

## 2.7 Effizienz

Problem:

Spielbäume können **RIESIG** werden!!

## Unsere Lösung:

- Wir erzeugen die ME-Strategie nicht für alle möglichen Spiel-Verläufe, sondern erst **nach dem ersten Zug** der Gegnerin. Spart ... **Faktor 9**
- Wir berücksichtigen **Zug-Zwang**. Spart ... **??!!...:-)**
- Wir sind mit **akzeptablen** ME-Zügen zufrieden. Spart ungefähr ... **Faktor 2**

## Achtung:

- Für Tic-Tac-Toe reicht das vollkommen aus: pro Spielverlauf werden zwischen 126 und 1142 MyChoice-Knoten angelegt ...
- Für komplexere Spiele wie Dame, Schach oder gar Go benötigen wir weitere Ideen ...

## 1. Idee: Eröffnungen

- Tabelliere Anfangs-Stücke optimaler Spiel-Verläufe.
- Konstruiere die Strategie erst ab der ersten Konfiguration, die von den tabellierten Eröffnungen abweicht ...

## Beispiel: Tic-Tac-Toe

Wir könnten z.B. beste Antworten auf jeden möglichen Eröffnungs-Zug tabellieren:

```
public interface Opening {  
    int[] OPENING = {  
        4, 4, 4, 4, 2, 4, 4, 4, 4  
    };  
}
```

- Die Funktion `int nextMove(int place);` schlägt dann den ersten Antwort-Zug in `OPENING` nach.

- Erst bei der zweiten Antwort (d.h. für den vierten Stein auf dem Brett) wird die ME-Strategie konstruiert.
- Dann bleiben grade mal höchstens  $6! = 720$  Spiel-Fortsetzungen übrig ... die Anzahl der tatsächlich benötigten MyChoice-Knoten scheint aber nur noch zwischen 9 und 53 zu schwanken (!!!)

## 2. Idee:           Bewertungen

Finde eine geeignete Funktion *advice*, die die Erfolgsaussichten einer Konfiguration direkt abschätzt, d.h. ohne Aufbau eines Spielbaums.

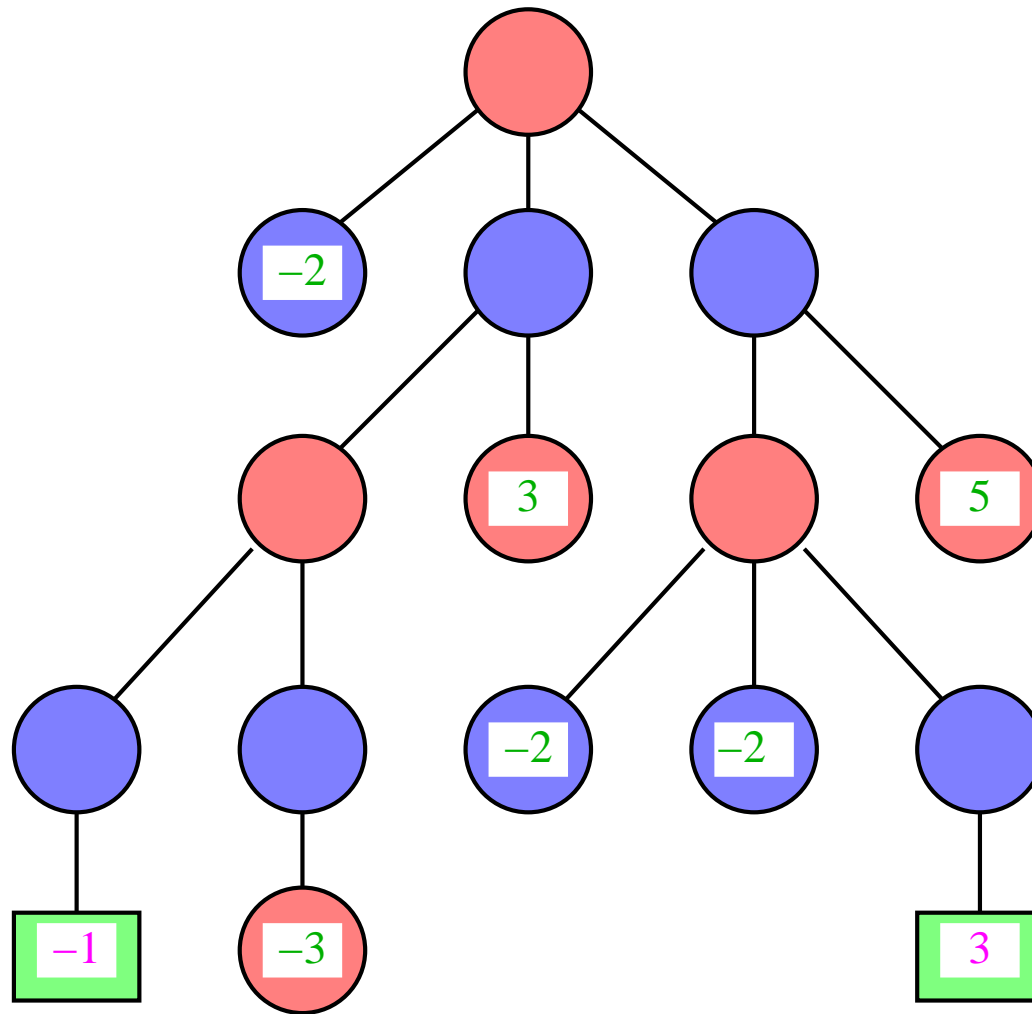
### Achtung:

- I.a. ist eine solche Funktion nicht bekannt   :-(
- Man muss mit unpräzisen bis fehlerhaften Bewertungs-Funktionen zurecht kommen ...

### 3. Idee: $(\alpha, \beta)$ -Pruning

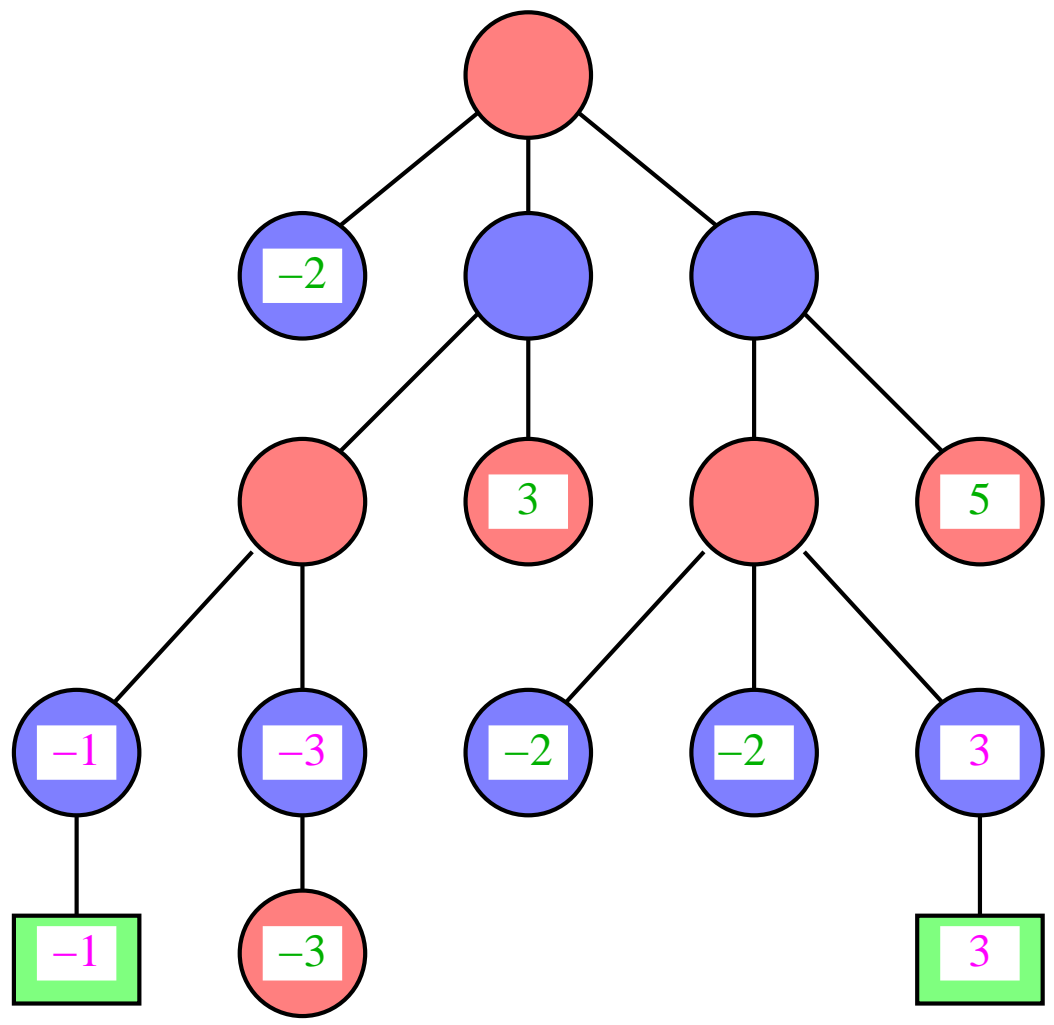
- Wir nehmen an, wir hätten eine halbwegs zuverlässige Bewertungsfunktion `advice`, d.h. es gibt Zahlen  $\alpha < 0 < \beta$  so dass für Konfigurationen `conf` gilt:
  - Ist `advice(conf) <  $\alpha$` , gewinnt **voraussichtlich** die Gegnerin;
  - Ist  `$\beta < \text{advice}(\text{conf})$` , gewinnt **voraussichtlich** das Programm.
- Zur Bestimmung unseres nächsten Zugs, betrachten wir sukzessive alle Nachfolger-Konfigurationen `conf`.
  - Ist  `$\beta < \text{advice}(\text{conf})$` , ist der Zug akzeptabel.

- Gibt es keinen akzeptablen Zug, betrachten wir rekursiv die Nachfolger aller Konfigurationen `conf`, für die  $\alpha < \text{advice}(\text{conf})$  .
- Für gegnerische Konfigurationen gehen wir dual vor ...

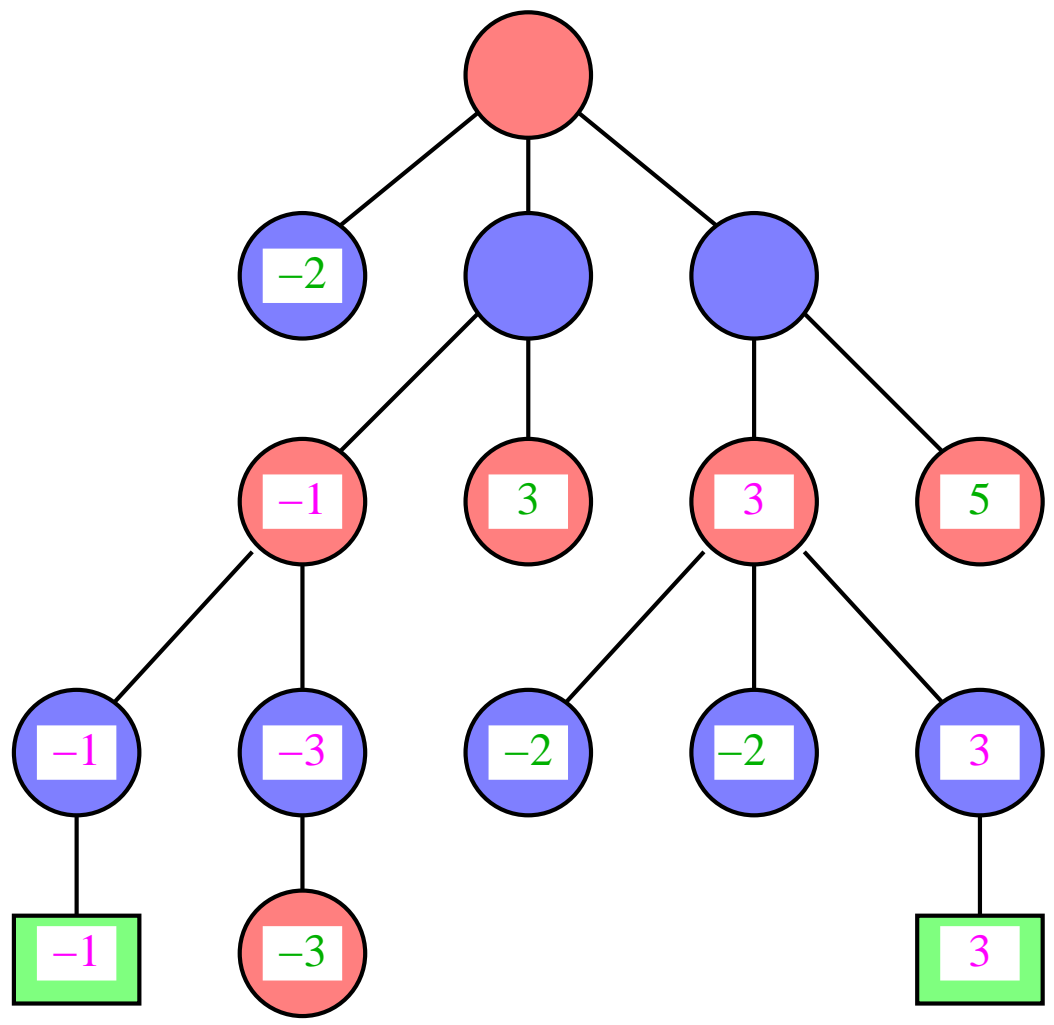


$\alpha = -1$

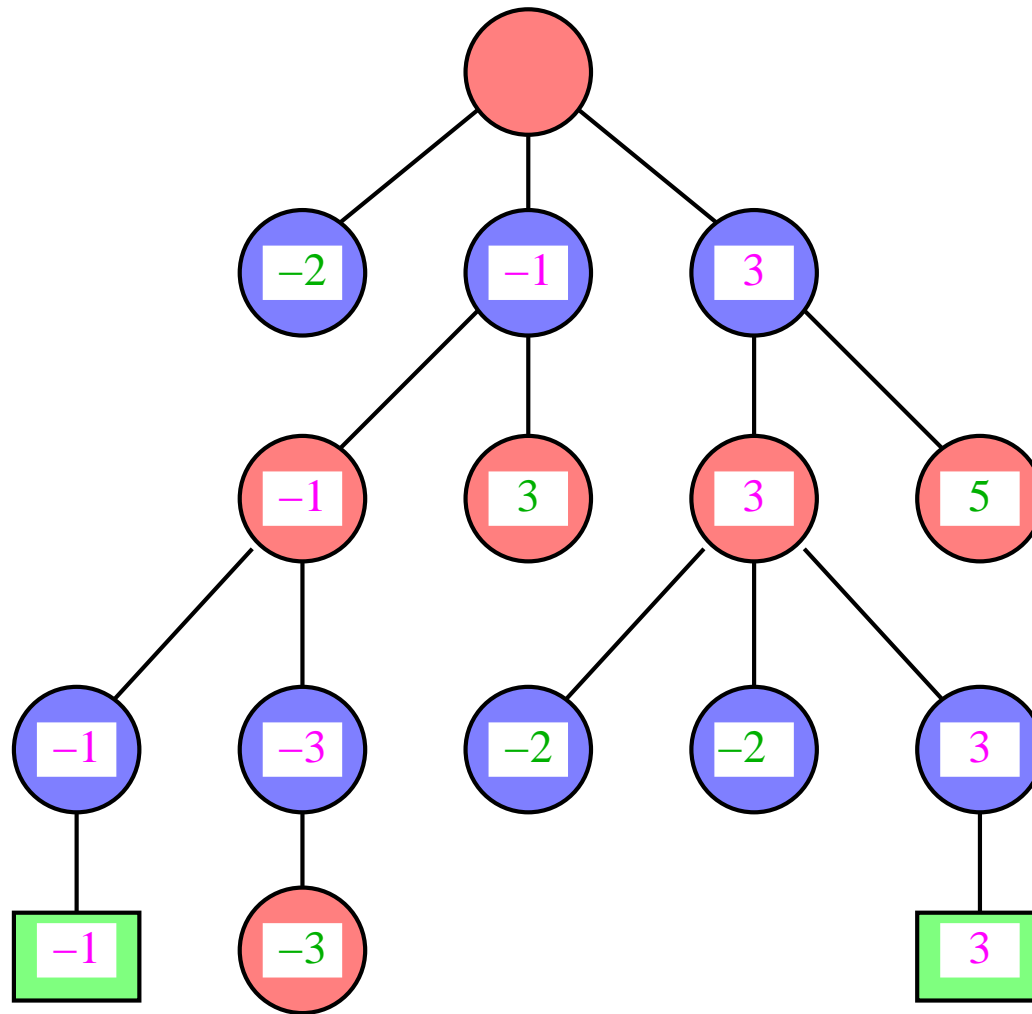
$\beta = 2$



$\alpha = -1$   
 $\beta = 2$

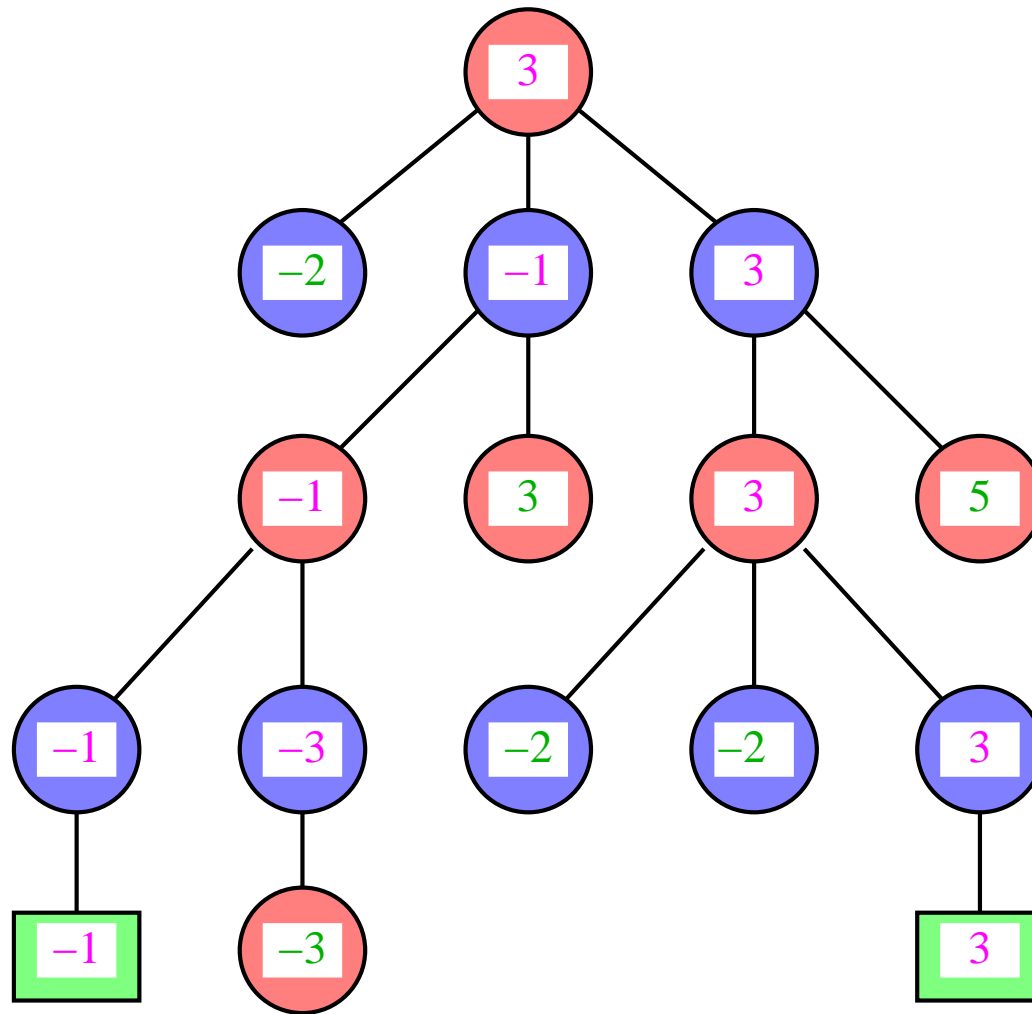


$\alpha = -1$   
 $\beta = 2$

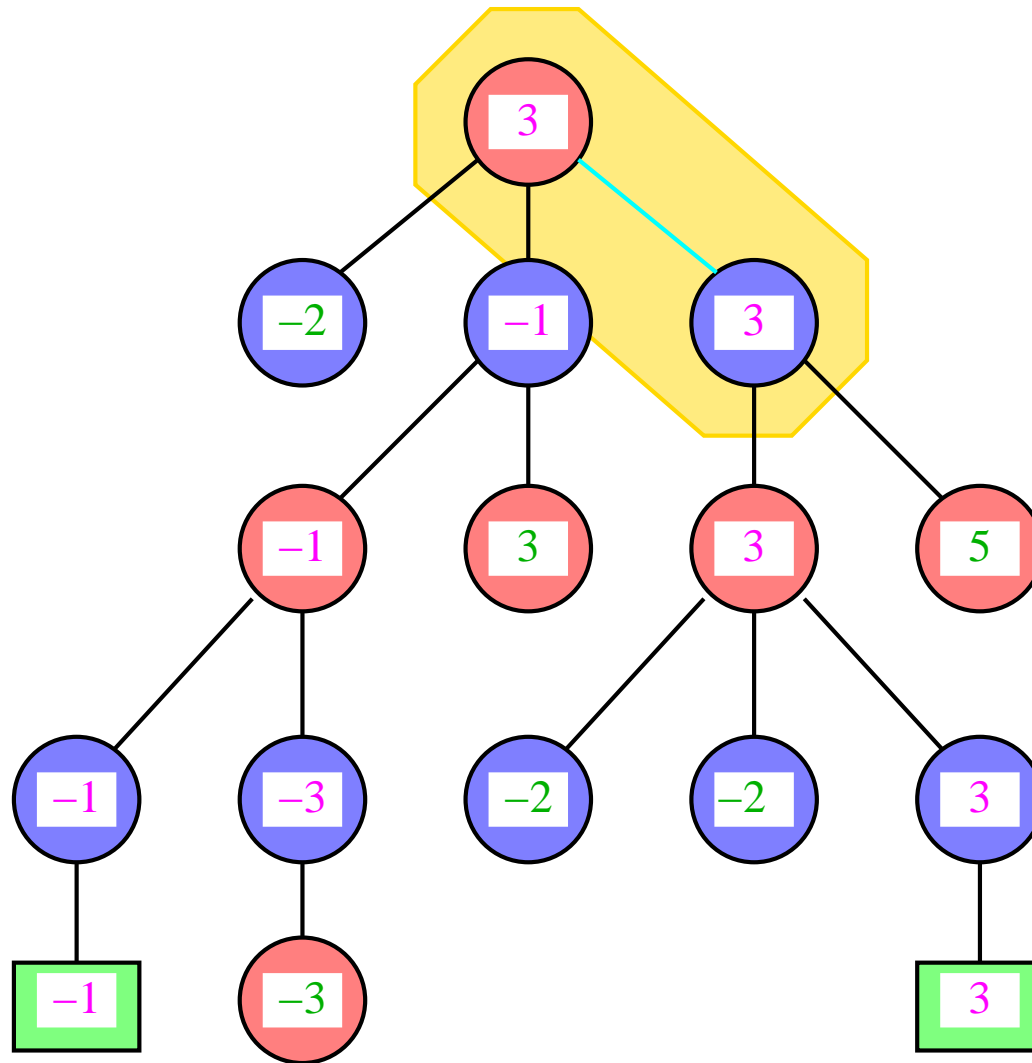


$\alpha = -1$

$\beta = 2$



$\alpha = -1$   
 $\beta = 2$



$\alpha = -1$   
 $\beta = 2$

## Vorteil:

Die Anzahl der zu untersuchenden Konfigurationen wird (hoffentlich ;-) beträchtlich eingeschränkt!

## Nachteil:

Ist die Bewertungsfunktion offensichtlich fehlerhaft, lässt sich das Programm austricksen ...

## Frage:

Wie findet man eine Bewertungsfunktion deren Fehlerhaftigkeit nicht so offensichtlich ist???

## Ausblick:

- Nicht alle 2-Personen-Spiele sind **endlich**.
- Gelegentlich hängt der Effekt eines Zugs zusätzlich vom **Zufall** ab.
- Eventuell ist die aktuelle Konfiguration nur **partiell bekannt**.

$\implies$  **Spieltheorie**

## 3 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 :-)

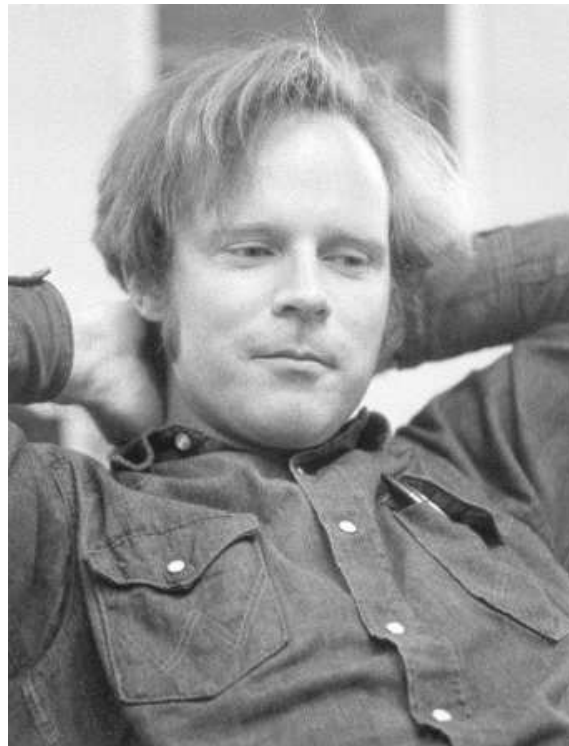
## Problem:

- 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** ...

## 3.1 Verifikation von Programmen



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

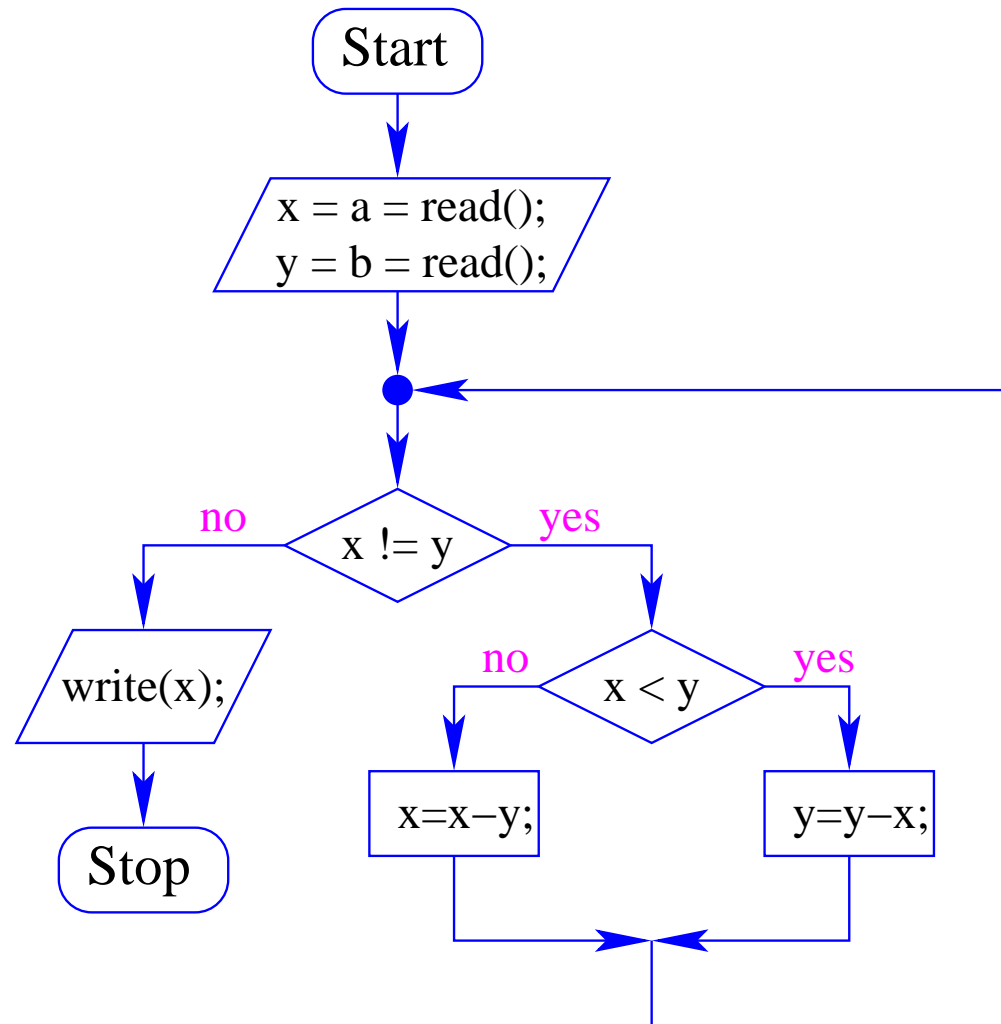
## Vereinfachung:

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:

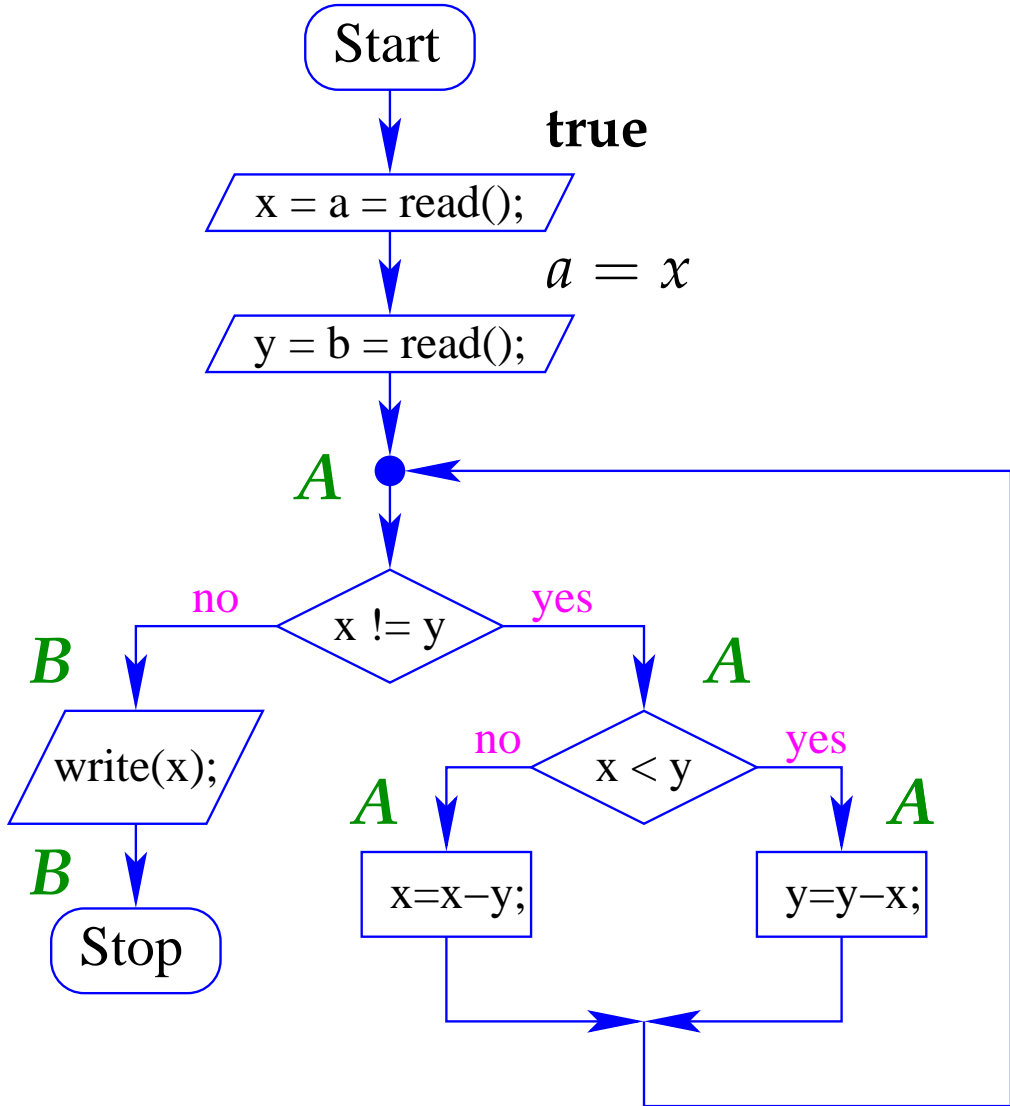
- Am Anfang gilt nix :-)
- Nach `a=read(); x=a;` gilt  $a = x$  :-)
- Vor Betreten und während der Schleife soll gelten:

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

- 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:	$x - y = 3 \quad :-)$