3, Create)) „gematched“.

- Ein Aufruf von laenge:

```
laenge Stock(1, Stock(2, Stock(3, Create)));;
```

Zusammenfassung

- Ein funktionales Programm besteht im Wesentlichen aus einer Reihe von Funktionsdeklarationen und einem Ausdruck, der Funktionsaufrufe enthält.
  - Zentrales Konzept ist dabei die Auswertung von Ausdrücken und dabei insbesondere die Auswertung von Funktionsaufrufen (***Funktionsapplikation***).
- ***Rekursion*** ist eine wichtige Programmiertechnik:
  - Rekursion erlaubt die Skalierbarkeit von Datenstrukturen und Programmen, d.h. ihre Anwendung auf Probleme, deren Größe nicht von vornherein feststeht.
  - Rekursion ist besonders dann zu empfehlen, wenn das Problem rekursiv ist, d.h. sich auf selbstähnliche kleinere Probleme zurückführen lässt.
- ***Korrekheit:*** Eine Funktion ist für gegebene Eingabewerte korrekt, wenn sie terminiert und partiell korrekt ist.
  - ***Terminierung***: es nicht zu einer unendlichen Folge rekursiver Funktionsaufrufe kommt
  - ***Partielle Korrektheit***: Die Funktion liefert das richtige Ergebnis wenn sie terminiert.
- Beweistechnik für die Terminierung ist die ***Abstiegsfunktion***.
- Beweistechnik für die partielle Korrektheit ist ***Induktion***.
- Pattern Matching ist in funktionalen Sprachen eine zusätzliche Möglichkeit der Parameterübergabe, die if-then-else-Verschachtelungen vermeidet und zusätzliche, auf spezielle Ausdrücke passende Parameter einführt.