

# Formale Methoden der Informatik

Prof. Dr. Karsten Wolf

# Teil I

# Semantik von Programmiersprachen

Prof. Dr. Karsten Wolf

# Schwerpunkt: *Programmiersprachen*

- Syntax = Struktur von Sätzen der Sprache  
(Theoretische Grundlagen der Informatik II: Grammatiken, Automaten)
- **Semantik = Bedeutung von Sätzen der Sprache**
- Pragmatik = Verwendung der Sprachen  
(in den Anwendungsgebieten)

# Wozu Semantik?

- Programme abarbeiten
- Programme erstellen
- Programme ändern/austauschen
- Programme übersetzen
- Programme mit Spezifikation vergleichen

# Wozu **formale** Semantik?

- Programme **automatisch** abarbeiten
  - Interpreter
- Programme **automatisch** erstellen
  - Konstruktion
- Programme **automatisch** ändern/austauschen
  - Transformation, Substitution, Optimierung
- Programme **automatisch** übersetzen
  - Compiler
- Programme **automatisch** mit Spezifikation vergleichen
  - Verifikation

# Außerdem

- Informale Semantiken (Standards, Reports, ...) sind
  - unvollständig
  - missverständlich
  - fehlerhaft
- Folgen:
  - Portabilität als Problem
  - Schwierigkeiten beim Ablösen veralteter Programmiersprachen
  - Unmöglichkeit seriöser Verifikation
  - ...
- Formalisierung hat schon oft solche Probleme aufgedeckt

## Nur mit formaler Semantik möglich:

- Beweisbar korrekte Interpreter/Compiler
- Beweisbar korrekte Programmoptimierung
- Konstruktion, Verifikation

Und schließlich...

Informatik = Syntax + Semantik + Pragmatik

Wir manipulieren Symbole, denken uns etwas dabei und verfolgen einen Zweck...



# Fahrplan

- Kapitel 1: Eine einfache Programmiersprache:

*ermöglicht Vergleich verschiedener Semantikarten*

# Fahrplan

- 2. Kapitel: Operationelle Semantik

*Wie (in welchen Schritten) entsteht der Effekt eines Programms?*

- Maschinensemantik
- Natürliche Semantik (Big-Step-Semantik)
- Strukturelle operationelle Semantik (Small-Step-Semantik)

Einsatz z.B. : Beweisbar korrekte Übersetzung

# Fahrplan

- 3. Kapitel: Denotationelle Semantik

*Was ist der Effekt eines Programms?*

Funktion, die Eingaben eine Ausgabe zuordnet

Einsatz z.B. : Programmoptimierung

Abstract Interpretation = Informationsbeschaffung zur  
Compilezeit

# Fahrplan

- 4. Kapitel: Axiomatische Semantik

*Welche Aussagen kann ich über den Effekt eines Programms beweisen?*

Logischer Kalkül

Einsatz z.B. : Programmverifikation

# Kapitel 1

## Eine einfache Programmiersprache

# W

```
cmd_seq      = command | cmd_seq „;“ command.
command      = „identifier“ „:=“ expression | „skip“
              | „while“ expression „do“ cmd_seq „end“
              | „if“ expression „then“ cmd_seq „end“
              | „if“ expression „then“ cmd_seq „else“
                cmd_seq „end“.
expression   = term | expression („+“ | „-“ | „OR“) term.
term         = factor | term („*“ | „/“ | „AND“) factor.
factor       = „NOT“ factor | „number“ | „identifier“
              | „true“ | „false“ | „(“ expression „)“
              | expression („<“ | „>“ | „<=“ | „>=“ | „<>“ | „=“)
                expression
```

# Abstrakte Syntax; Syntaxbaum

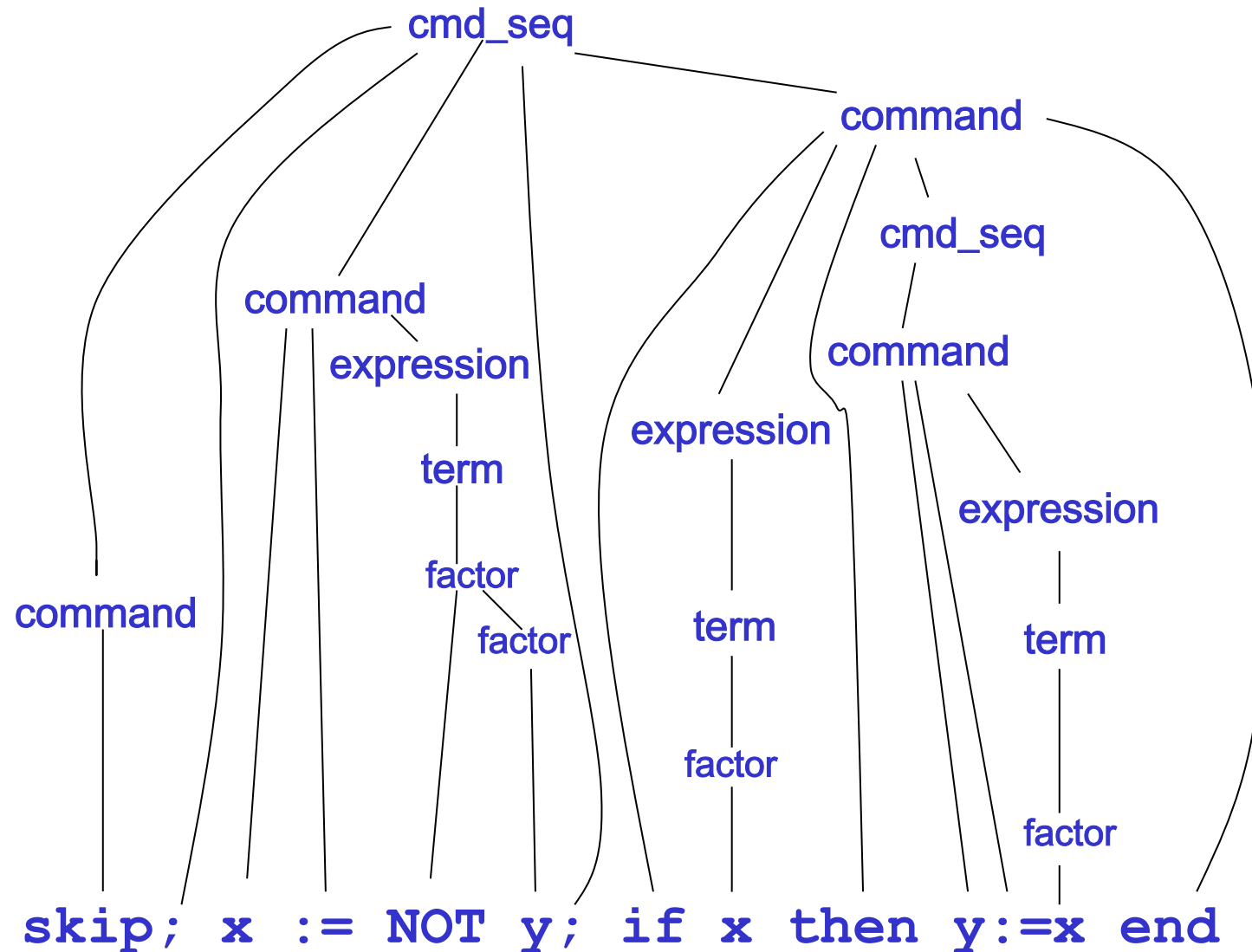
= Ableitungsbaum mit einigen Vereinfachungen, z.B.

- Verzicht auf struktursichernde Elemente: (,),;, end...
- Vereinheitlichung, z.B.

**if B then S end → if B then S else skip end**

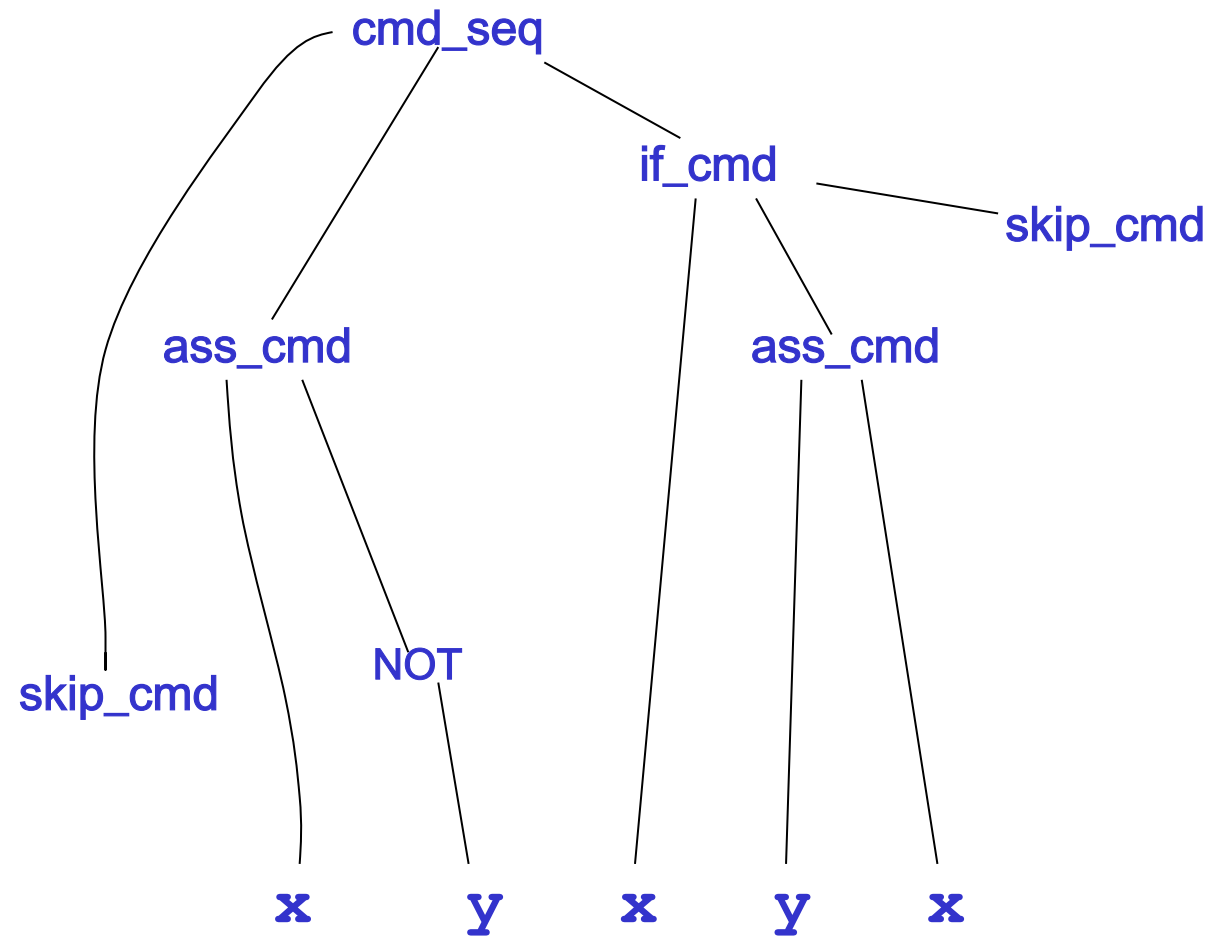
- Operanden in den Wurzelknoten von Expressions
- Listenstrukturen (z.B. cmd\_seq) einebnen
- Umbenennungen (z.B. Expression → Term → Faktor)  
überspringen
- ...

# Beispiel: Ableitungsbaum (schon etwas vereinfacht)





# Beispiel: Syntaxbaum



# Ausdruckstypen

Typ einer Expression:

- **integer**
  - Variable
  - Number
  - Expression mit Kopf +, -, \*, /
- **boolean**
  - true, false
  - Expression mit Kopf OR, AND, NOT, <, >, <=, >=, <>, =
- Vereinbarung (für Einfachheit): Typen von Bedingungen
- (IF, WHILE) sei **boolean**; Typ von Anweisung sei **integer**

# Kapitel 2

## Operationelle Semantik

# Idee

- Bedeutung eines Programms wird angegeben durch Transformation in ein Programm einer anderen, als bekannt vorausgesetzten Sprache/Maschine

# Varianten

- Maschinensemantik: Übersetzung in Code einer konkreten Maschine

*Jeder Compiler; schwach wegen Maschinenbesonderheiten*

- Abstrakte Maschinensemantik: Übersetzung in Code einer abstrakten Maschine

*WAM (Prolog), JVM (Java), P-Code (Pascal), Lilith/M-Code (Modula-2), .NET, ....; Pragmatisch, für Portabilität, einige nur durch informale Standards festgelegt*

- Selbstdefinition: Ein Interpreter für Sprache X, geschrieben in X

*LISP, PROLOG; informal ok, formal unbrauchbar*

- Structural Operational Semantics: Übersetzung in ein Automatenmodell (Small step)

*Zustand = Variablenbelegungen etc. Schritt = elementare Zustandstransformation durch Statement*

- Natural Semantics: dasselbe (Big Step)

*Zustand = Variablenbelegungen etc. Schritt = Zustandstransformation durch Statement*

# Vorbereitung: Betrachten W

- Zustand: ein Wert für jede Variable
- Typ: Semantik eines Typs  $t$  ist ein Wertebereich (Menge)  $WB(t)$ 
  - $WB(\text{integer}) := \{0, 1, -1, 2, -2, 3, -3, \dots\}$   $WB(\text{bool}) := \{tt, ff\}$
- Expression: wird interpretiert auf einem Zustand (liefert Wert), ändert Zustand selbst nicht
- Command: Ändert Zustand, liefert keinen Wert

# Zustand in W

- Formal: Ein Zustand  $s$  ist eine Abbildung
  - Definitionsbereich ist  $V$  (Menge der Variablen des Programms)
  - Für  $v \in V$  ist  $s(v)$  aus  $WB(\text{integer})$

Abstrahiert von Implementationsdetails wie

- Speicheradresse
- Binärkodierung

# Semantik einer Expression $E$

... ist eine Abbildung  $\text{States} \rightarrow \text{WB}(\text{typ}(E))$

In  $W$ : Zu einer arithmetischen Expression sei  $A \llbracket E \rrbracket$  ,  
zu einer booleschen Expression sei  $B \llbracket E \rrbracket$   
ihre Semantik,

also:  $A: \text{AExp} \rightarrow (\text{State} \rightarrow \text{WB}(\text{integer}))$   
 $B: \text{BExp} \rightarrow (\text{State} \rightarrow \text{WB}(\text{boolean}))$

Definition der Semantik: strukturelle Induktion über der Syntax  
einer Expression



# Semantik einer Expression E

Sei s Zustand.

$A \llbracket n \rrbracket (s) = \text{value}(n)$ , falls n *number* (vertiefen wir nicht weiter)

$A \llbracket x \rrbracket (s) = s(x)$ ,

$A \llbracket E1+E2 \rrbracket (s) = A \llbracket E1 \rrbracket (s) + A \llbracket E2 \rrbracket (s)$

$A \llbracket E1-E2 \rrbracket (s) = A \llbracket E1 \rrbracket (s) - A \llbracket E2 \rrbracket (s)$

$A \llbracket E1 * E2 \rrbracket (s) = A \llbracket E1 \rrbracket (s) * A \llbracket E2 \rrbracket (s)$

$A \llbracket E1/E2 \rrbracket (s) = \lfloor A \llbracket E1 \rrbracket (s) : A \llbracket E2 \rrbracket (s) \rfloor$

$B \llbracket \text{true} \rrbracket (s) = \text{tt}$

$B \llbracket \text{false} \rrbracket (s) = \text{ff}$

$B \llbracket E1 \text{ OR } E2 \rrbracket (s) = \text{ff}$ , falls  $B \llbracket E1 \rrbracket (s) = B \llbracket E2 \rrbracket (s) = \text{ff}$ , sonst tt

$B \llbracket E1 \text{ AND } E2 \rrbracket (s) = \text{tt}$ , falls  $B \llbracket E1 \rrbracket (s) = B \llbracket E2 \rrbracket (s) = \text{tt}$ , sonst ff

$B \llbracket \text{NOT } E \rrbracket (s) = \text{tt}$ , falls  $B \llbracket E \rrbracket (s) = \text{ff}$ , sonst ff

$B \llbracket E1 < E2 \rrbracket (s) = \text{tt}$ , falls  $A \llbracket E1 \rrbracket (s) < A \llbracket E2 \rrbracket (s)$ , sonst ff

analog:  $>$ ,  $>=$ ,  $<=$ ,  $=$ ,  $<>$

## 2.1 Natural Semantics (NatS)

- Es geht nun um *commands*.
- Idee: Relation  $\langle S, s \rangle \rightarrow s'$ 
  - „Gestartet in Zustand  $s$ , terminiert Command  $S$  und führt zu Zustand  $s'$ “
- Formulierung der Definitionen
  - (a) Axiome, z.B.  $\langle \text{skip}, s \rangle \rightarrow s$
  - (b) Regeln , z.B.

*Prämisse(n)*

$\langle S2, s \rangle \rightarrow s'$

$\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \rightarrow s'$

*Folgerung*

*Bedingung*

falls  $B \llbracket b \rrbracket (s) = \text{ff}$

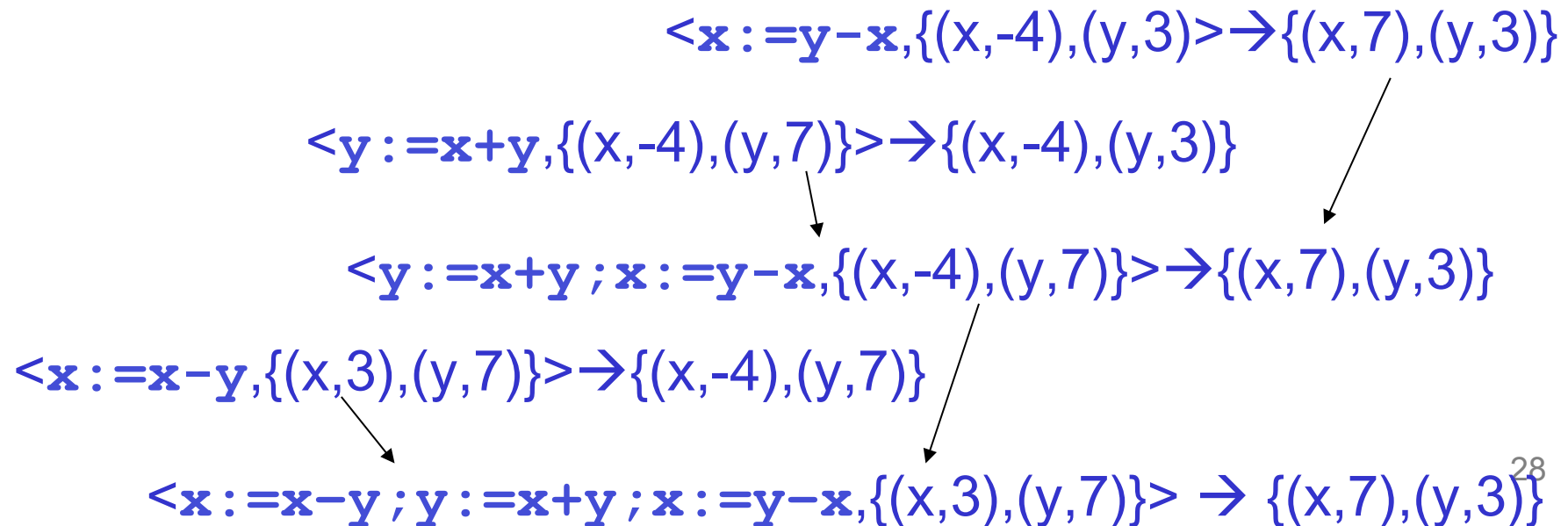
# Natural Semantics für W

- $[\text{skip}_{\text{NatS}}]$   $\langle \text{skip}, s \rangle \rightarrow s$ 

$s[x \rightarrow k](x) = k$   
 $s[x \rightarrow k](y) = s(y) \quad (y \neq x)$
- $[\text{ass}_{\text{NatS}}]$   $\langle x := E, s \rangle \rightarrow s[x \rightarrow A \llbracket E \rrbracket (s)]$
- $[\text{comp}_{\text{NatS}}]$  
$$\frac{\langle S1, s \rangle \rightarrow s' \quad \langle S2, s' \rangle \rightarrow s''}{\langle S1 ; S2, s \rangle \rightarrow s''}$$
- $[\text{if1}_{\text{NatS}}]$  
$$\frac{\langle S1, s \rangle \rightarrow s' \quad \text{falls } B \llbracket b \rrbracket (s) = \text{tt}}{\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \rightarrow s'}$$
- $[\text{if2}_{\text{NatS}}]$  
$$\frac{\langle S2, s \rangle \rightarrow s' \quad \text{falls } B \llbracket b \rrbracket (s) = \text{ff}}{\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \rightarrow s'}$$
- $[\text{while1}_{\text{NatS}}]$  
$$\frac{\langle S1, s \rangle \rightarrow s' \quad \langle \text{while } b \text{ do } S1 \text{ end}, s' \rangle \rightarrow s'' \quad \text{falls } B \llbracket b \rrbracket (s) = \text{tt}}{\langle \text{while } b \text{ do } S1 \text{ end}, s \rangle \rightarrow s''}$$
- $[\text{while2}_{\text{NatS}}]$  
$$\langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s \quad \text{falls } B \llbracket b \rrbracket (s) = \text{ff}$$

# Zielzustand ermitteln

- $x := x - y ; y := x + y ; x := y - x$
- Startzustand:  $s(x) = 3, s(y) = 7$
- ges: Zielzustand
- Weg: Folgerungsbaum aus Regel/Axiom-*Instanzen*



# Semantische Äquivalenz nachweisen

- wollen zeigen:
  - `while b do S end` ist semantisch äquivalent zu
  - `if b then S; while b do S end else skip end`
- Formal: Für alle  $s, s'$ :
  - $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$  genau dann, wenn
  - $\langle \text{if } b \text{ then } S; \text{while } b \text{ do } S \text{ end else skip end}, s \rangle \rightarrow s'$

# Beweis: Implikation

- Es gelte  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$ .
- Erster Fall: Es gelte  $B \llbracket b \rrbracket (s) = \text{tt}$ .
  - $\rightarrow [\text{while1}_{\text{NatS}}]:$  es gibt  $s''$  mit  $\langle S, s \rangle \rightarrow s''$  und  $\langle \text{while } b \text{ do } S \text{ end}, s'' \rangle \rightarrow s'$
  - $\rightarrow [\text{comp}_{\text{NatS}}]: \langle S; \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$
  - $\rightarrow [\text{if1}_{\text{NatS}}]: \langle \text{if } b \text{ then } S; \text{while } b \text{ do } S \text{ end else skip end}, s \rangle \rightarrow s'$
- Zweiter Fall: Es gelte  $B \llbracket b \rrbracket (s) = \text{ff}$ .
  - $\rightarrow [\text{while2}_{\text{NatS}}]: s = s'$
  - Gleiches Resultat liefert  $[\text{skip}_{\text{NatS}}]: \langle \text{skip}, s \rangle \rightarrow s$  und folglich
  - $[\text{if2}_{\text{NatS}}]: \langle \text{if } b \text{ then } S; \text{while } b \text{ do } S \text{ end else skip end}, s \rangle \rightarrow s'$ .

# Beweis: Replikation

- Es gelte  $\langle \text{if } b \text{ then } S; \text{ while } b \text{ do } S \text{ end else skip end}, s \rangle \rightarrow s'$ .
- Erster Fall: Es gelte  $B \llbracket b \rrbracket (s) = \text{tt}$ .
  - $\rightarrow [\text{if1}_{\text{NatS}}]: \langle S; \text{ while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$
  - $\rightarrow [\text{comp}_{\text{NatS}}]:$  es gibt  $s''$  mit  $\langle S, s \rangle \rightarrow s''$  und  $\langle \text{while } b \text{ do } S \text{ end}, s'' \rangle \rightarrow s'$
  - $\rightarrow [\text{while1}_{\text{NatS}}]: \langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$
- Zweiter Fall: Es gelte  $B \llbracket b \rrbracket (s) = \text{ff}$ .
  - $\rightarrow [\text{if2}_{\text{NatS}}]: \langle \text{skip}, s \rangle \rightarrow s'$ , also  $s = s'$
  - $\rightarrow [\text{while2}_{\text{NatS}}]: \langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$ .

# Terminierung

- Ausführung von  $S$  in  $s$ 
  - terminiert, falls es ein  $s'$  gibt mit  $\langle S, s \rangle \rightarrow s'$
  - kreist, falls es kein  $s'$  gibt mit  $\langle S, s \rangle \rightarrow s'$  (dann: kein endlicher Folgerungsbaum)
- Ausführung von  $S$ 
  - terminiert immer, falls  $S$  in allen Zuständen  $s$  terminiert
  - kreist immer, falls  $S$  in allen Zuständen  $s$  kreist
- Beispiele:
  - Jedes while-freie Statement in  $W$  terminiert immer
  - **while true do skip end** kreist immer
  - **while  $x > 1$  do  $y := 0$ ; while  $y < x$  do  $y := y + 2$  end; if  $x = y$  then  $x := x / 2$  else  $x := 3 * x + 1$  end end** immer-Terminierung ungeklärt



# Beweis: Jedes while-freie S terminiert immer

- (Induktion über Struktur des Folgerungsbaums)
- $[\text{skip}_{\text{NatS}}], [\text{ass}_{\text{NatS}}]$ : Zu jedem  $s$  ex.  $s'$ ; Baum hat Tiefe 0
- $[\text{if1}_{\text{NatS}}], [\text{if2}_{\text{NatS}}]$ : Ind.-Voraussetzung liefert: es gibt ein  $s'$  mit  $\langle S1, s \rangle \rightarrow s'$  bzw.  $\langle S1, s \rangle \rightarrow s'$
- $[\text{comp}_{\text{NatS}}]$ : Ind.-Voraussetzung liefert: Es gibt ein  $s'$  mit  $\langle S1, s \rangle \rightarrow s'$  und es gibt ein  $s''$  mit  $\langle S2, s' \rangle \rightarrow s''$

# Beweis: while true do skip end kreist immer

- Einzige anwendbare Regel:  $[\text{while1}_{\text{NatS}}]$

$\langle \text{skip}, s \rangle \rightarrow s$

$\langle \text{while true do skip end}, s \rangle \rightarrow s'$

---

$\langle \text{while true do skip end}, s \rangle \rightarrow s'$

- 2. Prämisse und Folgerung identisch  
→ kein endlicher Ableitungsbaum generierbar

# Determinismus

- Zu jedem Command  $S$  in  $W$  und jedem  $s$  gibt es max. ein  $s'$  mit  $\langle S, s \rangle \rightarrow s'$
- Beweis: (Induktion über Struktur des Folgerungsbaums)
- $[\text{skip}_{\text{NatS}}]$ ,  $[\text{ass}_{\text{NatS}}]$ ,  $[\text{while2}_{\text{NatS}}]$ :  $s'$  von  $s$  jeweils funktional abhängig
- $[\text{if1}_{\text{NatS}}]$ ,  $[\text{if2}_{\text{NatS}}]$ : Ind.-Voraussetzung liefert: es gibt max. ein  $s'$  mit  $\langle S1, s \rangle \rightarrow s'$  bzw.  $\langle S1, s \rangle \rightarrow s'$
- $[\text{comp}_{\text{NatS}}]$ : Ind.-Voraussetzung liefert: Es gibt max. ein  $s'$  mit  $\langle S1, s \rangle \rightarrow s'$  und es gibt max. ein  $s''$  mit  $\langle S2, s' \rangle \rightarrow s''$
- $[\text{while1}_{\text{NatS}}]$ : Ind.-Voraussetzung liefert: Es gibt max. ein  $s'$  mit  $\langle S, s \rangle \rightarrow s'$  und max. ein  $s''$  mit  $\langle \text{while } b \text{ do } S \text{ end}, s' \rangle \rightarrow s''$

## Zum Vergleich mit anderen Semantiken

- definieren semantische Funktion  $S_{\text{NatS}}: \text{Cmd} \rightarrow (\text{State} \hookrightarrow \text{State})$  wie folgt:
- $S_{\text{NatS}} \llbracket S \rrbracket (s) := s', \text{ falls } \langle S, s \rangle \rightarrow s', \text{ und } n.\text{def.}, \text{ sonst}$
- Rechtfertigung: Determiniertheit von W
- Bemerkung: Für Sprachen mit Nichtdeterminismus wäre  $S: \text{Cmd} \rightarrow (\text{State} \rightarrow \wp(\text{State}))$

# Zusammenfassung: Natural Semantics

- Zu jedem Command: Zustandsüberführung
  - Ermittlung per Folgerungsbaum
- Können mittels Semantik
  - Werte von Zielzuständen ermitteln
  - semantische Äquivalenz von Konstrukten nachweisen
  - über Terminierung und Determiniertheit argumentieren

## 2.2 Structural Operational Semantics

- Idee: Semantik eines Commands ist *Folge elementarer* Zustandstransformationen
- Zum Vergleich: Natural Semantics liefert eine Zustandsüberführung für das *gesamte* Command, gerechtfertigt durch Überführungen (meist) einfacherer Commands

## Notation

- $\langle S, s \rangle \Rightarrow \langle S', s' \rangle$  : Um  $S$  in  $s$  auszuführen, muss man erst nach  $s'$  übergehen und dort  $S'$  ausführen
- $\langle S, s \rangle \Rightarrow s'$  : Ausführung von  $S$  in  $s$  geht in  $s'$  über und terminiert dort

# Structural Operational Semantics für W

•  $[\text{skip}_{\text{SOS}}]$   $\langle \text{skip}, s \rangle \Rightarrow s$

•  $[\text{ass}_{\text{SOS}}]$   $\langle x := E, s \rangle \Rightarrow s[x \rightarrow A \llbracket E \rrbracket (s)]$

•  $[\text{comp1}_{\text{SOS}}]$  
$$\frac{\langle S1, s \rangle \Rightarrow s'}{\langle S1 ; S2, s \rangle \Rightarrow \langle S2, s' \rangle}$$

•  $[\text{comp2}_{\text{SOS}}]$  
$$\frac{\langle S1, s \rangle \Rightarrow \langle S1', s' \rangle}{\langle S1 ; S2, s \rangle \Rightarrow \langle S1' ; S2, s' \rangle}$$

•  $[\text{if1}_{\text{SOS}}]$   $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \Rightarrow \langle S1, s \rangle$   
falls  $B \llbracket b \rrbracket (s) = \text{tt}$

•  $[\text{if2}_{\text{SOS}}]$   $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \Rightarrow \langle S2, s \rangle$   
falls  $B \llbracket b \rrbracket (s) = \text{ff}$

•  $[\text{while}_{\text{SOS}}]$   $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \Rightarrow \langle \text{if } b \text{ then } S ;$   
 $\text{while } b \text{ do } S \text{ end else skip end}, s \rangle$



# Abarbeitungssequenzen

- endlich, terminierend, z.B. (mit  $s(x) = a$ ,  $s(y) = b$ )
  - $\langle x := x - y; y := x + y; x := y - x, s \rangle \Rightarrow$   
 $\langle y := x + y; x := y - x, s[x \rightarrow a - b] \rangle \Rightarrow$   
 $\langle x := y - x, s[x \rightarrow a - b, y \rightarrow a] \rangle \Rightarrow$   
 $s[x \rightarrow b, y \rightarrow a]$
- unendlich, nicht terminierend, z.B.
  - $\langle \text{while true do skip end}, s \rangle \Rightarrow$   
 $\langle \text{if true then skip; while true do skip end} \text{ else skip end}, s \rangle \Rightarrow$   
 $\langle \text{skip; while true do skip end}, s \rangle \Rightarrow$   
 $\langle \text{while true do skip end}, s \rangle \Rightarrow \dots$
- endlich, verklemmend (endet in einem  $\langle S', s' \rangle$  - kommt in  $W$  nicht vor, aber in Erweiterungen)

# Semantische Äquivalenz

- S1 und S2 sind semantisch äquivalent, falls folgende Bedingungen für alle  $s, s', S'$  gelten:
  - $\langle S1, s \rangle \Rightarrow^* s'$  gdw.  $\langle S2, s \rangle \Rightarrow^* s'$
  - Wenn  $\langle S1, s \rangle \Rightarrow^* \langle S', s' \rangle$  (ohne Nachfolger) so gibt es ein  $S2'$  mit  $\langle S2, s \rangle \Rightarrow^* \langle S2', s' \rangle$  (ohne Nachfolger)
  - Wenn  $\langle S2, s \rangle \Rightarrow^* \langle S', s' \rangle$  (ohne Nachfolger) so gibt es ein  $S1'$  mit  $\langle S1, s \rangle \Rightarrow^* \langle S1', s' \rangle$  (ohne Nachfolger)
  - Wenn  $\langle S1, s \rangle \Rightarrow^*$  (unendlich), so  $\langle S2, s \rangle \Rightarrow^*$  (unendlich).

# Argumentieren über Terminierung, Determiniertheit analog zu NatS

Unterschiede:

- SOS unterscheidet Terminierung und Verklemmung
- Argumentation durch Induktion über Abarbeitungssequenzen statt über Folgerungsbaum

## Zum Vergleich mit anderen Semantiken

- definieren semantische Funktion  $S_{SOS}: \text{Cmd} \rightarrow (\text{State} \hookrightarrow \text{State})$  wie folgt:
- $S_{SOS} \llbracket S \rrbracket (s) := s'$ , falls  $\langle S, s \rangle \Rightarrow^* s'$ , und *n.def.*, sonst
- Rechtfertigung: Determiniertheit von W

## Vergleich: SOS versus NatS

- Für jedes Command aus  $W$  gilt:  $S_{\text{SOS}} \llbracket S \rrbracket = S_{\text{NatS}} \llbracket S \rrbracket$
- Zum Beweis: Zeigen
  - Wenn  $\langle S, s \rangle \rightarrow s'$  so  $\langle S, s \rangle \Rightarrow^* s'$
  - Wenn  $\langle S, s \rangle \Rightarrow^* s'$  so  $\langle S, s \rangle \rightarrow s'$

# Beweis Teil 1

- per Induktion über dem Folgerungsbaum
- Es gelte  $\langle S, s \rangle \rightarrow s'$  (also ex. endlicher Folgerungsbaum)
- Fall  $[\text{skip}_{\text{NatS}}]$ : Also  $s' = s$ ; SOS liefert  $\langle \text{skip}, s \rangle \Rightarrow s$
- Fall  $[\text{ass}_{\text{NatS}}]$ : Also  $s' = s[x \rightarrow A \llbracket E \rrbracket (s)]$ ; SOS dito.
- Fall  $[\text{comp}_{\text{NatS}}]$ : Es gibt also  $s''$  mit  $\langle S1, s \rangle \rightarrow s''$  und  $\langle S2, s'' \rangle \rightarrow s'$ .  
Nach Ind.Vor:  $\langle S1, s \rangle \Rightarrow^* s''$  und  $\langle S2, s'' \rangle \Rightarrow^* s'$ . Aus  $\langle S1, s \rangle \Rightarrow^* s''$   
kann man folgern:  $\langle S1; S2, s \rangle \Rightarrow^* \langle S2, s'' \rangle$ . Also  $\langle S1; S2, s \rangle \Rightarrow^* s'$
- Fall  $[\text{if1}_{\text{NatS}}]$ : Nach Ind.Vor:  $\langle S1, s \rangle \Rightarrow^* s'$ . SOS liefert also  
 $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \Rightarrow \langle S1, s \rangle \Rightarrow^* s'$ .
- Fall  $[\text{if2}_{\text{NatS}}]$ : Analog.
- Fall  $[\text{while1}_{\text{NatS}}]$ : Nach Ind.Vor:  $\langle S, s \rangle \Rightarrow^* s''$  und  $\langle \text{while } b \text{ do } S \text{ end}, s'' \rangle \Rightarrow^* s'$ .  
Also  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \Rightarrow \langle \text{if } b \text{ then } S; \text{ while } b \text{ do } S \text{ end} \text{ else skip end}, s \rangle \Rightarrow \langle S; \text{ while } b \text{ do } S \text{ end}, s \rangle \Rightarrow^* \langle \text{while } b \text{ do } S \text{ end}, s'' \rangle \Rightarrow^* s'$
- Fall  $[\text{while2}_{\text{NatS}}]$ : Also  $s = s'$ ; SOS liefert  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \Rightarrow \langle \text{if } b \text{ then } S; \text{ while } b \text{ do } S \text{ end} \text{ else skip end}, s \rangle \Rightarrow \langle \text{skip}, s \rangle \Rightarrow s$ .

## Beweis Teil 2

- per Induktion über Abarbeitungssequenzen
- Es gelte  $\langle S, s \rangle \Rightarrow^* s'$  (also ex. endlicher Folgerungsbaum)
- Fall  $[\text{skip}_{\text{SOS}}]$ ,  $[\text{ass}_{\text{SOS}}]$ : leicht zu sehen
- Fall  $[\text{comp1/2}_{\text{SOS}}]$ : Man kann zeigen: Es gibt ein  $s''$  mit  $\langle S1; S2, s \rangle \Rightarrow^* s''$  und  $\langle S2, s'' \rangle \Rightarrow^* s'$ .  $\langle S1; S2, s \rangle \Rightarrow^* s''$  kann man folgern:  $\langle S1, s \rangle \Rightarrow^* s''$ . Nach Ind.Vor:  $\langle S1, s \rangle \rightarrow s''$  und  $\langle S2, s'' \rangle \rightarrow s'$ . Also:  $\langle S1; S2, s \rangle \rightarrow s'$ .
- Fall  $[\text{if1}_{\text{SOS}}]$ :  $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \Rightarrow \langle S1, s \rangle \Rightarrow^* s'$ . Nach Ind.Vor:  $\langle S1, s \rangle \Rightarrow^* s'$ . NatS liefert also  $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \rightarrow s'$ .
- Fall  $[\text{if2}_{\text{SOS}}]$ : Analog.
- Fall  $[\text{while}_{\text{SOS}}]$ :  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \Rightarrow \langle \text{if } b \text{ then } S; \text{while } b \text{ do } S \text{ end else skip end}, s \rangle \Rightarrow^* s'$ . Nach Ind.Vor:  $\langle \text{if } b \text{ then } S; \text{while } b \text{ do } S \text{ end else skip end}, s \rangle \rightarrow s'$ . Wegen sem. Äquivalenz auch  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$ .

## 2.3 Erweiterungen von W: 1. Abort

- Neues command: **abort**.
- Semantik informal: Programm abbrechen, also *nicht fortsetzen*
- SOS/NatS von W mit **abort** = SOS/NatS von W ohne **abort**!  
(sprich: auf abort keine Regeln/Axiome anwendbar)
- Dadurch in beiden Semantiken: **skip** und **abort** nicht äquivalent
- SOS → Ausführung von **abort** führt zu Verklemmung
- NatS → Ausführung von **abort** = Es gibt keinen Folgerungsbaum
- In SOS sind **abort** und **while true do skip end** nicht äquivalent
- In NatS sind **abort** und **while true do skip end** äquivalent!



## Erweiterungen von W: 2. Nichtdeterminismus

- Neues command: **S1 or S2**.
- Semantik informal: nichtdeterministische Wahl zw. S1 und S2
- $[or1_{NatS}]$ : 
$$\frac{\langle S1, s \rangle \rightarrow s'}{\langle S1 \text{ or } S2, s \rangle \rightarrow s'}$$
  $[or2_{NatS}]$ : 
$$\frac{\langle S2, s \rangle \rightarrow s'}{\langle S1 \text{ or } S2, s \rangle \rightarrow s'}$$
- $[or1_{SOS}]$ :  $\langle S1 \text{ or } S2, s \rangle \Rightarrow \langle S1, s \rangle$   
 $[or2_{SOS}]$ :  $\langle S1 \text{ or } S2, s \rangle \Rightarrow \langle S2, s \rangle$
- Betrachten (while true do skip end) or  $x:x-y; y:=x+y; x:=x-y;$
- NatS  $\rightarrow$  Es gibt einen endlichen Folgerungsbaum
- SOS  $\rightarrow$  Es gibt sowohl endliche als auch unendliche Abarbeitungssequenzen
- $\rightarrow$  NatS versteckt *mögliches* kreisendes Verhalten

# Erweiterungen von W: 3. Nebenläufigkeit

- Neues command: **S1 par S2.**
- Semantik informal: nebenläufige Ausführung von S1 und S2
- [par1<sub>SOS</sub>]:  $\frac{\langle S1, s \rangle \Rightarrow s'}{\langle S1 \text{ par } S2, s \rangle \Rightarrow \langle S2, s' \rangle}$  [par2<sub>SOS</sub>]:  $\frac{\langle S2, s \rangle \Rightarrow s'}{\langle S1 \text{ par } S2, s \rangle \Rightarrow \langle S1, s' \rangle}$
- [par3<sub>SOS</sub>]:  $\frac{\langle S1, s \rangle \Rightarrow \langle S1', s' \rangle}{\langle S1 \text{ par } S2, s \rangle \Rightarrow \langle S1' \text{ par } S2, s' \rangle}$
- [par4<sub>SOS</sub>]:  $\frac{\langle S2, s \rangle \Rightarrow \langle S2', s' \rangle}{\langle S1 \text{ par } S2, s \rangle \Rightarrow \langle S1 \text{ par } S2', s' \rangle}$
- In NatS nicht ausdrückbar, weil zu jedem Command ein *atomarer* Zustandsübergang definiert ist!

# Erweiterungen von W: 4. Blöcke

- mit Variablendeklarationen (dürfen weiter außen bereits deklariert sein)

- Syntax:

`command = ... | „begin“ declaration cmd_seq „end“`

- Verwendung eines Namens bezieht sich immer auf die innerste Deklaration.

```
begin    var x,y : integer;
        y := 1; x := 2;
        begin    var x : integer;
                x := 3; y := x + 1
        end ;
        x := x + y
end
```

# Natural Semantics für W + Blöcke

- setzen voraus: Für **declarationlist** L sei  $dv(L)$  die Menge der in L deklarierten Variablen
- Deklaration selbst hat keine Wirkung auf den Zustand
- Nach Verlassen von **block** müssen Variablenwerte auf den Inhalt vor Betreten des Blocks zurückgesetzt werden

$$[\text{block}_{\text{NatS}}]: \frac{\langle S, s \rangle \rightarrow s'}{\langle \text{begin DL S end}, s \rangle \rightarrow s'[\{x \rightarrow s(x) \mid x \in dv(DL)\}]}$$

# Erweiterungen von W: Prozeduren

- Syntax:

```
declaration = „var“ ...  
            | „proc“ „identifier“ „is“ block  
cmd         = „call“ „identifier“
```

- In Blockstrukturen können Bezeichner mehrfach deklariert sein. Es wird immer die innerste Deklaration/Definition verwendet. Unterscheiden:
- Statische/dynamische Scopes für
  - Prozeduren
  - Variablen
  - Statisch: Auswahl nach Einbettung von Blöcken im Quelltext
  - Dynamisch: Auswahl nach Einbettung in der Aufrufstruktur

# Beispiel statisch/dynamisch

```
begin  var x; x:= 0;
      proc p is begin x := x * 2 end
      proc q is begin call p end
      begin var x; x := 5;
            proc p is x := x + 1;
            call q;
              y := x;
            end
      end
end
```

Proc statisch/Var statisch:        y = 5  
Proc statisch/Var dynamisch:    y = 10  
Proc dynamisch/Var dynamisch: y = 6

# Prozedurumgebungen

- Brauchen: Zuordnung von Namen zu Blöcken:
  - $\text{env: Names} \rightarrow \text{Commands}$
- Betreten von Blöcken = Aktualisierung von env:
  - $\text{upd}(\text{proc } p \text{ is } S; \text{DL}, \text{env}) = \text{upd}(\text{DL}, \text{env}[p \mapsto S])$
  - $\text{upd}(\varepsilon, \text{env}) = \text{env}$
- Alle Semantikregeln stehen im Kontext einer Umgebung env:
  - $\text{env} \vdash \langle \text{skip}, s \rangle \rightarrow s$

# Natural Semantics für W mit Proz. (dyn. Scopes)

- $[\text{skip}_{\text{NatS}}] \quad \text{env} \vdash \langle \text{skip}, s \rangle \rightarrow s$
- $[\text{ass}_{\text{NatS}}] \quad \text{env} \vdash \langle \mathbf{x} := \mathbf{E}, s \rangle \rightarrow s[x \rightarrow A \llbracket E \rrbracket (s)]$
- $[\text{comp}_{\text{NatS}}] \quad \frac{\text{env} \vdash \langle \mathbf{S1}, s \rangle \rightarrow s' \quad \text{env} \vdash \langle \mathbf{S2}, s' \rangle \rightarrow s''}{\text{env} \vdash \langle \mathbf{S1}; \mathbf{S2}, s \rangle \rightarrow s''}$
- $[\text{if1}_{\text{NatS}}] \quad \frac{\text{env} \vdash \langle \mathbf{S1}, s \rangle \rightarrow s' \quad \text{falls } B \llbracket b \rrbracket (s) = \text{tt}}{\text{env} \vdash \langle \text{if } b \text{ then } \mathbf{S1} \text{ else } \mathbf{S2} \text{ end}, s \rangle \rightarrow s'}$
- $[\text{if2}_{\text{NatS}}] \quad \frac{\text{env} \vdash \langle \mathbf{S2}, s \rangle \rightarrow s' \quad \text{falls } B \llbracket b \rrbracket (s) = \text{ff}}{\text{env} \vdash \langle \text{if } b \text{ then } \mathbf{S1} \text{ else } \mathbf{S2} \text{ end}, s \rangle \rightarrow s'}$
- $[\text{while1}_{\text{NatS}}] \quad \frac{\text{env} \vdash \langle \mathbf{S}, s \rangle \rightarrow s' \quad \text{env} \vdash \langle \text{while } b \text{ do } \mathbf{S} \text{ end}, s' \rangle \rightarrow s''}{\text{env} \vdash \langle \text{while } b \text{ do } \mathbf{S1} \text{ end}, s \rangle \rightarrow s''} \quad \text{falls } B \llbracket b \rrbracket (s) = \text{tt}$
- $[\text{while2}_{\text{NatS}}] \quad \text{env} \vdash \langle \text{while } b \text{ do } \mathbf{S} \text{ end}, s \rangle \rightarrow s \quad \text{falls } B \llbracket b \rrbracket (s) = \text{ff}$
- $[\text{blockNatS}] \quad \frac{\text{upd}(\text{DL}, \text{env}) \vdash \langle \mathbf{S}, s \rangle \rightarrow s'}{\text{env} \vdash \langle \text{begin DL } \mathbf{S} \text{ end}, s \rangle \rightarrow s'[\{x \rightarrow s(x) \mid x \in \text{dv}(\text{DL})\}]}$
- $[\text{call}_{\text{NatS}}] \quad \frac{\text{env} \vdash \langle \mathbf{S}, s \rangle \rightarrow s'}{\text{env} \vdash \langle \text{call } p, s \rangle \rightarrow s'} \quad \text{falls } \text{env}(p) = S$



# Natural Sem. für W mit Proz. (gemischte Scopes)

- müssen env verfeinern: brauchen Zuordnung von Namen zu Blöcken *zum Zeitpunkt der Definition* einer Prozedur:
  - env: Names  $\rightarrow$  Commands x Environments
  - $\text{upd}(\text{proc } p \text{ is } S; \text{DL}, \text{env}) = \text{upd}(\text{DL}, \text{env}[p \mapsto (S, \text{env})])$
  - $\text{upd}(\varepsilon, \text{env}) = \text{env}$
- $[\text{call}_{\text{NatS}}] \quad \frac{\text{env}'[p \mapsto (S, \text{env}')] \vdash \langle S, s \rangle \rightarrow s'}{\text{env} \vdash \langle \text{call } p, s \rangle \rightarrow s'} \quad \text{falls } \text{env}(p) = (S, \text{env}')$

Nur relevant für rekursiv gerufene Prozeduren <sup>57</sup>

# Natural Sem. für W mit Proz. (static Scopes)

- müssen Konzept des Zustands verfeinern:
  - bisher: Variable verweist auf Wert. Hier:
    - Variable verweist auf *Location* ( $Env_V: Names \rightarrow Loc$ )
    - Location verweist auf Wert  
( $Sto: Loc \rightarrow WB(integer/boolean)$ )
  - new:  $\rightarrow Loc$  liefert eine bislang nicht verwendete Location
  - Assignments ändern Sto
  - Blöcke ändern Env
  - Ausdrucksauswertung bezieht Werte, indem von Namen über Env auf eine Location und von dieser über sto auf einen Wert verweisen wird:  
 $s = sto \circ env_V$

# Variablendeklarationen

- Variablendeklarationen definieren Variablenumgebungen:

$$[\text{var}_{\text{NatS}}]: \frac{\langle \text{DL}, \text{env}_V[x \rightarrow \text{new}()] \rangle \rightarrow \text{env}_V'}{\langle \text{var } x : \text{type}; \text{DL} \rangle \rightarrow \text{env}_V'}$$

$$[\text{none}_{\text{NatS}}]: \langle \varepsilon, \text{env}_V \rangle \rightarrow \text{env}_V$$

- Prozedurdekларationen aktualisieren  
Prozedurumgebungen:

$$\begin{aligned} \text{upd}(\text{proc } p \text{ is } S; \text{DL}, \text{env}_V, \text{env}_P) \\ = \text{upd}(\text{DL}, \text{env}_V, \text{env}_P[p \rightarrow (S, \text{env}_V, \text{env}_P)]) \end{aligned}$$

$$\text{upd}(\varepsilon, \text{env}_V, \text{env}_P) = \text{env}_P$$

# Natural Semantics für W mit Proz. (st. Scopes)

- $[\text{skip}_{\text{NatS}}] \quad \text{env}_V, \text{env}_P \vdash \langle \text{skip}, \text{sto} \rangle \rightarrow \text{sto}$
- $[\text{ass2}_{\text{NatS}}] \quad \text{env}_V, \text{env}_P \vdash \langle \mathbf{x} := \mathbf{E}, \text{sto} \rangle \rightarrow \text{sto}[\text{env}_V(\mathbf{x}) \rightarrow A \llbracket \mathbf{E} \rrbracket (\text{sto} \circ \text{env}_V)]$
- $[\text{comp}_{\text{NatS}}] \quad \text{env}_V, \text{env}_P \vdash \langle \mathbf{S1}, \text{sto} \rangle \rightarrow \text{sto}' \quad \text{env}_V, \text{env}_P \vdash \langle \mathbf{S2}, \text{sto}' \rangle \rightarrow \text{sto}''$   

$$\frac{}{\text{env}_V, \text{env}_P \vdash \langle \mathbf{S1} ; \mathbf{S2}, \text{sto} \rangle \rightarrow \text{sto}''}$$
- $[\text{if1}_{\text{NatS}}] \quad \frac{\text{env}_V, \text{env}_P \vdash \langle \text{sto} \rangle \rightarrow \text{sto}' \quad \text{falls } B \llbracket b \rrbracket (s) = \text{tt}}{\text{env}_V, \text{env}_P \vdash \langle \text{if } b \text{ then } \mathbf{S1} \text{ else } \mathbf{S2} \text{ end}, \text{sto} \rangle \rightarrow \text{sto}'}$
- $[\text{if2}_{\text{NatS}}] \quad \text{analog}$
- $[\text{while1}_{\text{NatS}}] \quad \text{analog}$
- $[\text{while2}_{\text{NatS}}] \quad \text{analog}$
- $[\text{blockNatS}] \quad \frac{\langle \text{DL}_{\text{Var}}, \text{env}_V \rangle \rightarrow \text{env}_V' \quad \text{upd}(\text{DL}_{\text{Proc}}, \text{env}_V', \text{env}_P) \vdash \langle \mathbf{S}, \text{sto} \rangle \rightarrow \text{sto}'}{\text{env}_V, \text{env}_P \vdash \langle \text{begin DL } \mathbf{S} \text{ end}, \text{sto} \rangle \rightarrow \text{sto}'}$
- $[\text{call}_{\text{NatS}}] \quad \frac{\text{env}_V', \text{env}_P' \vdash \langle \mathbf{S}, \text{sto} \rangle \rightarrow \text{sto}' \quad \text{mit } \text{env}_P(p) = (\mathbf{S}, \text{env}_V', \text{env}_P')}{\text{env}_V, \text{env}_P \vdash \langle \text{call } p, \text{sto} \rangle \rightarrow \text{sto}'}$

# Lektionen aus Erweiterungen

- SOS besser für Nichtdeterminismus und Parallelität
- NatS besser für Blockstrukturen
- Komplexe Sprachen erfordern angepasste Zustandsdefinitionen (env, Loc, Sto,...)

## 2.4 Eine beweisbar korrekte Implementation

- Brauchen:
  - Maschine
  - Formale Semantik dieser Maschine
  - Übersetzung von  $W$  in die Sprache der Maschine
  - Beweis der Übereinstimmung der Semantik eines  $W$ -Programms mit der seiner Übersetzung

# Die Maschine AM

Besitzt:

- einen Programmspeicher für eine Anweisungssequenz
- einen Kellerspeicher für die Ausdrucksauswertung  
Eine Kellerkonfiguration ist Element von  
 $(WB(integer) \cup WB(boolean))^*$
- einen allgemeinen Speicher für Variablenwerte  
 $s \in \text{State}$

# Anweisungen von AM

- PUSH  $n$ : Konstante  $n$  auf Keller schreiben
- TRUE, FALSE: analog
- ADD: oberste Kellersymbole entnehmen, addieren, Ergebnis auf Keller schreiben
- MULT, SUB, DIV, OR, NEG, EQ, NEQ, LE, GE, LT, GT: analog
- FETCH  $x$ : Variablenwert aus allgemeinem Speicher lesen und auf Keller schreiben
- STORE  $x$ : oberstes Kellersymbol entfernen und als Wert in Variable  $x$  speichern
- NOOP: nix tun
- BRANCH ( $c1, c2$ ): oberstes Kellersymbol (bool) entfernen und abhängig vom Wahrheitswert  $c1$  oder  $c2$  ausführen
- LOOP( $c1, c2$ ):  $c1$  ausführen, oberstes Kellersymbol (bool) entfernen, bei ff beenden, sonst  $c2$  ausführen und von vorn
- $c1 : c2$ : Hintereinanderausführung



# Operationelle Semantik für AM

- Zustand (Konfiguration) besteht aus
  - verbleibender Anweisungssequenz
  - Stackkonfiguration
  - Zustand (Variablenwerte)

# Operationelle Semantik für AM

- $\langle \text{PUSH } n:c,e,s \rangle$   $\triangleright \langle c, n \ e, s \rangle$
- analog: TRUE, FALSE
- $\langle \text{ADD}:c,x \ y \ e,s \rangle$   $\triangleright \langle c, (x+y) \ e, s \rangle$
- analog: SUB MULT DIV AND OR NEG EQ NEQ LT GT LE GE
- $\langle \text{FETCH } x:c,e,s \rangle$   $\triangleright \langle c, s(x) \ e, s \rangle$
- $\langle \text{STORE } x:c,y \ e,s \rangle$   $\triangleright \langle c, e, s[x \rightarrow y] \rangle$
- $\langle \text{NOOP}:c,e,s \rangle$   $\triangleright \langle c,e,s \rangle$
- $\langle \text{BRANCH}(c1,c2):c,x \ e,s \rangle$   $\triangleright \langle c1:c,e,s \rangle$ , falls  $x = \text{tt}$
- $\langle \text{BRANCH}(c1,c2):c,x \ e,s \rangle$   $\triangleright \langle c2:c,e,s \rangle$ , falls  $x = \text{ff}$
- $\langle \text{LOOP}(c1,c2):c,e,s \rangle$   $\triangleright$   
 $\langle c1:\text{BRANCH}(c2:\text{LOOP}(c1,c2),\text{NOOP}):c,e,s \rangle$

# Beispiel

- Ausführung von PUSH 1:FETCH x:ADD:STORE x bei leerem Stack und Initialzustand  $x = 3$ :

< PUSH 1:FETCH x:ADD:STORE x, $\varepsilon$ ,s> ▷

< FETCH x:ADD:STORE x,1,s> ▷

< ADD:STORE x,3 1,s> ▷

< STORE x,4,s> ▷

<  $\varepsilon$ ,  $\varepsilon$ ,s[x $\rightarrow$ 4]>

(Terminierung)

# Die Übersetzung: Expressions

- Ziel: expression  $E \rightarrow$  Anweisungssequenz  $C(E)$  mit:
  - $\langle C(E), e, s \rangle \triangleright^* \langle \varepsilon, A/B \llbracket E \rrbracket (s) e, s \rangle$
- $C(x) = \text{FETCH } x$
- $C(n) = \text{PUSH } n$
- $C(\text{true}) = \text{TRUE}$ ,  $C(\text{false}) = \text{FALSE}$
- $C(E1 + E2) = C(E2):C(E1):\text{ADD}$  ; andere Operationen analog

# Die Übersetzung: Statements

- $C(x := E) = C(E): \text{STORE } x$
- $C(\text{skip}) = \text{NOOP}$
- $C(S1; S2) = C(S1):C(S2)$
- $C(\text{if } b \text{ then } S1 \text{ else } S2 \text{ end}) = C(b):\text{BRANCH}(C(S1), C(S2))$
- $C(\text{while } b \text{ do } S \text{ end}) = \text{LOOP}(C(b), C(S))$

# Beispiel

$C(y:=1; \text{while NOT } (x=1) \text{ do } y:=y*x; x:=x-1 \text{ END})$

=

PUSH 1: STORE y:

LOOP(PUSH 1:FETCH x:EQ:NEG,

    FETCH x:FETCH y:MULT:STORE y:

    PUSH 1:FETCH x:SUB:STORE x)

# Korrektheitsbeweis: Plan

1. Zeigen für beliebige Expressions von W:  
 $\langle C(E), e, s \rangle \triangleright^* \langle \varepsilon, A/B \llbracket E \rrbracket (s) e, s \rangle$   
(und Zwischenkonfigurationen haben Stack der Form  $w e$ )
2. Zeigen für beliebige Commands:  
 $S_{\text{NatS}} \llbracket S \rrbracket = S_{\text{AM}} \llbracket S \rrbracket$  für alle  $S$  aus  $W$ .
  - 2.1 Wenn  $\langle S, s \rangle \rightarrow s'$ , so  $\langle C(S), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$
  - 2.2 Wenn  $\langle C(S), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, e, s' \rangle$ , so  $\langle S, s \rangle \rightarrow s'$  und  $e = \varepsilon$

# Korrektheit der Expressions

- z.Z.:  $\langle C(E), e, s \rangle \triangleright^* \langle \varepsilon, A/B \llbracket E \rrbracket (s) e, s \rangle$
- Induktion über Struktur von E:
  - $\langle C(n), e, s \rangle = \langle \text{PUSH } n, e, s \rangle \triangleright$  (nach Semantik AM)  
 $\langle \varepsilon, n = A \llbracket n \rrbracket (s) e, s \rangle$
  - $\langle C(x), e, s \rangle = \langle \text{FETCH } x, e, s \rangle \triangleright$  (nach Semantik AM)  
 $\langle \varepsilon, s(x) = A \llbracket x \rrbracket (s) e, s \rangle$
  - $\langle C(E1 + E2), e, s \rangle = \langle C(E2): C(E1): \text{ADD}, e, s \rangle \triangleright^*$  (nach IVor)  
 $\langle C(E1): \text{ADD}, A \llbracket E2 \rrbracket (s) e, s \rangle \triangleright^*$  (nach IVor)  
 $\langle \text{ADD}, A \llbracket E1 \rrbracket (s) A \llbracket E2 \rrbracket (s) e, s \rangle \triangleright$  (nach Sem. AM)  
 $\langle \varepsilon, A \llbracket E1 \rrbracket (s) + A \llbracket E2 \rrbracket (s) = A \llbracket E1 + E2 \rrbracket (s) e, s \rangle$
  - andere Operationen analog
  - Außerdem: Zwischenkonfigurationen haben Stack w e mit  $w \neq \varepsilon$



# Korrektheit der Statements I

- z.Z.: Wenn  $\langle S, s \rangle \rightarrow s'$ , so  $\langle \mathcal{C}(S), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$
- Induktion über Folgerungsbaum:
  - Sei  $\langle x := E, s \rangle \rightarrow s'$ .  $\langle \mathcal{C}(x := E), \varepsilon, s \rangle = \langle \mathcal{C}(E): \text{STORE } x, \varepsilon, s \rangle$   
 $\triangleright^* \langle \text{STORE } x, A \llbracket E \rrbracket (s), s \rangle$  (wg. Korrektheit Expressions)  
 $\triangleright \langle \varepsilon, \varepsilon, s[x \rightarrow A \llbracket E \rrbracket (s)] \rangle$  (nach Semantik STORE)
  - Sei  $\langle \text{skip}, s \rangle \rightarrow s'$ , also  $s = s'$ .  $\langle \mathcal{C}(\text{skip}), \varepsilon, s \rangle = \langle \text{NOOP}, \varepsilon, s \rangle = \langle \varepsilon, \varepsilon, s \rangle$
  - Sei  $\langle S1; S2, s \rangle \rightarrow s'$ . Also ex.  $s''$  mit  $\langle S1, s \rangle \rightarrow s''$  und  $\langle S2, s'' \rangle \rightarrow s'$ .  
 $\langle \mathcal{C}(S1; S2), \varepsilon, s \rangle = \langle \mathcal{C}(S1): \mathcal{C}(S2), \varepsilon, s \rangle \triangleright^* \langle \mathcal{C}(S2), \varepsilon, s'' \rangle$  (nach IVor)  
 $\triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$  (nach IVor)
  - Sei  $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \rightarrow s'$ .
- 1. Fall:  $B \llbracket b \rrbracket (s)(b) = \text{tt}$ , also  $\langle S1, s \rangle \rightarrow s'$ .  
 $\langle \mathcal{C}(\text{if } b \text{ then } S1 \text{ else } S2 \text{ end}), \varepsilon, s \rangle = \langle \mathcal{C}(b): \text{BRANCH}(\mathcal{C}(S1), \mathcal{C}(S2)), \varepsilon, s \rangle$   
 $\triangleright^* \langle \text{BRANCH}(\mathcal{C}(S1), \mathcal{C}(S2)), B \llbracket b \rrbracket (s) = \text{tt}, s \rangle$  (wg. Korrektheit Exp.)  
 $\triangleright \langle \mathcal{C}(S1), \varepsilon, s \rangle$  (wg. Semantik BRANCH)  
 $\triangleright \langle \varepsilon, \varepsilon, s' \rangle$  (IVor); 2. Fall analog.
- Sei  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$ .
  - 1. Fall:  $B \llbracket b \rrbracket (s)(b) = \text{tt}$ , also  $\langle S, s \rangle \rightarrow s''$  und  $\langle \text{while } b \text{ do } S \text{ end}, s'' \rangle \rightarrow s'$ .  
 $\langle \mathcal{C}(\text{while } b \text{ do } S \text{ end}), \varepsilon, s \rangle = \langle \text{LOOP}(\mathcal{C}(b), \mathcal{C}(S)), \varepsilon, s \rangle$   
 $\triangleright \langle \mathcal{C}(b): \text{BRANCH}(\mathcal{C}(S): \text{LOOP}(\mathcal{C}(b), \mathcal{C}(S)), \text{NOOP}), \varepsilon, s \rangle$  (Semantik LOOP)  
 $\triangleright^* \langle \text{BRANCH}(\mathcal{C}(S): \text{LOOP}(\mathcal{C}(b), \mathcal{C}(S)), \text{NOOP}), B \llbracket b \rrbracket (s) = \text{tt}, s \rangle$  (Korrektheit Exp.)  
 $\triangleright \langle \mathcal{C}(S): \text{LOOP}(\mathcal{C}(b), \mathcal{C}(S)), \varepsilon, s \rangle$  (Semantik BRANCH)  
 $\triangleright^* \langle \text{LOOP}(\mathcal{C}(b), \mathcal{C}(S)), \varepsilon, s'' \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$  (IVor) ; 2. Fall einfach.

# Korrektheit der Statements II

- z.Z.: Wenn  $\langle C(S), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, e, s' \rangle$ , so  $\langle S, s \rangle \rightarrow s'$  und  $e = \varepsilon$
- Induktion über Länge der Maschinenbefehlssequenz:
  - Sei  $\langle C(x := E), \varepsilon, s \rangle = \langle C(E):STORE\ x, \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$   
 $\langle C(E), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, A\ [E]\ (s), s \rangle$  (Nach IVor, Korrektheit Exp.)  
 $\rightarrow s' = s[x \rightarrow A\ [E]\ (s)]$  (Semantik STORE), also  $\langle x := E, s \rangle \rightarrow s'$ .
  - Skip: einfach
  - Sei  $\langle C(S1; S2), \varepsilon, s \rangle = \langle C(S1):C(S2), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$ . Zerlegen Abarbeitungsfolge:  
 $\langle C(S1):C(S2), \varepsilon, s \rangle \triangleright^* \langle C(S2), e, s'' \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$ . Nach IVor:  $e = \varepsilon$ ,  $\langle S1, s \rangle \rightarrow s''$  und  
 (wg. IVor)  $\langle S2, s'' \rangle \rightarrow s'$ . Also  $\langle S1; S2, s \rangle \rightarrow s'$ .
  - Sei  $\langle C(\text{if } b \text{ then } S1 \text{ else } S2 \text{ end}), \varepsilon, s \rangle = \langle C(b):BRANCH(C(S1), C(S2)), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$ .  
 Nach IVor, Korrektheit Exp.:  $\langle C(b), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, B\ [b]\ (s), s \rangle$ 
    - 1. Fall:  $B\ [b]\ (s)(b) = \text{tt}$ .  
 Dann  $\langle BRANCH(C(S1), C(S2)), B\ [b]\ (s), s \rangle \triangleright \langle C(S1), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$ .  
 Nach IVor  $\langle S1, s \rangle \rightarrow s'$ , also  $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, s \rangle \rightarrow s'$ . 2. Fall analog.
  - Sei  $\langle C(\text{while } b \text{ do } S \text{ end}), \varepsilon, s \rangle \triangleright$   
 $\langle C(b):BRANCH(C(S):LOOP(C(b), C(S)), NOOP), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$ . Nach IVor,  
 Korrektheit Exp.:  $\langle C(b), \varepsilon, s \rangle \triangleright^* \langle \varepsilon, B\ [b]\ (s), s \rangle$ .
    - 1. Fall:  $B\ [b]\ (s)(b) = \text{tt}$ , also  $\langle C(S):LOOP(C(b), C(S)), \varepsilon, s \rangle \triangleright^*$   
 $\langle LOOP(C(b), C(S)), e, s'' \rangle \triangleright^* \langle \varepsilon, \varepsilon, s' \rangle$   
 Nach IVor:  $e = \varepsilon$  und  $\langle S, s \rangle \rightarrow s''$  und  $\langle \text{while } b \text{ do } S \text{ end}, s'' \rangle \rightarrow s'$ .  
 Also  $\langle \text{while } b \text{ do } S \text{ end}, s \rangle \rightarrow s'$ . 2. Fall einfach.

# Zusammenfassung Kapitel 2

- Formale operationelle Semantik abstrahiert so weit wie möglich von technischen Rahmenbedingungen (Wertkodierung, Speicheradressen, ...) → verbleibende Flexibilität bei Implementierung
- Formale operationelle Semantik ermöglicht Argumentation über
  - Terminierung
  - semantische Äquivalenz
  - Determiniertheit
  - Korrektheit einer Übersetzung

## Kapitel 3

# Denotationelle Semantik

## Idee

- Hatten aus operationeller Semantik gewonnen:
  - $S: \text{Command} \rightarrow (\text{State} \leftrightarrow \text{State})$
  - Zur Bestimmung von  $S$ : Folgerungsbaum bzw. Abarbeitungssequenzen
- Ziel nun: Definition von  $S$  ohne Umwege, induktiv über Struktur eines Commands.
- Insbesondere:  $S(C)$  soll definierbar sein allein aus den Werten von  $S$  für die unmittelbaren Teilstrukturen von  $C$

*„Kompositionalität“*

## 3.1 Direct-Style Semantik für W

- $S_{ds} \llbracket x := E \rrbracket (s) = s[x \rightarrow A \llbracket E \rrbracket (s)]$
- $S_{ds} \llbracket \text{skip} \rrbracket = \text{id}$
- $S_{ds} \llbracket S1; S2 \rrbracket = S_{ds} \llbracket S2 \rrbracket \circ S_{ds} \llbracket S1 \rrbracket$
- $S_{ds} \llbracket \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \rrbracket =$   
 $\text{cond}(B \llbracket b \rrbracket, S_{ds} \llbracket S1 \rrbracket, S_{ds} \llbracket S2 \rrbracket)$
- $S_{ds} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket = \text{FIX}(F)$   
mit  $F(g) = \text{cond}(B \llbracket b \rrbracket, g \circ S_{ds} \llbracket S \rrbracket, \text{id})$

# Bedingte Anweisungen

- $\text{cond}: (\text{State} \rightarrow \{\text{tt}, \text{ff}\}) \times (\text{State} \hookrightarrow \text{State}) \times (\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})$
- $\text{cond}(p, g1, g2) (s) = \begin{cases} g1(s), & \text{falls } p(s) = \text{tt} \\ g2(s), & \text{falls } p(s) = \text{ff} \end{cases}$

## Bleibt: Semantik der While-Anweisung

- $S_{ds} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket = \text{FIX}(F)$   
mit  $F(g) = \text{cond}(B \llbracket b \rrbracket, g \circ S_{ds} \llbracket S \rrbracket, \text{id})$
- $\text{FIX}(F)$  („Fixpunkt von  $F$ “) ist Lösung der Gleichung  $X = F(X)$ .  
 $\text{FIX}: ((\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})) \rightarrow (\text{State} \hookrightarrow \text{State})$
- Wieso?
  - Sinnvoll:  $S_{ds} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket$   
=  $S_{ds} \llbracket \text{if } b \text{ then } S; \text{ while } b \text{ do } S \text{ end else skip end} \rrbracket$   
=  $\text{cond}(B \llbracket b \rrbracket, S_{ds} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket \circ S_{ds} \llbracket S \rrbracket, \text{id})$

also:  $S_{ds} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket$   
ist (eine!) Lösung von  $X = F(X)$



# Beispiel für Fixpunkt

- Betrachten  $S = \text{while } x \neq 0 \text{ do skip end}$

- $F_S(g)(s) = \begin{cases} g(s), & \text{falls } s(x) \neq 0 \\ s, & \text{falls } s(x) = 0 \end{cases}$

- $g_1$  mit  $g_1(s) = \begin{cases} \text{undef}, & s(x) \neq 0 \\ s, & s(x) = 0 \end{cases}$  ist Fixpunkt:

Falls  $s(x) = 0$ , so  $F(g_1)(s) = s = g_1(s)$

Falls  $s(x) \neq 0$ , so  $F(g_1)(s) = g_1(s) (= \text{undef})$

- $g_2$  mit  $g_2(s) = \text{undef}$  (für alle  $s$ ) ist kein Fixpunkt.

Für  $s$  mit  $s(x) = 0$ :  $F(g_2)(s) = s \neq g_2(s) = \text{undef}$

## Probleme:

1. Es gibt Funktionale, die keinen Fixpunkt haben, z.B.

$$F(g) = \begin{cases} g1, & \text{falls } g = g2 \\ g2, & \text{falls } g \neq g2 \end{cases}$$

Lösung: Werden zeigen, dass die in der Semantik von W verwendeten Funktionale Fixpunkte besitzen

2. Es gibt Funktionale, die mehr als einen Fixpunkt besitzen, z.B. ist jede Funktion  $g^*$  mit  $g^*(s) = s$ , falls  $s(x) = 0$ , ein Fixpunkt des Funktional der vorigen Folie

Lösung: Werden Bedingungen erarbeiten, die von genau einem Fixpunkt eines verwendeten Funktional erfüllt sind.

# Welcher Fixpunkt

- Betrachten while b do S end, gestartet in s0.
- Fall A: terminiert
- Fall B: terminiert nicht, weil ein Untercommand nicht terminiert
- Fall C: terminiert nicht, weil Schleife selbst nicht abbricht
- Fall A: Terminiert  $\rightarrow$  ex.  $s_1, \dots, s_n$  mit
  - $B \llbracket b \rrbracket (s_1) = \dots = B \llbracket b \rrbracket (s_{n-1}) = \text{tt}$ ,  $B \llbracket b \rrbracket (s_n) = \text{ff}$ .
  - $S_{ds} \llbracket S \rrbracket (s_i) = s_{i+1}$  für  $i < n$
  - Sei  $g_0$  Fixpunkt.  $\rightarrow$  (für  $i < n$ )  $g_0(s_i) = F(g_0)(s_i)$   
 $= \text{cond}(B \llbracket b \rrbracket (s_i), g_0 \circ S_{ds} \llbracket S \rrbracket (s_i), \text{id})$   
 $= g_0 \circ S_{ds} \llbracket S \rrbracket (s_i) = g_0(s_{i+1})$ .
  - Für  $i = n$ :  $g_0(s_n) = F(g_0)(s_n)$   
 $= \text{cond}(B \llbracket b \rrbracket (s_n), g_0 \circ S_{ds} \llbracket S \rrbracket (s_n), \text{id})$   
 $= \text{id}(s_n) = s_n$  Also:  $g_0(s_0) = s_n$ . Das ist, was wir wollen.

# Welcher Fixpunkt

- Fall B: terminiert nicht, weil ein Untercommand nicht terminiert
- ex.  $s_1, \dots, s_n$  mit
  - $B \llbracket b \rrbracket (s_1) = \dots = B \llbracket b \rrbracket (s_n) = tt$ ,
  - $S_{ds} \llbracket S \rrbracket (s_i) = s_{i+1}$  für  $i < n$
  - $S_{ds} \llbracket S \rrbracket (s_n) = \text{undef}$
- Sei  $g_0$  Fixpunkt.  $\rightarrow$  (für  $i < n$ )  $g_0(s_i) = g_0(s_{i+1})$  (wie vorher).
- Für  $i = n$ :  $g_0(s_n) = F(g_0)(s_n)$   
 $= \text{cond}(B \llbracket b \rrbracket (s_n), g_0 \circ S_{ds} \llbracket S \rrbracket (s_n), \text{id})$   
 $= \text{undef}$       Also:  $g_0(s_0) = \text{undef}$ . Das ist, was wir wollen.

# Welcher Fixpunkt

- Fall C: terminiert nicht, weil Schleife selbst nicht abbricht.
- ex.  $s_1, \dots, s_n, \dots$  mit
  - $B \llbracket b \rrbracket (s_1) = \dots = B \llbracket b \rrbracket (s_n) = \dots = tt,$
  - $S_{ds} \llbracket S \rrbracket (s_i) = s_{i+1}$  für  $i < n$
- Sei  $g_0$  Fixpunkt.  $\rightarrow$  (für alle  $i$ )  $g_0(s_i) = g_0(s_{i+1})$ .
- Aus diesen Gleichungen lässt sich kein Wert für  $g_0(s_0)$  ermitteln!

Für  $F$  mit  $F(g)(s) = \begin{cases} g(s), & \text{falls } x \neq 0 \\ s, & \text{falls } x = 0 \end{cases}$  (gebildet für  $\text{while } x \neq 0 \text{ do skip end}$ )  
 ist jedes  $g$  Fixpunkt, das  $g(s) = s$  für alle  $s$  mit  $s(x) = 0$  hat.

# Welcher Fixpunkt: Zusammenfassung

- Fall A und B: Jeder Fixpunkt liefert das gewünschte Ergebnis
- Fall C: Betrachten while  $x \neq 0$  do skip end
  - vorige Folie: jedes g Fixpunkt, das  $g(s) = s$  für alle s mit  $s(x) \neq 0$  hat.
  - intuitiv:  $S_{ds} \llbracket \text{while } x \neq 0 \text{ do skip end} \rrbracket (s_0) = \begin{cases} \text{undef}, s_0(x) \neq 0 \\ s_0, s_0(x) = 0 \end{cases}$

Dies ist der kleinste Fixpunkt  $g_0$  im folgenden Sinn:

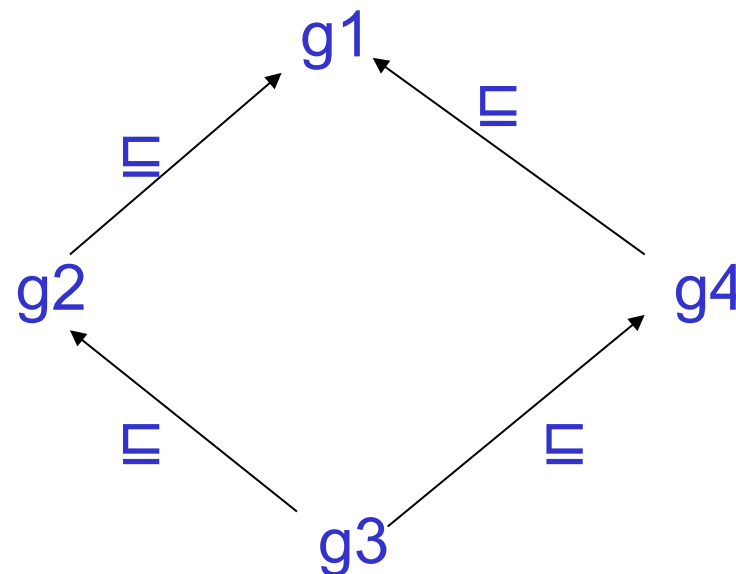
Wenn  $g_0(s) = s'$  (also definiert), so für alle anderen Fixpunkte  $g$ :  $g(s) = s'$ .

# Fixpunkttheorie

- Mathematik, die sich mit Fixpunkten beschäftigt
- Ziel: Handwerkszeug, um Existenz und Eindeutigkeit der in der Semantik von  $W$  vorkommenden Fixpunkte zu zeigen
- Reden über Halbordnungen
- Unsere Halbordnung:  $\sqsubseteq$  mit  $g1 \sqsubseteq g2$ , falls für alle  $s$ :  
 $g1(s) = s' \rightarrow g2(s) = s'$ 
  - ist reflexiv: für alle  $g$ :  $g \sqsubseteq g$
  - ist transitiv: für alle  $g1, g2, g3$ : Wenn  $g1 \sqsubseteq g2$  und  $g2 \sqsubseteq g3$ , so  $g1 \sqsubseteq g3$
  - ist antisymmetrisch: Wenn  $g1 \sqsubseteq g2$  und  $g2 \sqsubseteq g1$ , so  $g1 = g2$
- $\sqsubseteq$  entspricht der Relation  $\subseteq$  auf der Menge der zu  $g$  gehörenden geordneten Paare.

# Beispiel

- $g_1(s) = s$  für alle  $s$
- $g_2(s) = s$ , falls  $s(x) \geq 0$ , undef sonst
- $g_3(s) = s$ , falls  $s(x) = 0$ , undef sonst
- $g_4(s) = s$ , falls  $s(x) \leq 0$ , undef sonst



Hasse-Diagramm



# Begriffe für Halbordnungen

- $x$  heißt Minimum einer Halbordnung, falls für alle  $y$ :  $x \sqsubseteq y$
- Wenn eine Halbordnung ein Minimum besitzt, ist es eindeutig.
- In der Halbordnung  $\sqsubseteq$  auf der Menge  $(\text{State} \hookrightarrow \text{State})$  ist  $\perp$  mit  $\perp(s) = \text{undef}$  (für alle  $s$ ) das Minimum
- $M$  heißt Kette, falls für alle  $x, y \in M$ :  $x \sqsubseteq y$  oder  $y \sqsubseteq x$
- z.B.: Wenn  $g_i(s) = \begin{cases} s, & \text{falls } s(x) < i \\ \text{undef}, & \text{sonst,} \end{cases}$   
so  $\{g_i \mid i > 0\}$  unendliche Kette

# Begriffe für Halbordnungen II

- $x$  heißt obere Schranke einer Menge  $M$ , falls für alle  $y \in M: y \sqsubseteq x$ .
  - $x$  heißt kleinste obere Schranke von  $M$ , falls für alle Schranken  $x'$  von  $M$  gilt:  $x \sqsubseteq x'$ .
  - Falls  $M$  eine kleinste obere Schranke hat, so ist sie eindeutig bestimmt. Schreiben:  $\sqcup M$
  - $\sqsubseteq$  ist kettenvollständig, d.h., jede Kette  $M$  besitzt eine kleinste obere Schranke, nämlich
$$\sqcup M(s) = \begin{cases} s', \text{ ex. } g \in M \text{ mit } g(s) = s' \\ \text{undef, sonst} \end{cases}$$
- ( $\leq$  auf  $N$  ist nicht kettenvollständig –  $N$  hat keine obere Schranke)

## Begriffe für Halbordnungen III

- Jede kettenvollständige Halbordnung hat ein kleinstes Element  $\perp$ , nämlich  $\perp = \sqcup \emptyset$  (Jedes Element ist ob. Schranke von  $\emptyset$ )
- Sei  $F: (\text{State} \hookrightarrow \text{State}) \rightarrow (\text{State} \hookrightarrow \text{State})$
- $F$  heißt monoton, falls: Wenn  $g1 \sqsubseteq g2$ , so  $F(g1) \sqsubseteq F(g2)$
- Wenn  $F1, F2$  monoton, so auch  $F1 \circ F2$
- Wenn  $M$  Kette und  $F$  monoton, so  $\{F(x) \mid x \in M\}$  Kette und  $\sqcup \{F(x) \mid x \in M\} \sqsubseteq F(\sqcup M)$
- $F$  (monoton) heißt stetig, falls für alle nichtleeren Ketten  $M$ :  $\sqcup \{F(x) \mid x \in M\} = F(\sqcup M)$  und strikt, falls  $\sqcup \emptyset = F(\sqcup \emptyset)$

# Ein Fixpunkttheorem

- Sei  $F$  eine stetige (also auch monotone) Funktion in einer kettenvollständigen Halbordnung. Dann ist  $\text{FIX } F = \sqcup \{\perp, F(\perp), F(F(\perp)), \dots, F^i(\perp), \dots\}$  der kleinste Fixpunkt von  $F$ .

Beweis:

- $\{\perp, F(\perp), F(F(\perp)), \dots, F^i(\perp), \dots\}$  ist Kette, weil  $\perp \sqsubseteq F(\perp)$ , also  $F^i(\perp) \sqsubseteq F^{i+1}(\perp)$  (Monotonie), also  $F^i(\perp) \sqsubseteq F^{i+j}(\perp)$  (Transitivität). Also ist  $\sqcup \{\perp, F(\perp), F(F(\perp)), \dots, F^i(\perp), \dots\}$  wohldefiniert.
- $\text{FIX } F$  ist Fixpunkt, weil  $F(\text{FIX } F) = F(\sqcup \{\perp, F(\perp), F(F(\perp)), \dots, F^i(\perp), \dots\})$   
 $= \sqcup (\{F(\perp), F(F(\perp)), \dots, F(F^i(\perp)), \dots\})$  ( $F$  stetig)  
 $= \sqcup (\{\perp, F(\perp), F(F(\perp)), \dots, F(F^i(\perp)), \dots\})$  ( $\perp$  ist Minimum)  
 $= \text{FIX } F$ .
- $\text{FIX } F$  ist kleinster Fixpunkt, denn für anderen Fixpunkt  $d$  gilt:  
 $\perp \sqsubseteq d$  (Minimum), also  $F^n(\perp) \sqsubseteq F^n(d)$  (Monotonie), also  $F^n(\perp) \sqsubseteq d$  ( $d$  Fixp.), also  $d$  obere Schranke von  $\{\perp, F(\perp), F(F(\perp)), \dots, F^i(\perp), \dots\}$ , also  $\text{FIX } F \sqsubseteq d$ .

# Schlussfolgerung

- Zur Korrektheit der Semantikdefinition für  $W$  reicht es zu zeigen, dass die der Fixpunktbildung zugrundeliegenden Funktionale monoton und stetig sind.

# Stetigkeit der Funktionale in der Direct-Style-Semantik

- F in der Def. der Direct-Style-Semantik:
  - $F(g) = \text{cond}(B \llbracket b \rrbracket, g \circ S_{\text{ds}} \llbracket S \rrbracket, \text{id})$   
=  $F1(F2(g))$  mit  
 $F1(g) = \text{cond}(B \llbracket b \rrbracket, g, \text{id})$   
 $F2(g) = g \circ S_{\text{ds}} \llbracket S \rrbracket$

Zeigen

1.  $F1$  stetig
2.  $F2$  stetig
3. Wenn  $F1, F2$  stetig, so auch  $F1 \circ F2$

# Stetigkeit von F1

Für beliebige  $g_0: \text{State} \hookrightarrow \text{State}$  und  $p: \text{State} \rightarrow \{\text{tt}, \text{ff}\}$  ist  $F1(g) = \text{cond}(p, g, g_0)$  stetig.

Beweis:

(1) Zeigen: F1 monoton. Sei  $g_1 \sqsubseteq g_2$ , z.Z.  $F1(g_1) \sqsubseteq F1(g_2)$

Sei  $F1(g_1)(s) = s'$ .

- Fall:  $p(s) = \text{tt}$ . Dann  $F1(g_1)(s) = g_1(s) = s' = g_2(s) = F1(g_2)(s)$ .
- Fall:  $p(s) = \text{ff}$ . Dann  $F1(g_1)(s) = g_0(s) = F1(g_2)(s)$ .

(2) Zeigen: F1 stetig. Sei Y nichtleere Kette.

z.Z.  $F1(\sqcup Y) \sqsubseteq \sqcup \{F1(y) \mid y \in Y\}$  ( $\exists$  gilt immer)

Sei  $F1(\sqcup Y)(s) = s'$ . Zeigen: es gibt ein  $g \in Y$  mit  $F1(g)(s) = s'$ :

- Fall:  $p(s) = \text{tt}$ , dann  $F1(\sqcup Y)(s) = (\sqcup Y)(s)$ , also muss es ein  $g \in Y$  geben mit  $g(s) = s'$ .
- Fall:  $p(s) = \text{ff}$ , dann  $F1(\sqcup Y)(s) = g_0(s) = F1(g)(s)$  für bel.  $g \in Y$

Wenn es ein  $g \in Y$  mit  $F1(g)(s) = s'$  gibt, ist aber  $\sqcup \{F1(y) \mid y \in Y\}(s) = s'$ .

# Stetigkeit von F2

- Sei  $g_0: \text{State} \hookrightarrow \text{State}$  und  $F2(g) = g \circ g_0$ . Dann ist F2 stetig.

Beweis:

(1) Zeigen: F2 monoton.

Sei  $g_1 \sqsubseteq g_2$  und  $F2(g_1)(s) = s'$ . Also ex.  $s''$  mit  $g_0(s) = s''$ ,  $g_1(s'') = s'$ .

Wegen  $g_1 \sqsubseteq g_2$  ist  $g_2(s'') = s'$ , also  $F2(g_2)(s) = g_2(g_0(s)) = s'$ .

(2) Zeigen: F2 stetig.

Sei  $Y$  nichtleere Kette und  $F2(\sqcup Y)(s) = s'$ . Also

$\sqcup Y(g_0(s)) = s'$ . Also ex.  $g \in Y$  mit  $g(g_0(s)) = s'$ .

Also  $\sqcup \{F2(y) | y \in Y\}(s) = s'$ .



# Stetigkeit der Verkettung

Seien  $F1, F2$  stetig. Dann  $F1 \circ F2$  stetig.

Beweis:

(1) Da  $F1, F2$  monoton, so auch  $F1 \circ F2$ .

(2) Zeigen:  $F1 \circ F2$  stetig.

Sei  $Y$  nichtleere Kette.

Weil  $F2$  stetig:  $F2(\sqcup Y) = \sqcup \{F2(y) | y \in Y\}$

Weil  $F1$  stetig:  $F1(\sqcup \{F2(y) | y \in Y\}) = \sqcup \{F1(F2(y)) | y \in Y\}$ .

Also  $F1 \circ F2(\sqcup Y) = \sqcup \{F1 \circ F2(y) | y \in Y\}$ .

# Zusammenfassung Direct-Style-Semantik

- kompositional: Semantik eines Konstrukts allein auf der Basis seiner Teilkonstrukte definiert
- Kleinste Fixpunkte werden zur Definition der Semantik von Schleifen verwendet
- Fixpunkttheorie liefert Existenz solcher Fixpunkte
- ohne Beweis: Für alle Commands  $S$  von  $W$  ist
$$S_{ds} \llbracket S \rrbracket = S_{sos} \llbracket S \rrbracket$$

## 3.2 Erweiterung: Prozeduren

- Am Beispiel statischer Scopes
- Verwenden (z.T. erneut) die Konzepte
  - Loc: Menge von Locations (= Menge der ganzen Zahlen)
  - new:  $\text{Loc} \rightarrow \text{Loc}$  ( new(x) := x+1) /\* Ordnet einer Location die nächste zu \*/
  - Store:  $\text{Loc} \cup \{\text{next}\} \rightarrow \mathbb{Z}$  /\* Speicher; next speichert Adresse der 1. bislang unbenutzen Location \*/
  - $\text{Env}_V : \text{Var} \rightarrow \text{Loc}$
  - $\text{Env}_P : \text{Pnames} \rightarrow (\text{Store} \hookrightarrow \text{Store})$
  - Funktion lookup:  $\text{Env}_V \times \text{Store} \rightarrow \text{State}$ ;  $\text{lookup}(\text{env}, \text{sto})(x) = \text{sto}(\text{env}(x))$  /\* Transformiert env + sto in state \*/

# Direct-Style Semantik für W mit Env,Sto

- $S'_{ds} \llbracket x := E \rrbracket (env_V)(env_P)(sto) = sto[env_V(x) \rightarrow A \llbracket E \rrbracket (lookup (env_V, sto))]$
- $S'_{ds} \llbracket skip \rrbracket (env_V)(env_P) = id$
- $S'_{ds} \llbracket S1; S2 \rrbracket (env_V)(env_P) =$   
 $(S'_{ds} \llbracket S2 \rrbracket (env_V)(env_P)) \circ (S'_{ds} \llbracket S1 \rrbracket (env_V)(env_P))$
- $S'_{ds} \llbracket \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \rrbracket (env_V)(env_P) =$   
 $cond(B \llbracket b \rrbracket \circ lookup(env_V, .), S'_{ds} \llbracket S1 \rrbracket (env_V)(env_P), S'_{ds} \llbracket S2 \rrbracket (env_V)(env_P))$
- $S'_{ds} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket (env_V)(env_P) = FIX(F) \text{ mit}$   
 $F(g) = cond(B \llbracket b \rrbracket \circ lookup(env_V, .), g \circ S'_{ds} \llbracket S \rrbracket (env_V)(env_P), id)$

# Deklarationen

Effekt einer Variablendeklaration = Änderung von  $env_V$  ...  $DV$

Effekt einer Prozedurdekларation = Änderung von  $env_P$  ...  $DP$

$DV \llbracket e \rrbracket = id$

$DV \llbracket \text{var } x ; \text{ decl} \rrbracket (env_V, sto) = DV \llbracket \text{decl} \rrbracket$   
 $(env_V[x \rightarrow l], sto(l \rightarrow 0, next \rightarrow new\ l))$  mit  $l = sto(next)$

(0 = Initialwert für x)

$DP \llbracket e \rrbracket (env_V) = id$

$DP \llbracket \text{proc } p \text{ is } S ; \text{ decl} \rrbracket (env_V, env_P) =$   
 $DP \llbracket \text{decl} \rrbracket (env_V, env_P[p \rightarrow S'_{ds} \llbracket S \rrbracket (env_V)(env_P)])$

## Blöcke, Prozedurrufe

- $S'_{ds} \llbracket \text{begin Dec}_V \text{ Dec}_P S \text{ end} \rrbracket (\text{env}_V)(\text{env}_P)(\text{sto}) =$   
   $S'_{ds} \llbracket S \rrbracket (\text{env}'_V)(\text{env}'_P)(\text{sto}')$  mit  
     $DV \llbracket \text{Dec}_V \rrbracket (\text{env}_V, \text{sto}) = (\text{env}'_V, \text{sto}')$  und  
     $DP \llbracket \text{Dec}_P \rrbracket (\text{env}'_V, \text{env}_P) = \text{env}'_P$
- $S'_{ds} \llbracket \text{call } p \rrbracket (\text{env}_V)(\text{env}_P) = \text{env}_P(p)$

## 3.3 Continuation-Style Semantik

- Am Beispiel einer Erweiterung von W: Exceptions
- `command = ...`
  - | `begin S1 catch e: S2 end`
  - | `throw e`

# Continuation

- beschreibt den Effekt der Ausführung des restlichen Programms
- $c: \text{State} \hookrightarrow \text{State}$
- Continuation-style semantics:
- $S_{cs}: \text{Commands} \rightarrow (\text{Cont} \rightarrow \text{Cont})$
- $\dots ; S ; \dots \quad (c)$   
 $\rightarrow \dots ; S ; \dots \quad (c')$



# CS-Semantik für W (ohne Exceptions)

- $S_{cs} \llbracket x := E \rrbracket (c)(s) = c (s[x \rightarrow A \llbracket E \rrbracket (s)])$
- $S_{cs} \llbracket \text{skip} \rrbracket = \text{id}$
- $S_{cs} \llbracket S1; S2 \rrbracket = S_{cs} \llbracket S1 \rrbracket \circ S_{cs} \llbracket S2 \rrbracket$
- $S_{cs} \llbracket \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \rrbracket (c) =$   
 $\text{cond}(B \llbracket b \rrbracket, S_{cs} \llbracket S1 \rrbracket (c), S_{cs} \llbracket S2 \rrbracket (c))$
- $S_{cs} \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket = \text{FIX } G$   
wobei  $G(g)(c) = \text{cond}(B \llbracket b \rrbracket, S_{cs} \llbracket S \rrbracket (g(c)), c)$

Mit den bekannten Techniken kann man Existenz der Fixpunkte und Äquivalenz zu anderen Semantiken nachweisen:

Für alle Commands  $S$  und Continuations  $c$  gilt:

- $S_{cs} \llbracket S \rrbracket (c) = c \circ S_{ds} \llbracket S \rrbracket$

# Exceptions

- Brauchen Information, welcher Handler (catch-Block) zur welcher Exception gehört:
- $\text{Env}_E : \text{Exception} \rightarrow \text{Cont}$
- $S_{cs}' : (\text{Command} \times \text{Env}_E) \rightarrow (\text{Cont} \rightarrow \text{Cont})$

# CS-Semantik für W (mit Exceptions)

- $S_{cs}' \llbracket x := E \rrbracket (env)(c)(s) = c(s[x \rightarrow A/B \llbracket E \rrbracket (s)])$
- $S_{cs}' \llbracket skip \rrbracket (env) = id$
- $S_{cs}' \llbracket S1; S2 \rrbracket (env) = S_{cs}' \llbracket S1 \rrbracket (env) \circ S_{cs}' \llbracket S2 \rrbracket (env)$
- $S_{cs}' \llbracket \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \rrbracket (env)(c) =$   
 $\quad cond(B \llbracket b \rrbracket, S_{cs}' \llbracket S1 \rrbracket (env)(c), S_{cs}' \llbracket S2 \rrbracket (env)(c))$
- $S_{cs}' \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket (env) = FIX G$   
 $\quad \text{wobei } G(g)(c) = cond(B \llbracket b \rrbracket, S_{cs}' \llbracket S \rrbracket (env)(g(c)), c)$
- $S_{cs}' \llbracket \text{begin } S1 \text{ catch } e: S2 \text{ end} \rrbracket (env)(c) =$   
 $\quad S_{cs}' \llbracket S1 \rrbracket (env[e \rightarrow S_{cs}' \llbracket S2 \rrbracket (env)(c)])$
- $S_{cs}' \llbracket \text{throw } E \rrbracket (env)(c) = env(e)$

Mit den bekannten Techniken kann man Existenz der Fixpunkte und Äquivalenz zu anderen Semantiken nachweisen

## 3.3 Statische Programmanalyse

- Ziel: Information über die Semantik eines Programms ohne Ausführung
- Anwendung
  - Programmcodoptimierung
  - Programmverifikation
- Rahmenbedingungen
  - muss immer (und relativ schnell) terminieren (auch bei nicht terminierenden Programmen)
  - darf unscharfe („weiss nicht“), aber nie falsche Ergebnisse liefern

# Beispiele

- Expressions
  - Konstantenpropagation (Konstante Teilausdrücke gleich im Compiler ausrechnen)
  - Vorzeichenanalyse (vereinfacht z.B. Typkonvertierung, Ausnahmebehandlung bei Division durch 0)
  - Feldgrenzenüberwachung
- Daten
  - Pointeranalyse (may point to/must point to; Vereinfachung von Zugriffen)
  - Referenzanalyse (Wieviele Pointer zeigen auf mich?; Garbage collection)
- Kontrollfluss
  - Very busy expressions (Werte, die in jedem Kontrollzweig noch einmal verwendet werden)
  - Tote Zweige
  - .... und viele andere mehr

# Anwendungsbeispiel

- Übersetzung eines ALGOL (Pascal)-Arrays in ein C-Array
- Algol (Pascal):
  - `A : array [0:n,0:m] of integer`
  - Zugriff: `A[i,j]`
- C:
  - `int * A`
  - Zugriff: `* (A + i * (m+1) + j)`

# Brute-Force-Übersetzung ALGOL-C

ALGOL:

```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
    A[i,j] := B[i,j] + C[i,j];
    j := j + 1;
  end
  i := i + 1;
end
```

C:

```
i = 0;
while(i <=n) {
  j = 0;
  while(j <=m) {
    tmp = A + i * (m+1) + j;
    *tmp = *(B + i * (m+1)+j)
          + *(C + i*(m+1)+j);
    j = j + 1;
  }
  i = i + 1;
}
```

# Analyse: Available Expressions

## Ziel: Common Subexpression elimination

C alt:

```
i = 0;
while(i <=n) {
  j = 0;
  while(j <=m) {
    tmp = A + i * (m+1) + j;
    *tmp = *(B + i * (m+1)+j)
           + *(C + i*(m+1)+j);
    j = j + 1;
  }
  i = i + 1;
}
```

erste Berechnung

Folgeberechnung

C neu:

```
i = 0;
while(i <=n) {
  j = 0;
  while(j <=m) {
    t1 = i * (m+1)+j;
    tmp = A + t1;
    *tmp = *(B + t1)
           + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
}
```

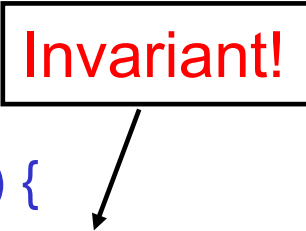


# Analyse: Schleifeninvarianten

## Ziel: Verschiebung von Code aus Schleife

C alt:

```
i = 0;
while(i <=n) {
  j = 0;
  while(j <=m) {
    t1 = i * (m+1)+j;
    tmp = A + t1;
    *tmp = *(B + t1)
          + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
}
```



C neu:

```
i = 0;
while(i <=n) {
  j = 0;
  t2 = i * (m+1);
  while(j <=m) {
    t1 = t2+j;
    tmp = A + t1;
    *tmp = *(B + t1)
          + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
}
```

# Analyse: Induktive Variablen

## Ziel: Komplexitätsreduktion

C alt:

```
i = 0;  
while(i <= n) {  
  j = 0;  
  t2 = i * (m+1);  
  while(j <= m) {  
    t1 = t2+j;  
    tmp = A + t1;  
    *tmp = *(B + t1)  
          + *(C + t1);  
    j = j + 1;  
  }  
  i = i + 1;  
}
```

A box labeled "Induktion" in red text has three arrows pointing to specific parts of the "C alt" code: one to the line `i = 0;`, one to the inner `while(j <= m)` loop, and one to the line `i = i + 1;`.

C neu:

```
i = 0;  
t3 = 0;  
while(i <= n) {  
  j = 0;  
  t2 = t3;  
  while(j <= m) {  
    t1 = t2+j;  
    tmp = A + t1;  
    *tmp = *(B + t1)  
          + *(C + t1);  
    j = j + 1;  
  }  
  i = i + 1;  
  t3 = t3 + m + 1;  
}
```

# Analyse: Equivalent expressions

## Ziel: Copy propagation

C alt:

```
i = 0;
t3 = 0;
while(i <=n) {
  j = 0;
  t2 = t3;
  while(j <=m) {
    t1 = t2+j;
    tmp = A + t1;
    *tmp = *(B + t1)
          + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
  t3 = t3 + m + 1;
}
```

äquivalent

C neu:


```
i = 0;
t3 = 0;
while(i <=n) {
  j = 0;
  t2 = t3;
  while(j <=m) {
    t1 = t3+j;
    tmp = A + t1;
    *tmp = *(B + t1)
          + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
  t3 = t3 + m + 1;
}
```

# Analyse: Live variables

## Ziel: dead code elimination

C alt:

```
i = 0;
t3 = 0;
while(i <=n) {
  j = 0;
  t2 = t3;
  while(j <=m) {
    t1 = t3+j;
    tmp = A + t1;
    *tmp = *(B + t1)
          + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
  t3 = t3 + m + 1;
}
```



C neu:

```
i = 0;
t3 = 0;
while(i <=n) {
  j = 0;
  while(j <=m) {
    t1 = t3+j;
    tmp = A + t1;
    *tmp = *(B + t1)
          + *(C + t1);
    j = j + 1;
  }
  i = i + 1;
  t3 = t3 + m + 1;
}
```

# Fazit des Beispiels

- Optimierungspotential kann auch ohne „Verschulden“ eines Programmierers entstehen  
weitere Beispiele:
  - Feldgrenzenüberwachung
  - Garbage collection
  - Virtuelle/Abstrakte Methoden
  - ...
- Optimierung setzt Analyse voraus
  - korrekt (was behauptet wird, muss stimmen)
  - nicht notwendig exakt (manches stimmt, wird aber nicht behauptet) – Preis: weniger Optimierung
  - effizient

# Eine Analyse im Detail: Abhängigkeitsanalyse

- Einige Variablen werden als „in“, andere als „out“ deklariert.
- Frage: Hängt der Wert der „out“-Variablen nach Ausführung des Programms funktional von den Werten der „in“-Variablen vor Ausführung ab?
- $y = x * z + 25$ ,  $y \in \text{out}$ 
  - ok, falls  $x, z \in \text{in}$
  - D? sonst (dubios = möglicherweise nicht ok)
- $y := 1$ ; while  $x \neq 1$  do  $y := y * x; x := x - 1$ ,  $x \in \text{in}$ ,  $y \in \text{out}$ 
  - ok
  - ohne Initialisierung von  $y$ : D?, weil  $y \notin \text{in}$  und Endwert von  $y$  vom Anfangswert von  $y$  abhängt

# Lösung: Nichtstandardsemantik

- Zustand  $s \rightarrow$  Abstrakter Zustand  $p$ ; pro Variable (statt Wert) Eigenschaft mit Werten ...

... z.B. 0,1,2,3, viele, irgendein (Referenzanalyse)

... z.B.  $<0, \leq 0, = 0, \geq 0, > 0$ , beliebig (Vorzeichenanalyse)

... z.B. 0,1,2,3,..., variabel (Konstantenpropagation)

... z.B. OK, D? (Abhängigkeitsanalyse)

Allen Wertebereichen gemeinsam: bilden kettenvollständige Halbordnungen

Grund: Wollen/Müssen Fixpunkte ausrechnen

## Rechnen mit OK und D?

- $OK \sqsubseteq D?$
- $(x \sqcup y \text{ meint } \sqcup\{x,y\})$
- $OK \sqcup OK = OK$
- $OK \sqcup D? = D? \sqcup OK = D? \sqcup D? = D?$



## Abstrakter Zustand

- $p : V \cup \{\text{control}\} \rightarrow \{\text{OK}, D?\}$
- control Bestandteil eines abstrakten Zustandes z.B. wegen
- if  $x = 1$  then  $y := 1$  else  $y := 2$
- $p(x) = D?$ , alle anderen ok
- Kontrollfluss hängt möglicherweise nicht von „in“ ab, also auch Wert von  $y$  möglicherweise nicht.

# Halbordnung auf abstrakten Zuständen

- Haben: Halbordnung  $\sqsubseteq$  auf  $\{OK, D?\}$
- Def.: Halbordnung auf abstrakten Zuständen:
  - $p \sqsubseteq p'$  falls für alle  $x \in V \cup \{\text{control}\}$ :  $p(x) \sqsubseteq p'(x)$
  - Ist Halbordnung
  - ist kettenvollständig und  $\sqcup M$  ist derjenige abstrakte Zustand mit  $\sqcup M(x) = \sqcup \{p(x) \mid p \in M\}$

# Beweis

- z.Z.:  $\sqsubseteq$  ist kettenvollständig und  $\sqcup M$  ist derjenige abstrakte Zustand mit  $\sqcup M(x) = \sqcup \{p(x) \mid p \in M\}$

(Wohldefiniertheit) Sei  $M$  Kette.

Nach Def.  $\sqsubseteq$  ist für jedes  $x$   $\{p(x) \mid p \in M\}$  Kette.

Da  $\sqsubseteq$  auf  $\{OK, D?\}$  kettenvollständig, ex.  $\sqcup \{p(x) \mid p \in M\}$  .

(Obere Schranke) Sei  $p \in M$ .

Weil  $\sqcup$  kleinste obere Schranke von  $\{p(x) \mid p \in M\}$ ,  
ist  $p(x) \sqsubseteq \sqcup \{p(x) \mid p \in M\}$  für alle  $x$ .

(Kleinste obere Schranke). Sei  $p^*$  obere Schranke von  $M$ .

Zeigen:  $\sqcup M \sqsubseteq p^*$ . Weil  $p^*$  obere Schranke von  $M$ , ist für alle  $p \in M$ :  $p \sqsubseteq p^*$ . Also für alle  $x$ :  $p(x) \sqsubseteq p^*(x)$ . Also  $\sqcup \{p(x) \mid p \in M\} \sqsubseteq p^*(x)$ .

Also:  $\sqcup M \sqsubseteq p^*$ .

# Die Analyse: Ausdrücke

- $PA: \text{Arithm.Expressions} \rightarrow (\text{Abstr.State} \rightarrow \{\text{OK}, D?\})$
- $PA \llbracket n \rrbracket (p) = \text{OK}$ , falls  $p(\text{control}) = \text{OK}$ , sonst  $D?$
- $PA \llbracket x \rrbracket (p) = p(x)$ , falls  $p(\text{control}) = \text{OK}$ , sonst  $D?$
- $PA \llbracket E1 + E2 \rrbracket (p) = PA \llbracket E1 \rrbracket (p) \sqcup PA \llbracket E2 \rrbracket (p)$  (-\*/ analog)
- $PB: \text{Bool.Expressions} \rightarrow (\text{Abstr.State} \rightarrow \{\text{OK}, D?\})$
- analog

# Die Analyse: Commands

- $PS: \text{Commands} \rightarrow (\text{Abstr.State} \rightarrow \text{Abstr.State})$
- $PS \llbracket x := E \rrbracket (p) = p[x \rightarrow PA/PB(E)(p)]$
- $PS \llbracket \text{skip} \rrbracket = \text{id}$
- $PS \llbracket S1; S2 \rrbracket = PS \llbracket S2 \rrbracket \circ PS \llbracket S1 \rrbracket$
- $PS \llbracket \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \rrbracket = \text{cnd}(PB(b), PS \llbracket S1 \rrbracket, PS \llbracket S2 \rrbracket)$ 
  - $\text{cnd}(f, h1, h2)(p) = h1(p) \sqcup h2(p)$ , falls  $f(p) = \text{OK}$ , sonst LOST.
  - $\text{LOST}(p)(x) = D?$  für alle  $x$  (einschließlich control)
- $PS \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket = \text{FIX } H \text{ mit}$   
 $H(g) = \text{cnd}(PB(b), goPS \llbracket S \rrbracket, \text{id})$

# Beispiele

- $y := x$ 
  - $p(x) = \text{OK}, p(y) = D?, p(\text{control}) = \text{OK}$   
 $\rightarrow \text{PS } \llbracket y:=x \rrbracket (p)(x) = \text{PS } \llbracket y:=x \rrbracket (p)(y) = \text{PS } \llbracket y:=x \rrbracket (p)(\text{control}) = \text{OK}$
  - $p(x) = D?, p(y) = \text{OK}, p(\text{control}) = \text{OK}$   
 $\rightarrow \text{PS } \llbracket y:=x \rrbracket (p)(x) = \text{PS } \llbracket y:=x \rrbracket (p)(y) = D?, \text{PS } \llbracket y:=x \rrbracket (p)(\text{control}) = \text{OK}$
- $\text{if } x=x \text{ then } z:=y \text{ else } y:=z$ 
  - $p(x) = p(y) = \text{OK}, p(z) = D?$   
 $\rightarrow \text{PS } \llbracket \text{if } x=x \text{ then } z:=y \text{ else } y:=z \rrbracket (p)(z) = D?$
  - $p(x) = D?, p(y) = p(z) = \text{OK}$   
 $\rightarrow \text{PS } \llbracket \text{if } x=x \text{ then } z:=y \text{ else } y:=z \rrbracket (p)(x,y,z,\text{control}) = D?$

# Beispiele

- $y := 1; \text{ while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ end}$ 
  - $p(x) = \text{OK}, p(y) = D?, p(\text{control}) = \text{OK}$

... müssen Fixpunkt berechnen:

FIX H mit

$H(g) = \text{cnd}(\text{PB } \llbracket x \neq 1 \rrbracket, \text{goPS } \llbracket y := y * x; x := x - 1 \rrbracket, \text{id}),$

also  $H(g)(p) = \text{LOST}$ , falls  $p(\text{control}) = D?$  oder  $p(x) = D?$ , sonst  $g(p) \sqcup p$

1.  $\perp(p) = \text{INIT}$  für alle  $p$  (INIT ordnet allen Variablen OK zu)

2.  $H(\perp)(p) = \text{LOST}$ , falls  $p(\text{control}) = D?$  oder  $p(x) = D?$ , sonst  $p$

3.  $H(H(\perp))(p) = \text{LOST}$ , falls  $p(\text{control}) = D?$  oder  $p(x) = D?$ , sonst  $p$

$\rightarrow \text{FIX } H = H(\perp)$

$\rightarrow \text{PS } \llbracket y := 1; \text{ while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ end } \rrbracket (p)(x)$

$= \text{PS } \llbracket y := x \rrbracket (p)(x, y, \text{control}) = \text{OK}$

# Aussagen

1. Diese Analyse ist wohldefiniert
  - Technik: Fixpunkttheorie
2. Diese Analyse ist sicher: Wenn die Analyse für eine Variable OK ergibt, ist diese funktional abhängig von den Inputs.
  - Def.:  $s \equiv_p s'$ , falls: Wenn  $p(\text{control})$  und  $p(x) = \text{ok}$ , so  $s(x) = s'(x)$
  - Satz: Für alle  $S$ : Wenn  $s \equiv_p s'$ , so kreist  $S$  sowohl in  $s$  als auch  $s'$ , oder  $S_{ds} \llbracket s \rrbracket \equiv_{pS} \llbracket S \rrbracket S_{ds} \llbracket s' \rrbracket$



# Terminierung der Analyse

- 1. Aussage (leicht zu sehen, aber schlecht geschätzt):  
Fixpunkt für while b do S end ist

$$\sqcup \{ \perp, F(\perp), F(F(\perp)), \dots, F^{(2^{m+1})2^{(m+1)}}(\perp) \} = F^{(2^{m+1})2^{(m+1)}}(\perp)$$

wobei m die Zahl der in b und S vorkommenden Variablen ist

Gründe:

- nicht vorkommende Variablen irrelevant
- für vorkommende Variablen + control:  $2^{m+1}$  Zustände
- für vorkommende Variablen + control:  $(2^{m+1})^{2^{(m+1)}}$  Funktionen

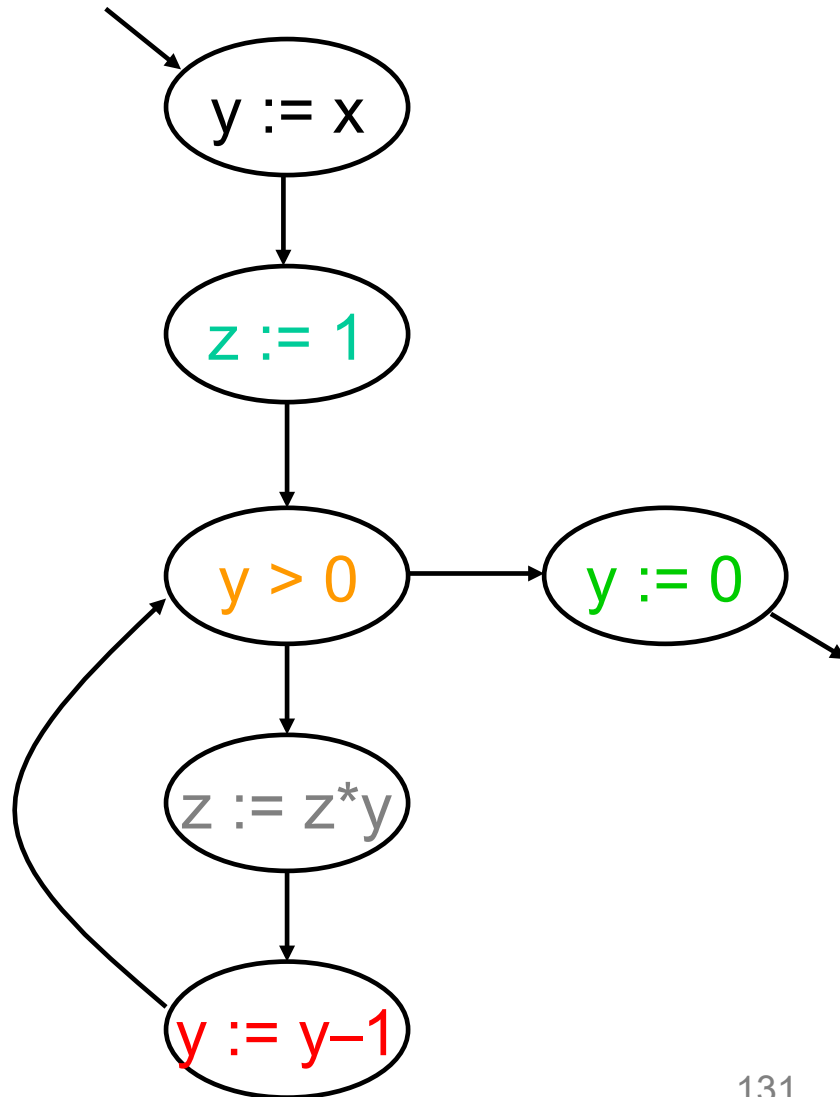
2. Aussage (besser geschätzt, aber mit detaillierter Analyse):  
 $(m+1)^2$  Iterationen reichen.

# Andere Analysen

- Nutzen: Flussgraph
  - Knoten für Zuweisungen und Tests
  - Kanten  $x \rightarrow y$  für „nach  $x$  kann möglicherweise  $y$  ausgeführt werden“
- , ohne semantikfreie (d.h. rein struktursichernde Elemente)

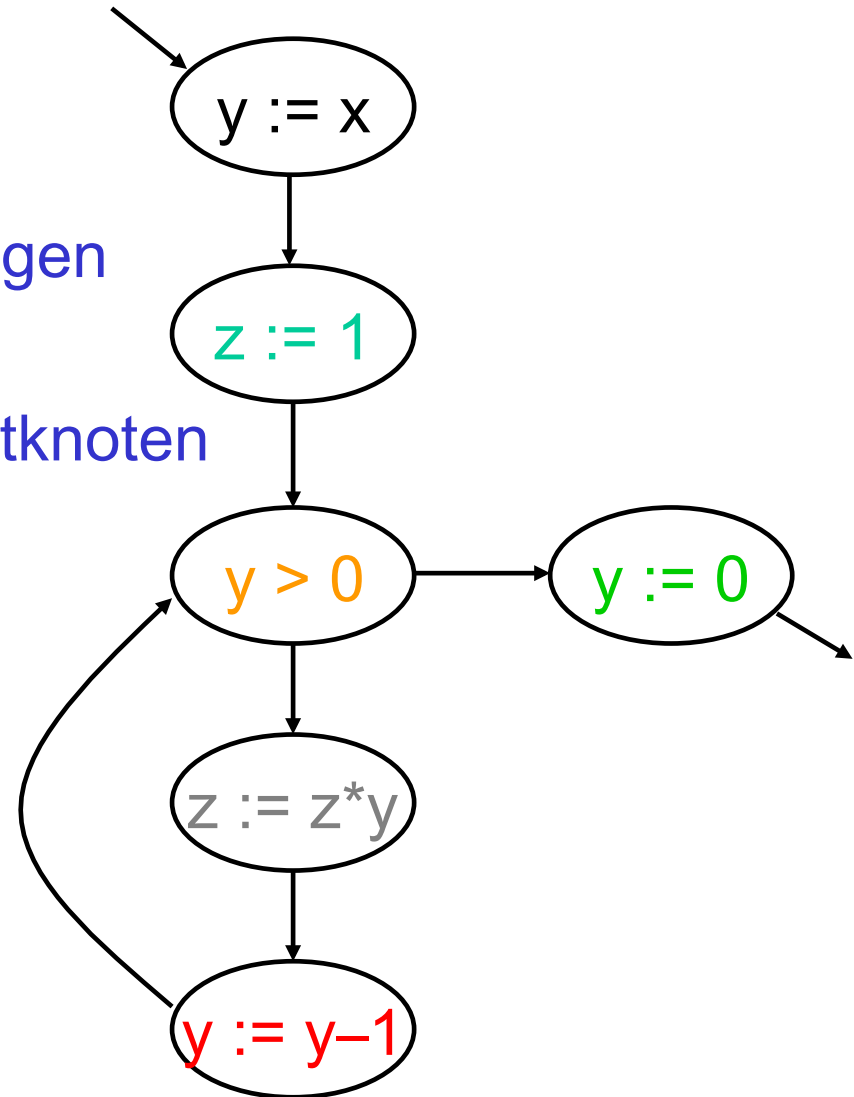
# Flussgraph: Beispiel

```
y := x;  
z := 1;  
while y > 0 do  
    z := z * y;  
    y := y - 1;  
end  
y := 0;
```



# Allgemein für Sprache W

- Für Statement S Knotenmengen
  - $\text{nodes}(S)$  – alle Knoten
  - $\text{init}(S)$  – ein Element, Startknoten
  - $\text{final}(S)$  – Endknoten
- Kantenmenge
  - $\text{flow}(S)$  – alle Kanten



# Induktive Definition

- für  $S = x := E$ 
  - $k := \text{new node}(\text{„}x := E\text{“})$
  - $\text{nodes}(S) = \text{init}(S) = \text{final}(S) = \{k\}$
  - $\text{flow}(S) = \emptyset$
- für  $S = \text{skip}$  analog
- für  $S = S1; S2$ 
  - $\text{nodes}(S) = \text{nodes}(S1) \cup \text{nodes}(S2)$
  - $\text{init}(S) = \text{init}(S1)$
  - $\text{final}(S) = \text{final}(S2)$
  - $\text{flow}(S) = \text{flow}(S1) \cup \text{flow}(S2) \cup \{k\} \times (\text{init}(S1) \cup \text{init}(S2))$
- für  $S = \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}$ 
  - $k := \text{new node}(\text{„}b\text{“})$
  - $\text{nodes}(S) = \text{nodes}(S1) \cup \text{nodes}(S2) \cup \{k\}$
  - $\text{init}(S) = \{k\}$
  - $\text{final}(S) = \text{final}(S1) \cup \text{final}(S2)$
  - $\text{flow}(S) = \text{flow}(S1) \cup \text{flow}(S2) \cup \{k\} \times (\text{init}(S1) \cup \text{init}(S2))$
- für  $S = \text{while } b \text{ do } S1 \text{ end}$ 
  - $k = \text{new node}(\text{„}b\text{“})$
  - $\text{nodes}(S) = \text{nodes}(S1) \cup \{k\}$
  - $\text{init}(S) = \text{final}(S) = \{k\}$
  - $\text{flow}(S) = \text{flow}(S1) \cup \{k\} \times (\text{init}(S1) \cup \text{final}(S1))$

# Analyse 1: Available Expressions

- Zu einem Knoten des Flussgraphen, bestimme alle Expressions, die auf allen Pfaden
  - bereits berechnet sind
  - seitdem nicht modifiziert wurden

- Beispiel

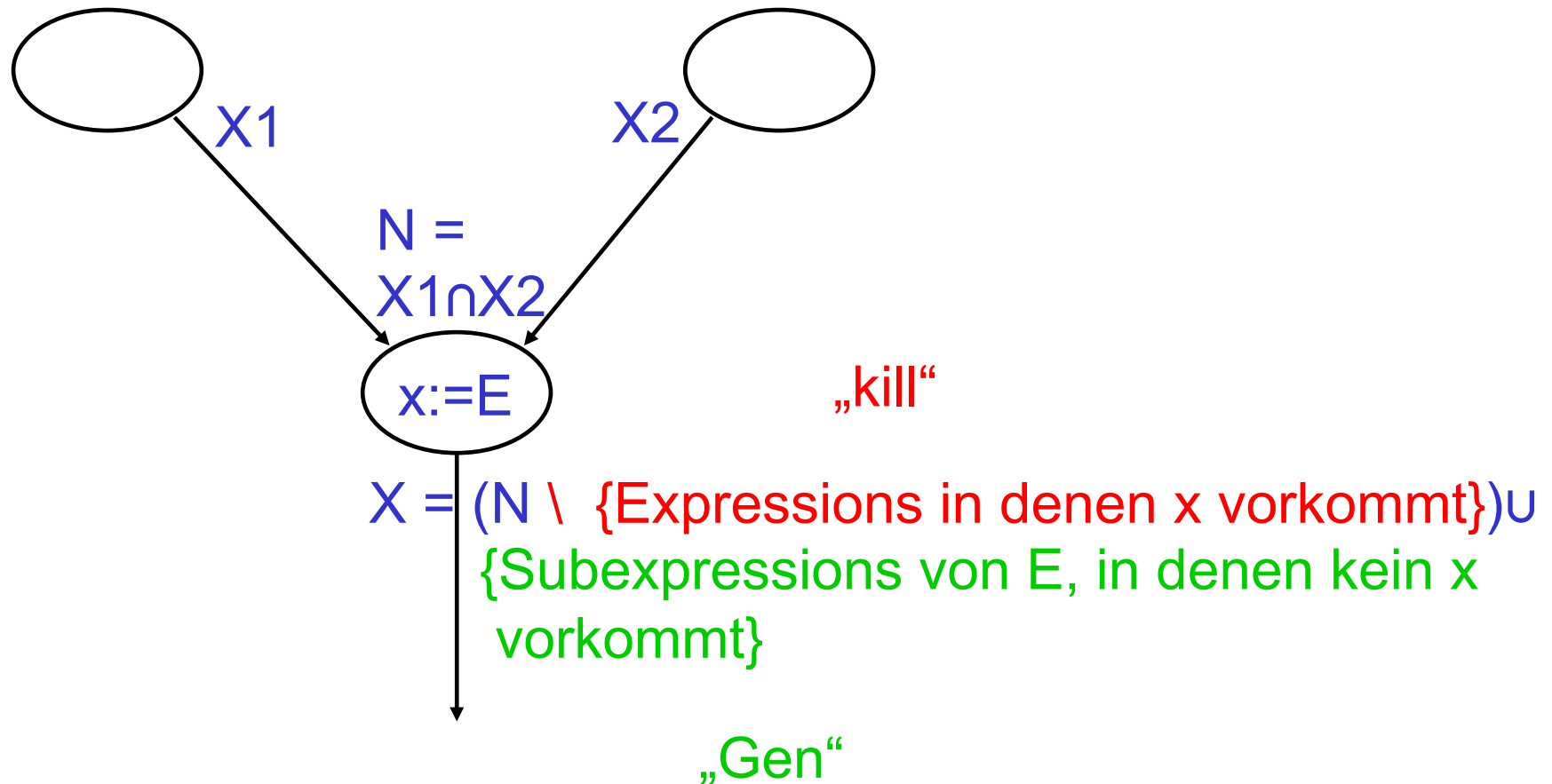


`x := a+b; y := a*b; while y > a+b do a := a + 1; x := a+b; end`

kann ggf umgeformt werden zu

`x := a+b; y := a*b; while y > x do a := a + 1; x := a+b; end`

## Available Expressions: Idee



# Available Expressions: Ausführung

$$\text{kill}(x := E) = \{ E' \in \text{EXP} \mid x \sqsubseteq E' \}$$

$$\text{kill}(\text{skip}) = \emptyset$$

$$\text{kill}(b) = \emptyset$$

$$\text{gen}(x := E) = \{ E' \mid E' \sqsubseteq E, !x \sqsubseteq E' \}$$

$$\text{gen}(\text{skip}) = \emptyset$$

$$\text{gen}(b) = \{ E' \mid E' \sqsubseteq b \}$$

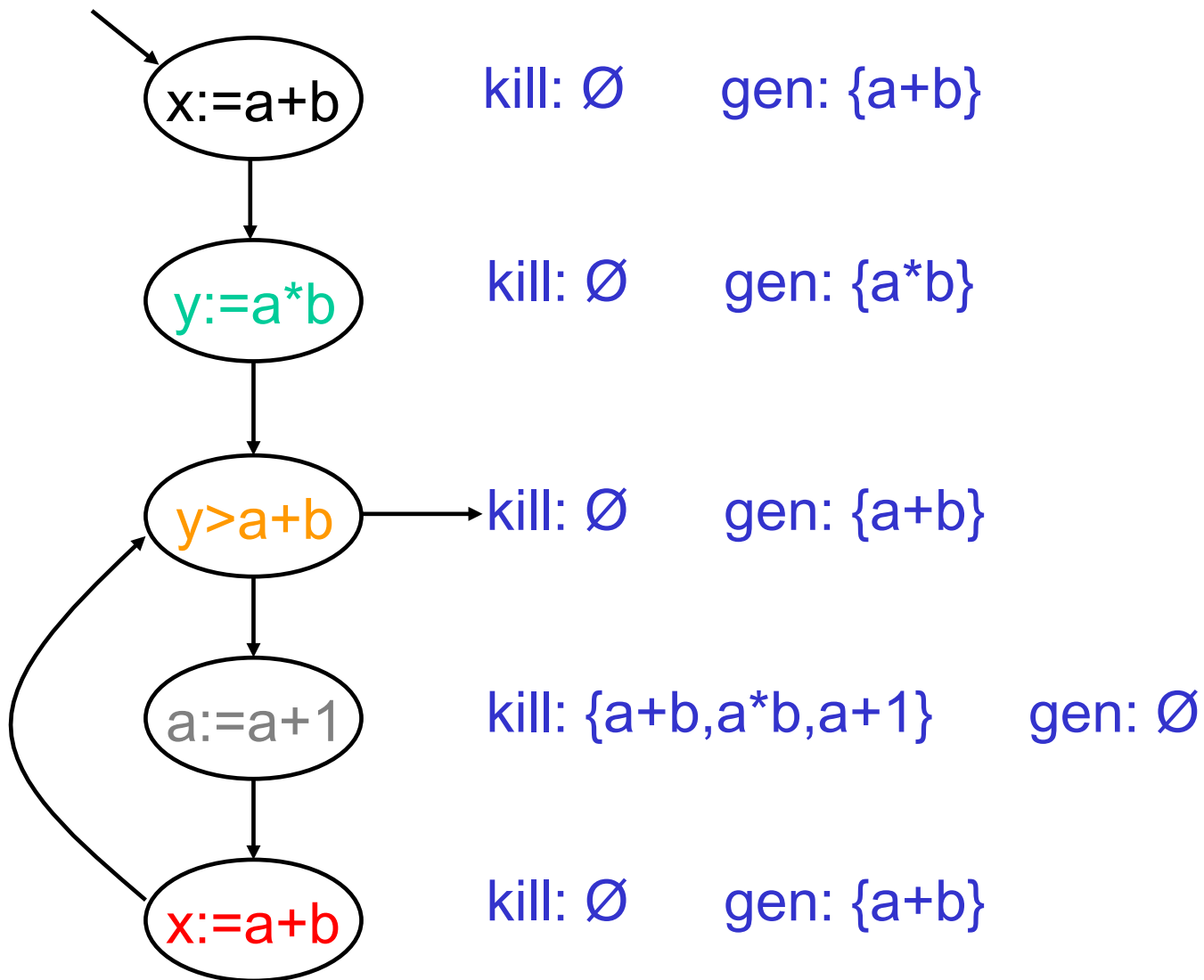
Bestimmung der Available Expressions mittels Lösung von Datenflussgleichungen

- Pro Knoten  $k$  Variable  $k_{\text{entry}}, k_{\text{exit}}$
- Gleichungen
  - $k_{\text{entry}} = \emptyset$  für  $k \in \text{init}(S)$
  - $k_{\text{entry}} = \bigcap_{(k', k) \in \text{flow}(S)} k'_{\text{exit}}$  für  $k \notin \text{init}(S)$
  - $k_{\text{exit}} = (k_{\text{entry}} \setminus \text{kill}(k)) \cup \text{gen}(k)$



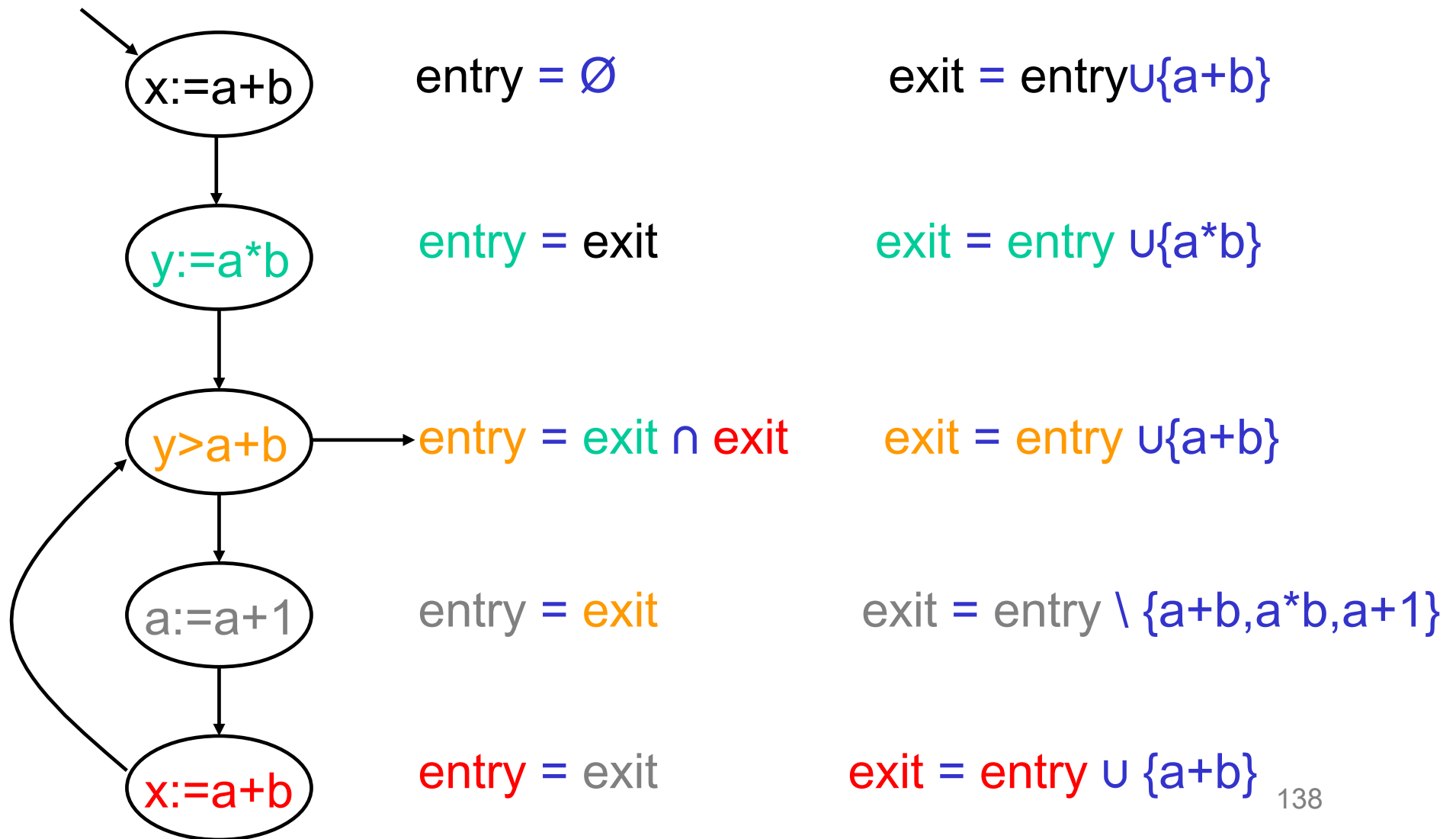
# Beispiel: kill, gen

$x := a+b$ ;  $y := a*b$ ; while  $y > a+b$  do  $a := a + 1$ ;  $x := a+b$ ; end



# Beispiel: Gleichungen

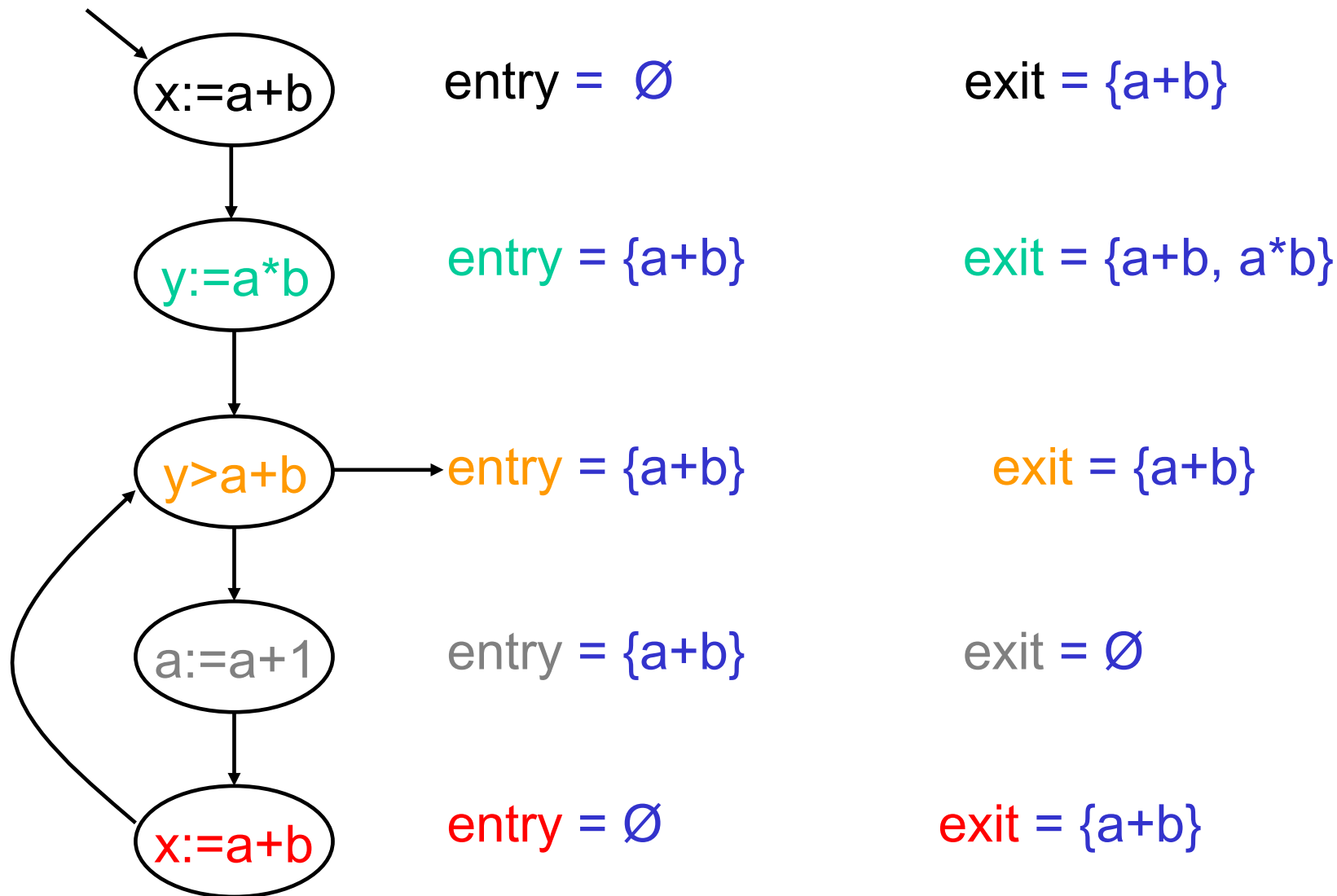
$x := a+b$ ;  $y := a*b$ ; while  $y > a+b$  do  $a := a + 1$ ;  $x := a+b$ ; end



## Beispiel:

Lösung = größte Lösung der Gleichungen

$x := a+b$ ;  $y := a*b$ ; while  $y > a+b$  do  $a := a + 1$ ;  $x := a+b$ ; end



# Bestimmung der größten Lösung

- Setze alle Variablen auf Menge aller Epressions
- REPEAT
  - für alle Gleichungen parallel
    - linke Seite := rechte Seite
  - UNTIL nothing changes

## Beispiel: Start

entry =  $\emptyset$   
a+b, a\*b, a+1

exit = entry  $\cup$  {a+b}  
a+b, a\*b, a+1

entry = exit  
a+b, a\*b, a+1

exit = entry  $\cup$  {a\*b}  
a+b, a\*b, a+1

entry = exit  $\cap$  exit  
a+b, a\*b, a+1

exit = entry  $\cup$  {a+b}  
a+b, a\*b, a+1

entry = exit  
a+b, a\*b, a+1

exit = entry  $\setminus$  {a+b, a\*b, a+1}  
a+b, a\*b, a+1

entry = exit  
a+b, a\*b, a+1

exit = entry  $\cup$  {a+b}  
a+b, a\*b, a+1

## Beispiel: 1. Iteration

entry =  $\emptyset$   
~~a+b, a\*b, a+1~~

exit = entry  $\cup$  {a+b}  
a+b, a\*b, a+1

entry = exit  
a+b, a\*b, a+1

exit = entry  $\cup$  {a\*b}  
a+b, a\*b, a+1

entry = exit  $\cap$  exit  
a+b, a\*b, a+1

exit = entry  $\cup$  {a+b}  
a+b, a\*b, a+1

entry = exit  
a+b, a\*b, a+1

exit = entry  $\setminus$  {a+b, a\*b, a+1}  
~~a+b, a\*b, a+1~~

entry = exit  
a+b, a\*b, a+1

exit = entry  $\cup$  {a+b}  
a+b, a\*b, a+1

## Beispiel: 2. Iteration

$$\text{entry} = \emptyset$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$
~~$$a+b, a*b, a+1$$~~

$$\text{entry} = \text{exit}$$
~~$$a+b, a*b, a+1$$~~

$$\text{exit} = \text{entry} \cup \{a*b\}$$
~~$$a+b, a*b, a+1$$~~

$$\text{entry} = \text{exit} \cap \text{exit}$$
~~$$a+b, a*b, a+1$$~~

$$\text{exit} = \text{entry} \cup \{a+b\}$$
~~$$a+b, a*b, a+1$$~~

$$\text{entry} = \text{exit}$$
~~$$a+b, a*b, a+1$$~~

$$\text{exit} = \text{entry} \setminus \{a+b, a*b, a+1\}$$

$$\text{entry} = \text{exit}$$
~~$$a+b, a*b, a+1$$~~

$$\text{exit} = \text{entry} \cup \{a+b\}$$
~~$$a+b, a*b, a+1$$~~

## Beispiel: 3. Iteration

$$\text{entry} = \emptyset$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$
~~$$a+b, a*b, a+1$$~~

$$\text{exit} = \text{entry} \cup \{a*b\}$$

$$a+b, a*b, a+1$$

$$\text{entry} = \text{exit} \cap \text{exit}$$

$$a+b, a*b, a+1$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$a+b, a*b, a+1$$

$$\text{entry} = \text{exit}$$

$$a+b, a*b, a+1$$

$$\text{exit} = \text{entry} \setminus \{a+b, a*b, a+1\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$
~~$$a+b, a*b, a+1$$~~



## Beispiel: 4. Iteration

$$\text{entry} = \emptyset$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a*b\}$$

$$\text{entry} = \text{exit} \cap \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \setminus \{a+b, a*b, a+1\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

## Beispiel: 5. Iteration

$$\text{entry} = \emptyset$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a*b\}$$

$$\text{entry} = \text{exit} \cap \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \setminus \{a+b, a*b, a+1\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

## Beispiel: 6. Iteration

$$\text{entry} = \emptyset$$

$$\text{exit}_{a+b} = \text{entry} \cup \{a+b\}$$

$$\text{entry}_{a+b} = \text{exit}_{a+b}$$

$$\text{exit}_{a+b, a*b} = \text{entry}_{a+b} \cup \{a*b\}$$

$$\text{entry}_{a+b} = \text{exit}_{a+b} \cap \text{exit}_{a*b}$$

$$\text{exit}_{a+b} = \text{entry}_{a+b} \cup \{a+b\}$$

$$\text{entry}_{a+b, a*b, a+1} = \text{exit}_{a+b, a*b}$$

$$\text{exit}_{a+b, a*b, a+1} = \text{entry}_{a+b, a*b, a+1} \cup \{a+1\}$$

$$\text{entry}_{a+b, a*b, a+1} = \text{exit}_{a+b, a*b, a+1}$$

$$\text{exit}_{a+b, a*b, a+1} = \text{entry}_{a+b, a*b, a+1} \cup \{a+b\}$$

## Beispiel: 7. Iteration = no change

$$\text{entry} = \emptyset$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a*b\}$$

$$\text{entry} = \text{exit} \cap \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \setminus \{a+b, a*b, a+1\}$$

$$\text{entry} = \text{exit}$$

$$\text{exit} = \text{entry} \cup \{a+b\}$$

# Analyse: Reaching definitions

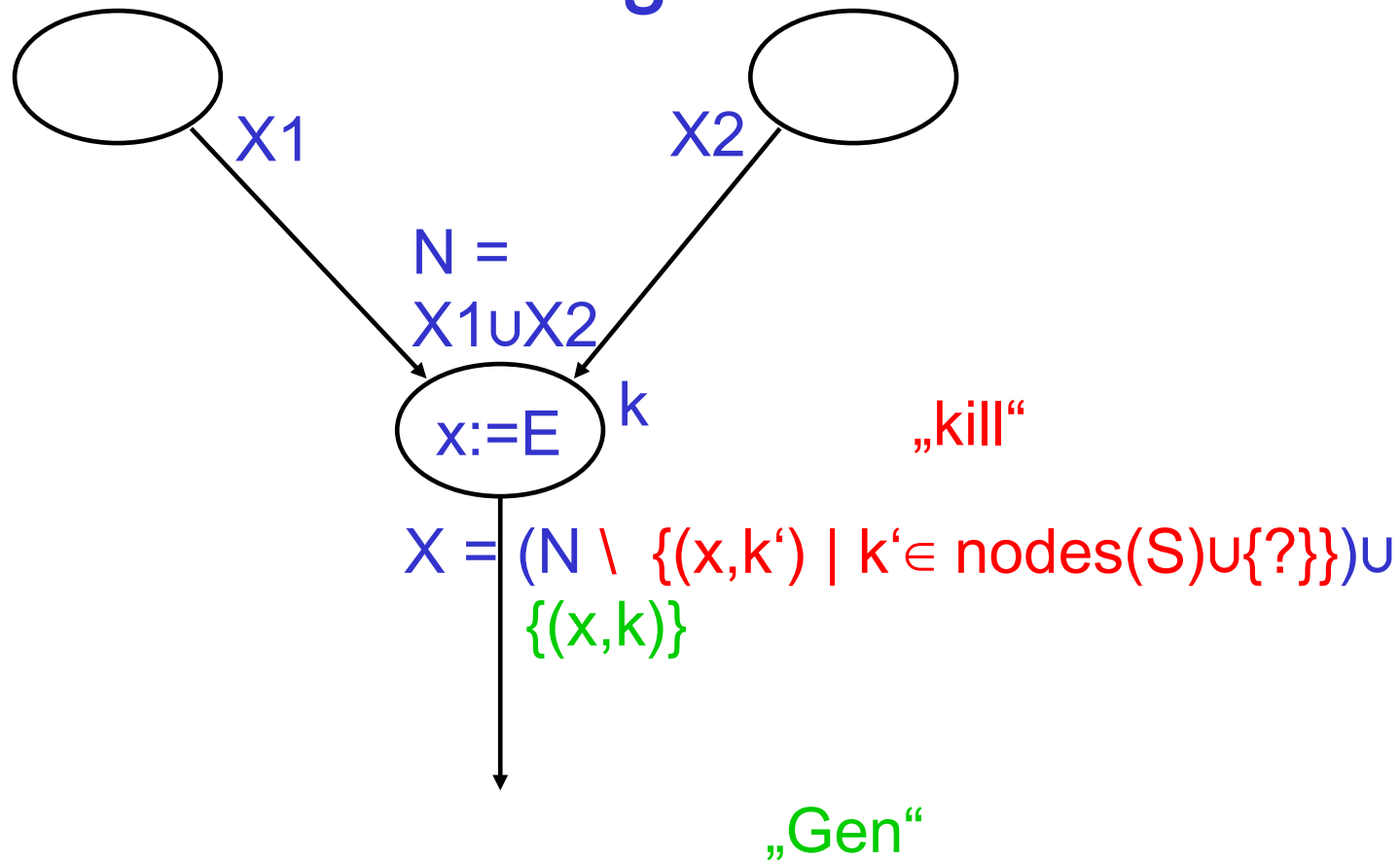
- Für einen Knoten k: Welche Zuweisungen sind auf mind. einem Pfad zu k noch nicht überschrieben?

- Beispiel



`x := 5; y := 1; while x > 1 do y := x*y; x := x - 1 end`

## Reaching definitions: Idee



# Reaching definitions: Ausführung

$$\text{kill}(x := E) = \{ (x, k) \mid (k \in \text{nodes}(S) \cup \{?\}) \}$$

$$\text{kill}(\text{skip}) = \emptyset$$

$$\text{kill}(b) = \emptyset$$

$$\text{gen}(x := E) = \{(x, \text{this})\}$$

$$\text{gen}(\text{skip}) = \emptyset$$

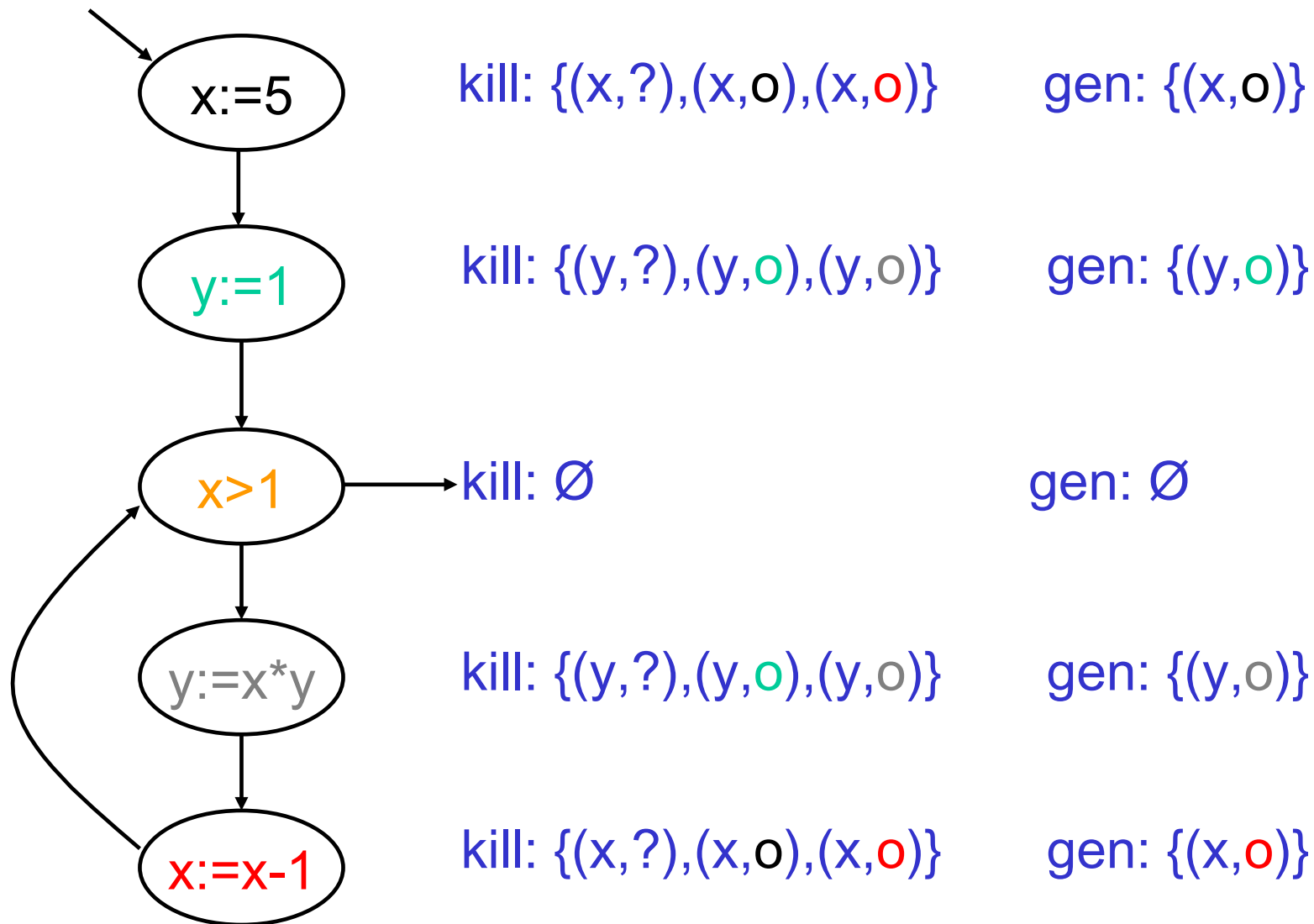
$$\text{gen}(b) = \emptyset$$

Bestimmung der reaching definitions mittels Lösung von Datenflussgleichungen

- Pro Knoten  $k$  Variable  $k_{\text{entry}}, k_{\text{exit}}$
- Gleichungen
  - $k_{\text{entry}} = \{(x, ?) \mid x \in \text{VAR}\}$  für  $k \in \text{init}(S)$
  - $k_{\text{entry}} = \bigcup_{(k', k) \in \text{flow}(S)} k'_{\text{exit}}$  für  $k \notin \text{init}(S)$
  - $k_{\text{exit}} = (k_{\text{entry}} \setminus \text{kill}(k)) \cup \text{gen}(k)$

## Beispiel: kill, gen

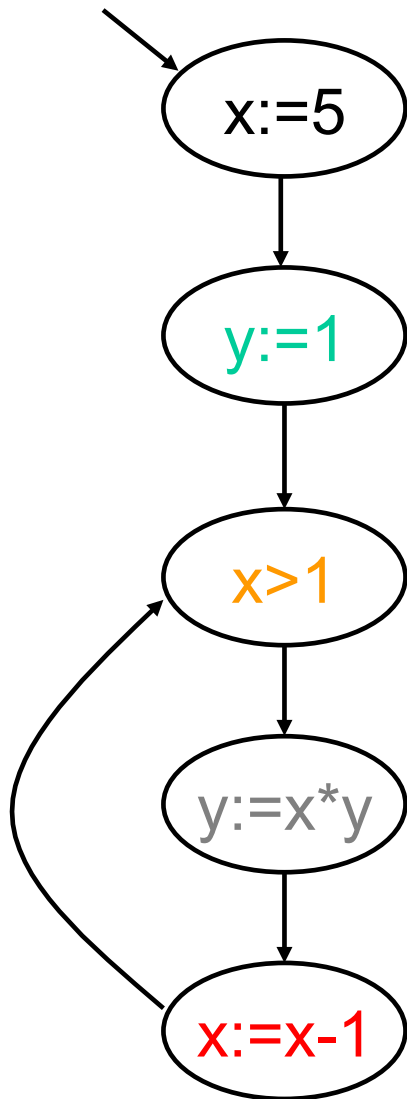
$x := 5; y := 1; \text{while } x > 1 \text{ do } y := x * y; x := x - 1; \text{end}$





# Beispiel: Gleichungen

$x := 5; y := 1; \text{while } x > 1 \text{ do } yx := x*y; x := x-1; \text{end}$



$\text{entry} = \{(x, ?), (y, ?)\}$

$\text{exit} = \text{entry} \setminus \{(x, ?), (x, o), (x, \textcolor{red}{o})\} \cup \{(x, o)\}$

$\text{entry} = \text{exit}$

$\text{exit} = \text{entry} \setminus \{(y, ?), (y, o), (y, \textcolor{gray}{o})\} \cup \{(y, \textcolor{teal}{o})\}$

$\text{entry} = \text{exit} \cup \text{exit}$

$\text{exit} = \text{entry}$

$\text{entry} = \text{exit}$

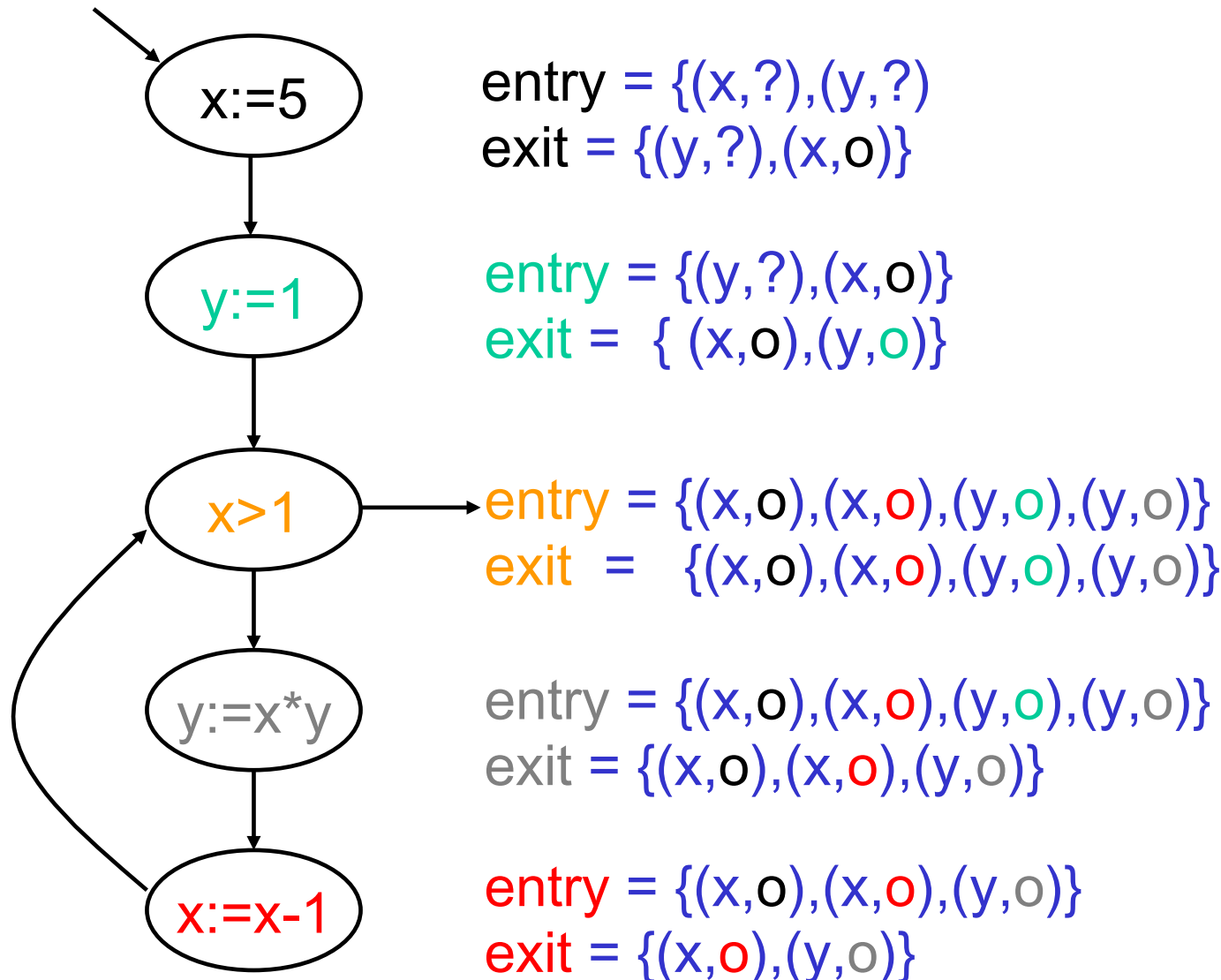
$\text{exit} = \text{entry} \setminus \{(y, ?), (y, \textcolor{teal}{o}), (y, \textcolor{gray}{o})\} \cup \{(y, \textcolor{gray}{o})\}$

$\text{entry} = \text{exit}$

$\text{exit} = \text{entry} \setminus \{(x, ?), (x, o), (x, \textcolor{red}{o})\} \cup \{(x, \textcolor{red}{o})\}$

# Beispiel: Lösung = kleinste Lösung

$x := 5; y := 1; \text{while } x > 1 \text{ do } yx := x * y; x := x - 1; \text{end}$



# Analyse: Very busy expressions

- Expression ist very busy an einem Knoten, falls ihr Wert auf jedem Kontrollpfad noch einmal benutzt wird (ohne dass vorkommende Variablen ihren Wert ändern)

Ziel: Very busy expressions können gleich berechnet werden

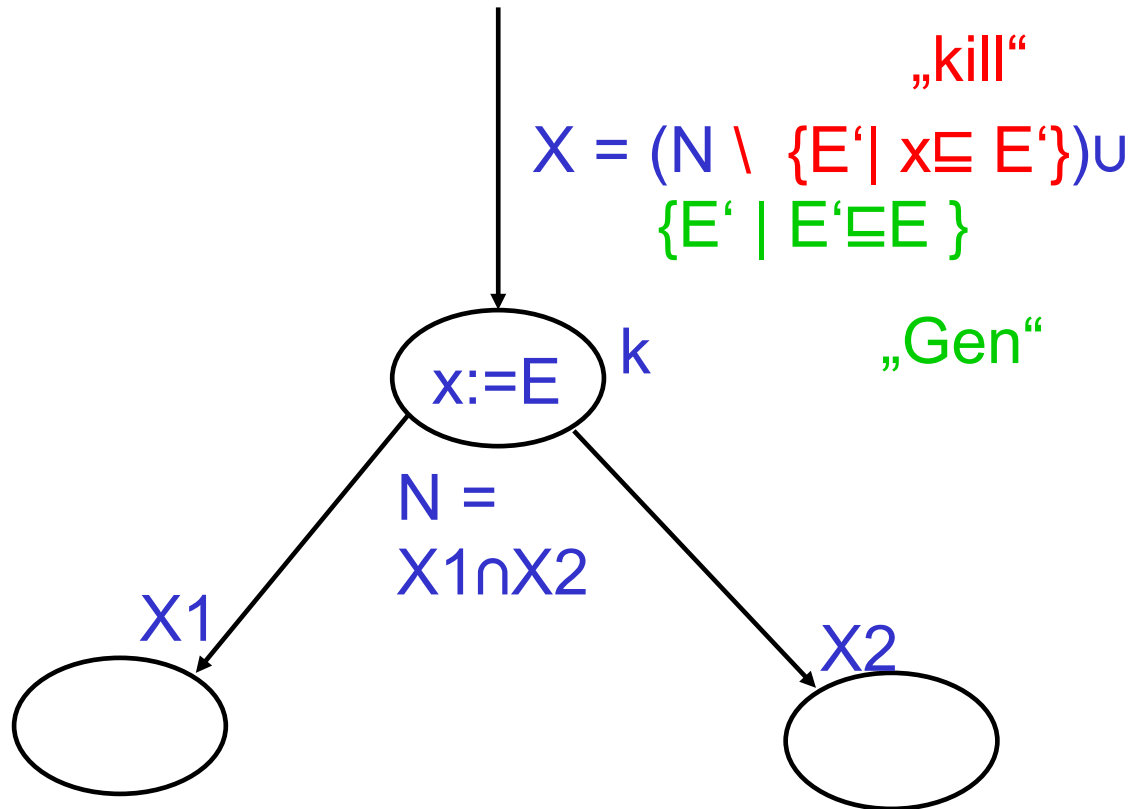
Beispiel:

very busy:  $b-a$ ,  $a-b$



```
if a>b then x:=  $b-a$  ;y := a-b else y:=  $b-a$  ;x:= a-b end
```

## Very busy expressions: Idee



# Very busy expressions: Ausführung

$$\text{kill}(x := E) = \{ E' \mid (x \sqsubseteq E') \}$$

$$\text{kill}(\text{skip}) = \emptyset$$

$$\text{kill}(b) = \emptyset$$

$$\text{gen}(x := E) = \{ E' \mid E' \sqsubseteq E \}$$

$$\text{gen}(\text{skip}) = \emptyset$$

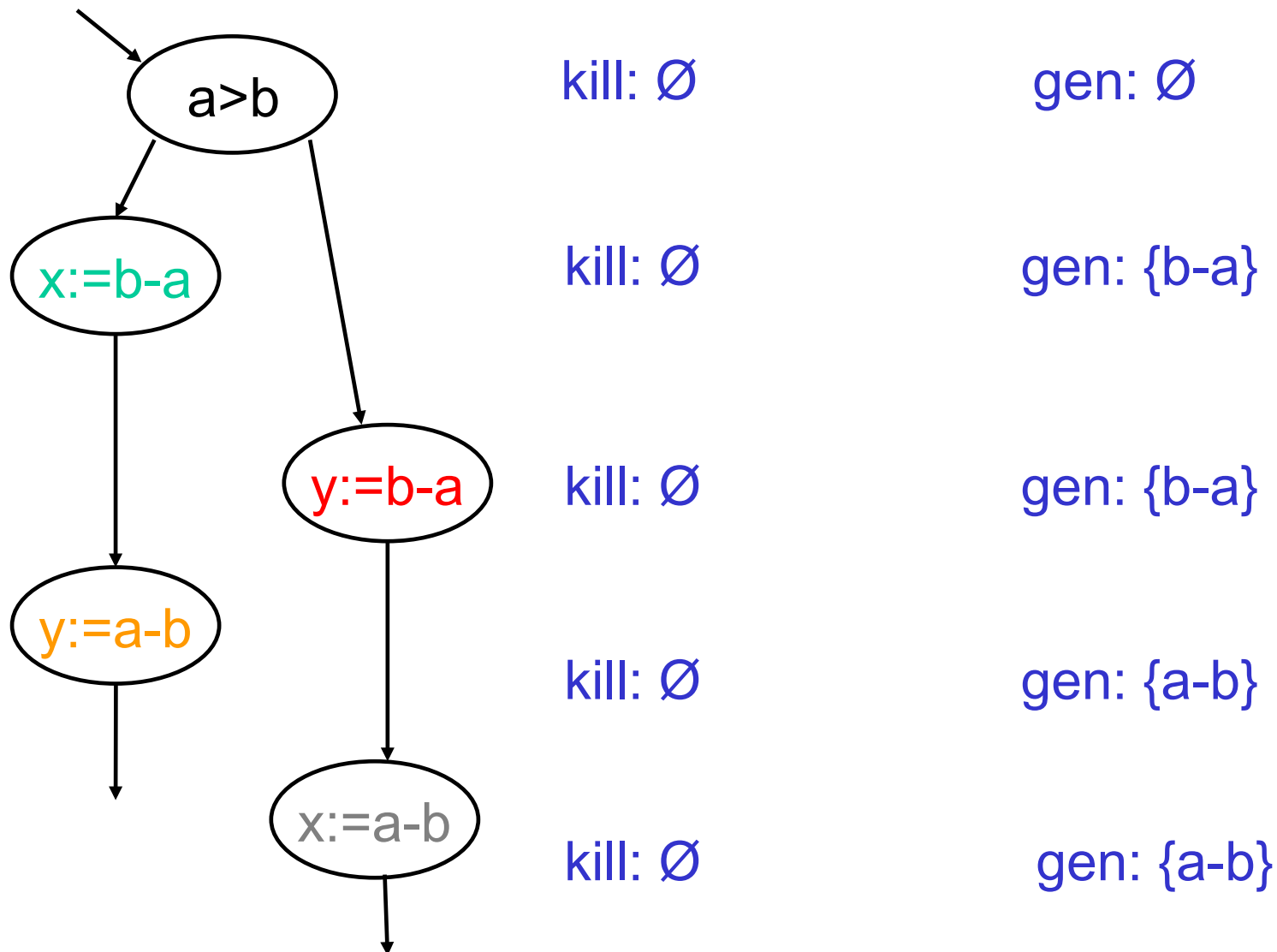
$$\text{gen}(b) = \{ E' \mid E' \sqsubseteq b \}$$

Bestimmung der Very busy Expressions mittels Lösung von Datenflussgleichungen

- $k_{\text{exit}} = \emptyset$  für  $k \in \text{final}(S)$
- $k_{\text{exit}} = \bigcap_{(k, k') \in \text{flow}(S)} k'_{\text{entry}}$  für  $k \notin \text{final}(S)$
- $k_{\text{entry}} = (k_{\text{exit}} \setminus \text{kill}(k)) \cup \text{gen}(k)$

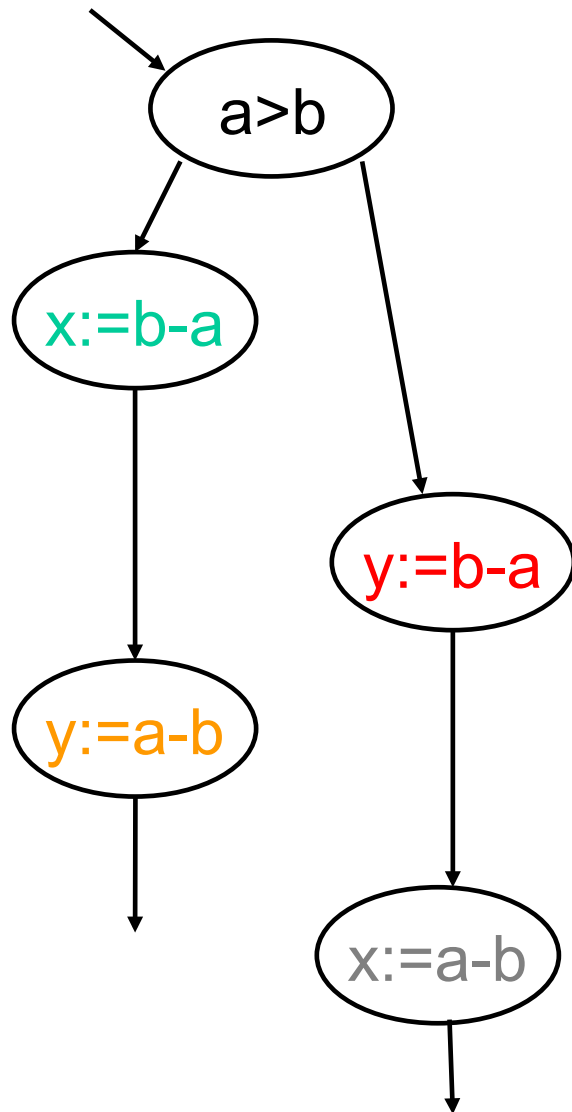
## Beispiel: kill, gen

if  $a > b$  then  $x := b - a$  ;  $y := a - b$  else  $y := b - a$  ;  $x := a - b$  end



# Beispiel: Gleichungen

if  $a > b$  then  $x := b - a$  ;  $y := a - b$  else  $y := b - a$  ;  $x := a - b$  end



entry = exit

exit = entry  $\cap$  entry

entry = exit  $\cup$  {b-a}

exit = entry

entry = exit  $\cup$  {b-a}

exit = entry

entry = exit  $\cup$  {a-b}

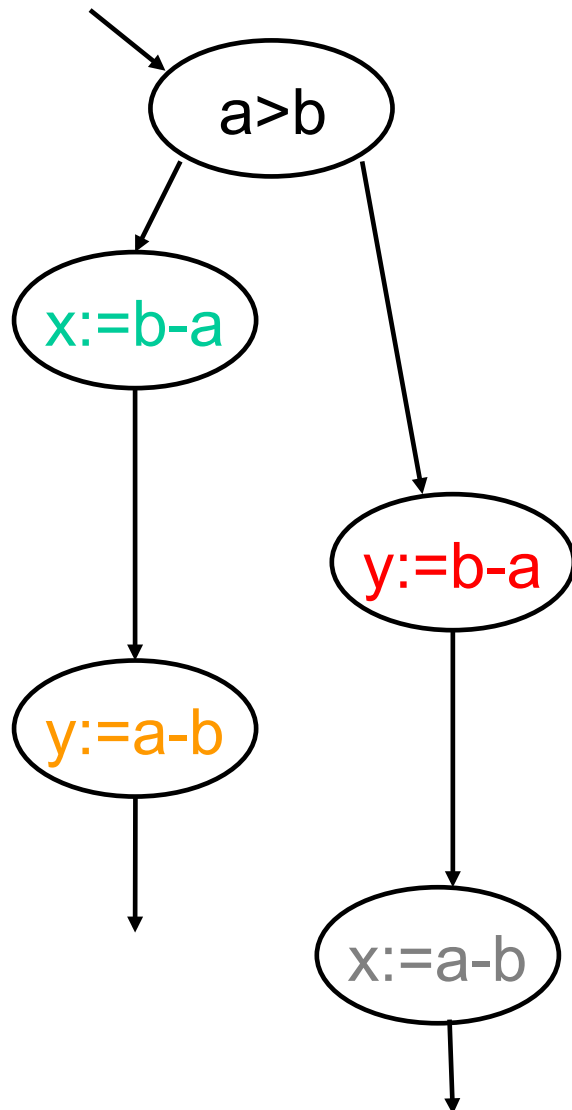
exit =  $\emptyset$

entry = exit  $\cup$  {a-b}

exit =  $\emptyset$

# Beispiel: Lösung = größter Fixpunkt

if  $a > b$  then  $x := b - a$  ;  $y := a - b$  else  $y := b - a$  ;  $x := a - b$  end



entry =  $\{a-b, b-a\}$

exit =  $\{a-b, b-a\}$

entry =  $\{a-b, b-a\}$

exit =  $\{a-b\}$

entry =  $\{a-b, b-a\}$

exit =  $\{a-b\}$

entry =  $\{a-b\}$

exit =  $\emptyset$

entry =  $\{a-b\}$

exit =  $\emptyset$



# Analyse: Live variables

- Variable ist live an einem Knoten, falls ihr Wert auf mindestens einem Kontrollpfad noch einmal benutzt wird (ohne dass ihr Wert überschrieben wurde)

Ziel: Zuweisungen an Variablen, die nicht live sind, können gestrichen werden

Beispiel:

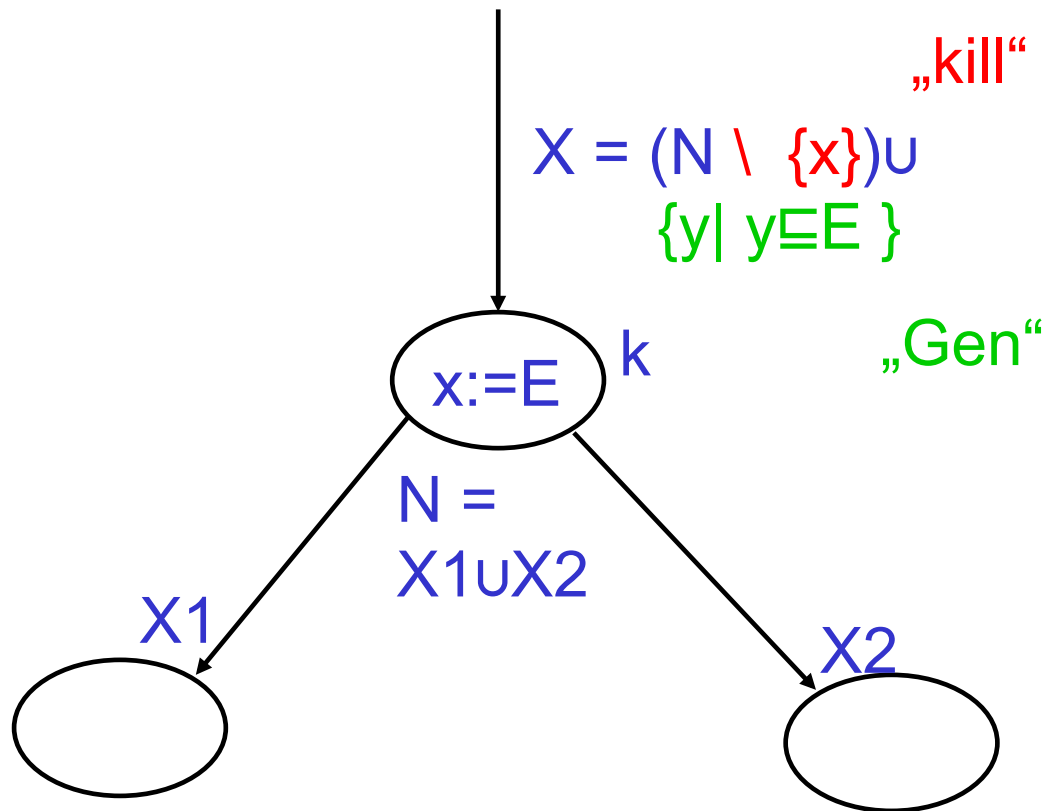


`x:=2;y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;`

erlaubt Transformation zu:

`y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;`

## Live variables: Idee



# Live variables: Ausführung

$$\text{kill}(x := E) = \{x\}$$

$$\text{kill}(\text{skip}) = \emptyset$$

$$\text{kill}(b) = \emptyset$$

$$\text{gen}(x := E) = \{ y \mid y \sqsubseteq E \}$$

$$\text{gen}(\text{skip}) = \emptyset$$

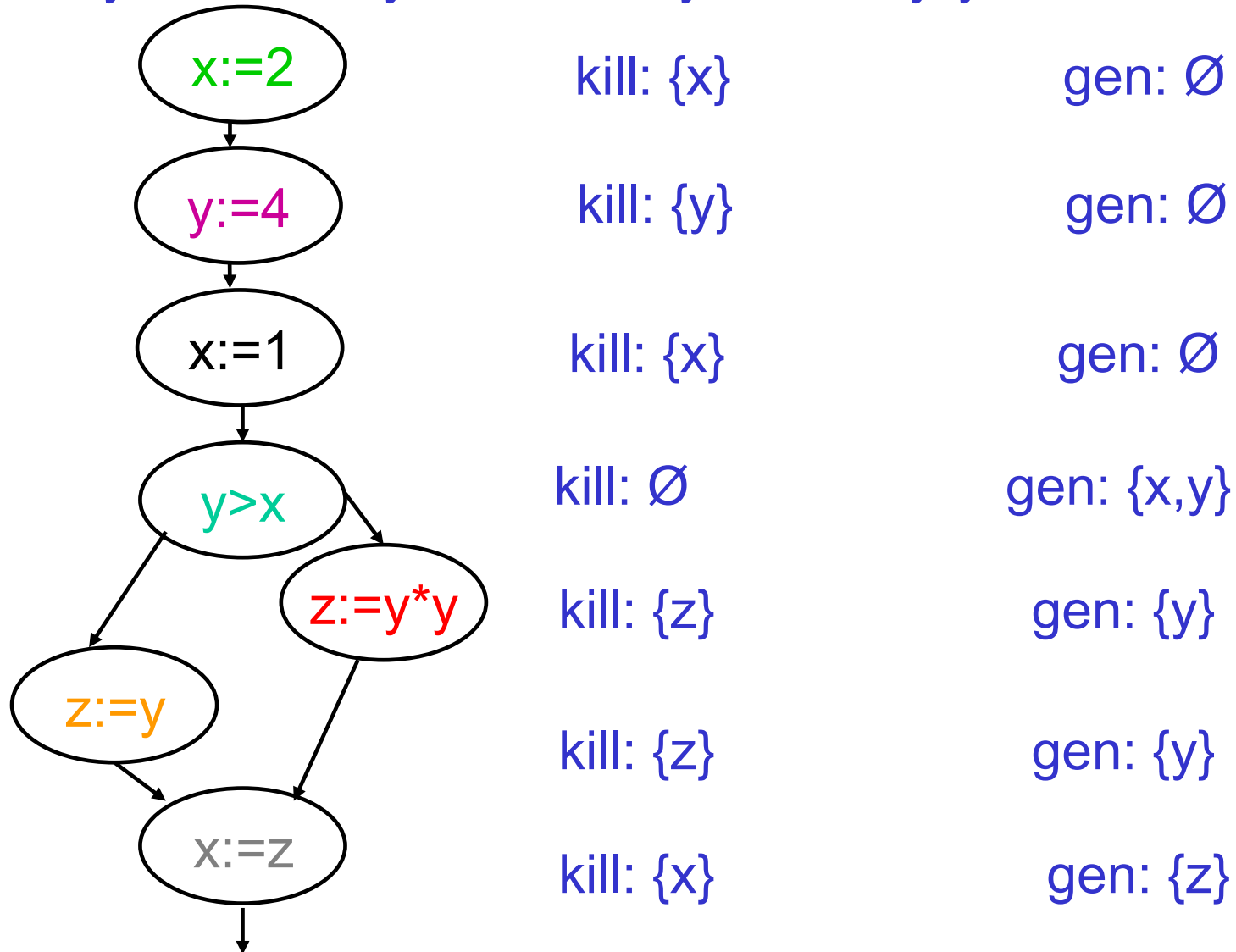
$$\text{gen}(b) = \{ y \mid y \sqsubseteq b \}$$

Bestimmung der Live variables mittels Lösung von Datenflussgleichungen

- $k_{\text{exit}} = \emptyset$  für  $k \in \text{final}(S)$
- $k_{\text{exit}} = \bigcup_{(k, k') \in \text{flow}(S)} k'_{\text{entry}}$  für  $k \notin \text{final}(S)$
- $k_{\text{entry}} = (k_{\text{exit}} \setminus \text{kill}(k)) \cup \text{gen}(k)$

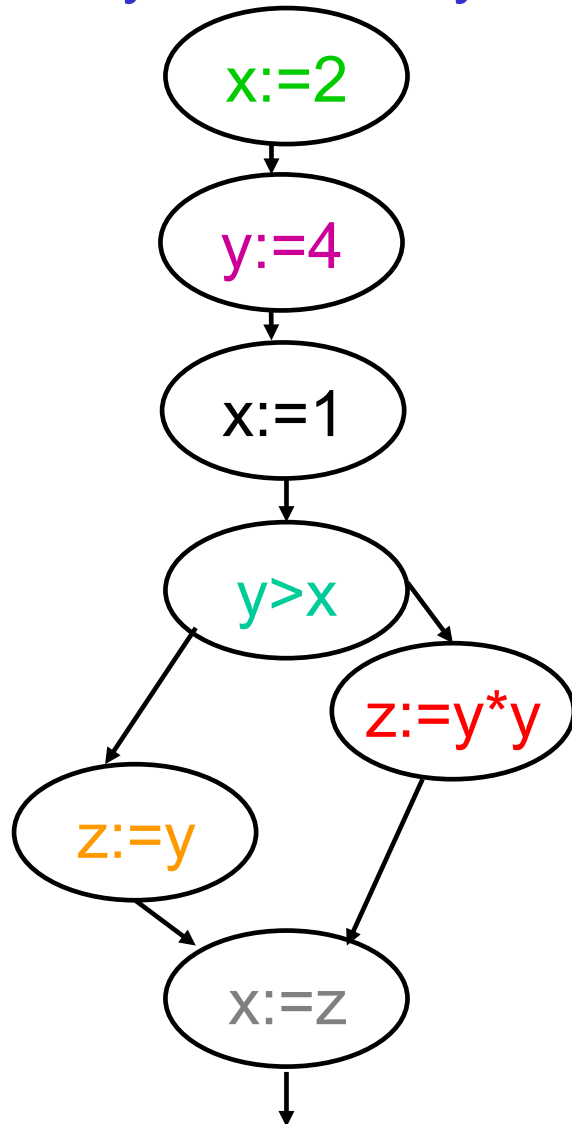
## Beispiel: kill, gen

$x:=2; y:=4; x:=1; \text{if } y > x \text{ then } z:=y \text{ else } z:=y*y \text{ end}; x := z;$



# Beispiel: Gleichungen

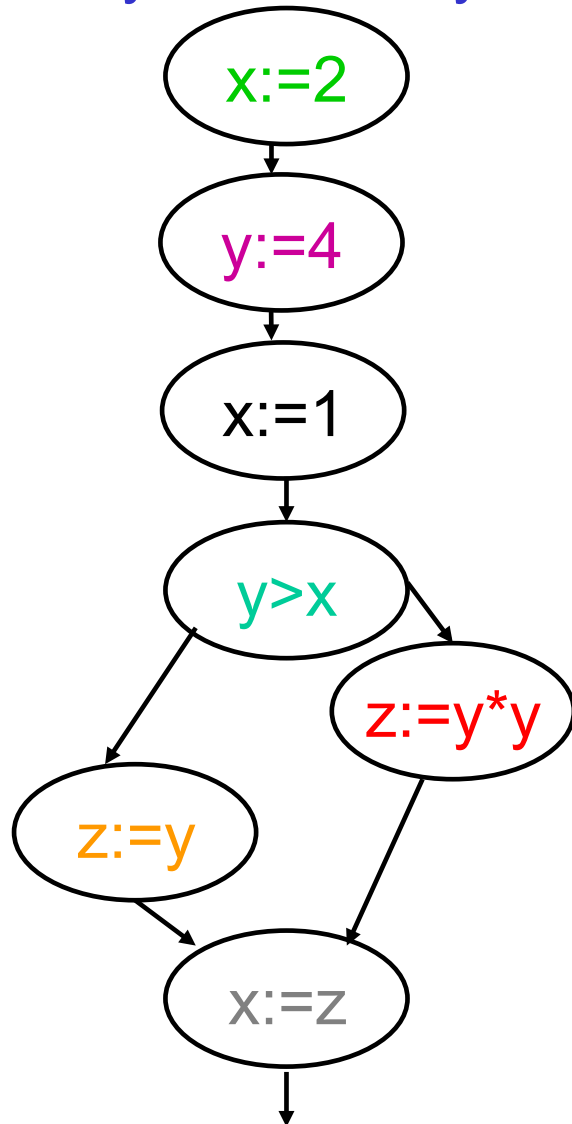
$x:=2; y:=4; x:=1; \text{if } y > x \text{ then } z:=y \text{ else } z:=y*y \text{ end}; x := z;$



$\text{entry} = \text{exit} \setminus \{x\}$   
 $\text{exit} = \text{entry}$   
 $\text{entry} = \text{exit} \setminus \{y\}$   
 $\text{exit} = \text{entry}$   
 $\text{entry} = \text{exit} \setminus \{x\}$   
 $\text{exit} = \text{entry}$   
 $\text{entry} = \text{exit} \cup \{x, y\}$   
 $\text{exit} = \text{entry} \cup \text{entry}$   
 $\text{entry} = \text{exit} \setminus \{z\} \cup \{y\}$   
 $\text{exit} = \text{entry}$   
 $\text{entry} = \text{exit} \setminus \{z\} \cup \{y\}$   
 $\text{exit} = \text{entry}$   
 $\text{entry} = \text{exit} \setminus \{x\} \cup \{z\}$   
 $\text{exit} = \emptyset$

## Beispiel: Lösung (kleinste)

$x:=2; y:=4; \textcolor{red}{x}:=1; \text{if } y > \textcolor{red}{x} \text{ then } z:=y \text{ else } z:=y*y \text{ end}; x := z;$



$\text{entry} = \emptyset$

$\text{exit} = \emptyset$

$\text{entry} = \emptyset$

$\text{exit} = \{y\}$

$\text{entry} = \{y\}$

$\text{exit} = \{x, y\}$

$\text{entry} = \{x, y\}$

$\text{exit} = \{y\}$

$\text{entry} = \{y\}$

$\text{exit} = \{z\}$

$\text{entry} = \{y\}$

$\text{exit} = \{z\}$

$\text{entry} = \{z\}$

$\text{exit} = \emptyset$

# Fazit bis hier

May-Analyse ( $\cup$ )  
(auf mind. einem Pfad...)

z.B. reaching definitions

Must-Analyse ( $\cap$ )  
(auf allen Pfaden ...)

z.B. available expressions

Vorwärtsanalyse

Rückwärtsanalyse

z.B. Live variables

z.B. very busy expressions

kleinste Fixpunkte

größte Fixpunkte

# Konstantenpropagation

- Für einen Knoten  $k$  und eine Variable  $x$  soll festgestellt werden,  
ob  $x$  in  $k$  stets den gleichen Wert liefert (wenn ja welchen)

Ziel: Ersetzung der Variable durch Konstante

Beispiel:

```
x := 6; y := 3; while x > y do x := x-1; z := y*y end
```

erlaubt Transformation

```
x := 6; y := 3; while x > 3 do x := x-1; z := 9 end
```



# Konstantenpropagation

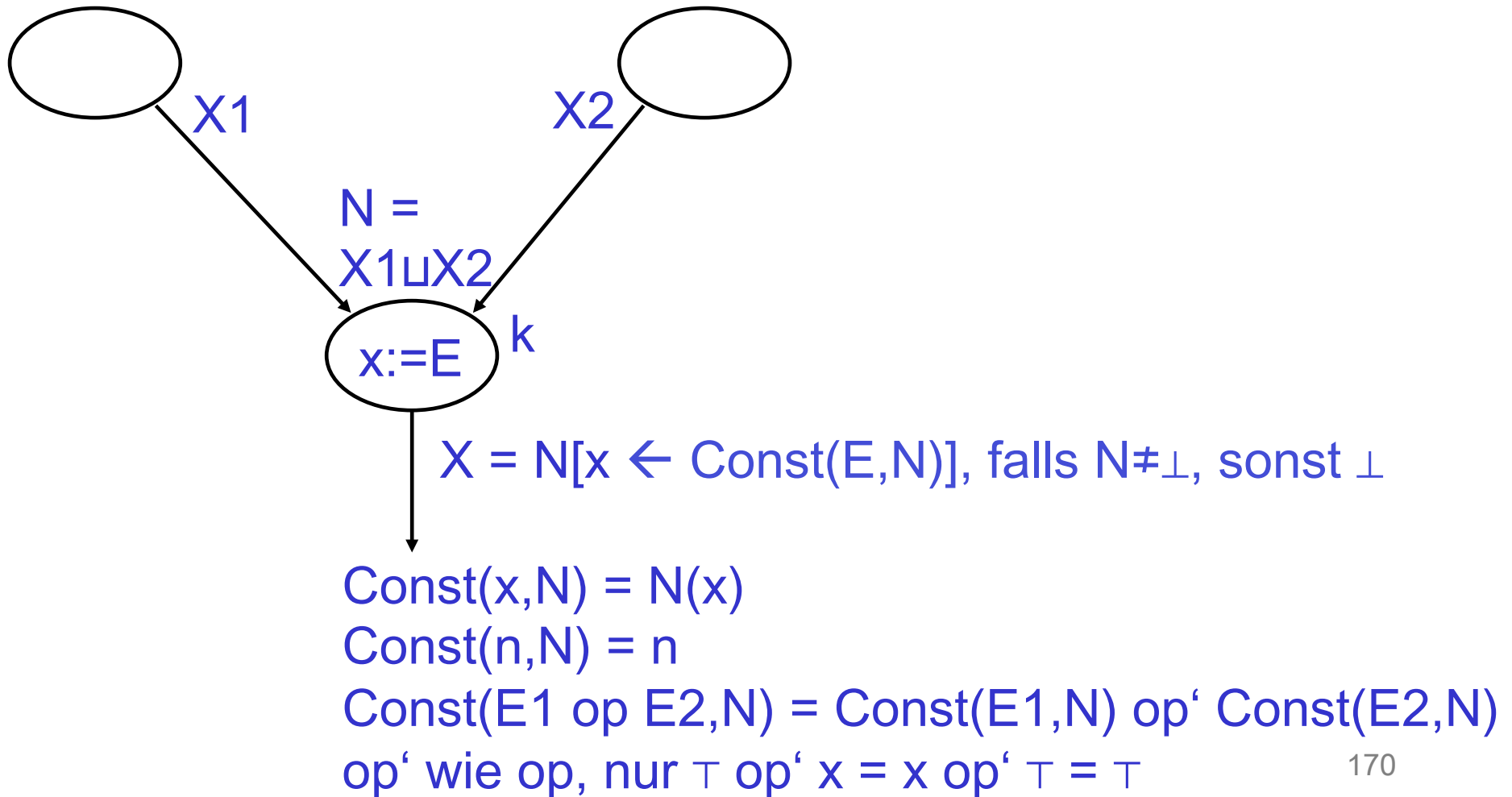
- Idee: Für jede Variable  $x$ , für jeden Punkt im Kontrollfluss
  - $\perp$  ... keine Information über  $x$  verfügbar
  - $n \in \mathbb{Z}$   $x$  hat hier immer Wert  $n$
  - $\top$  ...  $x$  könnte hier verschiedene Werte haben

$\perp \sqsubseteq n, n \sqsubseteq \top \quad n \neq n' \Rightarrow n \not\sqsubseteq n' \quad x \sqcup y = \text{kleinste obere Schranke von } x, y$

- Zustand:  $s: \text{VAR} \rightarrow \mathbb{Z} \cup \{\top, \perp\}$
- $s \sqcup s'$  definiert durch  $s \sqcup s' (x) = s(x) \sqcup s'(x)$  für alle  $x \in \text{VAR}$

# Konstantenpropagation

... ist Vorwärtsanalyse



# Kapitel 4

## Axiomatische Semantik

# Idee

Statt:

- Wie kommt der Effekt eines Programms zustande (operationell)
- Was ist der Effekt eines Programms? (denotationell)

Nun:

- Was kann ich über den Effekt eines Programms aussagen?

# Mittel: Hoare-Tripel

$$\{P\} S \{Q\}$$

Precondition

Postcondition

bedeutet:

Wenn S terminiert und vor Abarbeitung von S die Aussage P gilt,  
so gilt nach Abarbeitung von S die Aussage Q

*partielle Korrektheit*

$$\{P\} S \{\Downarrow Q\}$$

bedeutet:

Wenn vor Abarbeitung von S die Aussage P gilt, dann  
terminiert S und nach seiner Abarbeitung gilt die Aussage Q

*totale Korrektheit*

# Pre- und Postconditions

- P,Q sind Prädikate: Ordnen Variablen(belegung) einen Wahrheitswert zu
- Beispiele:  $x < y$ ,  $x > 0$ , usw.
- Variable: zwei Sorten
  - Programmvariablen: Wert ändert sich durch die Programmausführung
  - logische Variablen: Wert ändert sich nicht durch die Programmausführung („symbolische Konstanten“)
- Bsp:  $\{x = n\} \text{ } y:=1;\text{while } x \neq 1 \text{ do } y:=x*y;x:=x-1 \text{ end } \{y = n!\}$

## 4.1 Die axiomatische Semantik von W (partielle Korrektheit)

- $[\text{ass}]_{\text{par}}: \frac{\{P[x \rightarrow A \llbracket E \rrbracket]\} x := E \{P\}}{\{P\} x := E \{P\}}$
- $[\text{skip}]_{\text{par}}: \{P\} \text{ skip } \{P\}$
- $[\text{comp}]_{\text{par}}: \frac{\{P\} S1 \{Q\}, \{Q\} S2 \{R\}}{\{P\} S1 ; S2 \{R\}}$
- $[\text{if}]_{\text{par}}: \frac{\{P \wedge B \llbracket b \rrbracket\} S1 \{Q\}, \{P \wedge \neg B \llbracket b \rrbracket\} S2 \{Q\}}{\{P\} \text{ if } b \text{ then } S1 \text{ else } S2 \text{ end } \{Q\}}$
- $[\text{while}]_{\text{par}}: \frac{\{P \wedge B \llbracket b \rrbracket\} S \{P\}}{\{P\} \text{ while } b \text{ do } S \text{ end } \{P \wedge \neg B \llbracket b \rrbracket\}}$
- $[\text{cons}]_{\text{par}}: \frac{\{P\} S \{Q\} \quad \text{falls } P' \Rightarrow P \wedge Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$

# Weiteres Vorgehen

- Anwendung in der Programmverifikation, Erweiterungen
- Theorie: Widerspruchsfreiheit und Vollständigkeit



# Beispiel: Fakultätsberechnung

Def Fakultät (induktiv):  $0! = 1$ ;  $(n+1)! = (n+1) n!$

Ziel:  $\{x=n\} y:=1; \text{ while } x \neq 1 \text{ do } y:=y*x; x:=x-1 \text{ end } \{n>0 \wedge y=n!\}$

Beweis:

$I := \text{„ } x>0 \Rightarrow (y*x! = n! \wedge n \geq x) \text{“}$  Es folgt:

wegen  $[\text{ass}]_{\text{par}}: \{I[x \rightarrow x-1]\} x:=x-1 \{I\}$  und

$\{I[y \rightarrow y*x, x \rightarrow x-1]\} y:=y*x \{I[x \rightarrow x-1]\};$

wegen  $[\text{comp}]_{\text{par}}: \{I[y \rightarrow y*x, x \rightarrow x-1]\} y:=y*x; x:=x-1 \{I\};$

Es gilt:  $I \wedge x \neq 1 \Rightarrow I[y \rightarrow y*x, x \rightarrow x-1]$ , also

wegen  $[\text{cons}]_{\text{par}}: \{I \wedge x \neq 1\} y:=y*x; x:=x-1 \{I\};$

wegen  $[\text{while}]_{\text{par}}: \{I\} \text{ while } x \neq 1 \text{ do } y:=y*x; x:=x-1 \text{ end } \{I \wedge x=1\};$

Es gilt:  $I \wedge x=1 \Rightarrow (y = n! \wedge n > 0)$ , also

wegen  $[\text{cons}]_{\text{par}}: \{I\} \text{ while } x \neq 1 \text{ do } y:=y*x; x:=x-1 \text{ end } \{y = n! \wedge n > 0\};$

wegen  $[\text{ass}]_{\text{par}}: \{I[y \rightarrow 1]\} y:=1 \{I\};$

Es gilt:  $x=n \Rightarrow I[y \rightarrow 1]$ ; also

wegen  $[\text{cons}]_{\text{par}}: \{x=n\} y:= 1 \{I\}$  und

wegen  $[\text{comp}]_{\text{par}}: \{x=n\} y:=1; \text{ while } x \neq 1 \text{ do } y:=y*x; x:=x-1 \text{ end } \{n>0 \wedge y=n!\}$

## Beispiel: ggt-Berechnung

Zeigen:  $\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$

while  $x \neq y$  do

if  $x > y$  then

$x := x - y$

else

$y := y - x$

end

end

$\{x = \text{ggt}(a,b)\}$

$I = „x > 0 \wedge y > 0 \wedge \text{ggt}(x,y) = \text{ggt}(a,b)“$

# Beispiel: ggt-Berechnung

$\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$

$\{I\}$

while  $x \neq y$  do

  if  $x > y$  then

$x := x - y$

  else

$y := y - x$

  end

end

$I = „x > 0 \wedge y > 0 \wedge \text{ggt}(x,y) = \text{ggt}(a,b)“$

## Beispiel: ggt-Berechnung

$\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$

$\{I\}$

while  $x \neq y$  do

$\{I \wedge x \neq y\}$

if  $x > y$  then

$x := x - y$

else

$y := y - x$

end

end

$I = „x > 0 \wedge y > 0 \wedge \text{ggt}(x,y) = \text{ggt}(a,b)“$

# Beispiel: ggt-Berechnung

$\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$

$\{I\}$

while  $x \neq y$  do

$\{I \wedge x \neq y\}$

  if  $x > y$  then

$\{I \wedge x \neq y \wedge x > y\} \Rightarrow \{x-y > 0 \wedge y > 0 \wedge \text{ggt}(x-y, y) = \text{ggt}(a, b)\}$

$x := x - y$

  else

$\{I \wedge x \neq y \wedge x \leq y\} \Rightarrow \{I \wedge x < y\} \Rightarrow \{y-x > 0 \wedge x > 0 \wedge \text{ggt}(x, y-x) = \text{ggt}(a, b)\}$

$y := y - x$

  end

end

$I = „x > 0 \wedge y > 0 \wedge \text{ggt}(x, y) = \text{ggt}(a, b)“$

# Beispiel: ggt-Berechnung

```
{x=a ∧ y=b ∧ a>0 ∧ b>0}
{I}
while x<>y do
  {I ∧ x <>y}
  if x>y then
    {I ∧ x<>y ∧ x>y} ⇒ {x-y>0 ∧ y>0 ∧ ggt(x-y,y)=ggt(a,b)}
    x := x - y
    {I}
  else
    {I ∧ x<>y ∧ x≤y} ⇒ {I ∧ x<y} ⇒ {y-x>0 ∧ x>0 ∧ ggt(x,y-x)=ggt(a,b)}
    y := y - x
    {I}
  end
end
{I}
end
{I ∧ x=y} ⇒ {x = ggt(a,b)}
```

$I = „x > 0 \wedge y > 0 \wedge \text{ggt}(x,y) = \text{ggt}(a,b)“$

## Hilfssatz: $\text{ggt}(x,y) = \text{ggt}(x-y,y)$

- Vor:  $x > 0, y > 0, x - y > 0$
- $\text{ggt}(x,y)$  teilt  $x$ ,  $\text{ggt}(x,y)$  teilt  $y$ , also  $\text{ggt}(x,y)$  teilt  $x-y$ .
- Also  $\text{ggt}(x,y)$  ist gemeinsamer Teiler von  $x-y$  und  $y$
- Also  $\text{ggt}(x,y) \leq \text{ggt}(x-y,y)$
- $\text{ggt}(x-y,y)$  teilt  $x-y$ ,  $\text{ggt}(x-y,y)$  teilt  $y$ , also  $\text{ggt}(x-y,y)$  teilt  $y+(x-y)=x$
- Also  $\text{ggt}(x-y,y)$  ist gemeinsamer Teiler von  $x$  und  $y$
- Also  $\text{ggt}(x-y,y) \leq \text{ggt}(x,y)$
- Also:  $\text{ggt}(x,y) = \text{ggt}(x-y,y)$

# Arrays

- Hatten:  $[\text{ass1}]_{\text{par}}: \{P[x \rightarrow A \llbracket E \rrbracket ]\} x := E \{P\}$
- Naiv:  $\{a[3] = 1\} a[i] := 4 \{a[3] = 1\}$ 
  - nur richtig für  $i \neq 3$
- Problemanalyse: Verschiedene Zugriffe auf ein und dasselbe Element:  $a[3] \ a[i] \ (i=3) \dots$
- Lösung: Array als ganzheitliche Variable auffassen
- $a' = \text{write}(a, i, n)$  ist ein Array mit  $a'[j] = n$ , falls  $j=i$ ,  $a'[j]=a[j]$ , sonst.
- Also:
- $[\text{arr1}]_{\text{par}}: \{P[a \rightarrow \text{write}(a, A \llbracket i \rrbracket , A \llbracket E \rrbracket )]\} a[i] := E \{P\}$



# Beispiel: Maximum-Berechnung

```
Zeigen: {n>0}  
i := 0;  
m := a[0];  
while i<n do  
    i := i+1;  
    if a[i] > m then  
        m := a[i]  
    else  
        skip  
    end  
end  
{m = max {a[0],...a[n]}}
```

# Beispiel: Maximum-Berechnung

Zeigen:  $\{n \geq 0\}$

$i := 0;$

$\{n \geq i \wedge i = 0\}$

$m := a[0];$

$\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

while  $i < n$  do

$i := i + 1;$

if  $a[i] > m$  then

$m := a[i]$

else

skip

end

end

$\{m = \max \{a[0], \dots, a[n]\}$

# Beispiel: Maximum-Berechnung

```
Zeigen:  $\{n \geq 0\}$   
   $i := 0;$   
   $\{n \geq i \wedge i = 0\}$   
   $m := a[0];$   
   $\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
  while  $i < n$  do  
     $\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
     $i := i + 1;$   
    if  $a[i] > m$  then  
       $m := a[i]$   
    else  
      skip  
    end  
  end  
end  
 $\{m = \max \{a[0], \dots, a[n]\}$ 
```

# Beispiel: Maximum-Berechnung

Zeigen:  $\{n \geq 0\}$

$i := 0;$

$\{n \geq i \wedge i = 0\}$

$m := a[0];$

$\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

while  $i < n$  do

$\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

$i := i + 1;$

$\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

if  $a[i] > m$  then

$m := a[i]$

else

skip

end

end

$\{m = \max \{a[0], \dots, a[n]\}$

# Beispiel: Maximum-Berechnung

Zeigen:  $\{n \geq 0\}$   
   $i := 0;$   
   $\{n \geq i \wedge i = 0\}$   
   $m := a[0];$   
   $\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
  while  $i < n$  do  
     $\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
     $i := i + 1;$   
     $\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
    if  $a[i] > m$  then  
       $\{a[i] > m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
       $m := a[i]$   
    else  
       $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
      skip  
    end  
  end  
end  
 $\{m = \max \{a[0], \dots, a[n]\}$

# Beispiel: Maximum-Berechnung

Zeigen:  $\{n \geq 0\}$   
   $i := 0;$   
   $\{n \geq i \wedge i = 0\}$   
   $m := a[0];$   
 $\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
  while  $i < n$  do  
     $\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
       $i := i + 1;$   
       $\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
      if  $a[i] > m$  then  
         $\{a[i] > m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
       $\Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq a[i] \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = a[i]\}$   
         $m := a[i]$   
         $\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
      else  
         $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
      skip  
       $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
       $\Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
    end  
  end  
end  
 $\{m = \max \{a[0], \dots, a[n]\}$

# Beispiel: Maximum-Berechnung

Zeigen:  $\{n \geq 0\}$   
   $i := 0;$   
   $\{n \geq i \wedge i = 0\}$   
   $m := a[0];$   
 $\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \wedge a[y] = m\}$   
  while  $i < n$  do  
     $\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
       $i := i + 1;$   
       $\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
      if  $a[i] > m$  then  
         $\{a[i] > m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
       $\Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq a[i] \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = a[i]\}$   
         $m := a[i]$   
       $\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
      else  
         $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
      skip  
       $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
       $\Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
    end  
     $\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
  end  
   $\{m = \max \{a[0], \dots, a[n]\}$

# Beispiel: Maximum-Berechnung

Zeigen:  $\{n \geq 0\}$   
   $i := 0;$   
   $\{n \geq i \wedge i = 0\}$   
   $m := a[0];$   
 $\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
  while  $i < n$  do  
     $\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
       $i := i + 1;$   
       $\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
      if  $a[i] > m$  then  
         $\{a[i] > m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
         $m := a[i]$   
         $\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
      else  
         $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
        skip  
         $\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$   
       $\Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
    end  
     $\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
  end  
   $\{i \geq n \wedge i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$   
   $\Rightarrow \{m = \max \{a[0], \dots, a[n]\}$



## Beispiel: Array-Elemente tauschen

$\{a[i] = a \wedge a[j] = b\}$

$h := a[i];$

$\{h = a \wedge a[j] = b\}$

$a[i] := a[j];$

$\{a[i] = b \wedge h = a\}$

$a[j] := h;$

$\{a[i] = b \wedge a[j] = a\}$

funktioniert auch, wenn  $i = j$ !

# Beispiel: Dutch National Flag

- Problem:
- geg: Array  $a[0 \dots n]$  mit  $a[i] \in \{\text{blau}, \text{weiss}, \text{rot}\}$

Ziel: Array umsortieren (allein durch Swap-Operationen)  
derart, dass alle blau-Einträge vor allen weiss-  
Einträgen, und diese vor allen rot-Einträgen liegen.

# Algorithmus

```
b := 0; w:=0; r := n;  
while w ≤ r do  
    case a[w]  
        blau:  swap(a[b],a[w]);  
              w := w+1; b:= b+1;  
        weiss: w:=w+1;  
        rot:   swap(a[w],a[r]);  
              r := r-1;  
    end  
end
```

# Abschweifung: Hoare-Logik im Programmieralltag

- C, JAVA, EIFFEL ... unterstützen Assertions
- Schleifeninvarianten nach eigener Erfahrung extrem hilfreich
- Datenstruktur-Invarianten essentiell, um komplexe Strukturen zu beherrschen
- „Design by contract“    „Rely-Guarantee-Paradigm“:  
 $\{P\} S \{Q\}$  = „sicherst Du mir P zu, garantiere ich Dir Q“

# Weitere Beispiele

- Zeigen: Frühes Überlegen von Schleifeninvarianten hilft bei der Problemlösung
- Schleifeninvarianten kondensieren algorithmische Ideen

# Sattelsuche

- Gegeben Matrix von Zahlen
- Zeilen enthalten aufsteigende Werte
- Spalten enthalten aufsteigende Werte
- Wissen: k kommt irgendwo in der Matrix vor

- Frage: Wo?

- Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

# Sattelsuche

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

Zeilen: 0 .. m-1

Spalten: 0 .. n-1

Gesuchter Wert k bei [i,j]

Start:  $0 \leq i < m$  und  $0 \leq j < n$

Idee: Suchraum eingrenzen

Start mit  $i_u := 0$ ,  $i_o := m$ ,  $j_u := 0$ ,  $j_o := n$

Ziel:  $i_o = i_u + 1$ ,  $j_o = j_u + 1$

# Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?
- a) irgendwo in der Mitte  $a[i,j] > k$   $a[i,j] < k$



# Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?
- a) irgendwo in der Mitte  $a[i,j] > k$        $a[i,j] < k$
- b) irgendwo am Rand

# Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?
- a) irgendwo in der Mitte      $a[i,j] > k$       $a[i,j] < k$
- b) irgendwo am Rand
- c) Ecke links oben oder rechts unten

# Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?
- a) irgendwo in der Mitte      $a[i,j] > k$       $a[i,j] < k$
- b) irgendwo am Rand
- c) Ecke links oben oder rechts unten
- d) Ecke links unten oder rechts oben

# Algorithmus+Beweis

{k kommt in a vor}

iu := 0; io := m; ju := 0; jo := n;

{es gibt i,j mit  $iu \leq i < io$  und  $ju \leq j < jo$  und  $a[i,j] = k$ }

while io – iu > 1 OR jo – ju > 1 do

  case a[io-1,ju]

    = k: iu := io - 1; jo := ju+1;

    < k: ju := ju + 1;

    > k: io := io – 1;

  end

end

# Größtes True-Quadrat

Geg: Matrix aus Booleans

Ges: Größe einer größten quadratischen Teilmatrix, die nur TRUE-Einträge hat

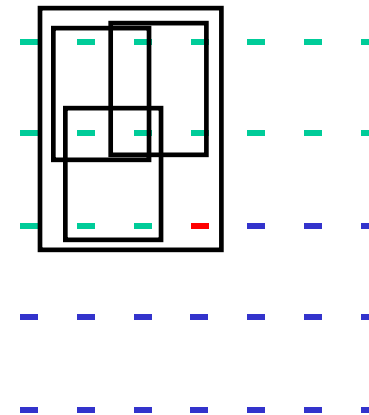
T	T	F	F	T	T	F
T	T	T	F	T	T	T
T	T	T	T	T	T	T
T	T	T	F	T	T	T
T	F	T	T	T	T	T

# Größtes True-Quadrat

Idee: Zahlenmatrix, die für jeden Matrixeintrag  $i,j$  die Größe des größten True-Quadrates mit  $i,j$  als rechter unterer Ecke angibt

Warum?

T	T	F	F	T	T	F
T	T	T	F	T	T	T
T	T	T	T	T	T	T
T	T	T	F	T	T	T
T	F	T	T	T	T	T



# Größtes True-Quadrat

Algorithmus und Beweis:

(Der Einfachheit halber Annahme:  $b[x,y] = 0$  für  $x < 0$  oder  $y < 0$ )

$z := 0; s := 0;$

{Für alle  $[i,j]$  mit  $i < z$  oder  $i = z$  und  $j < s$  ist  $b[i,j]$  die Größe des größten True-Quadrates mit  $[i,j]$  als rechter unterer Ecke }

while  $z < m$  do

    while  $s < n$  do

        case  $a[z,s]$

            F :  $b[z,s] := 0;$

            T :  $b[z,s] := \text{MIN}(b[z-1,s], b[z-1,s-1], b[z,s-1]) + 1$

        end;

$s := s + 1;$

    end

$z := z + 1; s := 0;$

end

# Erweiterung: Prozeduren

- nichtrekursiv:

- $$[\text{call1}]_{\text{par}}: \frac{\{P\} S \{Q\}}{\{P\} \text{ call } p \{Q\}} \quad (\text{proc } p \text{ is } S)$$

- rekursiv:

- $$[\text{call2}]_{\text{par}}: \frac{\{P\} \text{ call } p \{Q\} \vdash \{P\} S \{Q\}}{\{P\} \text{ call } p \{Q\}} \quad (\text{proc } p \text{ is } S)$$



# Beispiel: Quicksort

qs(0,n);

qs(von,bis) is

  b := von; w := von+1; r := bis;

  p := a[b];

  while w ≤ r do

    case

      a[w] < p: swap(a[b],a[w]);

              w := w+1; b:= b+1;

      a[w] = p: w:=w+1;

      a[w] > p: swap(a[w],a[r]);

              r := r-1;

    end

  end

  if von < b-1 then qs(von,b-1) end;

  if r+1 < bis then qs(r+1,bis) end;

end

Dutch National Flag

blau := <p

weiss := =p

rot := >p

# Beweis: Quicksort

$\{0 < n\}$   
qs(0,n);

$\{von < bis\}$   
qs(von,bis) is

```
b := von; w := von+1; r := bis;
  {b < w ≤ r+1}
  p := a[b];
  while w ≤ r do
    case
      a[w] < p:  swap(a[b],a[w]);
                w := w+1; b:= b+1;
      a[w] = p:  w:=w+1;
      a[w] > p:  swap(a[w],a[r]);
                r := r-1;
    end
  end
  {b < w ≤ r+1 ∧ für alle i : (i < b ⇒ a[i] < p) ∧ (i ≥ b ∧ i < w ⇒ a[i] = p) ∧ (i > r ⇒ a[i] > p)}
  if von < b-1 then qs(von,b-1) end;
  {b < w ≤ r+1 ∧ für alle i : (i < b ⇒ a[i] < p) ∧ (i ≥ b ∧ i < w ⇒ a[i] = p) ∧ (i > r ⇒ a[i] > p) ∧ sortiert(von,b-1) }
  if r+1 < bis then qs(r+1,bis) end;
  {b < w ≤ r+1 ∧ für alle i : (i < b ⇒ a[i] < p) ∧ (i ≥ b ∧ i < w ⇒ a[i] = p) ∧ (i > r ⇒ a[i] > p) ∧ sortiert(von,b-1) ∧ sortiert(r+1,bis) }
  ⇒ {sortiert(von,bis)}
end
{sortiert(von,bis)}
```

## 4.2 Axiomatische Semantik von W (totale Korrektheit)

- $[\text{ass1}]_{\text{tot}}: \frac{\{P[x \rightarrow A \llbracket E \rrbracket]\} x := E \quad \{\Downarrow P\}}{\text{falls } x \text{ Int-Variable}}$
- $[\text{ass2}]_{\text{tot}}: \frac{\{P[x \rightarrow B \llbracket E \rrbracket]\} x := E \quad \{\Downarrow P\}}{\text{falls } x \text{ Bool-Variable}}$
- $[\text{skip}]_{\text{tot}}: \frac{\{P\} \text{ skip} \quad \{\Downarrow P\}}{\text{ }}$
- $[\text{comp}]_{\text{tot}}: \frac{\{P\} S1 \quad \{\Downarrow Q\} \quad , \quad \{Q\} S2 \quad \{\Downarrow R\}}{\{P\} S1 ; S2 \quad \{\Downarrow R\}}$
- $[\text{if}]_{\text{tot}}: \frac{\{P \wedge B \llbracket b \rrbracket\} S1 \quad \{\Downarrow Q\} \quad , \quad \{P \wedge \neg B \llbracket b \rrbracket\} S2 \quad \{\Downarrow Q\}}{\{P\} \text{ if } b \text{ then } S1 \text{ else } S2 \text{ end} \quad \{\Downarrow Q\}}$
- $[\text{while}]_{\text{tot}}: \frac{\{P(z+1)\} S \quad \{\Downarrow P(z)\} \quad (z \in \mathbb{N})}{\{ \text{ex. } z: P(z) \} \text{ while } b \text{ do } S \text{ end} \quad \{\Downarrow P(0)\}}$   
wobei  $P(z+1) \Rightarrow B \llbracket b \rrbracket$  ,  $P(0) \Rightarrow \neg B \llbracket b \rrbracket$
- $[\text{cons}]_{\text{tot}}: \frac{\{P\} S \quad \{\Downarrow Q\}}{\{P'\} S \quad \{\Downarrow Q'\}} \quad \text{falls } P' \Rightarrow P \wedge Q \Rightarrow Q'$

## Liberaleres Konzept: Abstiegsfunktion

- Zu jeder While-Schleife Term  $t$  mit nat. Zahl als Wert und
  - $\{t = k\} S \{t < k\}$
  - $t = 0 \Rightarrow \neg B \llbracket b \rrbracket$
- Dann terminiert while  $b$  do  $S$  end

# Terminierung: Fakultätsberechnung

$y:=1; \text{ while } x \neq 1 \text{ do } y:=y*x; x:=x-1 \text{ end } \{n>0 \wedge y=n!\}$

Abstiegssfunktion:  $x - 1$  (Voraussetzung:  $x > 0$ )

# Terminierung: ggt-Berechnung

```
while  $x \neq y$  do  
  if  $x > y$  then  
     $x := x - y$   
  else  
     $y := y - x$   
  end  
end
```

Abstiegssfunktion:  $|x - y|$

# Terminierung: Maximum-Berechnung

```
i := 0;  
  m := a[0];  
  while i < n do  
    i := i + 1;  
    if a[i] > m then  
      m := a[i]  
    else  
      skip  
    end  
  end  
end
```

Abstiegssfunktion:  $n - i$

# Terminierung: Dutch National Flag

```
b := 0; w:=0; r := n;  
while w ≤ r do  
  case a[w]  
    blau:  swap(a[b],a[w]);  
           w := w+1; b:= b+1;  
    weiss: w:=w+1;  
    rot:   swap(a[w],a[r]);  
           r := r-1;  
  end  
end
```

Abstiegssfunktion:  $r + 1 - w$



# Terminierung: Sattelsuche

```
iu := 0; io := m; ju := 0; jo := n;  
while io - iu > 1 OR jo - ju > 1 do  
  case a[io-1,ju]  
    = k: iu := io - 1; jo := ju+1;  
    < k: ju := ju + 1;  
    > k: io := io - 1;  
  end  
end
```

Abstiegssfunktion:  $(io-iu) + (jo-ju)$

# Terminierung: Größtes True-Quadrat

```
z := 0; s := 0;
while z < m do
  while s < n do
    case a[i,j]
      F :   b[i,j] := 0;
      T :   b[i,j] := MIN(b[i-1,j],b[i-1,j-1],b[i,j-1])+1
    end;
    s := s + 1;
  end
  z := z + 1; s := 0;
end
```

Abstiegssfunktion:  $mn - nz - s$

# Terminierung: Prozeduren

- nichtrekursiv:
- $[\text{call1}]_{\text{par}}: \frac{\{P\} S \{\Downarrow Q\}}{\{P\} \text{ call } p \{\Downarrow Q\}} \quad (\text{proc } p \text{ is } S)$
- rekursiv:
- $[\text{call2}]_{\text{par}}: \frac{\{P(z)\} \text{ call } p \{\Downarrow Q\} \vdash \{P(z+1)\} S \{\Downarrow Q\}}{\{\text{ex. } z: P(z)\} \text{ call } p \{\Downarrow Q\}} \quad \text{wobei } \neg P(0)$   
 $(\text{proc } p \text{ is } S, \text{WB}(z) = N)$

# Beispiel: Quicksort

qs(0,n);

qs(von,bis) is

  b := von; w := von; r := bis;

  p := a[b];

  while w ≤ r do

    case

      a[w] < p: swap(a[b],a[w]);

              w := w+1; b:= b+1;

      a[w] = p: w:=w+1;

      a[w] > p: swap(a[w],a[r]);

              r := r-1;

    end

  end

  if von < b-1 then qs(von,b-1) end;

  if r+1 < bis then qs(r+1,bis) end;

end

Abstiegsfunktion: bis - von

## 4.3 Korrektheit und Vollständigkeit

- stellen Beziehung her zwischen partieller Korrektheit und Natural Semantics:
- $\models \{P\} S \{Q\}$  falls für alle Zustände  $s$  gilt:  
Wenn  $P(s)$  und  $\langle S, s \rangle \rightarrow s'$ , so  $Q(s')$

„ $\{P\} S \{Q\}$  ist richtig“

- Zum Vergleich:
- $\vdash \{P\} S \{Q\}$  falls sich diese Aussage mittels der Axiome und Regeln der partiellen Korrektheit herleiten lässt

„ $\{P\} S \{Q\}$  ist beweisbar“

# Korrektheit und Vollständigkeit

- Korrektheit: „Alles, was sich beweisen lässt, ist richtig“

Für alle  $P, Q, S$ : Wenn  $\vdash \{P\} S \{Q\}$ , so  $\models \{P\} S \{Q\}$

- Vollständigkeit: „Alles, was richtig ist, lässt sich beweisen“

Für alle  $P, Q, S$ : Wenn  $\models \{P\} S \{Q\}$ , so  $\vdash \{P\} S \{Q\}$

## Korrektheit: Wenn $\vdash \{P\} S \{Q\}$ , so $\models \{P\} S \{Q\}$

- Beweis durch Induktion über der Struktur einer Ableitung gemäß der Axiome und Regeln der partiellen Korrektheit:
- Anfang 1: Sei  $S = x := E$  und  $\langle S, s \rangle \rightarrow s'$ . Also  $s' = s[x \rightarrow A \llbracket E \rrbracket (s)]$ . Wenn nun  $P[x \rightarrow A \llbracket E \rrbracket ](s)$ , so gilt also auch  $P(s')$ .
- Anfang 2: Sei  $S = \text{skip}$  -- trivial
- Schritt 1: Sei  $S = S1;S2$  und  $\langle S1;S2, s \rangle \rightarrow s'$ . Also ex.  $s''$  mit  $\langle S1, s \rangle \rightarrow s''$  und  $\langle S2, s'' \rangle \rightarrow s'$ . Sei nun  $\vdash \{P\} S1 \{Q\}$  und  $\vdash \{Q\} S2 \{R\}$ . Nach IV:  $\models \{P\} S1 \{Q\}$  (\*) und  $\models \{Q\} S2 \{R\}$  (\*\*). Wenn nun  $P(s)$  gilt, so ist wegen (\*)  $Q(s'')$  und wegen (\*\*)  $R(s')$ .

## Forts. Korrektheit: Wenn $\vdash \{P\} S \{Q\}$ , so $\models \{P\} S \{Q\}$

- Schritt 2: Sei  $S = \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}$  und  $\langle S, s \rangle \rightarrow s'$ .  
1. Fall:  $B \llbracket b \rrbracket (s) = \text{true}$ . Also gilt  $(P \wedge B \llbracket b \rrbracket)(s)$ .  
Nach IV:  $\models \{P \wedge B \llbracket b \rrbracket\} S1 \{Q\}$  Also  $Q(s')$ .  
2. Fall analog
- Schritt 3: Sei  $S = \text{while } b \text{ do } S' \text{ end}$  und  $\langle S, s \rangle \rightarrow s'$   
Zeigen per Induktion über Tiefe der Ableitung für  $\langle S, s \rangle \rightarrow s'$ :  
 $(P \wedge \neg B \llbracket b \rrbracket)(s')$ .  
Anfang (Tiefe 0): Dann:  $B \llbracket b \rrbracket (s) = \text{false}$ . Also ist  $s = s'$ , also  
 $(P \wedge \neg B \llbracket b \rrbracket)(s')$ .  
Schritt: (Tiefe  $> 0$ ) Dann:  $B \llbracket b \rrbracket (s) = \text{true}$ . Also  $(P \wedge B \llbracket b \rrbracket)(s)$ .  
Dann gibt es  
 $s''$  mit  $\langle S', s \rangle \rightarrow s''$  und  $\langle S, s'' \rangle \rightarrow s'$ . Es gilt  $P(s'')$ . Nach IV  
folgt  $(P \wedge \neg B \llbracket b \rrbracket)(s')$ .
- Schritt 4 (Konsequenzregel) kein Problem.



# Vollständigkeit: Wenn $\models \{P\} S \{Q\}$ , so $\vdash \{P\} S \{Q\}$

- Betrachten folgende Assertions:
  - $\{x = 2\} \ x := x + 1 \ \{x \text{ ungerade}\}$
  - $\{x = 28\} \ x := x + 1 \ \{x \text{ ungerade}\}$
  - $\{x \text{ durch } 4 \text{ teilbar}\} \ x := x + 1 \ \{x \text{ ungerade}\}$
  - $\{x \text{ gerade}\} \ x := x + 1 \ \{x \text{ ungerade}\}$
- Alle richtig. Die letzte wohl am wertvollsten.

$wpr(S, Q)$ , die *schwächste Vorbedingung* für  $S$  und  $Q$ , ist dasjenige Prädikat  $P$ , wo  $P(s)$  gdw. für alle  $s'$  mit  $\langle S, s \rangle \rightarrow s'$ :  $Q(s')$ .

## Aussagen über wpr

- Für alle  $S, Q$ :  $\models \{wpr(S, Q)\} \ S \ \{Q\}$

Beweis: Wenn  $wpr(S, Q)(s)$  und  $\langle S, s \rangle \rightarrow s'$ , so per Def. von wpr:  $Q(s')$ .

- Für alle  $P, S, Q$ : Wenn  $\models \{P\} \ S \ \{Q\}$ , so  $P \Rightarrow wpr\{S, Q\}$

Beweis: Sei  $s$  Zst. mit  $P(s)$  und  $\langle S, s \rangle \rightarrow s'$ . Wegen  $\models \{P\} \ S \ \{Q\}$  ist  $Q(s')$ . Also  $wpr(S, Q)(s)$ . Also  $P \Rightarrow wpr\{S, Q\}$

# Vollständigkeit: Wenn $\models \{P\} S \{Q\}$ , so $\vdash \{P\} S \{Q\}$

- Wir zeigen für alle  $S, Q$ :  $\vdash \{wpr(S, Q)\} S \{Q\}$ . Originalaussage folgt dann mit den Aussagen der vorigen Folie
- Fall  $x := E$ . Offenbar  $wpr(x := E, Q) = Q[x \rightarrow A \llbracket E \rrbracket]$
- Fall skip: Offenbar  $wpr(\text{skip}, Q) = Q$ .
- Fall  $S = S1; S2$ . Nach IV:  $\vdash \{wpr(S2, Q)\} S2 \{Q\}$  und  $\vdash \{wpr(S1, wpr(S2, Q))\} S1 \{wpr(S2, Q)\}$ . Damit ist  $\vdash \{wpr(S1, wpr(S2, Q))\} S1; S2 \{Q\}$  Bleibt z.Z:  
 $wpr(S1; S2, Q) \Rightarrow wpr(S1, wpr(S2, Q))$   
Betrachten  $s$  mit  $wpr(S1; S2, Q)(s)$   
Sei  $\langle S1, s \rangle \rightarrow s''$  und  $\langle S2, s'' \rangle \rightarrow s'$  (wenn es  $s'$  bzw.  $s''$  nicht gibt, wird die Aussage trivial). Also  $Q(s')$ . Also  $wpr(S2, Q)(s'')$ . Also  
 $wpr(S1, wpr(S2, Q))(s)$ .

# Vollständigkeit: Wenn $\models \{P\} S \{Q\}$ , so $\vdash \{P\} S \{Q\}$

- Wir zeigen für alle  $S, Q$ :  $\vdash \{wpr(S, Q)\} S \{Q\}$ .

- Fall  $S = \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}$ .

Mit IV:  $\vdash \{wpr(S1, Q)\} S1 \{Q\}$  und  $\vdash \{wpr(S2, Q)\} S2 \{Q\}$ .

Sei  $P = (B \llbracket b \rrbracket \wedge wpr(S1, Q)) \vee (\neg B \llbracket b \rrbracket \wedge wpr(S2, Q))$

Mit [cons]<sub>P</sub>:  $\vdash \{B \llbracket b \rrbracket \wedge P\} S1 \{Q\}$  und  $\vdash \{\neg B \llbracket b \rrbracket \wedge P\} S2 \{Q\}$ .

Also mit [if]<sub>P</sub>:  $\{P\} \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \{Q\}$ . Bleibt z.Z.:

$wpr(\text{if } b \text{ then } S1 \text{ else } S2 \text{ end}, Q) \Rightarrow P$ .

Sei  $\langle S, s \rangle \rightarrow s'$  mit  $Q(s')$ . 1. Fall:  $B \llbracket b \rrbracket (s)$ . Also gilt linke Alternative in  $P$ . 2. Fall analog.

# Vollständigkeit: Wenn $\models \{P\} S \{Q\}$ , so $\vdash \{P\} S \{Q\}$

- Wir zeigen für alle  $S, Q: \vdash \{wpr(S, Q)\} S \{Q\}$ .
  - Fall  $S = \text{while } b \text{ do } S' \text{ end}$ . Sei  $P = wpr(\text{while } b \text{ do } S' \text{ end}, Q)$ .
    1. Zeigen:  $\neg B \llbracket b \rrbracket \wedge P \Rightarrow Q$   
Sei  $s$  Zst. mit  $(\neg B \llbracket b \rrbracket \wedge P)(s)$ . Also  $\langle \text{while } b \text{ do } S' \text{ end}, s \rangle \rightarrow s$ , also  $Q(s)$ .
    2. Zeigen:  $B \llbracket b \rrbracket \wedge P \Rightarrow wpr(S', P)$   
Sei  $s$  Zst. mit  $(B \llbracket b \rrbracket \wedge P)(s)$ . Sei  $\langle S', s \rangle \rightarrow s'$ . (Wenn  $s'$  nicht ex., ist Aussage trivial). Zeigen:  $P(s')$ 
      1. Fall: Es gibt ein  $s''$  mit  $\langle \text{while } b \text{ do } S' \text{ end}, s' \rangle \rightarrow s''$ .  
Dann gilt auch  $\langle \text{while } b \text{ do } S' \text{ end}, s \rangle \rightarrow s''$ , und wegen Wahl von  $P$  auch  $Q(s'')$ . Damit muss aber auch  $P(s')$  gelten
      2. Fall: Es gibt kein  $s''$  mit  $\langle \text{while } b \text{ do } S' \text{ end}, s' \rangle \rightarrow s''$ .  
Dann gilt  $P(s')$  trivialerweise.
- Mit IV:  $\vdash \{wpr(S', P)\} S' \{P\}$ . Mit (2):  $\vdash \{P \wedge B \llbracket b \rrbracket\} S' \{P\}$ . Mit  $[while]_P$ :  
 $\vdash \{P\} \text{while } b \text{ do } S' \text{ end} \{\neg B \llbracket b \rrbracket \wedge P\}$ . Mit  $[cons]$  und (1):  
 $\vdash \{P\} \text{while } b \text{ do } S' \text{ end} \{Q\}$ .

# Gödelscher Unvollständigkeitssatz

„In jedem hinreichend ausdrucksstarken formalen System gibt es Aussagen, die richtig, aber nicht beweisbar sind“

hinreichend ausdrucksstark: im wesentlichen Rechnen mit natürlichen Zahlen und etwas Logik

Formales System = Aussagen formulieren mittels Sprache;  
Beweisschritte mittels Ersetzungsregeln

# Gödels Beweisidee 1: Übersetzung in Zahlentheorie

- Aussagen der Sprache übersetzen in Zahlen:

Sprache hat Alphabet: Zeichen  $\rightarrow$  Zahl

Satz der Sprach ist Folge von Zeichen.

Kodierung: 2<sup>erster Bu.</sup> 3<sup>zweiter Bu</sup> 5<sup>dritter Bu</sup>

Anwendung einer Regel: Rechnen mit Zahlen

## Gödels Beweisidee 2: Selbstaussage

- Lügnerparadoxon: Ein Kreter sagt: „Alle Kreter lügen“
- Gödels Selbstaussage:
- $H = \text{„}H \text{ ist nicht beweisbar“}$ 
  - 1. Fall: beweisbar, dann falsch, also System nicht korrekt
  - 2. Fall: nicht beweisbar, dann aber richtig!



# Gödels Argument trägt nicht in unserem Setting

- Grund: Lassen beliebige Prädikate zu.
- Haben: Abzählbar viele Variablen
- → überabzählbar viele Prädikate
- → es können nicht alle Prädikate in einer Sprache formuliert werden
- → Voraussetzung für Gödels Satz treffen nicht zu

# Zusammenfassung Semantik

- Drei Konzepte, entsprechen drei grundlegenden Herangehensweisen an Modellbildung in der Informatik
  - Operationell: Nutzen abstrakte Maschinenmodelle; im Mittelpunkt steht der *Schritt*
  - Denotationell: Nutzen mathematische Formalismen wie Funktionen, Relationen, Gleichungen
  - Logisch: Nutzen logische Aussagen und Beweiskalküle

# Zusammenfassung Semantik

- All diese Konzepte finden sich auch in Ansätzen zur (semi-) formalen Spezifikation
  - Operationell: State Charts, Activity diagrams, Petrinetze, Message sequence diagrams, ...
  - Denotationell: Z,  $\mu$ -Kalkül
  - Logisch: Larch, temporale Logik

# Zusammenfassung Semantik

- Operationelle Semantik:
  - relativ leicht zu definieren
  - relativ schwer auszuwerten
  - momentan Hauptmethode für verteilte bzw. interaktive Systeme (Protokolle, verteilte Algorithmen, GUI,...)
  - Anwendung: Vergleich von Systemen, Model Checking

# Zusammenfassung Semantik

- Denotationelle Semantik:
  - relativ schwer zu definieren
  - reichhaltige Auswertungsmöglichkeiten
  - verlangt mathematische Fertigkeiten
  - Anwendung: Statische Analyse
    - kompositional

# Zusammenfassung Semantik

- Axiomatische Semantik:
  - mittelschwer zu definieren
  - leichte Auswertung
  - verlangt logische Fertigkeiten
  - Anwendung: Programmverifikation
    - kompositional

# Modellbildung in der Informatik

- Informatik = Übergang von informalen Ideen zu formalen Konstrukten
- vernünftiger Zwischenschritt: Modelle
- konsequente Umsetzung: Modellbasierte Softwareentwicklung
- State of the art: Informelle/semiformelle Modellierungskonzepte (vornehmlich UML)

# Vorteile formaler Semantik/formaler Modelle

- Missverständnisarm
- Möglichkeit bedeutungstreuer Übersetzung
- Verfügbarkeit z.T. automatisierter Methoden zur Auswertung, zur Fehlererkennung
- Konstruktionsmethodik („correct by construction“)
- Proof-carrying code
- tieferes Verständnis



# Nachteile formaler Modelle

- Erstellungsaufwand
- Schwer lesbar für Quereinsteiger
- Aber:
  - Aufwand zahlt sich oft aus
  - Tools verstecken Teil der Schwierigkeiten
  - sichere Jobs für Experten
  - aktuelle Forschung: Komplexitätsreduktion