

Korrekte Software: Grundlagen und Methoden  
Vorlesung 1 vom 19.04.22  
Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2022

# Organisatorisches

## ▶ Veranstalter:



Serge Autexier  
serge.autexier@dfki.de  
Cartesium 1.49<sup>1</sup>, Tel. 59834



Christoph Lüth  
christoph.lueth@dfki.de  
MZH 4186, Tel. 59830

## ▶ Termine:

▶ Dienstag, 10 – 12, MZH 1450

▶ Donnerstag, 8 – 10, MZH 1450

▶ **Webseite:** <https://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss22>

# Organisatorisches

## ▶ Veranstalter:



Serge Autexier  
serge.autexier@dfki.de  
Cartesium 1.49<sup>1</sup>, Tel. 59834



Christoph Lüth  
christoph.lueth@dfki.de  
MZH 4186, Tel. 59830

## ▶ Termine:

- ▶ Dienstag, 10 – 12, MZH 1450
- ▶ Donnerstag, **8:30 – 10:00**, MZH 1450

▶ **Webseite:** <https://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss22>

# Veranstaltungskonzept

- ▶ Aus den letzten Jahren: **integrierte Veranstaltung** statt **langer Vorlesung**.
- ▶ Kürzere **Vortragseinheiten**, dazwischen **Arbeitsfragen** (Kurzübungen)
- ▶ Wöchentliche **Übungsaufgaben** zur Vertiefung
- ▶ Technisch:
  - ▶ Fragen/Kurzübungen in **HedgeDoc**: <http://hackmd.informatik.uni-bremen.de/>
  - ▶ Übungsblätter als **Markdown**, Abgabe über gitlab.

# Prüfungsform

- ▶ 10 Übungsblätter (geplant)
- ▶ **Bewertung:**
  - ▶ A (sehr gut, 1.3) — nichts zu meckern, keine/kaum Fehler
  - ▶ B (gut, 2.3) — kleine Fehler, sonst gut
  - ▶ C (befriedigend, 3.3) — größere Fehler oder Mängel
  - ▶ Nicht bearbeitet — oder mehr Fehler als Bearbeitung
- ▶ **Prüfungsleistung:**
  - ▶ **Mündliche Prüfung:** Einzelprüfung ca. 20– 30 Minuten
  - ▶ **Übungsbetrieb** (bis zu 15% Bonuspunkte, keine Voraussetzung)

# Übungsbetrieb

- ▶ Abgabe und Korrektur des Übungsbetriebs erfolgt über **gitlab**.
- ▶ Dazu legt **pro Gruppe** ein Repository an.
- ▶ Ladet uns (`clueth`, `autexier`) als Developer ein.
- ▶ Für jedes Übungsblatt:
  - 1 Das Übungsblatt ladet ihr von der Webseite herunter und bearbeitet es **elektronisch**.
  - 2 Die Lösung wird als Markdown abgelegt (bitte Namen `uebung-XX.md` nicht verändern; Zusatzmaterial als `uebung-XX-...` wenn nötig), und ladet es **vor** dem Abgabezeitpunkt hoch (`push`).
  - 3 Nach dem Abgabezeitpunkt laden wir die Änderungen herunter (`pull`), korrigieren direkt im Markdown, fügen die Bewertung hinzu, und laden die Korrektur wieder hoch (`push`)

# Arbeitsblatt 1.1: Jetzt seid ihr dran!

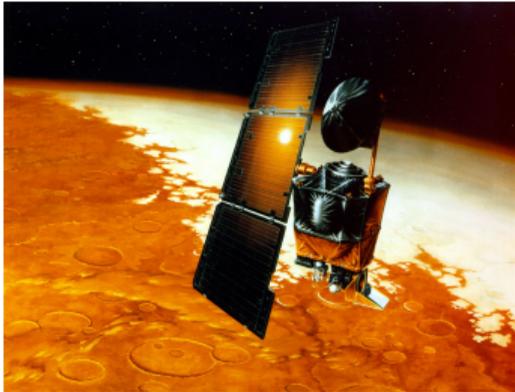
- ▶ Gruppiert euch in Gruppen zu drei Teilnehmenden!
- ▶ Zu jeder Gruppe gibt es ein Arbeitsblatt:  
<https://hackmd.informatik.uni-bremen.de/Ds7wVjXAQoiszJJPNJff0A#>
- ▶ Auf diesem Arbeitsblatt bearbeitet ihr die Arbeitsfragen im Laufe des Kurses.
- ▶ Bitte nur in “eurem” Arbeitsblatt arbeiten
- ▶ Die Arbeitsblätter sind nicht notenrelevant.

# I. Warum Korrekte Software?

# Software-Disaster I: Therac-25



# Software-Disasters II: Space



## Software-Disaster III: AT&T (15.01.1990)

```
while (! empty(ring_rcv_buffer)
      && ! empty(side_buffer empty)) {
  initialize pointer to first message buffer;
  get copy of buffer;
  switch (message) {
    case (incoming_message):
      if (sender is out_of_service) {
        if (empty(ring_wrt_buffer)) {
          send "in service" to status map;
        } else {
          break;
        }
      }
      process incoming message, set up pointers;
      break;
    }
  }
do optional parameter work;
}
```

# Software-Disaster IV: Ugeplantes Übergewicht



- ▶ „A software mistake caused a Tui flight to take off heavier than expected as female passengers using the title “Miss” were classified as children [...]“
- ▶ 38 erwachsene Passagiere als Kinder (35kg) statt als Erwachsene (69kg) klassifiziert.

$$38 \cdot (69 \text{ kg} - 35 \text{ kg}) = 1292 \text{ kg}$$

- ▶ Software „was programmed in an unnamed foreign country where the title “Miss” is used for a child and “Ms” for an adult female.“

Quelle: *Guardian*, 09.04.2021.

<https://www.theguardian.com/world/2021/apr/09/tui-plane-serious-incident-every-miss-on-board-child-weight-birmingham-majorca>

## Arbeitsblatt 1.2: Jetzt seid ihr dran!

- ▶ Sucht im Netz nach weiteren Software-Disastern:
  - ① Was ist passiert?
  - ② Wie ist es passiert?
  - ③ Was war der Softwarefehler?
- ▶ Quellen: Suchmaschine nach Wahl (“software disasters”), The Risks Digest, <https://catless.ncl.ac.uk/Risks/>

# II. Inhalt der Vorlesung

# Themen



Korrekte Software im Lehrbuch:

- ▶ Spielzeugsprache
- ▶ Wenig Konstrukte
- ▶ Kleine Beispiele



Korrekte Software im Einsatz:

- ▶ Richtige Programmiersprache
- ▶ Mehr als nur ganze Zahlen
- ▶ Skalierbarkeit — wie können große Programme verifiziert werden?

# Inhalt

- ▶ Grundlagen:
  - ▶ Beweis der **Korrektheit** von Programmen: der **Floyd-Hoare-Kalkül**
  - ▶ **Bedeutung** von Programmen: **Semantik**
- ▶ Betrachtete Programmiersprache: “C0” (erweiterte Untermenge von C)
- ▶ Erweiterung der Programmkonstrukte und des Hoare-Kalküls:
  - 1 Referenzen (Zeiger)
  - 2 Funktion und Prozeduren (Modularität)
  - 3 Reiche **Datenstrukturen** (Felder, struct)

# Fahrplan

- ▶ Einführung
- ▶ Operationale Semantik
- ▶ Denotationale Semantik
- ▶ Äquivalenz der Operationalen und Denotationalen Semantik
- ▶ Der Floyd-Hoare-Kalkül I
- ▶ Der Floyd-Hoare-Kalkül II: Invarianten
- ▶ Korrektheit des Floyd-Hoare-Kalküls
- ▶ Strukturierte Datentypen
- ▶ Verifikationsbedingungen
- ▶ Vorwärts mit Floyd und Hoare
- ▶ Funktionen und Prozeduren I
- ▶ Funktionen und Prozeduren II
- ▶ Referenzen und Speichermodelle
- ▶ Ausblick und Rückblick

# III. Warum Semantik?

# Idee

- ▶ Was wird hier berechnet?

```
p= 1;  
c= 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1;  
}
```

# Idee

- ▶ Was wird hier berechnet?  $p = n!$
- ▶ Warum? Wie können wir das **beweisen**?

```
p= 1;  
c= 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1;  
}
```

# Idee

- ▶ Was wird hier berechnet?  $p = n!$
- ▶ Warum? Wie können wir das **beweisen**?
- ▶ Wir berechnen symbolisch, welche Werte Variablen über den Programmverlauf annehmen.

```
p= 1;  
c= 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1;  
}
```

# Semantik von Programmiersprachen

Drei wesentliche Möglichkeiten:

- ▶ **Operationale Semantik:** Ausführung auf einer **abstrakten** Maschine
- ▶ **Denotationale Semantik:** Abbildung in ein **mathematisches Objekt**
- ▶ **Axiomatische Semantik:** Beschreibung anhand der **Eigenschaften**

# Arbeitsblatt 1.3: Maschinen und Funktionen

Was genau kann man sich unter “abstrakten Maschine” vorstellen?

Betrachtet als Beispiele:

- ▶ Eine Taschenlampe
- ▶ Eine Waschmaschine
- ▶ Einen Taschenrechner

Was ist hier die Abstraktion?

# Unsere Sprache C0

- ▶ C0 ist eine **Untermenge** der Sprache C
- ▶ C0-Programme sind **ausführbare** C-Programme
- ▶ Grundausbaustufe:
  - ▶ Zuweisungen, Fallunterscheidungen, Schleifen
  - ▶ Datentypen: ganze Zahlen mit Arithmetik
  - ▶ Relationen: Vergleich ( $=$ ,  $\leq$ )
  - ▶ Boolesche Operatoren: Konjunktion, Disjunktion, Negation
- ▶ 1. Ausbaustufe: Felder und Strukturen
- ▶ 2. Ausbaustufe: Funktionen und Prozeduren (nur Ausblick)
- ▶ 3. Ausbaustufe: Referenzen (nur Ausblick)
- ▶ Fehlt: **union**, **goto**, ...

# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

```
p = 1;  
c = 1;  
while (c <= n) {  
  p = p * c;  
  c = c + 1; }  
}
```

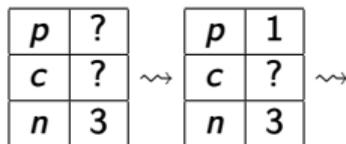
$p$	?
$c$	?
$n$	3

↔

# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

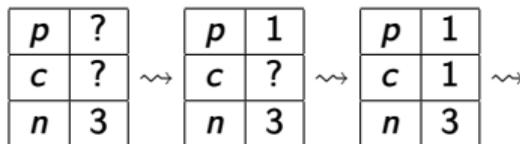
```
p = 1;  
c = 1;  
while (c <= n) {  
  p = p * c;  
  c = c + 1; }  
}
```



# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

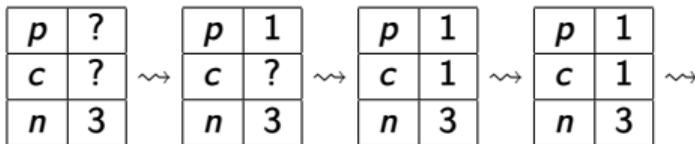
```
p = 1;  
c = 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1; }  
}
```



# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

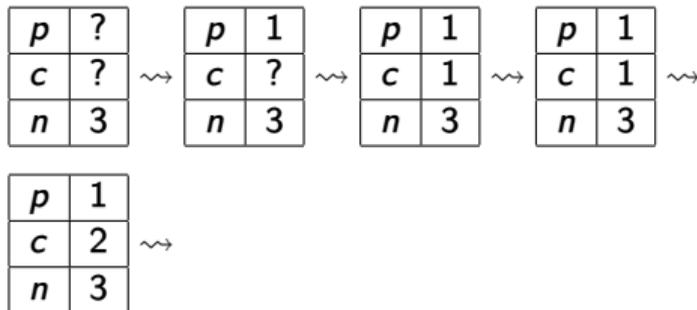
```
p = 1;  
c = 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1;  
}
```



# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

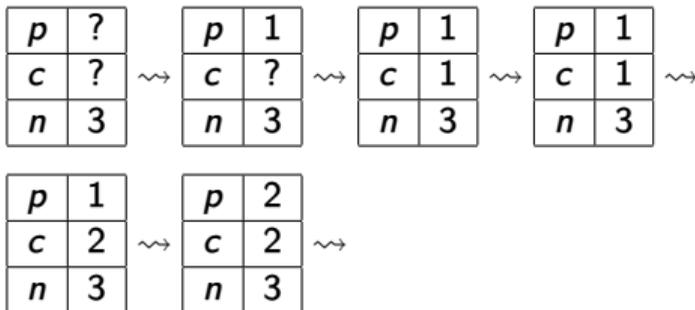
```
p = 1;  
c = 1;  
while (c <= n) {  
  p = p * c;  
  c = c + 1;  
}
```



# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

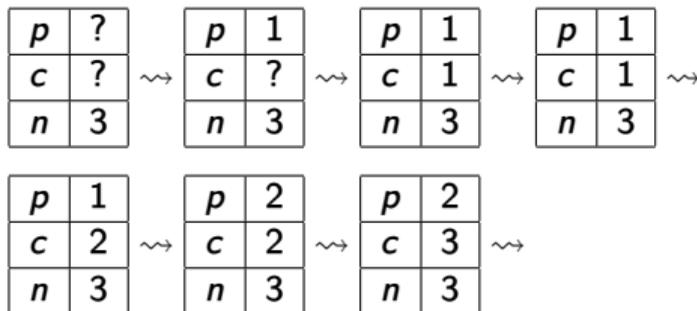
```
p = 1;  
c = 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1;  
}
```



# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

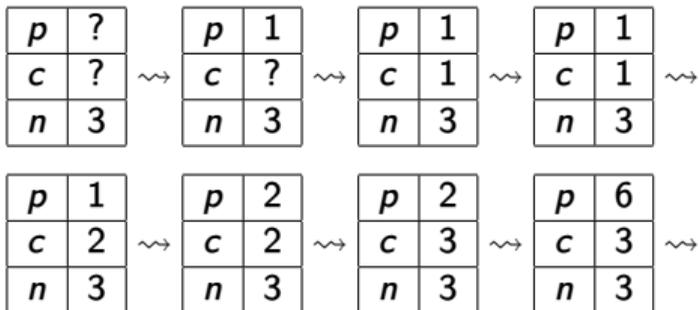
```
p = 1;  
c = 1;  
while (c <= n) {  
    p = p * c;  
    c = c + 1;  
}
```



# Operationale Semantik

- ▶ Kernkonzept: Zustandsübergänge einer abstrakten Maschine
- ▶ Abstrakte Maschine hat **impliziten Zustand**
- ▶ Zustand ordnet **Adressen** veränderliche **Werte** zu
- ▶ Konkretes Beispiel:  $n \mapsto 3$ ,  $p$  und  $c$  undefiniert

```
p = 1;  
c = 1;  
while (c <= n) {  
  p = p * c;  
  c = c + 1;  
}
```





# Arbeitsblatt 1.4: Operationale Semantik

Gegeben folgendes C0-Programm:

```
1 x= 0;  
2 while (n > 0) {  
3   x= x+ n*n;  
4   n= n-1;  
5 }
```

Entwickeln Sie die ersten zehn Schritte der operationalen Semantik wie im Beispiel oben für den initialen Zustand

$n$	4
$x$	?

$\rightsquigarrow \dots$

# Denotationale Semantik

- ▶ Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- ▶ **Partielle** Funktionen zwischen Zuständen  $\llbracket c \rrbracket : \sigma \rightarrow \sigma$
- ▶ Beispiel:

```
p = 1;  
c = 1; // p1  
while (c <= n) {  
    p = p * c;  
    c = c + 1; // p2  
}  
// p3
```

$$\llbracket p_1 \rrbracket(\sigma) = \sigma[p \mapsto 1][c \mapsto 1]$$

$$\llbracket p_2 \rrbracket(\sigma) = \sigma[p \mapsto \sigma(p) * \sigma(c)][c \mapsto \sigma(c) + 1]$$

$$\llbracket p_3 \rrbracket(\sigma) = ???$$

# Denotationale Semantik

- ▶ Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- ▶ **Partielle** Funktionen zwischen Zuständen  $\llbracket c \rrbracket : \sigma \rightarrow \sigma$
- ▶ Beispiel:

```
p = 1;  
c = 1; // p1  
while (c <= n) {  
    p = p * c;  
    c = c + 1; // p2  
}  
// p3
```

$$\llbracket p_1 \rrbracket(\sigma) = \sigma[p \mapsto 1][c \mapsto 1]$$

$$\llbracket p_2 \rrbracket(\sigma) = \sigma[p \mapsto \sigma(p) * \sigma(c)][c \mapsto \sigma(c) + 1]$$

$$\llbracket p_3 \rrbracket(\sigma) = ???$$

$$\Gamma(\llbracket c \leq n \rrbracket)(\llbracket p_2 \rrbracket)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } \llbracket c \leq n \rrbracket(\sigma) = 0 \\ (\varphi \circ \llbracket p_2 \rrbracket)(\sigma) & \text{if } \llbracket c \leq n \rrbracket(\sigma) = 1 \end{cases}$$

# Denotationale Semantik

- ▶ Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- ▶ **Partielle** Funktionen zwischen Zuständen  $\llbracket c \rrbracket : \sigma \rightarrow \sigma$
- ▶ Beispiel:

```
p = 1;  
c = 1; // p1  
while (c <= n) {  
  p = p * c;  
  c = c + 1; // p2  
}  
// p3
```

$$\llbracket p_1 \rrbracket(\sigma) = \sigma[p \mapsto 1][c \mapsto 1]$$

$$\llbracket p_2 \rrbracket(\sigma) = \sigma[p \mapsto \sigma(p) * \sigma(c)][c \mapsto \sigma(c) + 1]$$

$$\llbracket p_3 \rrbracket(\sigma) = \text{fix}(\Gamma(\llbracket c \leq n \rrbracket)(\llbracket p_2 \rrbracket))(\llbracket p_1 \rrbracket(\sigma))$$

$$\Gamma(\llbracket c \leq n \rrbracket)(\llbracket p_2 \rrbracket)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } \llbracket c \leq n \rrbracket(\sigma) = 0 \\ (\varphi \circ \llbracket p_2 \rrbracket)(\sigma) & \text{if } \llbracket c \leq n \rrbracket(\sigma) = 1 \end{cases}$$

$$\Gamma(\beta)(\rho)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } \beta(\sigma) = 0 \\ (\varphi \circ \rho)(\sigma) & \text{if } \beta(\sigma) = 1 \end{cases}$$

# Denotationale Semantik

- ▶ Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- ▶ **Partielle** Funktionen zwischen Zuständen  $\llbracket c \rrbracket : \sigma \rightarrow \sigma$
- ▶ Beispiel:

```
p = 1;  
c = 1; // p1  
while (c <= n) {  
    p = p * c;  
    c = c + 1; // p2  
}  
// p3
```

$$\llbracket p_1 \rrbracket(\sigma) = \sigma[p \mapsto 1][c \mapsto 1]$$

$$\llbracket p_2 \rrbracket(\sigma) = \sigma[p \mapsto \sigma(p) * \sigma(c)][c \mapsto \sigma(c) + 1]$$

$$\llbracket p_3 \rrbracket = \text{fix}(\Gamma(\llbracket c \leq n \rrbracket)(\llbracket p_2 \rrbracket)) \circ \llbracket p_1 \rrbracket$$

$$\Gamma(\llbracket c \leq n \rrbracket)(\llbracket p_2 \rrbracket)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } \llbracket c \leq n \rrbracket(\sigma) = 0 \\ (\varphi \circ \llbracket p_2 \rrbracket)(\sigma) & \text{if } \llbracket c \leq n \rrbracket(\sigma) = 1 \end{cases}$$

$$\Gamma(\beta)(\rho)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } \beta(\sigma) = 0 \\ (\varphi \circ \rho)(\sigma) & \text{if } \beta(\sigma) = 1 \end{cases}$$

# Axiomatische Semantik

- ▶ Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- ▶ Zusicherungen sind zustandsabhängige Prädikate
- ▶ Beispiel (mit  $n = 3$ )

```
// (1)
p = 1; // (2)
c = 1; // (3)
while (c <= n) {
  // (4)
  p = p * c;
  c = c + 1;
  // (5)
}
// (6)
```

- (1)  $n = 3$
- (2)  $p = 1 \wedge n = 3$
- (3)  $p = 1 \wedge c = 1 \wedge n = 3$
- (4) ???
- (5)
- (6)  $p = 6 \wedge c = 4 \wedge n = 3$

# Axiomatische Semantik

- ▶ Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- ▶ Zusicherungen sind zustandsabhängige Prädikate
- ▶ Beispiel (mit  $n = 3$ )

```
// (1)
p = 1; // (2)
c = 1; // (3)
while (c <= n) {
  // (4)
  p = p * c;
  c = c + 1;
  // (5)
}
// (6)
```

- (1)  $n = 3$
- (2)  $p = 1 \wedge n = 3$
- (3)  $p = 1 \wedge c = 1 \wedge n = 3$
- (4)  $(p = 1 \wedge c = 1 \vee p = 1 \wedge c = 2 \vee p = 2 \wedge c = 3) \wedge n = 3$
- (5)  $(p = 1 \wedge c = 2 \vee p = 2 \wedge c = 3 \vee p = 6 \wedge c = 4) \wedge n = 3$
- (6)  $p = 6 \wedge c = 4 \wedge n = 3$

# Axiomatische Semantik

- ▶ Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- ▶ Zusicherungen sind zustandsabhängige Prädikate
- ▶ Beispiel (mit  $n = 3$ )

```
// (1)
p = 1; // (2)
c = 1; // (3)
while (c <= n) {
  // (4)
  p = p * c;
  c = c + 1;
  // (5)
}
// (6)
```

- (1)  $n = 3$
- (2)  $p = 1 \wedge n = 3$
- (3)  $p = 1 \wedge c = 1 \wedge n = 3$
- (4)  $p = (c - 1)! \wedge c \leq n \wedge n = 3$
- (5)  $p = (c - 1)! \wedge n = 3$
- (6)  $p = 6 \wedge c = 4 \wedge n = 3$

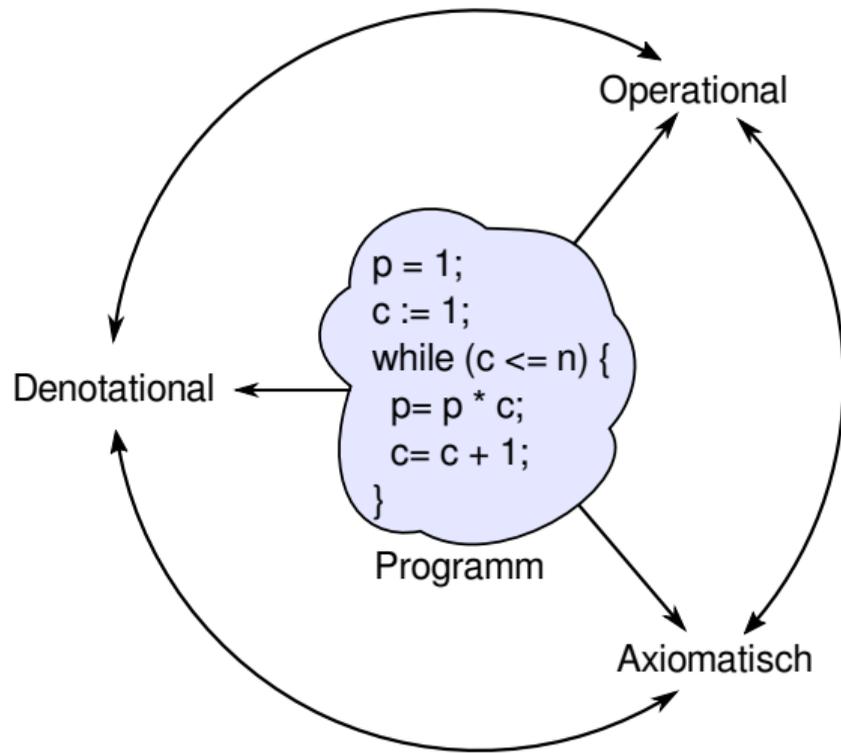
# Arbeitsblatt 1.5: Zusicherungen

Betrachten Sie folgende Variation des Programms von oben:

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
  // (4)
  c = c + 1;
  p = p * c;
}
// (5)
```

- ▶ Welche der Zusicherungen (1) – (5) von oben gelten noch?
- ▶ Welche nicht?
- ▶ Was gilt stattdessen?

# Drei Semantiken — Eine Sicht



# Zusammenfassung

- ▶ Wir wollen die **Bedeutung** (Semantik) von Programmen beschreiben, um ihre Korrektheit beweisen zu können.
- ▶ Dazu gibt es verschiedene Ansätze, die wir betrachten werden.
- ▶ Nächste Woche geht es mit dem ersten los: **operationale** Semantik