

# Informatik 2

Sommersemester 2007

Helmut Seidl

Institut für Informatik  
TU München

# 0 Allgemeines

## Inhalt dieser Vorlesung:

- Korrektheit von Programmen;
- Funktionales Programmieren mit OCaml :-)

# 1 Korrektheit von Programmen

- Programmierer machen Fehler :-)
- Programmierfehler können **teuer** sein, z.B. wenn eine Rakete explodiert, ein firmenwichtiges System für **Stunden** ausfällt ...
- In einigen Systemen dürfen **keine** Fehler vorkommen, z.B. Steuerungssoftware für Flugzeuge, Signalanlagen für Züge, Airbags in Autos ...

## Problem:

Wie können wir sicherstellen, dass ein Programm das **richtige** tut?

## Ansätze:

- Sorgfältiges Vorgehen bei der Software-Entwicklung;
- Systematisches Testen
  - ⇒ formales Vorgehensmodell (**Software Engineering**)
- Beweis der Korrektheit
  - ⇒ **Verifikation**

## Ansätze:

- Sorgfältiges Vorgehen bei der Software-Entwicklung;
- Systematisches Testen
  - ⇒ formales Vorgehensmodell (**Software Engineering**)
- Beweis der Korrektheit
  - ⇒ **Verifikation**

## Hilfsmittel:

Zusicherungen

## Beispiel:

```
public class GGT extends MiniJava {
    public static void main (String[] args) {
        int x, y, a, b;
        a = read(); b = read();
        x = a; y = b;
        while (x != y)
            if (x > y) x = x - y;
            else      y = y - x;

        assert(x != y);

        write(x);
    } // Ende der Definition von main();
} // Ende der Definition der Klasse GGT;
```

## Kommentare:

- Die statische Methode `assert()` erwartet ein Boolesches Argument.
- Bei normaler Programm-Ausführung wird jeder Aufruf `assert(e);` ignoriert :-)
- Starten wir **Java** mit der Option: `-ea` (**enable assertions**), werden die `assert`-Aufrufe ausgewertet:
  - ⇒ Liefert ein Argument-Ausdruck `true`, fährt die Programm-Ausführung fort.
  - ⇒ Liefert ein Argument-Ausdruck `false`, wird ein **Fehler** `AssertionError` geworfen.

## Achtung:

Der Laufzeit-Test soll eine **Eigenschaft** des Programm-Zustands bei Erreichen eines Programm-Punkts überprüfen.

Der Test sollte **keineswegs** den Programm-Zustand verändern !!!

Sonst zeigt das beobachtete System ein anderes Verhalten als das unbeobachtete ???



## Achtung:

Der Laufzeit-Test soll eine **Eigenschaft** des Programm-Zustands bei Erreichen eines Programm-Punkts überprüfen.

Der Test sollte **keineswegs** den Programm-Zustand verändern !!!

Sonst zeigt das beobachtete System ein anderes Verhalten als das unbeobachtete ???

## Tipp:

Um Eigenschaften komplizierterer Datenstrukturen zu überprüfen, empfiehlt es sich, getrennt **Inspector**-Klassen anzulegen, deren Objekte eine Datenstruktur **störungsfrei** besichtigen können :-)

## Problem:

- Es gibt i.a. sehr viele Programm-Ausführungen :-)
- Einhalten der Zusicherungen kann das Java-Laufzeit-System immer nur für eine Program-Ausführung überprüfen :-)



Wir benötigen eine generelle Methode, um das Einhalten einer Zusicherung zu **garantieren** ...

## 1.1 Verifikation von Programmen



Robert W Floyd, Stanford U. (1936 – 2001)

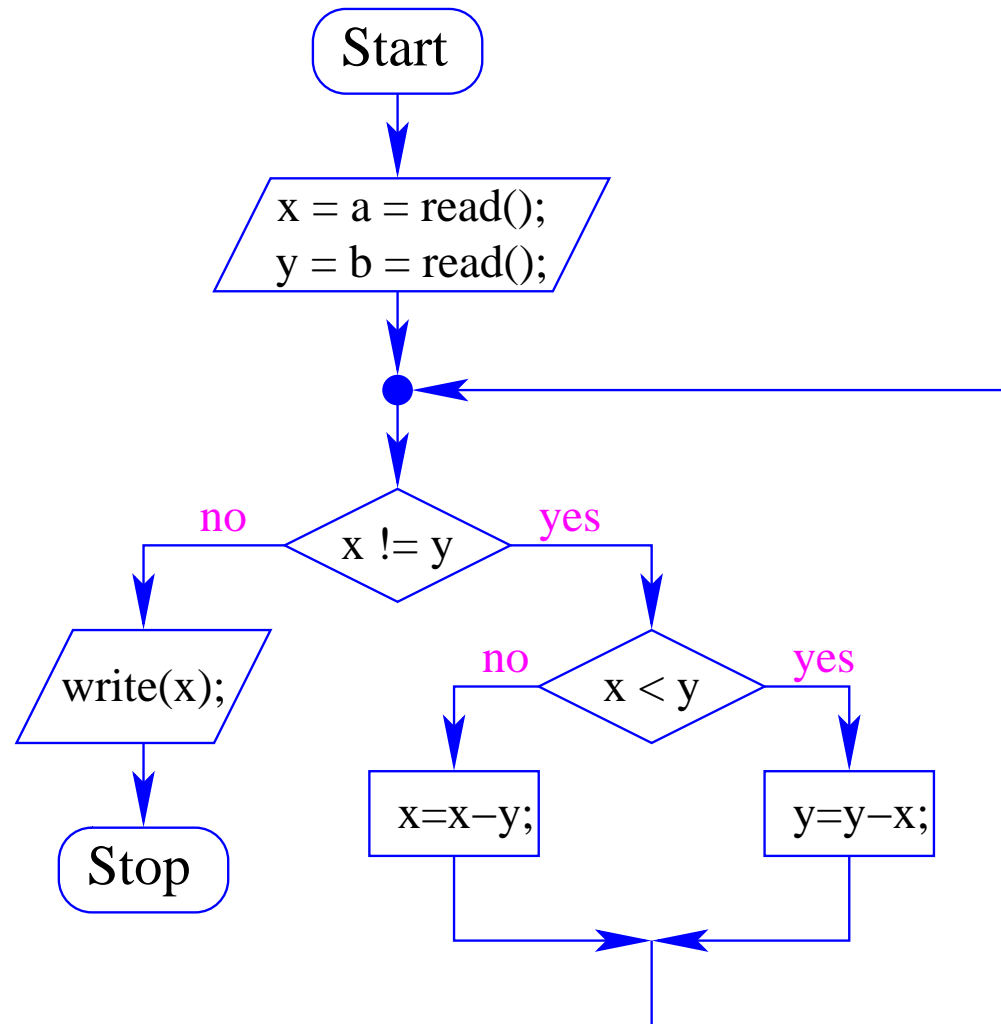
## Vereinfachung:

Wir betrachten erst mal nur **MiniJava** ;-)

## Idee:

- Wir schreiben eine Zusicherung an **jeden** Programmpunkt :-)
- Wir überprüfen **lokal**, dass die Zusicherungen von den einzelnen Anweisungen im Programm eingehalten werden.

# Unser Beispiel:



## Diskussion:

- Die Programmpunkte entsprechen den **Kanten** im Kontrollfluss-Diagramm :-)
- Wir benötigen eine Zusicherung pro Kante ...

## Hintergrund:

$d \mid x$  gilt genau dann wenn  $x = d \cdot z$  für eine ganze Zahl  $z$ .

Für ganze Zahlen  $x, y$  sei  $\text{ggT}(x, y) = 0$ , falls  $x = y = 0$  und andernfalls die größte ganze Zahl  $d$ , die  $x$  und  $y$  teilt.

Dann gelten unter anderem die folgenden Gesetze:

$$\begin{aligned}ggT(x, 0) &= |x| \\ggT(x, x) &= |x| \\ggT(x, y) &= ggT(x, y - x) \\ggT(x, y) &= ggT(x - y, y)\end{aligned}$$

## Idee für das Beispiel:

- Am Anfang gilt nix :-)
- Nach `a=read(); x=a;` gilt  $a = x$  :-)
- Vor Betreten und während der Schleife soll gelten:

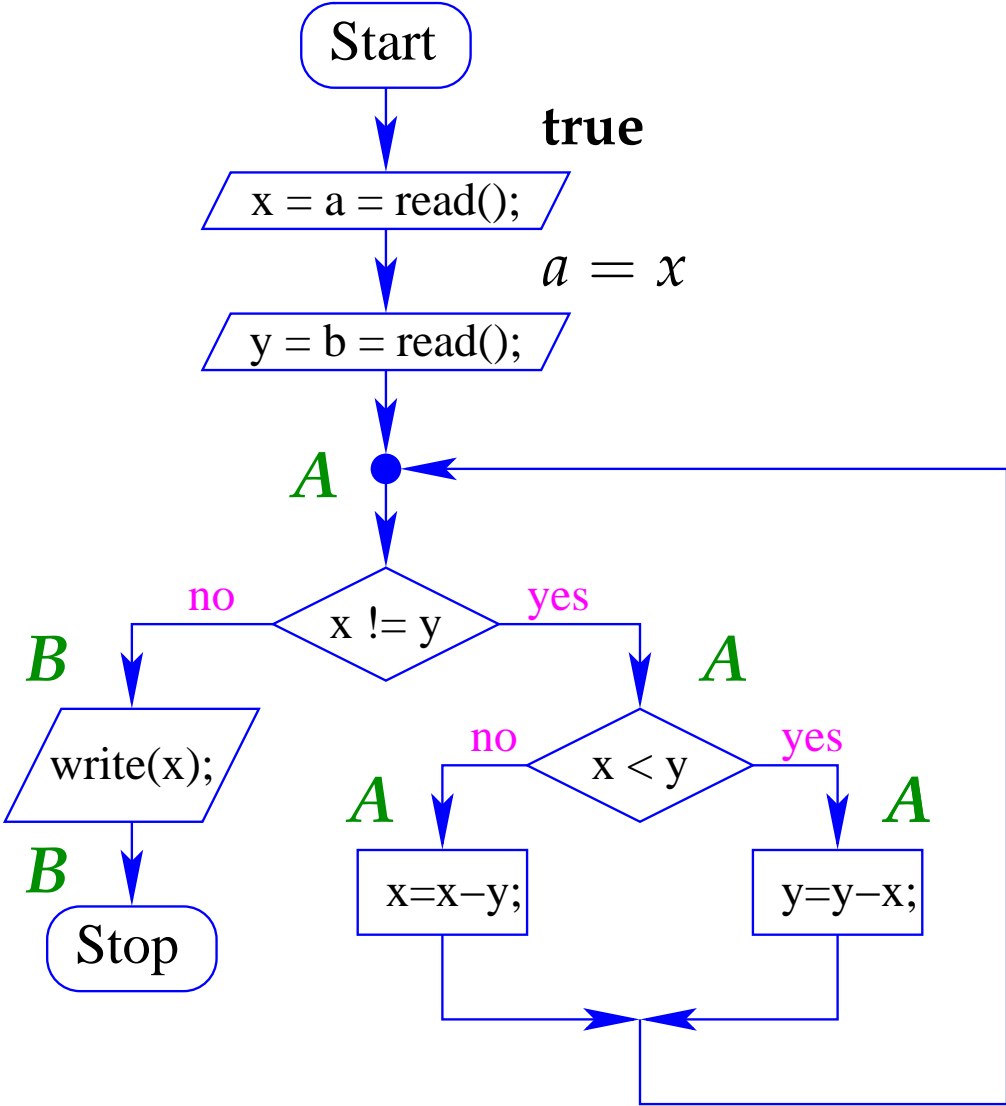
$$A \equiv \text{ggT}(a, b) = \text{ggT}(x, y)$$

- Am Programm-Ende soll gelten:

$$B \equiv A \wedge x = y$$



# Unser Beispiel:



## Frage:

Wie überprüfen wir, dass Zusicherungen lokal zusammen passen?

## Teilproblem 1: Zuweisungen

Betrachte z.B. die Zuweisung:  $x = y+z;$

Damit **nach** der Zuweisung gilt:  $x > 0,$  // **Nachbedingung**

muss **vor** der Zuweisung gelten:  $y + z > 0.$  // **Vorbedingung**

# Allgemeines Prinzip:

- Jede Anweisung transformiert eine Nachbedingung  $B$  in eine **minimale** Anforderung, die **vor** Ausführung erfüllt sein muss, damit  $B$  **nach** der Ausführung gilt :-)

# Allgemeines Prinzip:

- Jede Anweisung transformiert eine Nachbedingung  $B$  in eine **minimale** Anforderung, die **vor** Ausführung erfüllt sein muss, damit  $B$  **nach** der Ausführung gilt :-)
- Im Falle einer Zuweisung  $x = e;$  ist diese **schwächste Vorbedingung** (engl.: **weakest precondition**) gegeben durch

$$\mathbf{WP}[[x = e;]] (B) \equiv B[e/x]$$

Das heißt: wir **substituieren** einfach in  $B$  überall  $x$  durch  $e$  !!!

# Allgemeines Prinzip:

- Jede Anweisung transformiert eine Nachbedingung  $B$  in eine **minimale** Anforderung, die **vor** Ausführung erfüllt sein muss, damit  $B$  **nach** der Ausführung gilt :-)
- Im Falle einer Zuweisung  $x = e;$  ist diese **schwächste Vorbedingung** (engl.: **weakest precondition**) gegeben durch

$$\mathbf{WP}[\![x = e;\!] (B) \equiv B[e/x]$$

Das heißt: wir **substituieren** einfach in  $B$  überall  $x$  durch  $e$  !!!

- Eine beliebige Vorbedingung  $A$  für eine Anweisung  $s$  ist **gültig**, sofern

$$A \Rightarrow \mathbf{WP}[\![s]\!] (B)$$

//  $A$  **impliziert** die schwächste Vorbedingung für  $B$ .

## Beispiel:

Zuweisung:	$x = x - y;$
Nachbedingung:	$x > 0$
schwächste Vorbedingung:	$x - y > 0$
stärkere Vorbedingung:	$x - y > 2$
noch stärkere Vorbedingung:	$x - y = 3$ :-)

... im GGT-Programm (1):

Zuweisung:  $x = x - y;$

Nachbedingung:  $A$

schwächste Vorbedingung:

$$\begin{aligned} A[x - y/x] &\equiv \text{ggT}(a, b) = \text{ggT}(x - y, y) \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, y) \\ &\equiv A \end{aligned}$$

... im GGT-Programm (2):

Zuweisung:  $y = y - x;$

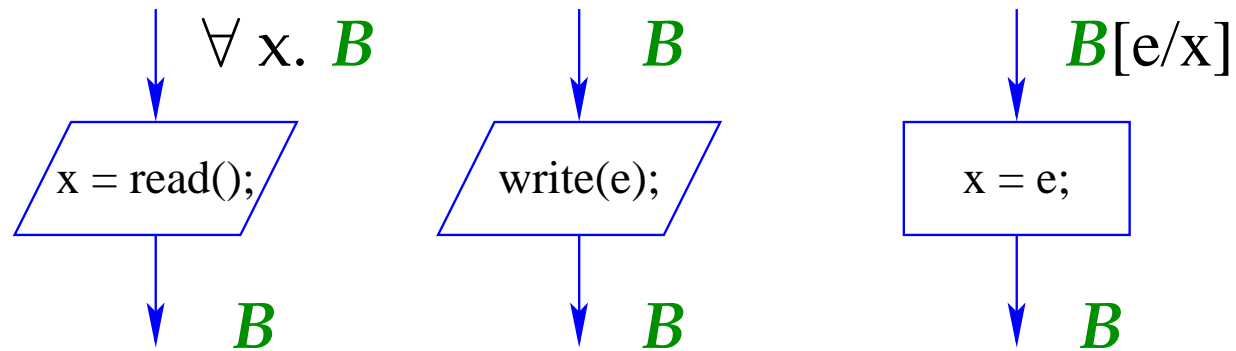
Nachbedingung:  $A$

schwächste Vorbedingung:

$$\begin{aligned} A[y - x/y] &\equiv \text{ggT}(a, b) = \text{ggT}(x, y - x) \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, y) \\ &\equiv A \end{aligned}$$



# Zusammenstellung:



$$\mathbf{WP}[\text{;}] (B) \equiv B$$

$$\mathbf{WP}[\text{x = e;}] (B) \equiv B[e/x]$$

$$\mathbf{WP}[\text{x = read();}] (B) \equiv \forall x. B$$

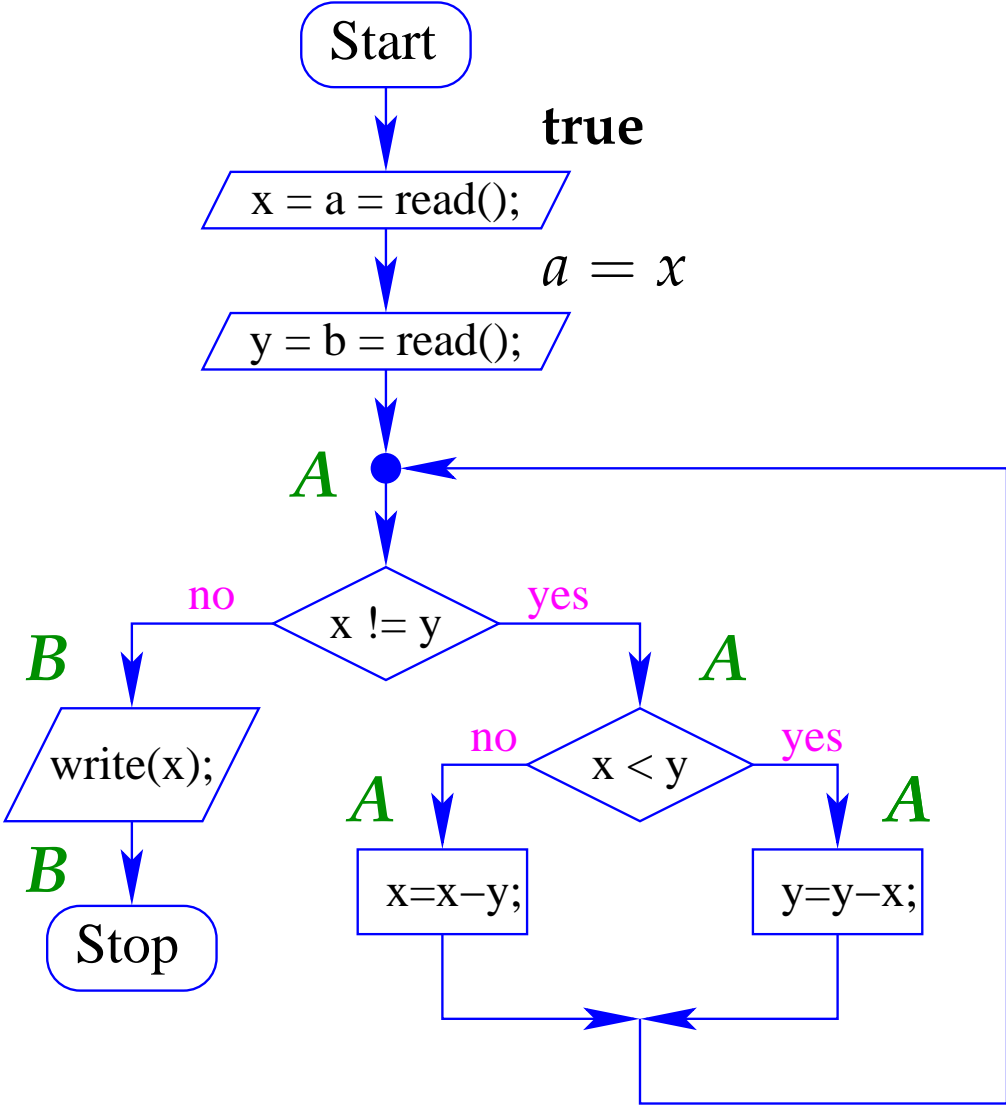
$$\mathbf{WP}[\text{write(e);}] (B) \equiv B$$

## Diskussion:

- Die Zusammenstellung liefert für alle Aktionen jeweils die **schwächsten** Vorbedingungen für eine Nachbedingung  $B$ .
- Eine Ausgabe-Anweisung ändert keine Variablen. Deshalb ist da die schwächste Vorbedingung  $B$  selbst ;-)
- Eine Eingabe-Anweisung  $x = \text{read}()$ ; ändert die Variable  $x$  auf unvorhersehbare Weise.

Damit nach der Eingabe  $B$  gelten kann, muss  $B$  vor der Eingabe für jedes mögliche  $x$  gelten ;-)

# Orientierung:



Für die Anweisungen: `b = read(); y = b;` berechnen wir:

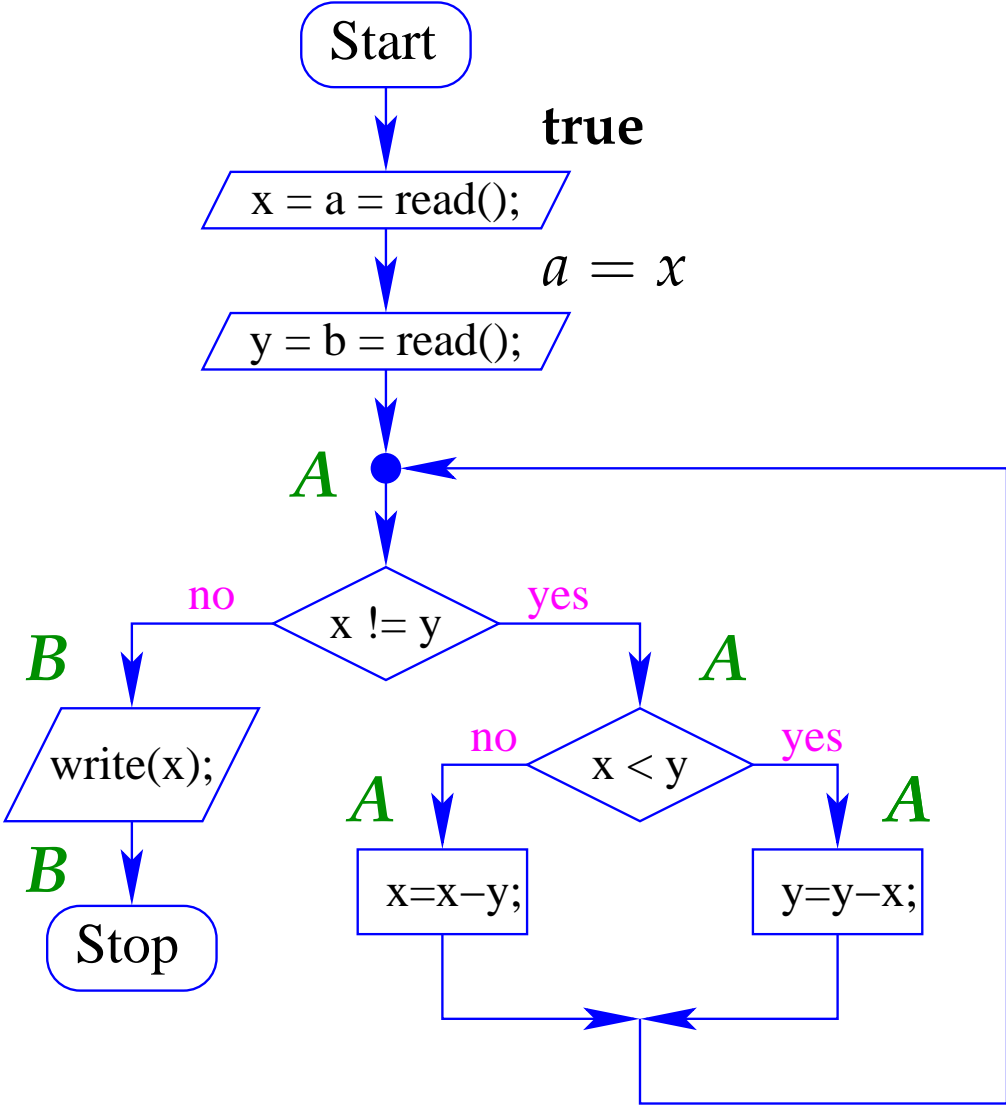
$$\begin{aligned} \mathbf{WP}[[y = b;]] (A) &\equiv A[b/y] \\ &\equiv ggT(a, b) = ggT(x, b) \end{aligned}$$

Für die Anweisungen:  $b = \text{read}(); y = b;$  berechnen wir:

$$\begin{aligned}\mathbf{WP}[[y = b;]] (A) &\equiv A[b/y] \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, b)\end{aligned}$$

$$\begin{aligned}\mathbf{WP}[[b = \text{read}();]] (\text{ggT}(a, b) = \text{ggT}(x, b)) \\ &\equiv \forall b. \text{ggT}(a, b) = \text{ggT}(x, b) \\ &\Leftarrow a = x \quad \text{:-)}\end{aligned}$$

# Orientierung:



Für die Anweisungen: `a = read(); x = a;` berechnen wir:

$$\begin{aligned} \mathbf{WP}[\mathbf{x} = \mathbf{a};] (a = x) &\equiv a = a \\ &\equiv \mathbf{true} \end{aligned}$$

$$\begin{aligned} \mathbf{WP}[\mathbf{a} = \mathbf{read}();] (\mathbf{true}) &\equiv \forall a. \mathbf{true} \\ &\equiv \mathbf{true} \quad \text{: -)} \end{aligned}$$