# Exkurs: Aussagenlogik

Aussagen: "Alle Menschen sind sterblich",

"Sokrates ist ein Mensch", "Sokrates ist sterblich"

# Exkurs: Aussagenlogik

Aussagen: "Alle Menschen sind sterblich",

"Sokrates ist ein Mensch", "Sokrates ist sterblich"

$$\forall\, x.\, \mathsf{Mensch}(x) \Rightarrow \mathsf{sterblich}(x)$$

$$\mathsf{Mensch}(\mathsf{Sokrates}),\ \mathsf{sterblich}(\mathsf{Sokrates})$$

# Exkurs: Aussagenlogik

Aussagen: "Alle Menschen sind sterblich",

"Sokrates ist ein Mensch", "Sokrates ist sterblich"

$\forall\, x.\, \mathsf{Mensch}(x) \Rightarrow \mathsf{sterblich}(x)$

$\mathsf{Mensch}(\mathsf{Sokrates}),\ \mathsf{sterblich}(\mathsf{Sokrates})$

Schließen: Falls $\forall\, x.\, P(x)$ gilt, dann auch $P(a)$ für ein konkretes $a$ !

Falls $A \Rightarrow B$ und $A$ gilt, dann muss auch $B$ gelten !

# Exkurs: Aussagenlogik

Aussagen: "Alle Menschen sind sterblich",

"Sokrates ist ein Mensch", "Sokrates ist sterblich"

$\forall\, x.\, \mathsf{Mensch}(x) \Rightarrow \mathsf{sterblich}(x)$

$\mathsf{Mensch}(\mathsf{Sokrates}),\ \mathsf{sterblich}(\mathsf{Sokrates})$

Schließen: Falls $\forall\, x.\, P(x)$ gilt, dann auch $P(a)$ für ein konkretes $a$ !

Falls $A \Rightarrow B$ und $A$ gilt, dann muss auch $B$ gelten !

Tautologien: $A \vee \neg A$

$\forall\, x \in \mathbb{Z}.\, x < 0 \vee x = 0 \vee x > 0$

# Exkurs: Aussagenlogik (Forts.)

Gesetze:

$$\neg\neg A \equiv A$$

$$A \wedge A \equiv A$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$A \vee (B \wedge A) \equiv A$$

$$A \wedge (B \vee A) \equiv A$$

## 1.2 Korrektheit

### Fragen:

- Welche Programm-Eigenschaften können wir mithilfe lokal konsistenter Annotierungen garantieren ?
- Wie können wir nachweisen, dass unser Verfahren keine falschen Ergebnisse liefert ??

# Erinnerung (1):

- In MiniJava können wir ein Zustand $\sigma$ aus einer Variablen-Belegung, d.h. einer Abbildung der Programm-Variablen auf ganze Zahlen (ihren Werten), z.B.:

$$\sigma = \{x \mapsto 5, y \mapsto -42\}$$

# Erinnerung (1):

- In MiniJava können wir ein Zustand $\sigma$ aus einer Variablen-Belegung, d.h. einer Abbildung der Programm-Variablen auf ganze Zahlen (ihren Werten), z.B.:

$$\sigma = \{x \mapsto 5, y \mapsto -42\}$$

- Ein Zustand $\sigma$ erfüllt eine Zusicherung $A$, falls

$$A[\sigma(x)/x]_{x \in A}$$

// wir substituieren jede Variable in $A$ durch ihren Wert in $\sigma$

eine wahre Aussage ist, d.h. äquivalent **true**.

**Wir schreiben:** $\sigma \models A$.

## Beispiel:

$$
\begin{aligned}
\sigma & = \{x \mapsto 5, y \mapsto 2\} \\
A & \equiv (x > y) \\
A[5/x, 2/y] & \equiv (5 > 2) \\
& \equiv \textbf{true}
\end{aligned}
$$

# Beispiel:

$$
\begin{aligned}
\sigma &= \{x \mapsto 5, y \mapsto 2\} \\
A &\equiv (x > y) \\
A[5/x, 2/y] &\equiv (5 > 2) \\
&\equiv \textbf{true}
\end{aligned}
$$

$$
\begin{aligned}
\sigma &= \{x \mapsto 5, y \mapsto 12\} \\
A &\equiv (x > y) \\
A[5/x, 12/y] &\equiv (5 > 12) \\
&\equiv \textbf{false}
\end{aligned}
$$

# Triviale Eigenschaften:

$$\sigma \models \textbf{true} \quad \text{für jedes} \quad \sigma$$

$$\sigma \models \textbf{false} \quad \text{für kein} \quad \sigma$$

$\sigma \models A_1$ und $\sigma \models A_2$ ist äquivalent zu

$$\sigma \models A_1 \wedge A_2$$

$\sigma \models A_1$ oder $\sigma \models A_2$ ist äquivalent zu

$$\sigma \models A_1 \vee A_2$$

# Erinnerung (2):

- Eine Programmausführung $\pi$ durchläuft einen Pfad im Kontrollfluss-Graphen :-)
- Sie beginnt in einem Programmpunkt $u_0$ in einem Anfangszustand $\sigma_0$. und führt in einen Programmpunkt $u_m$ und einen Endzustand $\sigma_m$.
- Jeder Schritt der Programm-Ausführung führt eine Aktion aus und ändert Programmpunkt und Zustand :-)

# Erinnerung (2):

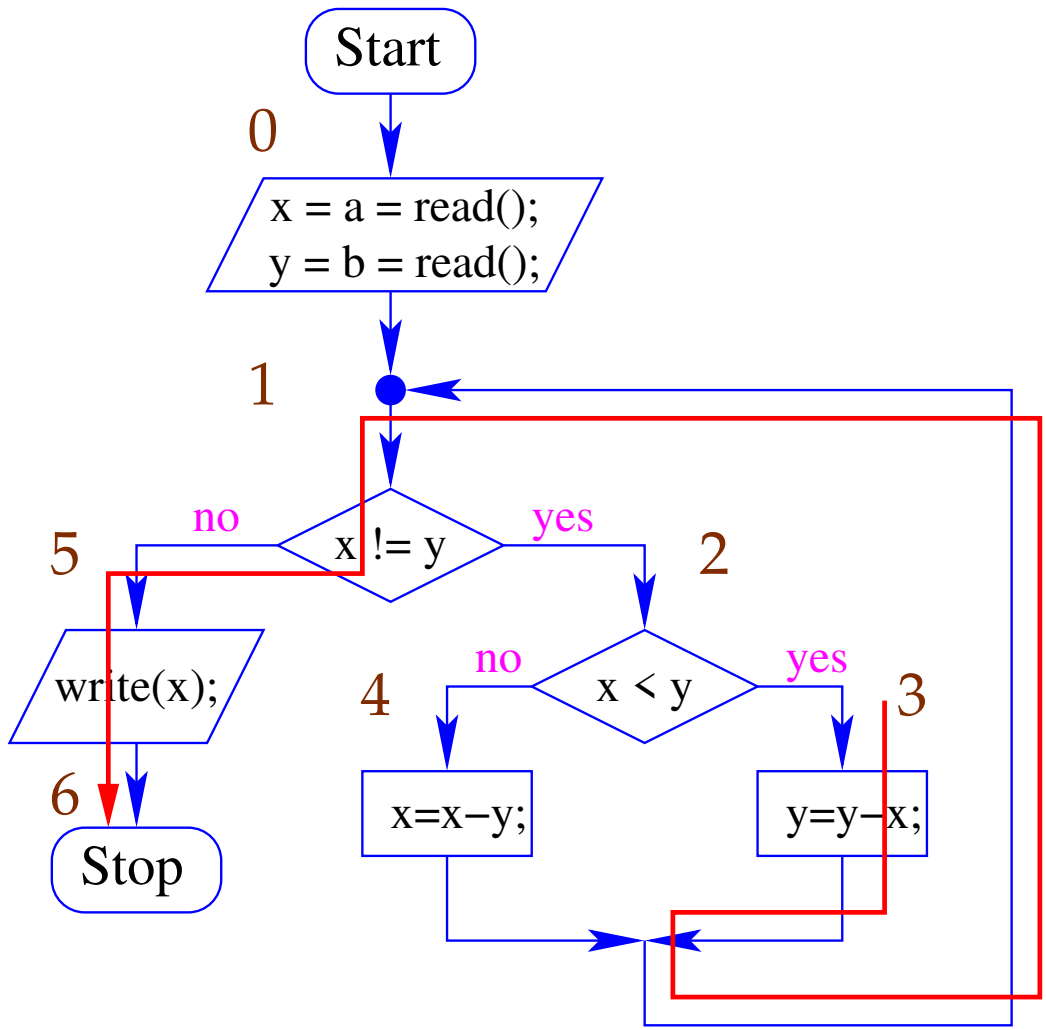
- Eine Programmausführung $\pi$ durchläuft einen Pfad im Kontrollfluss-Graphen :-)
- Sie beginnt in einem Programmpunkt $u_0$ in einem Anfangszustand $\sigma_0$. und führt in einen Programmpunkt $u_m$ und einen Endzustand $\sigma_m$.
- Jeder Schritt der Programm-Ausführung führt eine Aktion aus und ändert Programmpunkt und Zustand :-)

$\Longrightarrow$ Wir können $\pi$ als Folge darstellen:

$$(u_0, \sigma_0) s_1 (u_1, \sigma_1) \ldots s_m (u_m, \sigma_m)$$

wobei die $s_i$ Elemente des Kontrollfluss-Graphen sind, d.h. Anweisungen oder Bedingungen ...

# Beispiel:


Start
0
x = a = read();
y = b = read();
1
no
x != y
yes
5
2
write(x);
no
x < y
yes
4
3
6
x=x−y;
y=y−x;
Stop

Nehmen wir an, wir starten in Punkt 3 mit $\{x \mapsto 6, y \mapsto 12\}$.

Dann ergibt sich die Programmausführung:

$$
\begin{array}{rcll}
\pi & = & (3, \{x \mapsto 6, y \mapsto 12\}) & \texttt{y = y-x;} \\
 & & (1, \{x \mapsto 6, y \mapsto 6\}) & \texttt{!(x != y)} \\
 & & (5, \{x \mapsto 6, y \mapsto 6\}) & \texttt{write(x);} \\
 & & (6, \{x \mapsto 6, y \mapsto 6\}) &
\end{array}
$$

## Satz:

Sei $p$ ein MiniJava-Programm, Sei $\pi$ eine Programmausführung, die im Programmpunkt $u$ startet und zum Programmpunkt $v$ führt.

**Annahmen:**

- Die Programmpunkte von $p$ seien lokal konsistent mit Zusicherungen annotiert.
- Der Programmpunkt $u$ sei mit $A$ annotiert.
- Der Programmpunkt $v$ sei mit $B$ annotiert.

# Satz:

Sei $p$ ein MiniJava-Programm, Sei $\pi$ eine Programmausführung, die im Programmpunkt $u$ startet und zum Programmpunkt $v$ führt.

**Annahmen:**

- Die Programmpunkte von $p$ seien lokal konsistent mit Zusicherungen annotiert.
- Der Programmpunkt $u$ sei mit $A$ annotiert.
- Der Programmpunkt $v$ sei mit $B$ annotiert.

**Dann gilt:**

Erfüllt der Anfangszustand von $\pi$ die Zusicherung $A$, dann erfüllt der Endzustand die Zusicherung $B$.

# Bemerkungen:

- Ist der Startpunkt des Programms mit **true** annotiert, dann erfüllt jede Programmausführung, die den Programmpunkt $v$ erreicht, die Zusicherung an $v$ :-)
- Zum Nachweis, dass eine Zusicherung $A$ an einem Programmpunkt $v$ gilt, benötigen wir eine lokal konsistente Annotierung mit zwei Eigenschaften:

  (1) der Startpunkt ist mit **true** annotiert;

  (2) Die Zusicherung an $v$ impliziert $A$ :-)

# Bemerkungen:

- Ist der Startpunkt des Programms mit **true** annotiert, dann erfüllt jede Programmausführung, die den Programmpunkt $v$ erreicht, die Zusicherung an $v$ :-)
- Zum Nachweis, dass eine Zusicherung $A$ an einem Programmpunkt $v$ gilt, benötigen wir eine lokal konsistente Annotierung mit zwei Eigenschaften:

  (1) der Startpunkt ist mit **true** annotiert;

  (2) Die Zusicherung an $v$ impliziert $A$ :-)
- Unser Verfahren gibt (vorerst) keine Garantie, dass $v$ überhaupt erreicht wird !!!
- Falls ein Programmpunkt $v$ mit der Zusicherung **false** annotiert werden kann, kann $v$ nie erreicht werden :-))

# Beweis:

Sei $\pi = (u_0, \sigma_0) s_1 (u_1, \sigma_1) \ldots s_m (u_m, \sigma_m)$

Gelte: $\sigma_0 \models A$.

Wir müssen zeigen: $\sigma_m \models B$.

# Idee:

Induktion nach der Länge $m$ der Programmausführung :-)

# Fazit:

- Das Verfahren nach Floyd ermöglicht uns zu beweisen, dass eine Zusicherung $B$ bei Erreichen eines Programmpunkts stets (bzw. unter geeigneten Zusatzannahmen :-) gilt ...
- Zur Durchführung benötigen wir:
  - Zusicherung **true** am Startpunkt.
  - Zusicherungen an jedem weiteren Programmpunkt :-(
  - Nachweis, dass die Zusicherungen lokal konsistent sind

$\Longrightarrow$ Logik, automatisches Beweisen

## 1.3 Optimierung

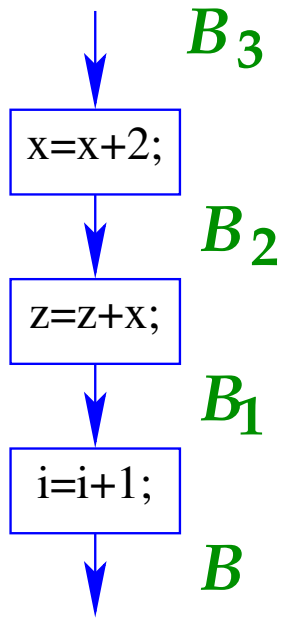
**Ziel:** Verringerung der benötigten Zusicherungen

**Beobachtung:**

Hat das Programm keine Schleifen, können wir für jeden Programmpunkt eine hinreichende Vorbedingung ausrechnen !!!

# Beispiel:

```
x = x+2;
z = z+x;
i = i+1;
```



# Beispiel (Fort.):

Sei $\quad B \equiv z = i^2 \;\wedge\; x = 2i - 1$

Dann rechnen wir:

$$
\begin{aligned}
B_1 \;\equiv\; \mathbf{WP}[\![\texttt{i = i+1;}]\!](B) \quad &\equiv \quad z = (i+1)^2 \wedge x = 2(i+1) - 1 \\
&\equiv \quad z = (i+1)^2 \wedge x = 2i + 1
\end{aligned}
$$

# Beispiel (Fort.):

Sei $\quad B \equiv z = i^2 \;\wedge\; x = 2i - 1$

Dann rechnen wir:

$$
\begin{aligned}
B_1 \;&\equiv\; \mathbf{WP}[\![\texttt{i = i+1;}]\!](B) &&\equiv\; z = (i+1)^2 \wedge x = 2(i+1) - 1 \\
&&&\equiv\; z = (i+1)^2 \wedge x = 2i + 1 \\
B_2 \;&\equiv\; \mathbf{WP}[\![\texttt{z = z+x;}]\!](B_1) &&\equiv\; z + x = (i+1)^2 \wedge x = 2i + 1 \\
&&&\equiv\; z = i^2 \wedge x = 2i + 1
\end{aligned}
$$

# Beispiel (Fort.):

Sei $\quad B \equiv z = i^2 \;\wedge\; x = 2i - 1$

Dann rechnen wir:

$$
\begin{array}{lcll}
B_1 & \equiv & \mathbf{WP}[\![\texttt{i = i+1;}]\!](B) & \equiv \quad z = (i+1)^2 \wedge x = 2(i+1) - 1 \\
 & & & \equiv \quad z = (i+1)^2 \wedge x = 2i + 1 \\
B_2 & \equiv & \mathbf{WP}[\![\texttt{z = z+x;}]\!](B_1) & \equiv \quad z + x = (i+1)^2 \wedge x = 2i + 1 \\
 & & & \equiv \quad z = i^2 \wedge x = 2i + 1 \\
B_3 & \equiv & \mathbf{WP}[\![\texttt{x = x+2;}]\!](B_2) & \equiv \quad z = i^2 \wedge x + 2 = 2i + 1 \\
 & & & \equiv \quad z = i^2 \wedge x = 2i - 1 \\
 & & & \equiv \quad B \qquad \text{;-)}
\end{array}
$$

# Idee:

- Für jede Schleife wähle einen Programmpunkt aus.

  Sinnvolle Auswahlen:

  $\longrightarrow$ Vor der Bedingung;

  $\longrightarrow$ Am Beginn des Rumpfs;

  $\longrightarrow$ Am Ende des Rumpfs ...

- Stelle für jeden gewählten Punkt eine Zusicherung bereit

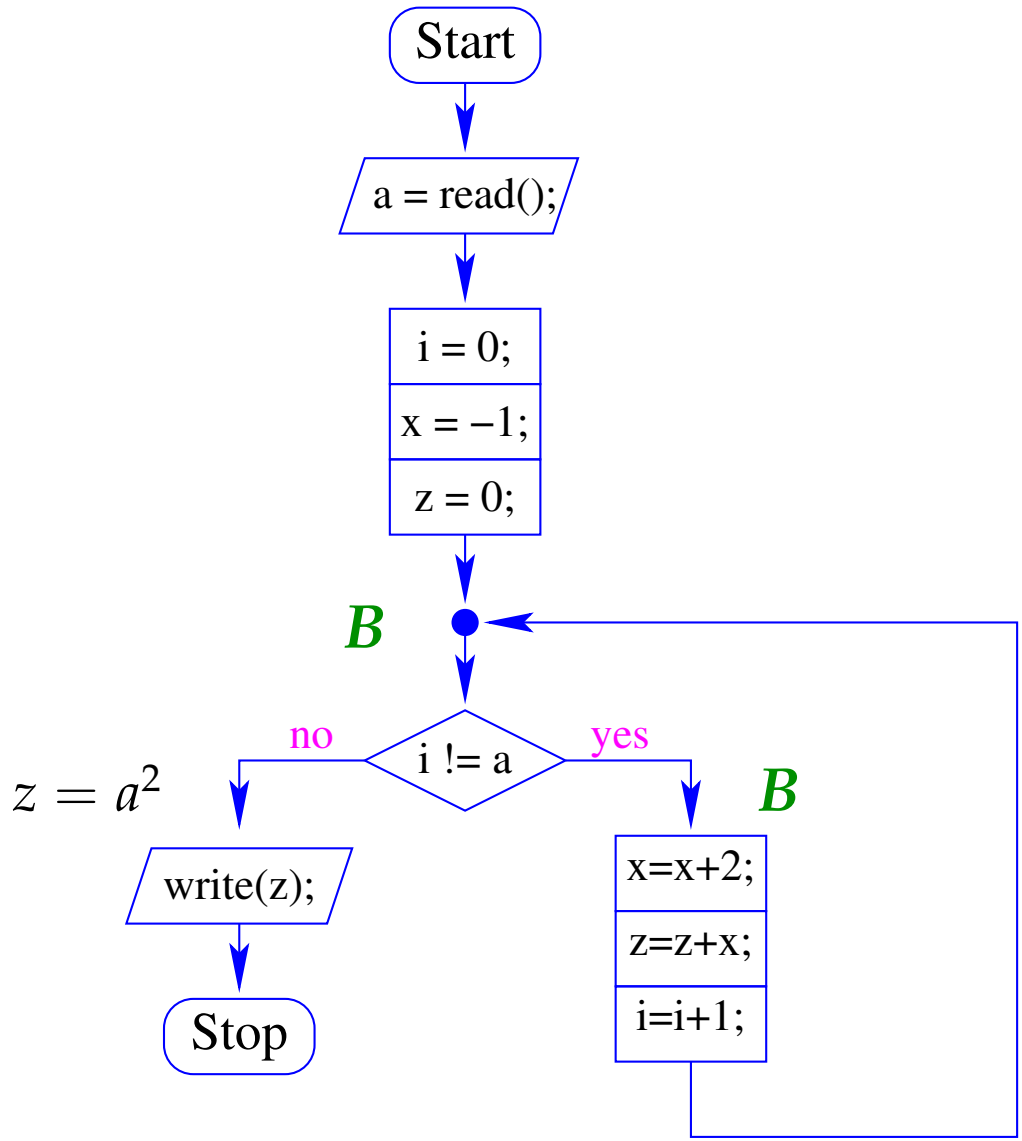
  $\Longrightarrow$ Schleifen-Invariante

- Für alle übrigen Programmpunkte bestimmen wir Zusicherungen mithilfe $\mathbf{WP}[\![\ldots]\!]()$ :-)

# Beispiel:

```
int a, i, x, z;
a = read();
i = 0;
x = -1;
z = 0;
while (i != a) {
   x = x+2;
   z = z+x;
   i = i+1;
}
assert(z==a*a);
write(z);
```

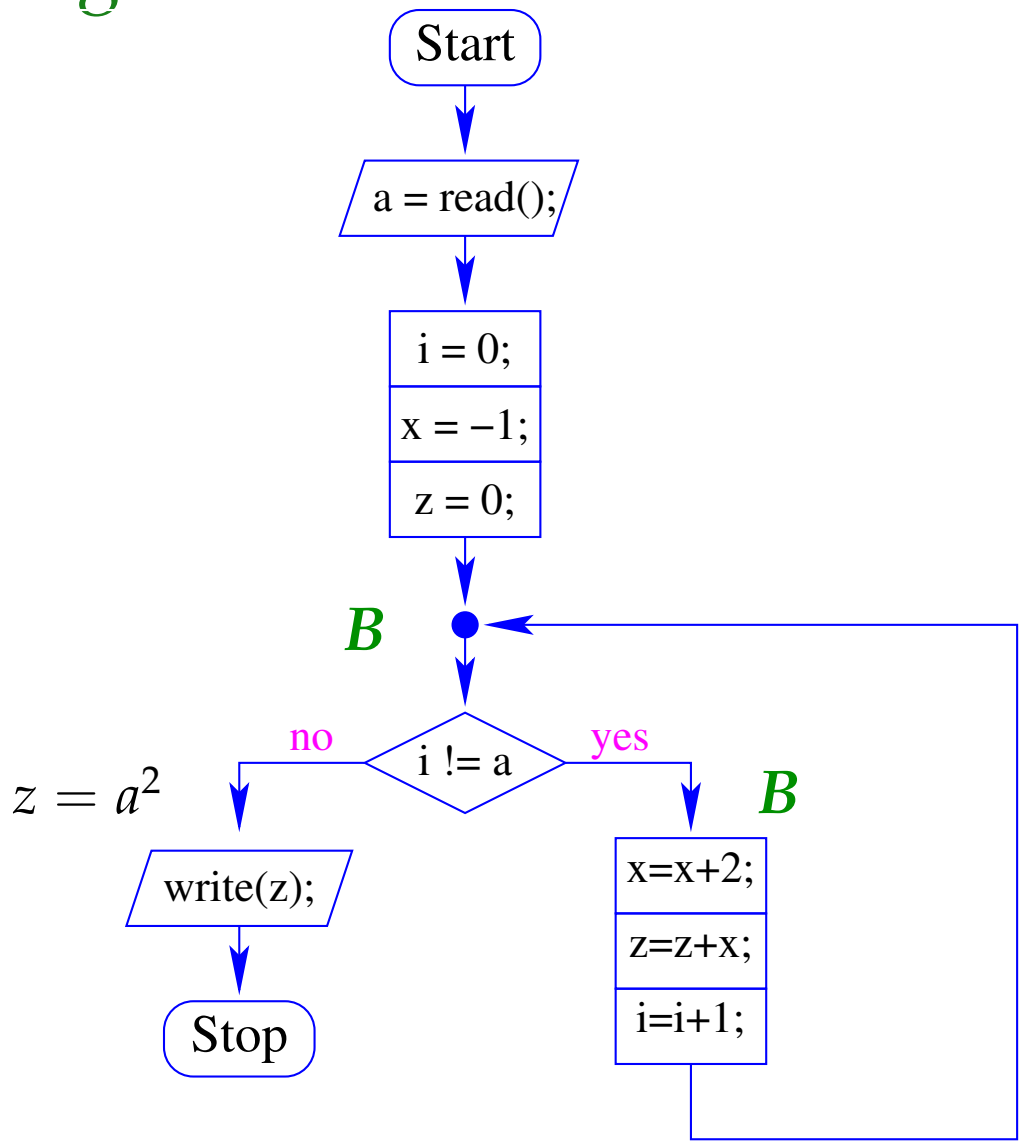
# Beispiel:



## Wir überprüfen:

$$
\begin{aligned}
&\mathbf{WP}[\![\texttt{i != a}]\!](z = a^2, B) \\
&\qquad \equiv \quad (i = a \wedge z = a^2) \;\vee\; (i \neq a \wedge B) \\
&\qquad \equiv \quad (i = a \wedge z = a^2) \;\vee\; (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \\
&\qquad \Leftarrow \quad (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \;\vee\; (i = a \wedge z = i^2 \wedge x = 2i - 1) \\
&\qquad \equiv \quad z = i^2 \wedge x = 2i - 1 \quad \equiv \quad B \qquad\qquad \text{:-)}
\end{aligned}
$$

# Orientierung:



## Wir überprüfen:

$$
\begin{array}{lcl}
\mathbf{WP}[\![\texttt{z = 0;}]\!](B) & \equiv & 0 = i^2 \wedge x = 2i - 1 \\
 & \equiv & i = 0 \wedge x = -1 \\
\mathbf{WP}[\![\texttt{x = -1;}]\!](i = 0 \wedge x = -1) & \equiv & i = 0 \\
\mathbf{WP}[\![\texttt{i = 0;}]\!](i = 0) & \equiv & \mathbf{true} \\
\mathbf{WP}[\![\texttt{a = read();}]\!](\mathbf{true}) & \equiv & \mathbf{true} \qquad \text{:-))}
\end{array}
$$

## 1.4 Terminierung

### Problem:

- Mit unserer Beweistechnik können wir nur beweisen, dass eine Eigenschaft gilt wann immer wir einen Programmpunkt erreichen !!!
- Wie können wir aber garantieren, dass das Programm immer terminiert ?
- Wie können wir eine Bedingung finden, unter der das Programm immer terminiert ??

## Beispiele:

- Das ggT-Programm terminiert nur für Eingaben $a, b$ mit: $a > 0$ und $b > 0$.
- Das Quadrier-Programm terminiert nur für Eingaben $a \geq 0$.
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer :-)

## Beispiele:

- Das ggT-Programm terminiert nur für Eingaben $a, b$ mit: $a > 0$ und $b > 0$.
- Das Quadrier-Programm terminiert nur für Eingaben $a \geq 0$.
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer :-)

Lässt sich dieses Beispiel verallgemeinern ??

## Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
   j = read();
   while (j>0) { t = t+1; j = j-1; }
   i = i-1;
}
write(t);
```

- Die gelesene Zahl `i` (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-) die Summe der positiven für j gelesenen Werte

**Beispiel:**

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
   j = read();
   while (j>0) { t = t+1; j = j-1; }
   i = i-1;
}
write(t);
```

- Die gelesene Zahl `i` (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-) die Summe der positiven für j gelesenen Werte

$\Longrightarrow$ das Programm terminiert immer !!!

Programme nur mit `for`-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
```

// im Rumpf wird `i` nicht modifiziert

... terminieren ebenfalls immer :-))

Programme nur mit `for`-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
```

// im Rumpf wird `i` nicht modifiziert

... terminieren ebenfalls immer :-))

## Frage:

Wie können wir aus dieser Beobachtung eine Methode machen, die auf beliebige Schleifen anwendbar ist ?

# Idee:

- Weise nach, dass jede Scheife nur endlich oft durchlaufen wird ...
- Finde für jede Schleife eine Kenngröße $r$, die zwei Eigenschaften hat:

  (1) Wenn immer der Rumpf betreten wird, ist $r > 0$;

  (2) Bei jedem Schleifen-Durchlauf wird $r$ kleiner :-)

- Transformiere das Programm so, dass es neben der normalen Programmausführung zusätzlich die Kenngrößen $r$ mitberechnet.
- Verifiziere, dass (1) und (2) gelten :-)

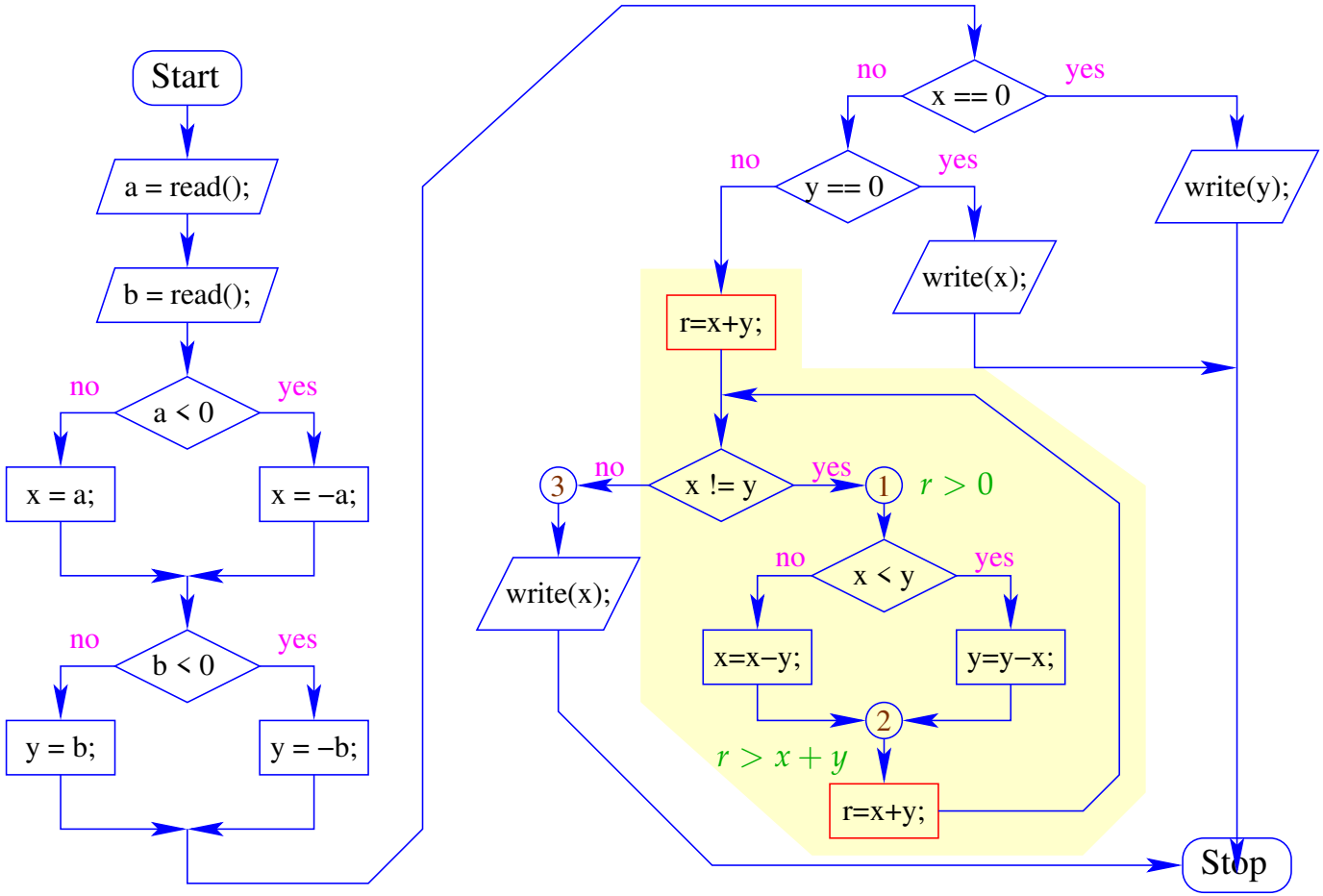
## Beispiel: Sicheres ggT-Programm

```
int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else {
        while (x != y)
           if (y > x) y = y-x;
           else       x = x-y;
        write(x);
}
```

Wir wählen: $r = x + y$

Transformation:

```
int a, b, x, y, r;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else { r = x+y;
        while (x != y) {
           if (y > x) y = y-x;
           else       x = x-y;
           r = x+y; }
        write(x);
}
```


Start
a = read();
b = read();
a < 0
no
yes
x = a;
x = −a;
b < 0
no
yes
y = b;
y = −b;
x == 0
no
yes
write(y);
y == 0
no
yes
write(x);
r=x+y;
x != y
no
yes
3
1
$r > 0$
write(x);
x < y
no
yes
x=x−y;
y=y−x;
2
$r > x + y$
r=x+y;
Stop