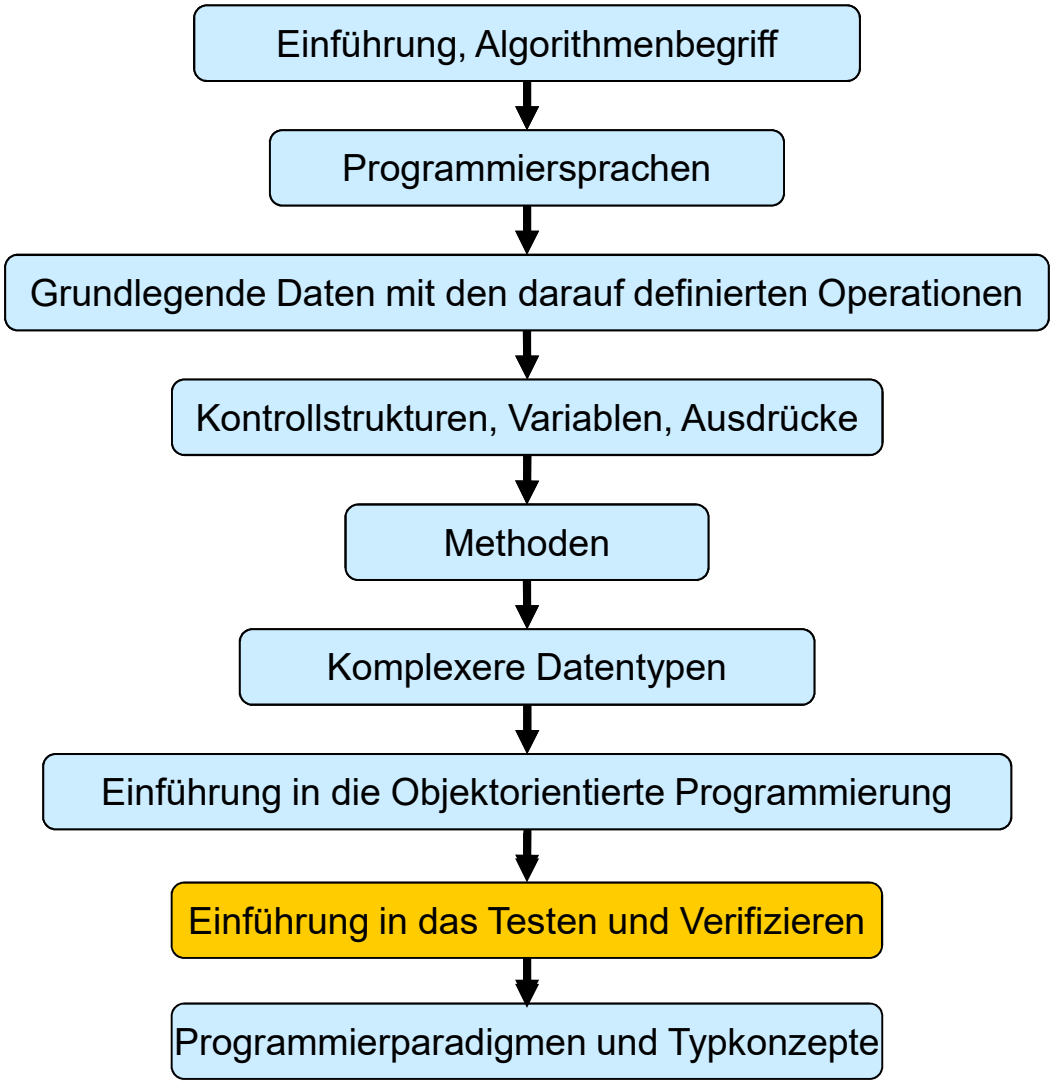
Inhalt dieser Veranstaltung



Mögliche Fehlerquellen in der Programmierphase

- Syntaktischer Fehler
  - Beispiel: `(3+)4`
  - werden **alle** (!) vom Compiler über die Sprachgrammatik erkannt
- Semantischer Fehler
  - Beispiel: `int[] a = {1,2,3};  int x = a[3];`
  - können zu einem geringen Teil vom Compiler/Werkzeugen erkannt werden
- Logischer Fehler
  - Beispiel:

```
double radius = 3;
// berechne Kreisumfang
double kreisumfang = Math.PI * radius;
```

  - snytaktisch und semantisch korrekt, ist aber nicht das, was man eigentlich ausdrücken wollte
  - kaum durch Werkzeuge erkennbar

Testen

- Frage: arbeitet ein entwickeltes Programm entsprechend der Spezifikation?

- Antwort 1: Teste systematisch alle möglichen Fälle (oder eine sinnvolle Teilmenge möglicher Fälle)
- In beiden nachfolgenden Beispielen: Eingabe ist eine ganze Zahl aus $[0,2^{31}-1]$
- Beispiel 1: Behauptung: `b==2a`

```
int a = Integer.parseInt(args[0]);
long b = 0;
for(int i=0; i<a; i++) b += 2;
```

Beispiel 2: Behauptung: `c==???`

```
int a = Integer.parseInt(args[0]);
int b = Integer.parseInt(args[1]);
long c = 0;
for(int i=0; i<a; i++)
  for(int j=0; j<b; j++) c++;
```

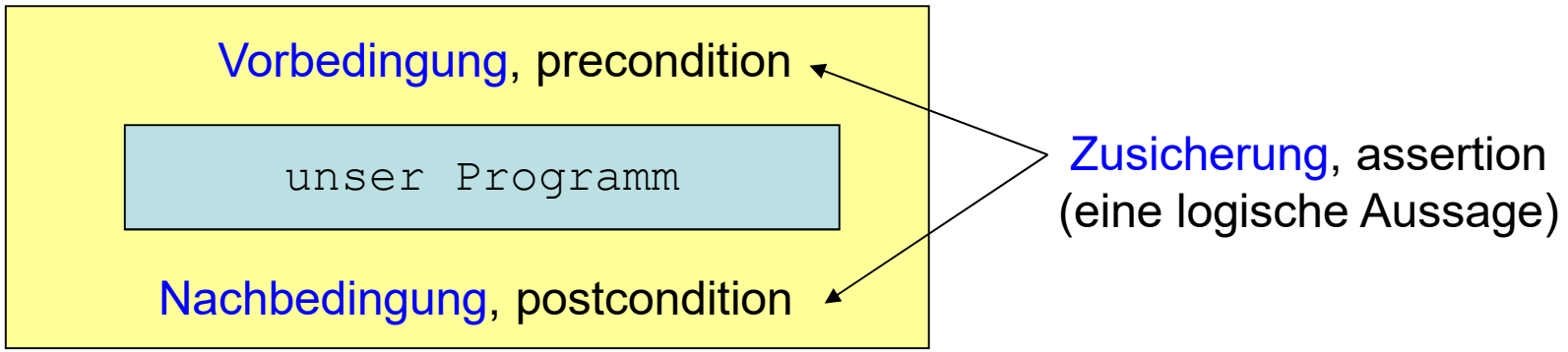
- Mögliche Fälle Beispiel 1: 2^{31}= 2.147.483.648
- Mögliche Fälle Beispiel 2: $(2^{31})^2$ = ca. $4{,}6 \cdot 10^{18}$
- Was ist zum Beispiel, wenn als Eingabe `float` statt `int` erlaubt wäre?

- Durch Testen kann man zwar das Vorkommen, nicht aber das Fehlen von Fehlern (Korrektheit) nachweisen!

Testen ist gut, formal beweisen ist besser

- Wünschenswert: formaler Beweis, dass ein Programm / Programmteil gewisse Eigenschaften garantiert besitzt
- Deshalb Antwort 2: beweise **allgemein** die Korrektheit für alle möglichen Fälle

- Verfolgter Ansatz:
  - Das Ein-/Ausgabeverhalten eines Programms wird durch logische Formeln beschrieben (Vorbedingung / Nachbedingung)
  - Das Verhalten komplexer Programme wird auf das Verhalten der Basiskonstrukte zurückgeführt, indem man den syntaktischen Aufbau der Sprache berücksichtigt (strukturelle Induktion).

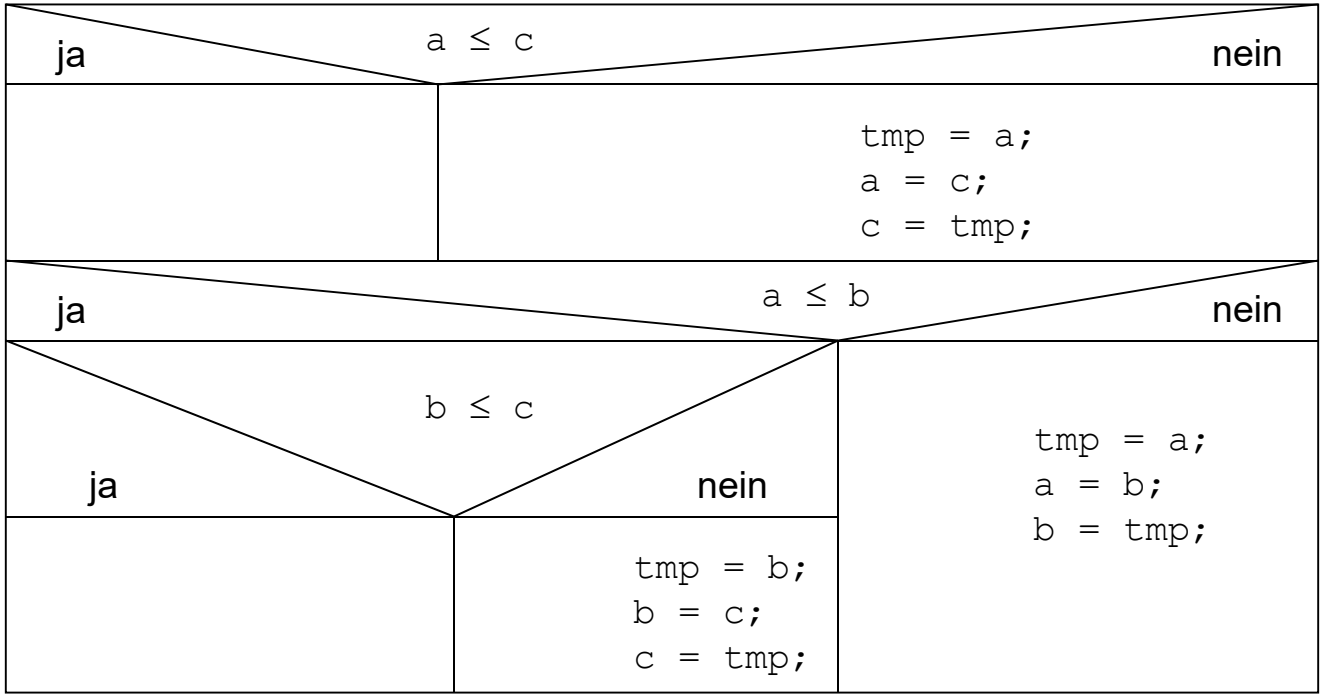


Beispiel

- Aufgabenstellung:
  Es sollen die Werte dreier Variablen `a`, `b` und `c` so vertauscht werden, dass anschließend gilt: $a \leq b \leq c$.

- Lösung:
  Vergleiche jeweils zwei Werte direkt miteinander und vertausche sie gegebenenfalls mit Hilfe einer temporären Variablen.

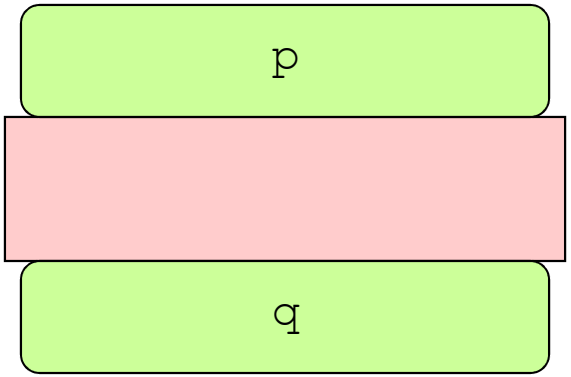
Struktogramm zur Bestimmung von a ≤ b ≤ c



Zusicherungen in Struktogrammen

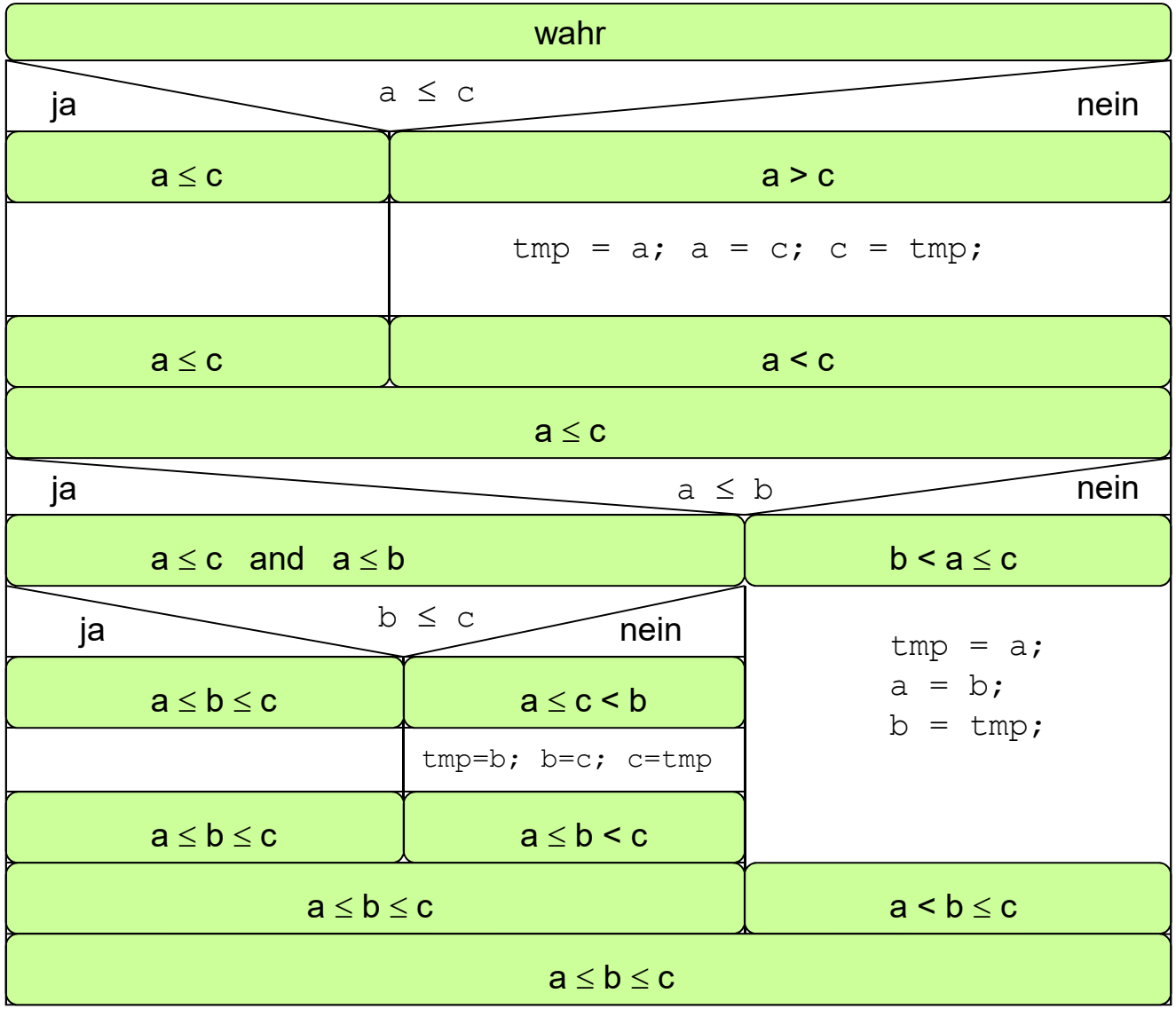
Für ein beliebiges Struktogramm der Form

und Vor- und Nachbedingung `p` bzw. `q` lässt sich die Wirkung des Programms darstellen durch:



d.h. in grafischer Form durch Vierecke mit abgerundeten Ecken.

Struktogramm mit Zusicherungen



Zwischenstand

- Testen kann existierende Fehler aufzeigen, aber nicht grundsätzlich ausschließen
- Ein Ansatz zum Beweisen von Programmeigenschaften geht über den Beweis des Verhaltens eines Programms/Programmteils: wenn bestimmte Vorbedingungen gelten, kann man gewisse Nachbedingungen aufgrund der Programmeigenschaften garantieren

Reflektion

- Was ist die Ausgabe des Programms bei einem Übergabewert $n \in N$?

```
public class Test {
    public static void main(String[] args){
        int n = Integer.parseInt(args[0]);
        System.out.println(n*n);
}}
```

Hoare-Kalkül

- Allgemein wird die Korrektheit eines Programms S ausgedrückt durch eine Korrektheitsformel der Form {p} S {q}, die das Ein-/Ausgabeverhalten des Programms S spezifiziert.
- p wird Vorbedingung genannt, q Nachbedingung oder Zusicherung.

- Eine Korrektheitsformel {p} S {q} heißt partiell korrekt, wenn jede terminierende Berechnung von S, die in einem p-Zustand beginnt, in einem q-Zustand endet.

- Eine Korrektheitsformel {p} S {q} heißt total korrekt, wenn jede Berechnung von S, die in einem p-Zustand beginnt, **terminiert** und ihr Endzustand q erfüllt.

- Unterschied partiell korrekt / total korrekt:
  Im Falle der partiellen Korrektheit wird über nicht-terminierende Berechnungen von S keine Aussagen gemacht.

Hoare-Kalkül

- Das Hoare-Kalkül ist ein System zum Nachweis der partiellen Korrektheit deterministischer Programme
- Eine partielle Programmkorrektheit lässt sich damit induktiv über den syntaktischen Aufbau eines Programms beweisen
- Das System besteht aus Regeln, die die Form haben:

$$\frac{\text{Prämissen}}{\text{Konklusion}}$$

- Prämissen sind Korrektheitsformeln oder Zusicherungen
- Eine Konklusion ist eine Korrektheitsformel

- Die Bedeutung solcher Regeln ist, dass man aus der Gültigkeit der Prämissen auf die Gültigkeit der Konklusion schließen kann

- Axiome sind Regeln ohne Prämissen

Einfache Beispielprogrammiersprache

- Konstanten und Variablen (ohne Deklaration):
- Beispiel: `x, y, 4711, 0`

- Leere Anweisung: **`skip`**
- Die leere Anweisung macht nichts.

- Zuweisung: `u = t`
- Der Variablen `u` wird der Wert des Ausdrucks `t` zugewiesen

- Sequenz: S_1 `;` S_2
- Wenn S_1 und S_2 beliebige Konstrukte der Programmiersprache sind, so werden diese durch die Sequenz hintereinander ausgeführt.

- Selektion: **`if`** `Bedingung` **`then`** S_1 **`else`** S_2 **`end`**
- Falls die Bedingung wahr ist, wird S_1 ausgeführt, ansonsten S_2. S_1 und S_2 können wieder beliebige Konstrukte der Programmiersprache sein.

- Iteration: **`while`** `Bedingung` **`do`** `S` **`end`**
- Solange `Bedingung` erfüllt ist, führe `S` aus. `S` kann eine beliebige Anweisung sein.

Beispielprogramm

- Aufgabenstellung: Berechne den ganzzahligen Anteil und den Rest der Division zweier natürlicher Zahlen `x` und `y`

- Lösung als Programm:

```
quotient = 0 ;
rest = x ;
while rest ≥ y do
      rest = rest - y ;
      quotient = quotient + 1
end
```

- Wir werden später beweisen, dass dieses Programm korrekt den ganzzahligen Anteil (in `quotient`) und den Rest (in `rest`) bei der Division zweier natürlicher Zahlen `x` und `y` $\neq 0$ berechnet

Axiom 1: Leere Anweisung

Leere Anweisung: **skip**

Regel:

$$\frac{\{\ \}}{\{p\}\ \textbf{skip}\ \{p\}}$$

Da diese Regel ein Axiom ist, ist die Prämisse leer und damit diese Regel immer anwendbar.

Dieses Axiom besagt, dass durch die Ausführung der skip-Anweisung die Aussage p der Vorbedingung ebenfalls als Nachbedingung gültig ist.

Beispiel

Regel:

$$\frac{\{\,\}}{\{p\}\ \texttt{skip}\ \{p\}}$$

- Behauptung: {x>0} `skip` {x>0}
- Beweis:
  Direkte Anwendung der skip-Regel mit p ≡ x>0 liefert die Aussage.
  Also ist die Gesamtaussage {x>0} `skip` {x>0} bewiesen.

- Hiermit haben wir also bewiesen, dass, wenn vor Anwendung des Programms (der skip-Anweisung) x>0 ist, so gilt nach Ausführung des Programms ebenfalls x>0.

Axiom 2: Zuweisung

Zuweisung: `u = t`

Regel:

$$\frac{\{\}}{\{p\,[u\,/\,t]\}\ \ u = t\ \ \{p\}}$$

wobei p [u / t] bedeutet: Alle Vorkommen von `u` werden in der Bedingung p syntaktisch durch `t` ersetzt. `u` ist eine beliebige Variable der Programmiersprache (z.B. `x`) und `t` ist ein beliebiger Ausdruck der Sprache (z.B. `(3+x)*5`).

Über dieses Axiom lassen sich also Aussagen machen über die Zuweisung eines Wertes (des Wertes des Ausdrucks `t`) an eine Variable (`u`).

Beispiel

Regel:

$$\frac{\{\}}{\{p\,[u\,/\,t]\}\ \ u\ =\ t\ \ \{p\}}$$

- Behauptung: {y<32} `y=y+23` {y<55}
- Beweis:
  - Wende die Zuweisungsregel rückwärts an, indem ausgehend von der Nachbedingung der Aussage die sich durch die Regel ergebende Vorbedingung ermittelt wird

    {y<55 [y/y+23] } `y=y+23` {y<55}
    ⇔ {y+23<55} `y=y+23` {y<55}
    ⇔ {y<32} `y=y+23` {y<55}
  - Die berechnete Vorbedingung dieser Anweisung entspricht der Vorbedingung der Gesamtaussage, also ist die Behauptung bewiesen.
- Hiermit haben wir bewiesen, wenn in y ein beliebiger Wert kleiner als 32 steht, nach Ausführung des Programms ein Wert kleiner als 55 in y steht.

Regel 3: Sequenz

Sequenz: $S_1 \; ; \; S_2$

Regel:

$$\frac{\{p\}\ S_1\ \{q\} \quad , \quad \{q\}\ S_2\ \{r\}}{\{p\}\ S_1\ ;\ S_2\ \{r\}}$$

Diese Regel ist ähnlich einem Transitivitätsgesetz. Wenn die Nachbedingung der ersten Anweisung S_1 gleich der Vorbedingung der zweiten Anweisung S_2 ist, so kann man Aussagen machen über die Hintereinanderausführung der beiden Anweisungen $S_1; \; S_2$.

Beispiel

- Behauptung: {x<1} `x=x+1; x=x+2` {x<4}
- Beweis:
  - Das Programm besteht aus einer Sequenz von zwei Zuweisungen
  - Um die Sequenzregel anwenden zu können (Konklusion), müssen wir erst die Aussagen in der Prämisse beweisen
  - Wir gehen von der Nachbedingung x<4 aus und "schauen", welche Vorbedingung wir für die letzte Zuweisung damit bekommen
  - Damit gehen wir als Nachbedingung in die erste Zuweisung und "schauen", welche Vorbedingung sich damit ergibt
  - Teilbeweis 1 (Zuweisungsregel rückwärts angewandt):
    {x<4 [x/x+2] } `x=x+2` {x<4} ⇔ {x+2<4} `x=x+2` {x<4} ⇔ {x<2} `x=x+2` {x<4}
  - Teilbeweis 2 (Zuweisungsregel rückwärts angewandt):
    {x<2 [x/x+1] } `x=x+1` {x<2} ⇔ {x+1<2} `x=x+1` {x<2} ⇔ {x<1} `x=x+1` {x<2}
  - Damit haben wir folgende Teilaussagen gezeigt (Prämissen der Sequenzregel im konkreten Fall):
    {x<1} `x=x+1` {x<2} und {x<2} `x=x+2` {x<4}
  - Damit können wir die Konklusion der Sequenzregel anwenden und erhalten die Aussage über das Gesamtprogramm:
    {x<1} `x=x+1; x=x+2` {x<4}

Regel 4: Selektion

Selektion: **if** Bedingung **then** S_1 **else** S_2 **end**

Regel:

$$\frac{\{p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}\ \{q\}}$$

Liegt eine gemeinsame Vorbedingung p und eine gemeinsame Nachbedingung q vor, so kann man für die Selektion eine Aussage machen.

Beispiel

- Behauptung: {a≠b} `if a≥b then x=a-b else x=b-a end` {x>0}
- Beweis:
  - Um die Selektionsregeln anwenden zu können, müssen wir erst die beiden Teilaussagen der Prämisse zeigen
  - Wir gehen von der Nachbedingung x>0 aus und "schauen", welche Vorbedingung sich für die beiden Anweisungen jeweils ergibt
  - Teilbeweis 1 (Zuweisungsregel rückwärts angewandt):
    {x>0[x/a-b]} `x=a-b` {x>0} ⇔ {a-b>0} `x=a-b` {x>0} ⇔ {a>b} `x=a-b` {x>0}
    Eigentlich zu zeigen: {a≠b∧a≥b} `x=a-b` {x>0}
    Aber aus (a≠b∧a≥b) ⇒ a>b folgt die Teilbehauptung (später mehr dazu).
  - Teilbeweis 2 (Zuweisungsregel rückwärts angewandt):
    {x>0[x/b-a]} `x=b-a` {x>0} ⇔ {b-a>0} `x=b-a` {x>0} ⇔ {b>a} `x=b-a` {x>0}
    Eigentlich zu zeigen: {a≠b∧¬a≥b} `x=a-b` {x>0}
    Aber aus (a≠b∧¬a≥b) ⇒ a<b folgt die Teilbehauptung (später mehr dazu).
  - Damit können wir die Konklusion der Selektionsregel anwenden und erhalten die Aussage über das Gesamtprogramm:
    {a≠b} `if a≥b then x=a-b else x=b-a end` {x>0}

Regel 5: Iteration

Iteration: `while B do S end`

Regel:

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{p \wedge \neg B\}}$$

- Die Bedingung p nennt man auch Schleifeninvariante.
- Die Regel besagt, dass die Schleifeninvariante p nach der Terminierung der Schleife gültig ist, wenn sie nach jeder Ausführung des Schleifenrumpfes `S` gültig ist.

- Praktische Herausforderung: Finden einer geeigneten Schleifeninvariante

- Beispiel zur Regel gleich im großen Beispiel

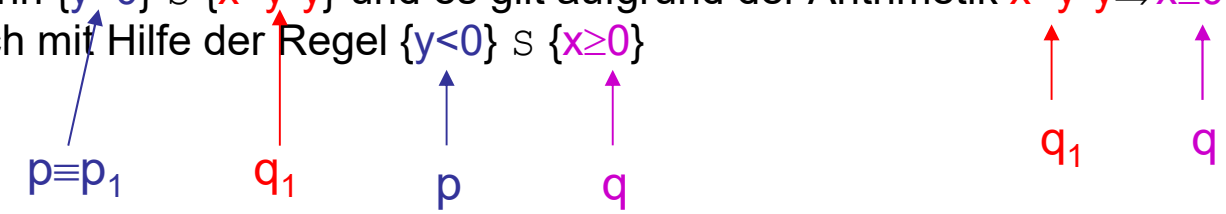
Regel 6: Konsequenzregel

Regel:

$$\frac{p \Rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \Rightarrow q}{\{p\}\ S\ \{q\}}$$

- Manchmal ist es im Beweisgang erforderlich, dass die Vorbedingung verstärkt und/oder die Nachbedingung abgeschwächt werden muss
- Verstärken in Vorwärtsrichtung u.a.: Weglassen Und-Term, Hinzufügen Oder-Term
- Abschwächen in Rückwärtsrichtung: Hinzufügen Und-Term, Weglassen Oder-Term

- Ebenso lassen sich logische Aussagen außerhalb des Hoare-Kalküls hierbei verwenden.
- Beispiel: für alle ganzen Zahlen $x \in Z$ gilt: $x \cdot x \geq 0$. Also kann man mit Hilfe der Konsequenzregel und dieser Aussage der Arithmetik schließen:
  Wenn $\{y<0\}\ S\ \{x=y*y\}$ und es gilt aufgrund der Arithmetik $x=y*y \Rightarrow x \geq 0$, also gilt dann auch mit Hilfe der Regel $\{y<0\}\ S\ \{x \geq 0\}$



Beispiel (von eben aufgegriffen)

Regel:

$$\frac{p \Rightarrow p_1,\ \{p_1\}\ \mathtt{S}\ \{q_1\},\ q_1 \Rightarrow q}{\{p\}\ \mathtt{S}\ \{q\}}$$

- Behauptung: $\{a \neq b\}$ `if a≥b then x=a-b else x=b-a end` $\{x>0\}$
- Beweis:
  - ...
  - Teilbeweis 1 (Zuweisungsregel rückwärts angewandt):
    $\{x>0[x/a-b]\}$ `x=a-b` $\{x>0\} \Leftrightarrow \{a-b>0\}$ `x=a-b` $\{x>0\} \Leftrightarrow \{a>b\}$ `x=a-b` $\{x>0\}$
  - Eigentlich zu zeigen: $\{a \neq b \wedge a \geq b\}$ `x=a-b` $\{x>0\}$
  - Jetzt kommt der Einsatz der Konsequenzregel mit p, p_1, q, q_1 der Prämisse:
    - $p \equiv (a \neq b \wedge a \geq b)$, $p_1 \equiv a>b$, $p \Rightarrow p_1$
    - $q \equiv x>0$, $q_1 \equiv q$, $q \Rightarrow q_1$
  - Es gilt $p \Rightarrow p_1$, $q \Rightarrow q_1$ mit obigen p, p_1, q, q_1. Also können wir die Konklusion ziehen: $\{a \neq b \wedge a \geq b\}$ `x=a-b` $\{x>0\}$
  - und damit ist Teilaussage 1 bewiesen

Zwischenstand

- Der Beweisgang beim Hoare-Kalkül geht immer über die Struktur eines Programms
- Zu jedem Programmkonstrukt gibt es eine Regel (abhängig von der Programmiersprache)
- Axiome sind immer anwendbar
- Die Konklusion von Nicht-Axiom-Regeln ist anwendbar, sobald man die Prämisse dieser Regel bewiesen hat

Reflektion

- Die Anwendung von Axiomen ist einfacher / weniger aufwändig als die von Nicht-Axiom-Regeln. Wieso hat Herr Hoare nicht grundsätzlich nur Axiome definiert?

Zusammenhängender Beweis

Aufgabenstellung: Ganzzahlige Division mit Rest:
In `x` und `y` stehen die Eingabewerte. Das Resultat soll nach der Berechnung in `quotient` und `rest` stehen.

```
quotient = 0;
rest = x;
while rest ≥ y
    rest = rest - y;
    quotient = quotient + 1
end
```

div(x,y)

Die Behauptung, dass obiges Modul `quotient` und `rest` korrekt berechnet, lässt sich wie folgt formulieren:

{ (x≥0) ∧ (y≥0) } `div(x,y)` { (quotient*y+rest=x) ∧ (rest<y) }

Vorbedingung obiges Programm Nachbedingung

Plan zum Beweisgang

Zu zeigen: { (x≥0) ∧ (y≥0) } `div(x,y)` { (quotient*y+rest=x) ∧ (rest<y) }

Wahl einer geeigneten Schleifeninvariante p:

(quotient*y+rest = x) ∧ (rest ≥0)

Dann lässt sich zeigen (das ist unsere Beweisstrategie):

(1) Die Schleifeninvariante p gilt vor der erstmaligen Ausführung der Schleife, d.h. es gilt:

{ (x ≥ 0) ∧ (y ≥ 0) } `quotient:=0; rest:=x` { (quotient*y+rest = x) ∧ (rest ≥0) }

(2) p bleibt bei einmaliger Ausführung des Schleifenrumpfes wahr, d.h. (`rest ≥ y` ist Bedingung in Schleifenkopf):

{ p ∧ (rest ≥ y) } `rest := rest-y ; quotient := quotient+1` { p }

(3) p ∧ ¬ B impliziert die gewünschte Nachbedingung:

((quotient*y+rest = x) ∧ (rest ≥ 0)) ∧ ¬ (rest ≥ y)

Beweis (1)

(1) Die Schleifeninvariante p gilt vor der erstmaligen Ausführung der Schleife, d.h. es gilt:

$\{ (x \geq 0) \wedge (y \geq 0) \}$ `quotient:=0; rest:=x` $\{ (quotient*y+rest = x) \wedge (rest \geq 0) \}$

Beweis:

Vorgehen: Wir nehmen das gewünschte Endresultat und wenden 2x Axiom 2 an (Rückwärtsschließen), die Sequenzregel und anschließend die Konsequenzregel.

Anwendung Axiom 2:

$\{(quotient*y+rest = x)\wedge(rest \geq 0)[rest|x]\}$ rest:=x $\{ (quotient*y+rest = x)\wedge(rest \geq 0) \}$

$\Leftrightarrow \{(quotient*y+x = x)\wedge(x \geq 0)\}$ rest:=x $\{ (quotient*y+rest = x)\wedge(rest \geq 0) \}$

Anwendung Axiom 2:

$\{(quotient*y+x = x)\wedge(x \geq 0)\ [quotient|0] \}$ quotient:=0 $\{(quotient*y+x = x)\wedge(x \geq 0) \}$

$\Leftrightarrow \{(0*y+x = x)\wedge(x \geq 0) \}$ quotient:=0 $\{(quotient*y+x = x)\wedge(x \geq 0) \}$

$\Leftrightarrow \{(x = x)\wedge(x \geq 0) \}$ quotient:=0 $\{(quotient*y+x = x)\wedge(x \geq 0) \}$

$\Leftrightarrow \{ x \geq 0 \}$ quotient:=0 $\{(quotient*y+x = x)\wedge(x \geq 0) \}$

Anwendung Sequenzregel:

$\{ x \geq 0 \}$ quotient:=0; rest:=x $\{ (quotient*y+rest = x)\wedge(rest \geq 0) \}$

Anwendung Konsequenzregel (mit $((x \geq 0)\wedge(y \geq 0)) \Rightarrow (x \geq 0)$)

$\{(x \geq 0) \wedge (y \geq 0)\}$ quotient:=0; rest:=x $\{ (quotient*y+rest = x)\wedge(rest \geq 0) \}$

Beweis (2)

(2) p bleibt bei einmaliger Ausführung des Schleifenrumpfes wahr, d.h.:

$\{ p \wedge (rest \geq y) \}$ rest := rest-y ; quotient := quotient+1 $\{ p \}$

Beweis:

Vorgehen: Mit Endresultat 2x Axiom 2 (rückwärts), Sequenz- + Konsequenzregel.

Anwendung Axiom 2:

$\{(quotient*y+rest=x)\wedge(rest\geq 0)[quotient|quotient+1]\}$ quotient := quotient+1 $\{(quotient*y+rest=x)\wedge(rest\geq 0)\}$

$\Leftrightarrow \{((quotient+1)*y+rest=x)\wedge(rest\geq 0)\}$ quotient := quotient+1 $\{(quotient*y+rest=x)\wedge(rest\geq 0)\}$

Anwendung Axiom 2:

$\{((quotient+1)*y+rest=x)\wedge(rest\geq 0)[rest|rest-y]\}$ rest := rest-y $\{((quotient+1)*y+rest=x)\wedge(rest\geq 0)\}$

$\Leftrightarrow \{((quotient+1)*y+(rest-y)=x)\wedge(rest-y\geq 0)\}$ rest := rest-y $\{((quotient+1)*y+rest=x)\wedge(rest\geq 0)\}$

$\Leftrightarrow \{(quotient*y+y+rest-y=x)\wedge(rest-y\geq 0)\}$ rest := rest-y $\{((quotient+1)*y+rest=x)\wedge(rest\geq 0)\}$

$\Leftrightarrow \{(quotient*y+rest=x)\wedge(rest-y\geq 0)\}$ rest := rest-y $\{((quotient+1)*y+rest=x)\wedge(rest\geq 0)\}$

$\Leftrightarrow \{(quotient*y+rest=x)\wedge(rest\geq y)\}$ rest := rest-y $\{((quotient+1)*y+rest=x)\wedge(rest\geq 0)\}$

Sequenzregel:

$\{(quotient*y+rest=x)\wedge(rest\geq y)\}$ rest:=rest-y; quotient:=quotient+1 $\{(quotient*y+rest=x)\wedge(rest\geq 0)\}$

Konsequ.regel (mit $(quotient*y+rest=x)\wedge(rest\geq 0)\wedge(rest\geq y) \Rightarrow (quotient*y+rest=x)\wedge(rest\geq y)$):

$\{((quotient*y+rest=x)\wedge(rest\geq 0)\wedge(rest\geq y)\}$ rest := rest-y ; quotient := quotient+1 $\{(quotient*y+rest=x)\wedge(rest\geq 0)\}$

Beweis (3)

(3) $p \wedge \neg B$ impliziert die gewünschte Nachbedingung:

$$((quotient*y+rest = x) \wedge (rest \geq 0)) \wedge \neg (rest \geq y)$$

Lässt sich umformen zu

$$(quotient*y+rest = x) \wedge (0 \leq rest < y)$$

Bemerkungen

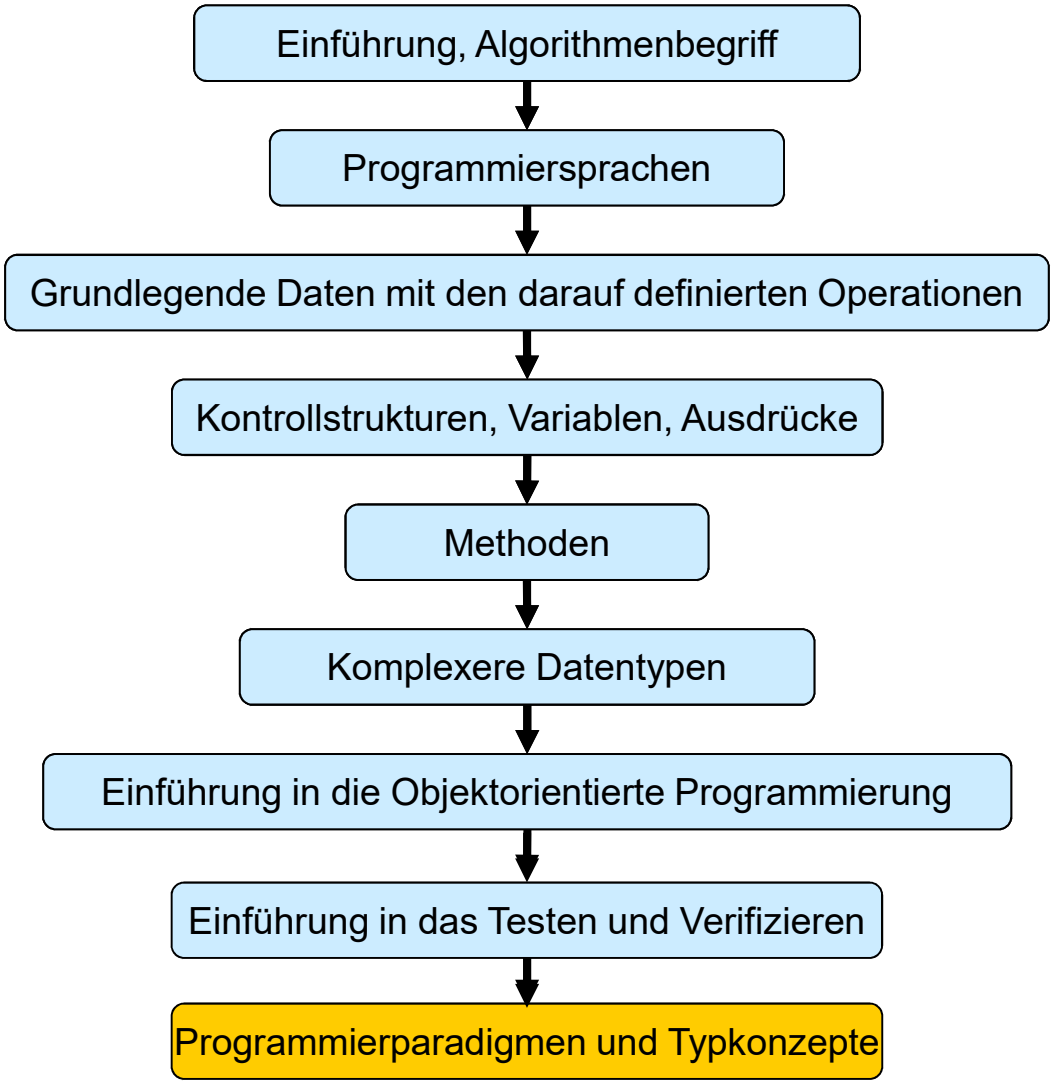
- Der Beweis zeigt die partielle, aber nicht die totale Korrektheit. Das Programm terminiert nicht, wenn man auch y=0 gestatten würde.

- Die totale Korrektheit, dass das Programm auch zusätzlich terminiert, muss getrennt nachgewiesen werden (nicht hier).

Zusammenfassung

- Durch Testen kann man zwar das Vorkommen, nicht aber das Fehlen von Fehlern (Korrektheit) nachweisen
- Kalkül von Hoare ist ein Ansatz zum formalen Beweis gewisser Eigenschaften eines Programms
- Beweisgang geht immer über die Struktur eines Programms (strukturelle Induktion)

Inhalt dieser Veranstaltung



Programmierparadigmen

- Es gibt viele verschiedene grundsätzliche Ansätze zur Formulierung einer Programmierlösung

- Beispiele bis jetzt:
  - "Java ohne Objekte" (nur Variablen, Kontrollstrukturen, statische Methoden)
  - Objektorientierter Programmierstil

- Einen grundsätzlichen Programmierstil nennt man ein Programmierparadigma.

- Verbreitete Programmierparadigmen sind u.a.:
  - Imperatives Paradigma
  - Objektorientiertes Paradigma
  - Funktionales Paradigma
  - Logikorientiertes Paradigma

Imperatives Programmierparadigma

- Tritt in einem Programmiermodell der Speicher und die Zustandstransformation des Speichers durch eine Folge von Befehlen (gemäß dem Kontrollfluss) in den Vordergrund, so spricht man von imperativer Programmierung (imperare: lat. befehlen, herrschen).

- Wesentliche Grundbausteine in imperativen Programmiersprachen sind das Variablenkonzept und die Zuweisung, wodurch ein globaler Zustand (Inhalt des Hauptspeichers) verändert wird:

```
Variable = Ausdruck;
```

- Programmiersprachen dazu:
  Java (1. Teil), C, C++, Pascal, Basic, Fortran,...

Objektorientiertes Programmierparadigma

- Im objektorientierten Ansatz wird ein System als eine Menge miteinander kooperierender und kommunizierender Objekte aufgefasst, die untereinander Nachrichten austauschen (üblicherweise konkret realisiert über Methodenaufrufe).

- Hierbei erfolgt eine Aufteilung des Problems also nicht mehr nach dem Kontrollfluss (funktionale Aufteilung, wie sie bei der Schrittweisen Verfeinerung oft entsteht), sondern die Datenobjekte des Problems und ihr Zusammenwirken bilden die Grundlage für einen Lösungsansatz.

- Das Denken in und entwickeln mit Objekten kommt der zu modellierenden Realität meist näher. So lassen sich einfache wie komplexe Systeme in einem Software-System abbilden und die Strukturen sind ähnlich der Realität (gleiche Sicht wie Anwender).

- Klassen entsprechen dabei auch einem Dienstleistungsgedanken (bieten Methoden für andere an).

Funktionales Programmierparadigma

- Im funktionalen Berechnungsmodell ist wichtig, was berechnet wird, nicht wie dies konkret auf einem Rechner ausgeführt wird. Aus Grundfunktionen kann man über funktionale Programmiermuster (z.B. Einsetzungsschema, Primitive Rekursion, Bedingte Iteration) weitere Funktionen definieren.

- In dem rein funktionalen Ansatz gibt es entsprechend dem funktionalen Berechnungsmodell nur Funktionen und Anwendungen von Funktionen.

- Es gibt im Gegensatz zum imperativen Ansatz im funktionalen Ansatz **keine**:
  - Variablen (außer formale Funktionsparameter)
  - Speicher
  - Zuweisung
  - Zustandstransformationen eines globalen Speichers

- Im funktionalen Ansatz ist eine Programmausführung im Grunde die Auswertung der Funktion main() mit aktuellen Argumenten.

Beispiel

- ML ist eine funktionale Sprache (die allerdings auch imperative Anteile hat).
- Die Fibonacci-Funktion

$$fib(n) = \begin{cases} 1 & \text{falls } n=0 \vee n=1 \\ fib(n-2) + fib(n-1) & \text{sonst} \end{cases}$$

- würde in ML notiert als:

```
> fun fib n =  if n = 0 then 1
               else if n = 1 then 1
               else fib(n-2) + fib(n-1);
val fib = fn: int -> int
>
```

Funktionen sind 1.Klasse

- Funktionen sind dabei Objekte "1. Klasse". Eine Funktion kann z.B. selbst Argument einer Funktion sein oder ihr Ergebniswert.

- Beispiel: map-Funktion in ML

```
> fun map f [] = []
= | map f (h::t) = (f h)::(map f t);
val map = fn : ('a -> 'b) -> 'a list -> 'b list
>
```

- Anwenden der map-Funktion:

```
> map (fn x=> x+1) [1,2,3];
val it = [2,3,4] : int list
>
```

Logikorientierter Ansatz

- Im logikorientierten Ansatz betrachtet man:
  - Eine Menge von Fakten (Datenbasis)
  - Eine Menge von allgemeinen logischen Regeln (Implikationen)
  - Anfragen, die das System mit Hilfe des Datenbasis und den Regeln beantworten soll.

- Eine Programmiersprache, die diesem Ansatz folgt, ist Prolog.
- Fakten werden in Prolog in Form von Relationen zwischen Objekten notiert.
- Eine Relation der Form

  $\texttt{relation}(x_1,\ldots,x_n)$

- bedeutet, dass $x_1,\ldots,x_n$ in dieser Relation stehen.
- Zur Schaffung der Faktenbasis gibt man bekannte Relationen zwischen Objekten an.

Beispiel (1)

- Es sollen verwandtschaftliche Beziehungen zwischen Personen untersucht werden. Zum Aufbau der Datenbasis werden zwei Relationenpaare definiert:
  - `maennlich(x)` bzw. `weiblich(x)` gibt an, dass ein Objekt `x` männlich bzw. weiblich ist.
  - `sohn(x,y)` bzw. `tochter(x,y)` gibt an, dass `x` Sohn bzw. Tochter von `y` ist.

- Angabe der Datenbasis:

```
maennlich(karl).
maennlich(peter).
weiblich(karin).
weiblich(susanne).
sohn(peter, karl).
sohn(peter, karin).
tochter(susanne, karl).
tochter(susanne, karin).
```

Beispiel (2)

- Allgemein gültige Regeln werden in Prolog als Wenn-Dann-Regeln (Horn-Klauseln) formuliert. Statt

  $x_1 \wedge x_2 \wedge \ldots \wedge x_n \Rightarrow y$

  schreibt man in Prolog:

  y :- x_1, x_2,..., x_n

- (Allgemeingültige) Regeln für unser Beispiel:

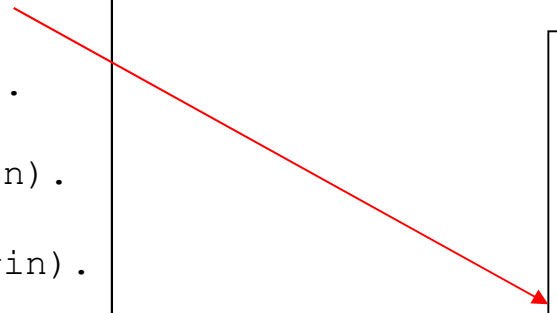
```
vater(X,Y)        :- maennlich(X), sohn(Y,X).
vater(X,Y)        :- maennlich(X), tochter(Y,X).
mutter(X,Y)       :- weiblich(X), sohn(Y,X).
mutter(X,Y)       :- weiblich(X), tochter(Y,X).
kind(X,Y)         :- sohn(X,Y).
kind(X,Y)         :- tochter(X,Y).
ehefrau(X,Y)      :- weiblich(X), maennlich(Y), (kind(Z,X), kind(Z,Y)).
ehemann(X,Y)      :- maennlich(X), weiblich(Y), (kind(Z,X), kind(Z,Y)).
```

- Eine Komma auf der rechten Seite bedeutet eine Und-Verknüpfung.
- Oder-Verknüpfungen schreibt man als Regeln untereinander.
- Fakten werden streng von oben nach unten versucht anzuwenden.

Beispiel (3)

- Nach Eingabe der Fakten und Regeln kann man nun Anfragen an das Prolog-System stellen:

```
?- sohn(peter, karl).
ja
?- vater(karl, peter).
ja
?- ehemann(karl, karin).
ja
?- ehemann(peter, karin).
nein
?- ehemann(karl,X).
karin
```



Fakten:

```
1.  maennlich(karl).
2.  maennlich(peter).
3.  weiblich(karin).
4.  weiblich(susanne).
5.  sohn(peter, karl).
6.  sohn(peter, karin).
7.  tochter(susanne, karl).
8.  tochter(susanne, karin).
```

Regeln:

```
1.  vater(X,Y)    :- maennlich(X), sohn(Y,X).
2.  vater(X,Y)    :- maennlich(X), tochter(Y,X).
3.  mutter(X,Y)   :- weiblich(X), sohn(Y,X).
4.  mutter(X,Y)   :- weiblich(X), tochter(Y,X).
5.  kind(X,Y)     :- sohn(X,Y).
6.  kind(X,Y)     :- tochter(X,Y).
7.  ehefrau(X,Y)  :- weiblich(X), maennlich(Y), (kind(Z,X), kind(Z,Y)).
8.  ehemann(X,Y)  :- maennlich(X), weiblich(Y), (kind(Z,X), kind(Z,Y)).
```

Zwischenstand

- Ein Programmierparadigma ist ein grundsätzlicher Programmierstil
- Java unterstützt i.W. das imperative und objektorientierte Paradigma
- Weitere sind u.a. das funktionale und logikorientierte Paradigma

Reflektion

- Kann man in Java eine map-Funktionen mit den Mitteln programmieren, die wir bis jetzt kennen?

Typkonzepte in Programmiersprachen

- Wiederholung: Mit einem Typ in einer Programmiersprache wird festgelegt:
  - Wertemenge
  - Operationen auf den Werten
  - Darstellung von Werten (Anzahl und Kodierung von Bits)

- Bis jetzt kennen wir in Java: jede Variable muss deklariert werden. In einer Deklaration wird u.a. der zugehörige Typ angegeben.

- Aus einer Deklaration kann ein Compiler
  - viele wichtige Sachen ableiten (Gültigkeitsbereich, Sichtbarkeit, Lebensdauer, Speicherbedarf, Interpretation der Bits, mögliche Operatoren, Interpretation/Überladen von Operatoren wie +,…)
  - und gewisse Fehler erkennen

Beispiel

- Überladen von Operatoren (und analog Funktionen)

```
int i1=1, i2=1, i3=1;
double d1=1, d2=1, d3=1;

i1 = i2 * i3;          // int-Multiplikation
                       // Bytecode imul wird generiert

d1 = d2 * d3;          // double-Multiplikation
                       // Bytecode dmul wird generiert
```

- Erkennen von Fehlern

```
String s1, s2, s3 = "hallo";

s1 = 2 + s3;    // OK. Nach Typanpassung Stringkonkatenation
s2 = 2 * s3;    // Fehler: Operation nicht möglich
```

Statische Typisierung

- Bei einer **statischen Typisierung** **muss** jede Variable in ihrem Gültigkeitsbereich deklariert werden

- Oft muss eine Deklaration einer Variablen vor der Nutzung im Programmcode geschehen (in vielen Programmiersprachen ausschließlich möglich, z.B. C)
  - Beispiel: lokale Variablen einer Java-Methode
  - Aber Gegenbeispiel „Deklaration vor Nutzung“ in Java: Instanz- und Klassenvariablen

- Vorteile einer statischen Typisierung (siehe eben):
  - umfangreiche Möglichkeit der Fehlerprüfung zur Übersetzungszeit
  - automatisches Erkennen der richtigen Version eines überladenen Operators zur Übersetzungszeit
- Nachteile:
  - Prinzipiell lauffähiger Code ist nicht übersetzbar
  - Zusätzlicher Programmcode für eigentlich nicht-genutzten Code

Beispiel zur Fehlerüberprüfung

```
public String isGerade(int number) {

    if(number) {
      String result = "Zahl ist gerade";
    } else {
      String result = "Zahl ist ungerade";
    }

    return result;
}
```

- Ist die Java-Methode so korrekt?
- Was, wann und wieso kann ein Compiler bei diesem Beispiel erkennen?

Beispiel zu Nachteilen einer statischen Typisierung

Beispiel 1:

```
public int verdopple(int zahl) {
    int doppelt = 2 *zahl;

    if(doppelt % 2 == 0) {
        return doppelt;
    } else {
        return "man glaubt es nicht: die Zahl ist ungerade";
    }
}
```

- Ist die Java-Methode so korrekt?
- Was, wann und wieso kann ein Compiler bei diesem Beispiel erkennen?
- Ist das ein Problem?

Beispiel 2:

```
public interface MeinInterface {
    int methode1();
    int methode2();
}

public class MeineKlasse implements MeinInterface {
    int methode1() { /* mein Code dazu */ }
    int methode2() { // die brauche ich eigentlich nicht */ }
}
```

Dynamische Typisierung

- In einer **dynamisch typisierten Sprache** werden Typüberprüfungen und Typentscheidungen erst zur Laufzeit getroffen
- Oft bei Skriptsprachen anzutreffen (python,…)

- Eine Typbehandlung erst zur Laufzeit kann die Effizienz des Programmcodes teilweise erheblich reduzieren (Faktor 10-100 und mehr langsamer)
- Eine (Typ-)Fehlerbehandlung kann je nach Sprache unterschiedlich sein

- Vorteile:
  - siehe Nachteile statische Typisierung
- Nachteile:
  - Effizienz
  - Fehlende Fehlererkennung

Beispiele

JavaScript

```
var a = 1;
var b = "hallo";
var c = a / b;
```

Mit speziellem Wert NaN (Not a Number) wird weitergerechnet

Visual Basic

```
a = 1
b = "hallo"
c = a / b
```

Programmabbruch zur Laufzeit

Starkes / schwaches Typsystem

- In einem **schwachen Typsystem** sind auch Operationen mit typmäßig unzulässigen Operationsparametern möglich
- In einem **starken Typsystem** sind ausschließlich zulässige Operanden erlaubt
- Java hat ein starkes Typsystem
- Beispiele für ein schwaches Typsystem (Programmiersprache C):

```
void check(int i) {
    // int wird als boolean missbraucht
    if(i)
        // %f bedeutet: interpretiere 32 Bit als Fließkommazahl (float)
        printf("Variableninhalt ist %f\n", i);
}
```

Was passiert hier?

```
void istZehn(int i) {
    if(i = 10)
        printf("Variable hat den Wert 10\n");
}
```

Achtung: in Java auch!

```
void istZehn(boolean b) {
    if(b = true)
        System.out.println("Argument b ist wahr");
}
```

Explizite / implizite Typisierung

- **Explizite Typisierung**: bei der Deklaration ist der Typ mit anzugeben.

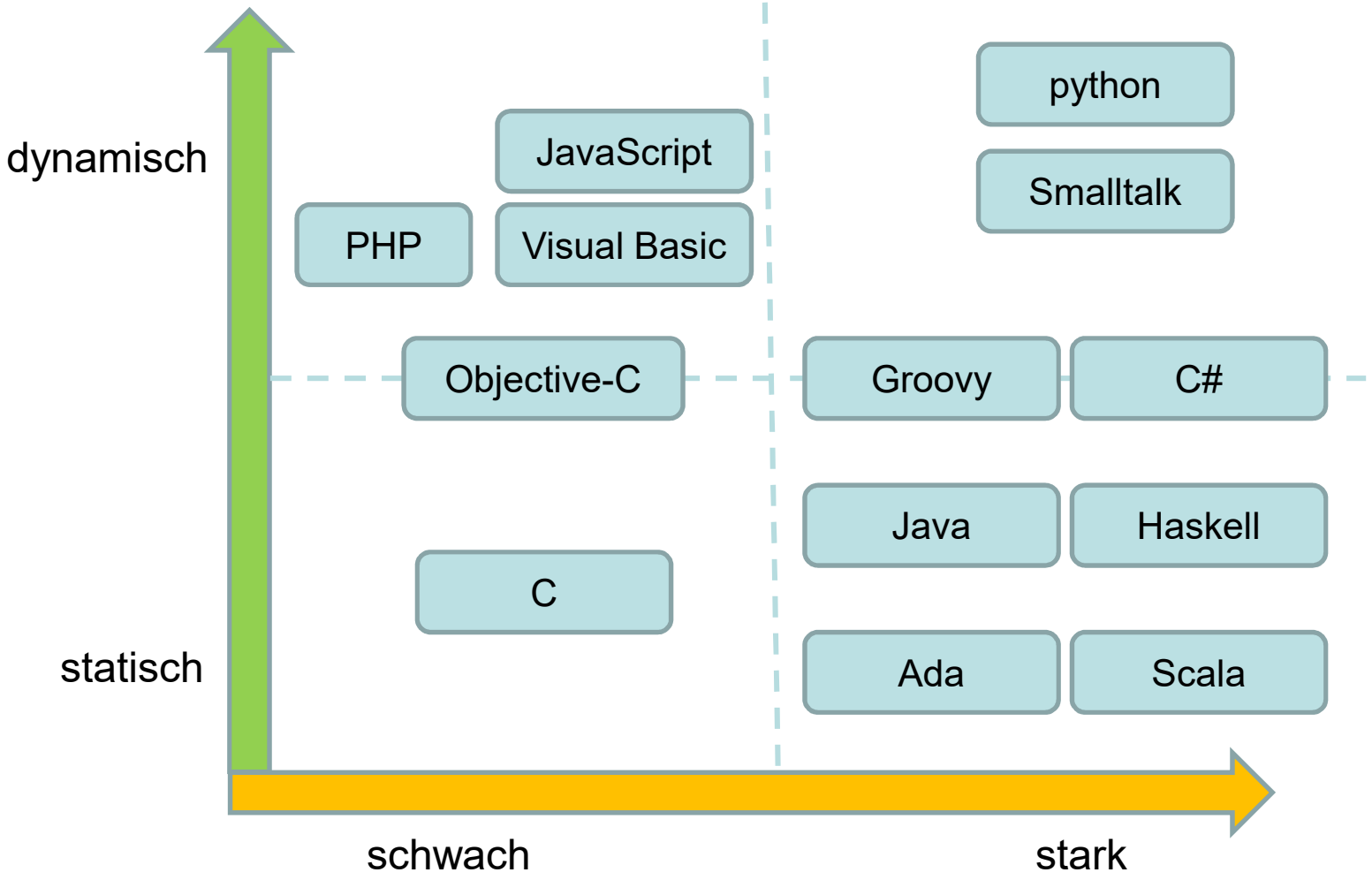
```
int i;
int j = i+1;
```

- **Implizite Typisierung**:
  - Neben einer Typangabe in einer Deklaration kann auch durch einen eindeutigen Typ in einer Initialisierung ein Typ bestimmt werden.
  - Ist der Typ festgelegt, können spätere Typverletzungen erkannt werden (egal, wie Typ festgelegt wurde)
  - Beispiel: Scala

```
val x: Int;      // expliziter Typangabe in Deklaration
val y = 1 + 2;   // implizite Typfestlegung durch Initialisierung

val a = "hallo"; // implizite Typfestlegung durch Initialisierung
val b = a / 2;   // Fehler zur Übersetzungszeit (Scala ist statisch typisiert)
```

Klassifizierung einiger Programmiersprachen



Beispiel

Wie würden Sie die Programmiersprache zu folgendem Code charakterisieren?

- dynamisch / statisch
- schwach / stark
- implizit / explizit

Die (Stellen der) Fehlermeldungen geben wesentliche Hinweise!

```
a = 1
b = "2"
c = "X"
if c > 0:          # hier Laufzeitfehler 1
    print("c > 0")
d = a + b          # hier Laufzeitfehler 2
```

- (Dies ist python-Code.)
- Laufzeitfehler 1: kein Fehler zur Übersetzungszeit, sondern erst zur Laufzeit
  → dynamisch typisiert
- Laufzeitfehler 1: Vergleich "x" (Inhalt von c) und 0 → stark typisiert
- In Zeile mit Laufzeitfehler 2: implizite Typisierung von d

Zwischenstand

- Es gibt verschieden Ansätze zur Typisierung in Programmiersprachen, die sich auch kombinieren lassen
- Mit eher laxen Ansätzen erhält man mehr Programmierfreiheiten, verliert dadurch aber teilweise sinnvolle Fehlerüberprüfungen

Reflektion

- Wenn man das letzte python-Beispiel sinngemäß nach Java übertragen würde: Wo gäbe es weshalb Probleme?

Inhalt dieser Veranstaltung