Übersetzung und Analyse von Programmen

Vorlesung im Wintersemester 2025/26

Prof. Dr. Stephan Diehl
Informatik, Universität Trier

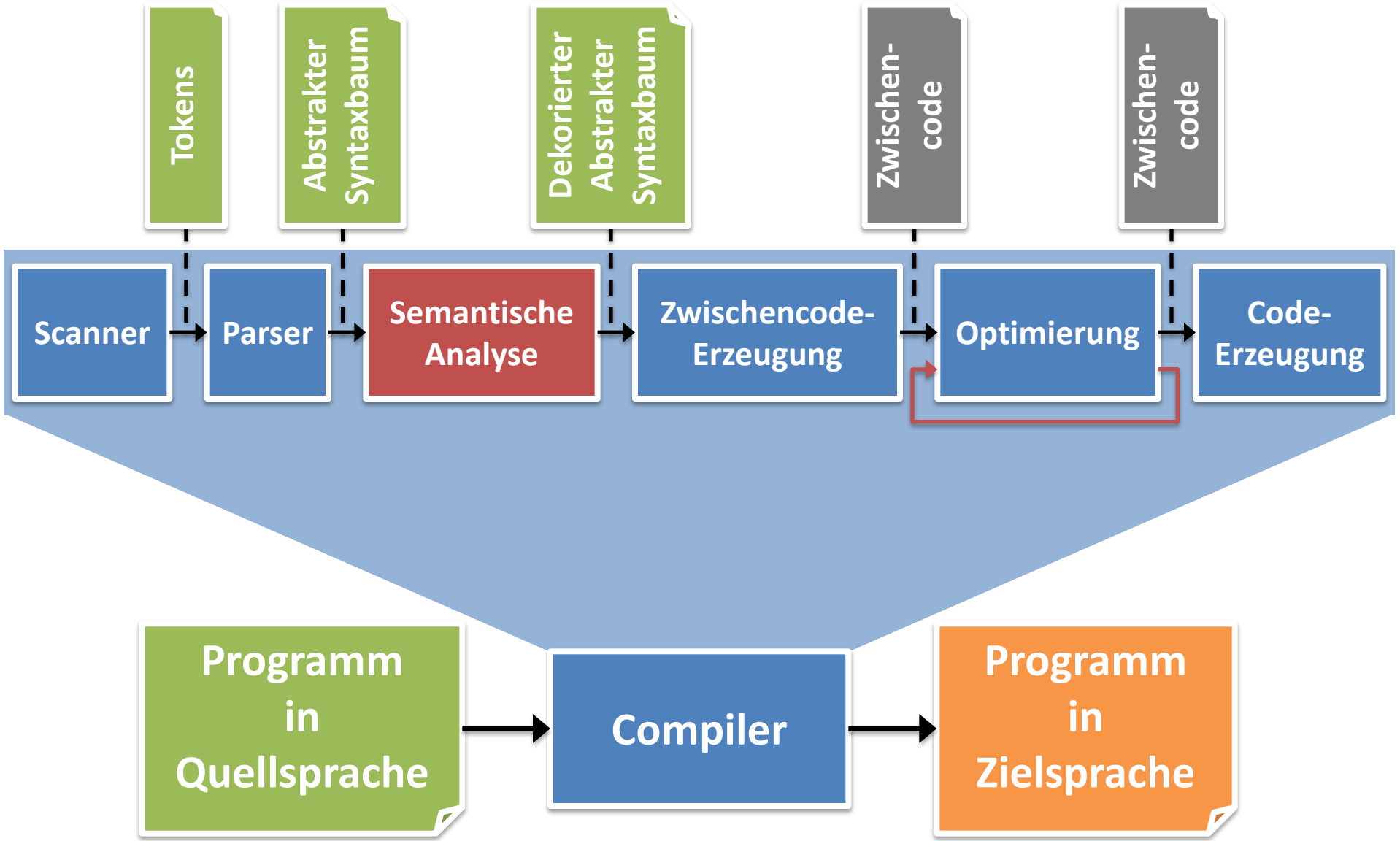
Zeitplan bis zum Jahresende

11.12.2025	Abstrakte Interpretation (von TRIPLA)	
16.12.2025	Weihnachtsdings + Vortrag Wölwer	Abgabe Übungsblatt „Übersetzung von Java“
18.12.2025		Abgabe Projekt III
08.01.2026	Datenflussanalyse	Übungsblatt „Datenflussanalyse“
13.01.2026	Datenflussanalyse von TRIPLA	
15.01.2026	Projekt IV (Kontrollflussanalyse)	Abgabe Übungsblatt „Datenflussanalyse“
20.01.2026	Code Clone Detection	
22.01.2026	Reverse Engineering, Erkennung von Entwurfsmustern	Abgabe Projekt IV
27.01.2026	Gastvortrag: „Programm Analysis and Transformation from a Practitioner‘s Point of View“ (Jonas Eckstein, itestra)	Projekt V (Datenflussanalyse)

Datenflussanalyse

Bildquelle: mit KI erzeugt (Bing, DALL-E)

Struktur eines Compilers



Statische Programmanalysen

- Kontrollflussanalyse
- Datenflussanalyse

Statische Programmanalyse

- Berechnet Eigenschaften eines Programms, die für alle Ausführungen des Programms gelten.
  - Nicht jede Eigenschaft kann berechnet werden wegen des Halteproblems.
  - Approximative Ansätze
  - Abstraktion, Filtern von Daten (je nach Ziel der Analyse)

- Dynamische vs. Statische Eigenschaften
  - Wie oft wurde ein Programmpunkt für eine gegebene Eingabe ausgeführt? (dynamisch)
  - Gibt es eine Eingabe, sodass ein bestimmter Programmpunkt jemals ausgeführt/besucht wird? (statisch)

Statische Programmanalyse

Einsatzgebiete

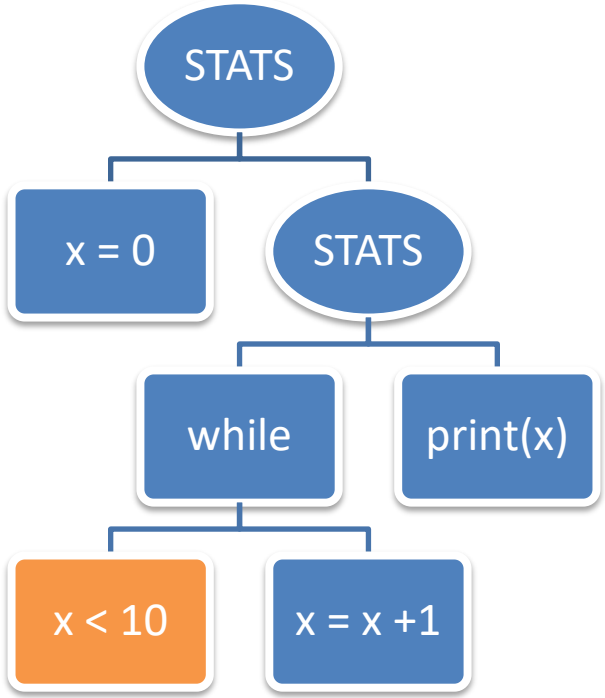
- Fehlererkennung, Debugging, Testen
  - Aufdecken von Bad Smells, Code Clones, etc.
- Program Comprehension, Code Review
  - In Kombination mit Visualisierungen
    - Beispiel: Darstellen von Abhängigkeiten
- Maschinenunabhängige Optimierungen im Compiler
  - Semantikerhaltende Transformationen zur Eliminierung von Redundanzen
    - Beispiel: Common Subexpression oder Dead Code Elimination

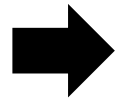
- *Software Metriken* und *Reverse Engineering* als Varianten der statischen Programmanalyse

Syntaxbaum → Kontrollflussgraph

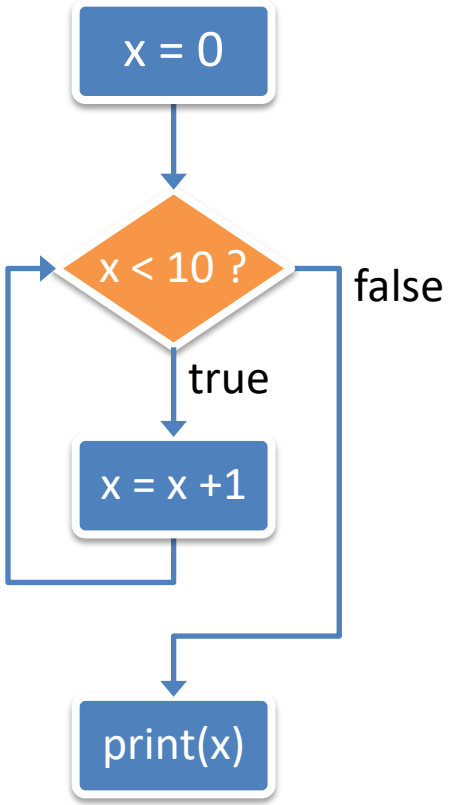
```
x = 0;
while (x < 10) {
    x = x + 1;
}

print(x);
```

Syntaktische Analyse

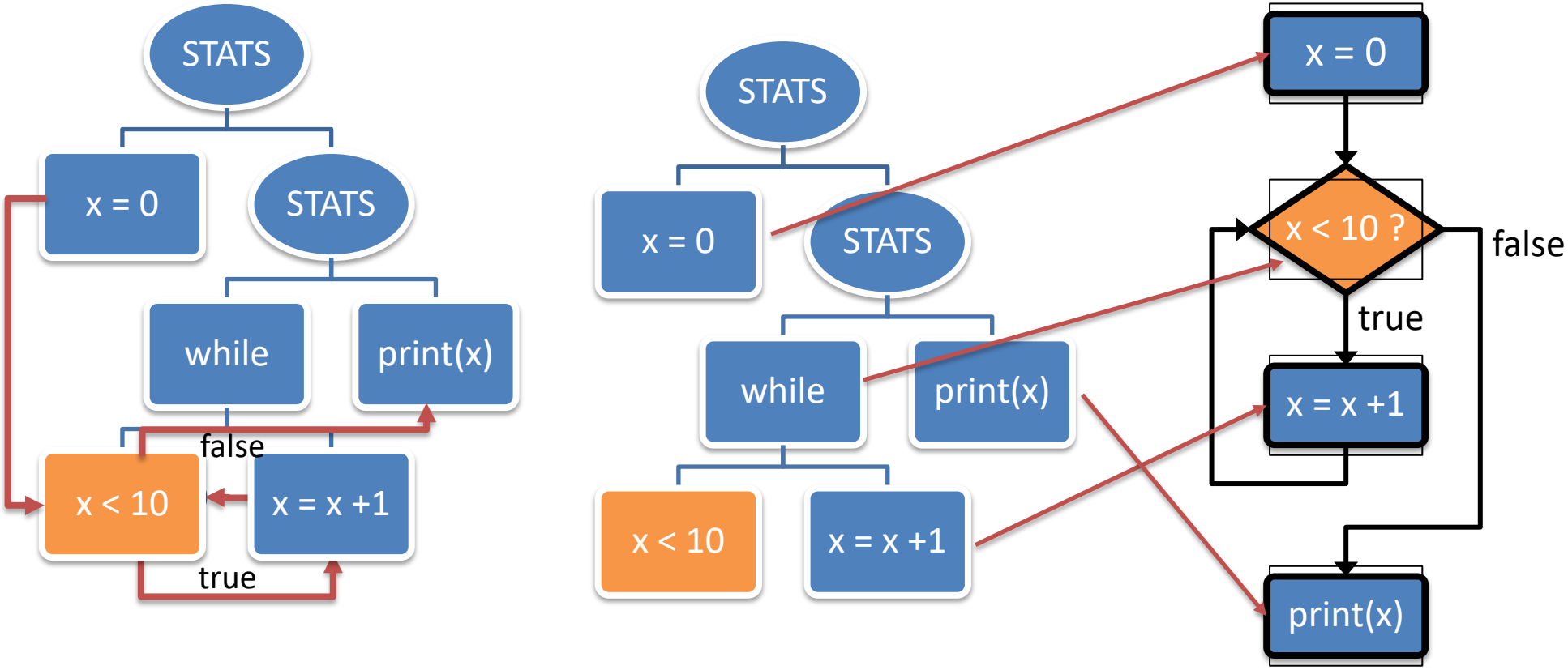




Kontrollfluss-analyse

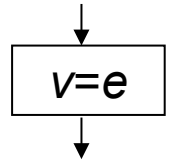


Implementierung: Verbindung von Syntaxbaum und KFG

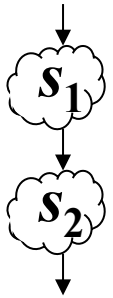


Berechnung eines Kontrollflussgraphen

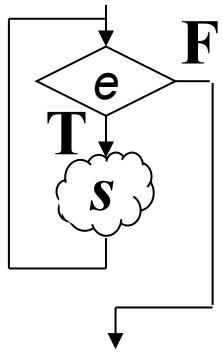
$v{=}e^p$



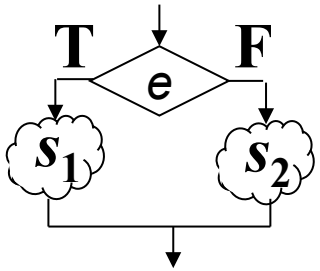
$s_1;s_2$



`while(`e`) {`s`}`^{p}



`if(`e`) {`s_1`} else {`s_2`}`^{p}



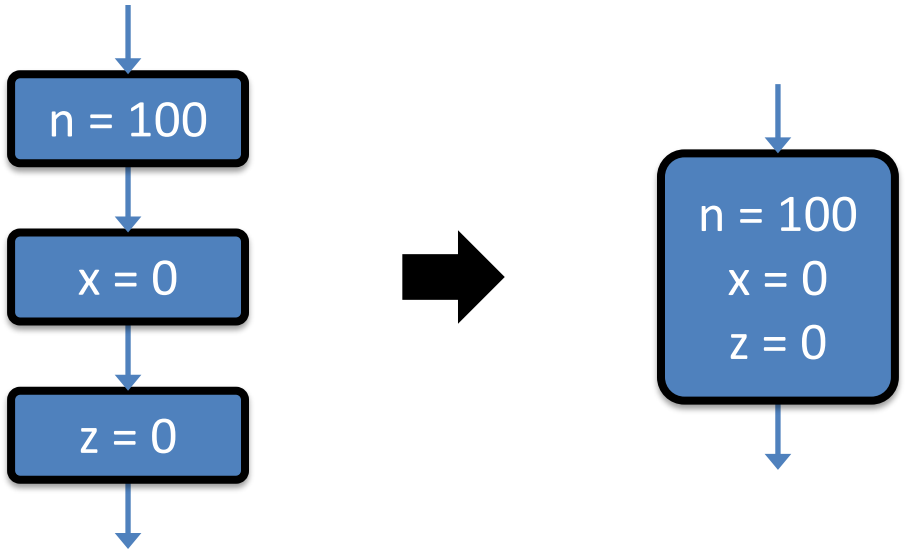
Formale Definition eines Kontrollflussgraphen

A CFG is a tuple (V, E, in, out) where

- V is a set of nodes,
- $E \subseteq V \times L \times V$ a set of edges,
- $L = \{\epsilon, \mathtt{T}, \mathtt{F}\}$ a set of labels,
- and $in, out \in V$ are start and end nodes.

Kontrollflussgraphen

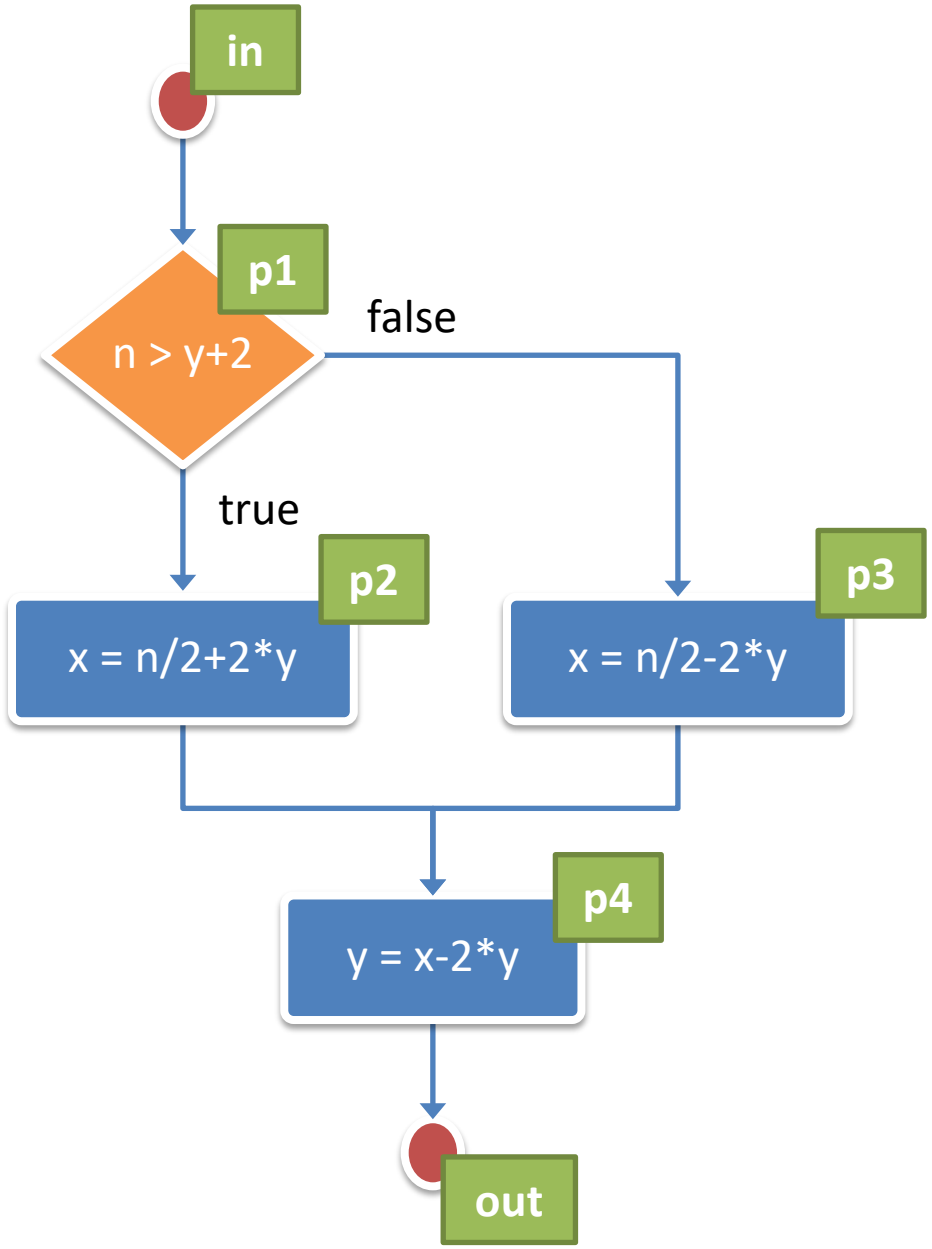
- Bemerkung:
  - Oft werden Folgen von einfachen Anweisungen, in die hinein oder aus denen heraus keine weiteren Kanten zeigen, zu einzelnen Knoten, sogenannten Grundblöcken (*basic blocks)*, zusammengefasst.



Kontrollflussanalyse

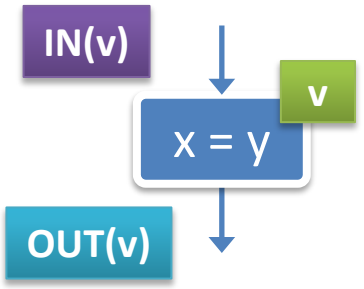
- Berechnet den Kontrollflussgraphen (KFG) eines Programms
  - Probleme bei echten Programmiersprachen:
    - Funktions- und Prozeduraufrufe
      → interprozeduraler KFG
    - Funktionszeiger (in funktionalen Sprachen höherer Ordnung, aber auch in C)
    - Dynamische Auswahl (dispatch) in objekt-orientierten Sprachen wie Java

```
if (n>y+2)
    x = n/2+2*y
else
    x = n/2-2*y
y=x-2*y
```



Datenflussanalyse

- Berechnet für jeden Programmpunkt (Knoten in KFG) Eigenschaften der Daten, die diesen Programmpunkt während der Ausführung erreichen können.
- Im allgemeinen werden zwei Informationen für jeden Knoten *v* berechnet:
  - **IN(v)**: Information über den Zustand, bevor der Programmpunkt ausgeführt wird.
  - **OUT(v)**: Information über den Zustand, nachdem der Programmpunkt ausgeführt wird.



- **Methode:** Lokal verfügbare Information wird entlang der Pfade des KFG propagiert.

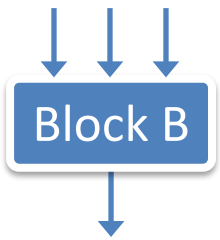
Datenflussanalyse

Für alle Knoten v

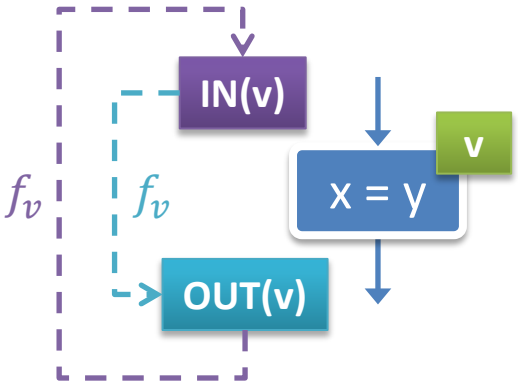
- **Ziel:** Finden einer Lösung für $IN(v)$ bzw. $OUT(v)$ unter einer Reihe von Bedingungen:
  - Transferfunktionen (Semantik einer Anweisung)
    - Vorwärtsfluss: $OUT(v) = f_v(IN(v))$
    - Rückwärtsfluss: $IN(v) = f_v(OUT(v))$
  - Einschränkungen im Kontrollfluss
    - Innerhalb eines Grundblocks:

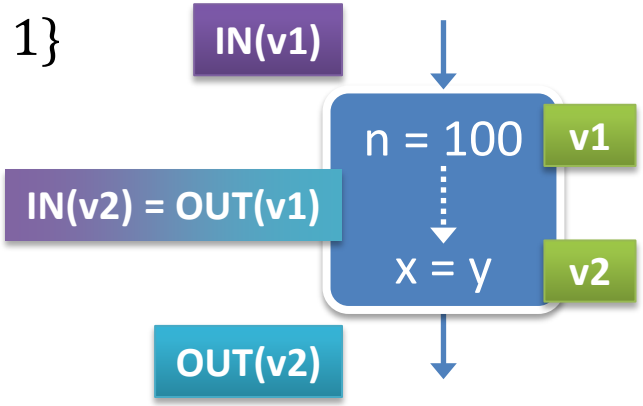
$$IN(v_{i+1}) = OUT(v_i) \quad \forall i \in \{1, \dots, n-1\}$$

    - Kanten zwischen Grundblöcken, z.B.:



$$IN(B) = \bigcup_{P \text{ Vorgänger von } B} OUT(P)$$





Datenflussanalyse

- **Vorwärtsflussprobleme:** was kann passieren, **bevor** die Ausführung den Programmpunkt erreicht, z.B.
  - Erreichende Definitionen (reaching definitions)
  - Verfügbare Ausdrücke (available expressions)

- **Rückwärtsflussprobleme**: was kann passieren, **nachdem** die Ausführung den Programmpunkt verlässt, z.B.
  - Lebendige Variablen (live variables)
  - Vielgefragte Ausdrücke (very busy expressions)
  - Erreichbare Verwendungen (reached uses)

Erreichende Definitionen

(Reaching Definitions)

- Sei *x* die von *d* definierte Variable, dann gilt:
  Definition *d* **erreicht** Programmpunkt *p*, wenn ein Pfad von *d* nach *p* existiert und dieser keine andere Definition *d'* enthält, die x (erneut) definiert.
- Nicht immer leicht zu ermitteln (z.B. indirekte Referenzen)
  - **Konservative Vorgehensweise:** Weiß man nicht, ob eine andere Wertzuweisung erfolgt ist, so nimmt man an, dass dies grundsätzlich möglich sein kann.
- **Anwendungsbeispiel:** Debugging
  - Ist Variable *x* an Programmpunkt *p* undefiniert?
- Vorwärtsflussproblem

Verfügbare Ausdrücke

(Available Expressions)

- Ein binärer Ausdruck e_1 *op* e_2 ist **verfügbar** am Programmpunkt *p*, wenn er auf jeden Fall zuvor berechnet wurde, d.h. entlang jeden Pfades über den *p* erreicht werden kann.

- **Anwendungsbeispiel:** Programmoptimierung
  - Erzeuge temporäre Variable, um erneute Berechnung zu vermeiden.

- Vorwärtsflussproblem

Beispiel

a=0

b=1

c=2

b=a*3+b*4+c*5

c=a*3-b*4

d=a*3+c*5

Verfügbare Ausdrücke

Definitionen

Sei G = (V, E, *in*, *out*) ein Kontrollflussgraph und Γ die Menge aller binärer Ausdrücke des Programms:

$$GEN, KILL : V \to L_G(E)$$

Knoten v generiert einen Ausdruck:

$$GEN(\mathrm{v}) = \begin{cases} \{e' \mid e' \text{ binärer Teilausdruck von } e\}, & \text{falls } v = \boxed{e} \\ \{e' \mid e' \text{ binärer Teilausdruck von } e \text{ und } x \text{ kommt nicht darin vor}\}, & \text{falls } v = \boxed{x = e} \\ \emptyset, & \text{sonst} \end{cases}$$

Knoten v zerstört einen Ausdruck:

$$KILL(\mathrm{v}) = \begin{cases} \{e' \mid e' \in \Gamma \text{ und } x \text{ kommt in } e' \text{vor}\}, & \text{falls } v = \boxed{x = e} \\ \emptyset & \text{sonst} \end{cases}$$

VA: Kill/Gen berechnen

$$GEN, KILL : V \rightarrow L_G(E)$$

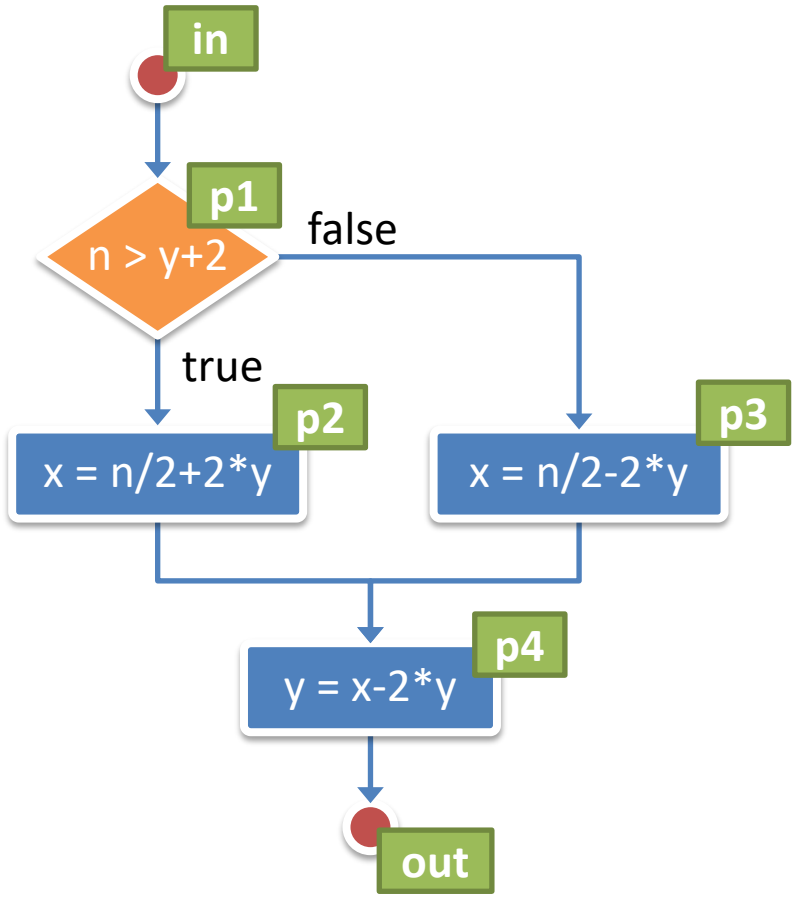
$$GEN(\mathrm{v}) = \begin{cases} \{e' \mid e' \text{ binärer Teilausdruck von } e\}, & \text{falls } v = \langle e \rangle \\ \{e' \mid e' \text{ binärer Teilausdruck von } e \text{ und } x \text{ kommt nicht darin vor}\}, & \text{falls } v = [x = e] \\ \emptyset, & \text{sonst} \end{cases}$$

$$KILL(\mathrm{v}) = \begin{cases} \{e' \mid e' \in \Gamma \text{ und } x \text{ kommt in } e' \text{vor}\}, & \text{falls } v = [x = e] \\ \emptyset & \text{sonst} \end{cases}$$

① ② ③ ④ ⑤ ⑥

$\Gamma = \{ \mathrm{y+2},\ \mathrm{n/2},\ \mathrm{2*y},\ \mathrm{n/2+2*y},\ \mathrm{n/2-2*y},\ \mathrm{x-2*y} \}$

v	GEN(v) =	KILL(v) =
in	∅	∅
p1		
p2		
p3		
p4		
out	∅	∅



Verfügbare Ausdrücke

Algorithmus

$$IN(in) = \emptyset$$
$$OUT(in) = \emptyset$$

forall $v \in V \backslash \{in\}$ **do**
  $IN(v) = \Gamma$
  $OUT(v) = \big(IN(v) \backslash \mathrm{KILL}(v)\big) \cup GEN(v)$
od

while there are changes **do**
  forall $v \in V \backslash \{in\}$ **do**
    $IN(v) = \bigcap_{(p,l,v) \in E} OUT(p)$
    $OUT(v) = \big(IN(v) \backslash \mathrm{KILL}(v)\big) \cup GEN(v)$
  od
od

Ein Ausdruck ist vor einem Programmpunkt nur dann verfügbar, wenn er nach **allen** vorangehenden Programmpunkten verfügbar ist.

Ein Ausdruck ist nach einen Programmpunkt verfügbar, wenn er davor verfügbar ist und nicht zerstört wurde …

… oder wenn er im Programmpunkt generiert wird.

Beispiel: Verfügbare Ausdrücke

$$GEN, KILL : V \rightarrow L_G(E)$$

$$GEN(\mathrm{v}) = \begin{cases} \{e' \mid e' \text{ binärer Teilausdruck von } e\}, & \text{falls } v = \boxed{e} \\ \{e' \mid e' \text{ binärer Teilausdruck von } e \text{ und } x \text{ kommt nicht darin vor}\}, & \text{falls } v = \boxed{x = e} \\ \emptyset, & \text{sonst} \end{cases}$$

$$KILL(\mathrm{v}) = \begin{cases} \{e' \mid \boxed{e' \in \Gamma} \text{ und } x \text{ kommt in } e' \text{vor}\}, & \text{falls } v = \boxed{x = e} \\ \emptyset & \text{sonst} \end{cases}$$

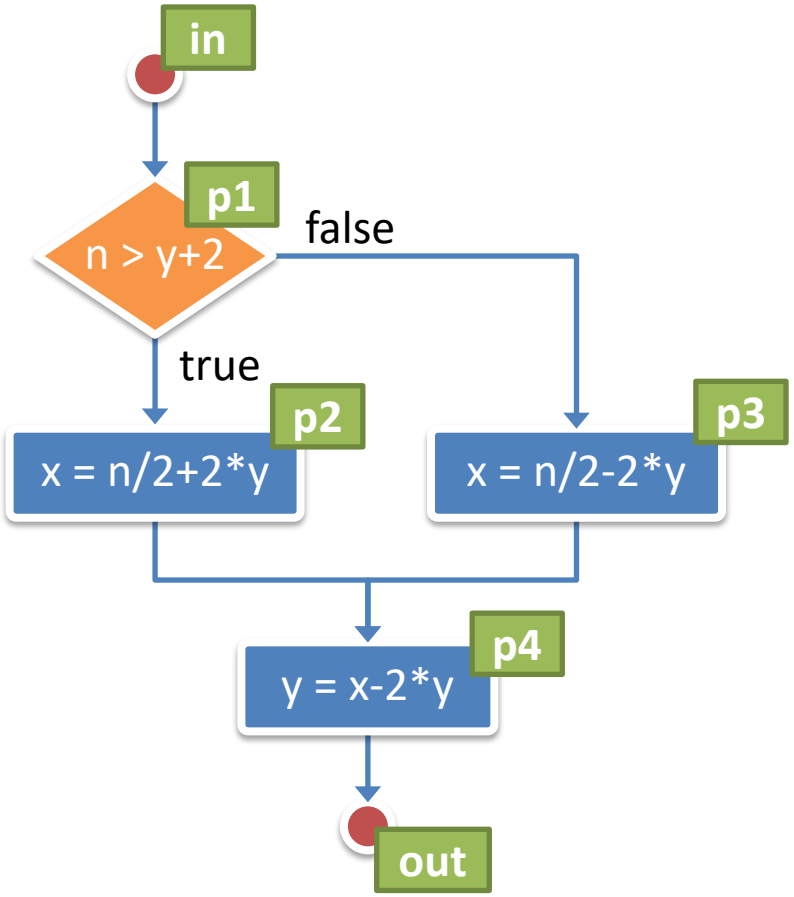
① ② ③ ④ ⑤ ⑥

$\Gamma = \{y+2, n/2, 2*y, n/2+2*y, n/2-2*y, x-2*y\}$

$IN(in) = \emptyset$
$OUT(in) = \emptyset$

forall $v \in V \backslash \{in\}$ **do**
  $IN(v) = \Gamma$
  $OUT(v) = \big(IN(v) \backslash \mathrm{KILL}(v)\big) \cup GEN(v)$
od

while there are changes **do**
  forall $v \in V \backslash \{in\}$ **do**
    $IN(v) = \bigcap_{(p,l,v) \in E} OUT(p)$
    $OUT(v) = \big(IN(v) \backslash \mathrm{KILL}(v)\big) \cup GEN(v)$
  od
od



v	GEN(v) =	KILL(v) =
in	Ø	Ø
p1	{1}	Ø
p2	{2,3,4}	{6}
p3	{2,3,5}	{6}
p4	Ø	{1, 3, 4, 5, 6}
out	Ø	Ø

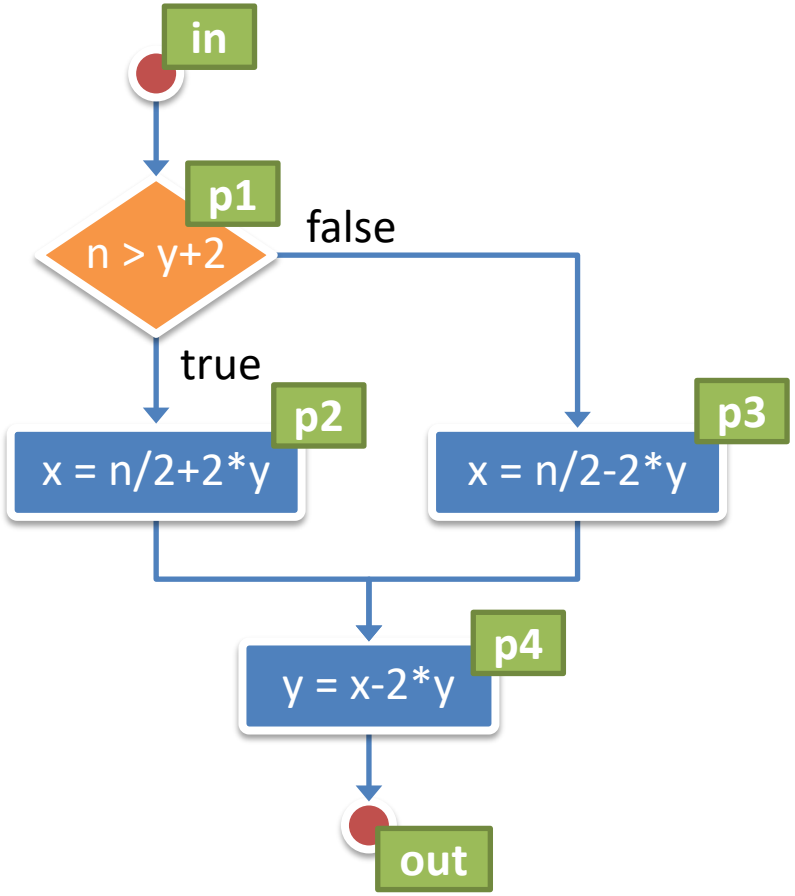
Γ = { y+2, ①
n/2, ②
2*y, ③
n/2+2*y, ④
n/2-2*y, ⑤
x-2*y} ⑥

Initialisierung:

$IN(in) = \emptyset$
$OUT(in) = \emptyset$

forall $v \in V \backslash \{in\}$ **do**
$IN(v) = \Gamma$
$OUT(v) = \left(IN(v) \backslash \text{KILL}(v)\right) \cup GEN(v)$
od

VA: Initialisierung



v	IN(v) =	OUT(v) =
in	Ø	Ø
p1	Γ	
p2	Γ	
p3	Γ	
p4	Γ	
out	Γ	

v	GEN(v) =	KILL(v) =
in	Ø	Ø
p1	{1}	Ø
p2	{2,3,4}	{6}
p3	{2,3,5}	{6}
p4	Ø	{1, 3, 4, 5, 6}
out	Ø	Ø

Initialisierung:

v	IN(v) =	OUT(v) =
in	Ø	Ø
p1	Γ	Γ
p2	Γ	{1,2,3,4,5}
p3	Γ	{1,2,3,4,5}
p4	Γ	{2}
out	Γ	Γ

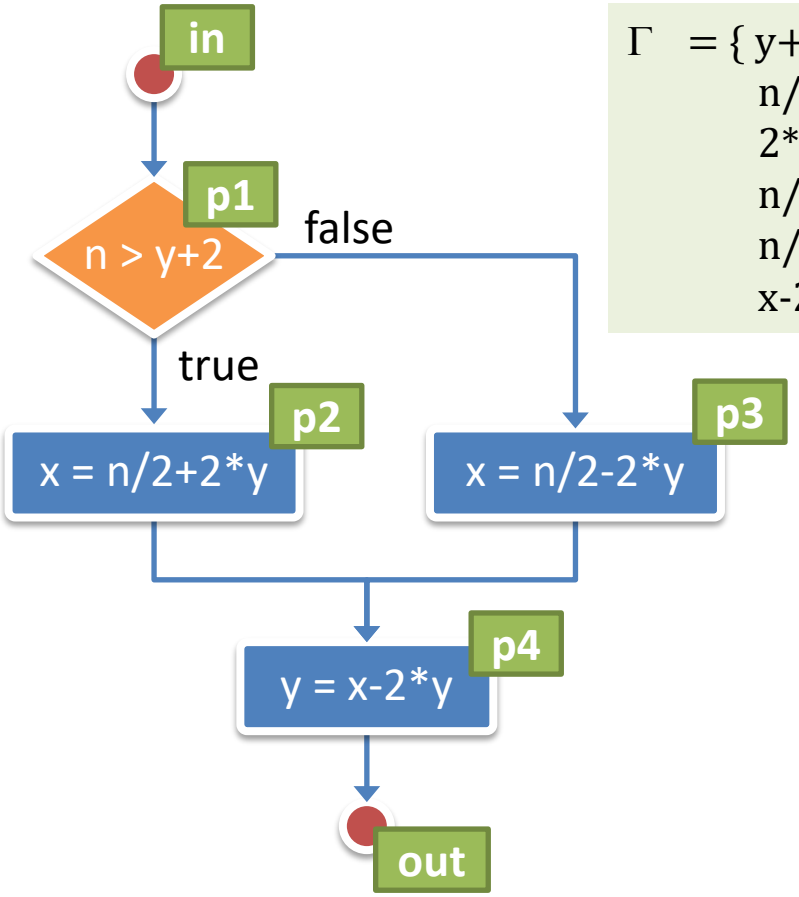
VA: Erste Iteration

while there are changes **do**
  forall $v \in V \backslash \{in\}$ **do**

$$IN(v) = \bigcap_{(p,l,v) \in E} OUT(p)$$

$$OUT(v) = \left(IN(v) \backslash \text{KILL}(v)\right) \cup GEN(v)$$

  od
od



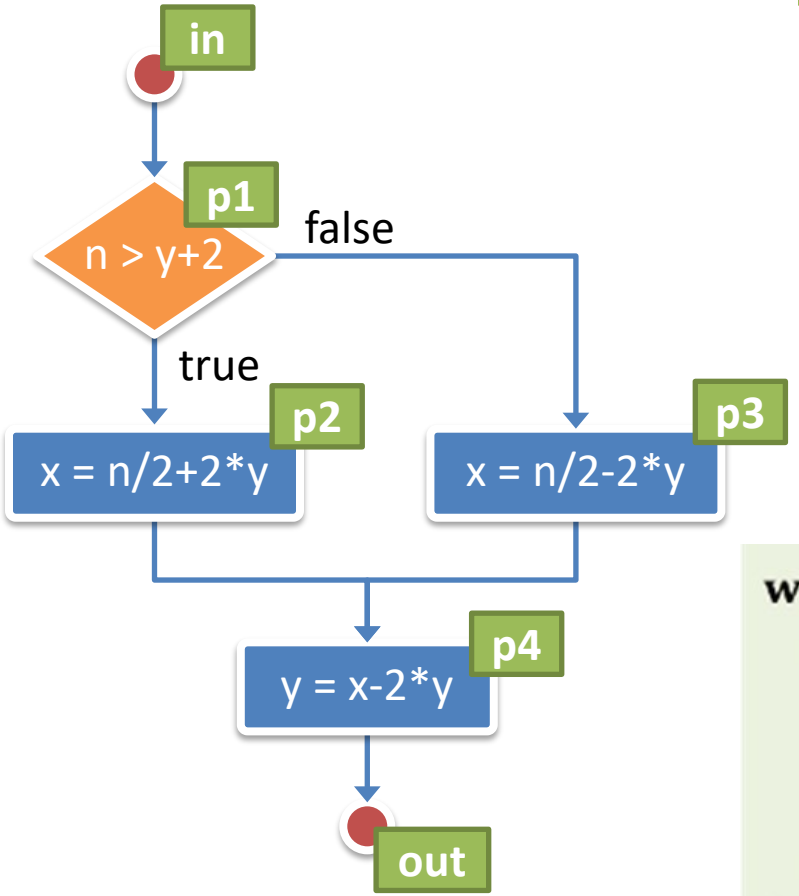
Γ = { y+2, (1)
n/2, (2)
2*y, (3)
n/2+2*y, (4)
n/2-2*y, (5)
x-2*y} (6)

Erste Iteration:

v	IN(v) =	OUT(v) =
in	Ø	Ø
p1		
p2		
p3		
p4		
out		

Beispiel: Verfügbare Ausdrücke

v	GEN(v) =	KILL(v) =
in	Ø	Ø
p1	{1}	Ø
p2	{2,3,4}	{6}
p3	{2,3,5}	{6}
p4	Ø	{1, 3, 4, 5, 6}
out	Ø	Ø



Erste Iteration:

v	IN(v) =	OUT(v) =
in	Ø	Ø
p1	Ø	{1}
p2	{1}	{1,2,3,4}
p3	{1}	{1,2,3,5}
p4	{1,2,3}	{2}
out	{2}	{2}

Zweite Iteration:

v	IN(v) =	OUT(v) =
in	Ø	Ø
p1	Ø	{1}
p2	{1}	{1,2,3,4}
p3	{1}	{1,2,3,5}
p4	{1,2,3}	{2}
out	{2}	{2}

$\Gamma = \{ y+2, \; n/2, \; 2*y, \; n/2+2*y, \; n/2-2*y, \; x-2*y \}$ (1) (2) (3) (4) (5) (6)

Keine Änderung → Fixpunkt

while there are changes **do**
  forall $v \in V \backslash \{in\}$ **do**

$$IN(v) = \bigcap_{(p,l,v) \in E} OUT(p)$$

$$OUT(v) = (IN(v) \backslash \text{KILL}(v)) \cup GEN(v)$$

  od
od

Lebendige Variablen
(Live Variables)

- Eine Variable *x* ist **lebendig** am Programmpunkt *p,* wenn es einen Pfad von *p* nach *p‘* gibt und
  - *x* von *p‘* verwendet (kommt im Ausdruck vor) wird
  - und es keine Redefinition von *x* (Zuweisung an *x*) entlang dieses Pfades gibt.

- **Anwendungsbeispiel:** Dead Code Elimination
  - z.B. Entfernen von „toten“ Zuweisungsblöcken

- Rückwärtsflussproblem

Beispiel

```
a=b+c
if (d>0)
        a=5
   else
        b=5
d=a+c
```

Lebendige Variablen

Definitionen

Sei G = (V, E, *in*, *out*) ein Kontrollflussgraph:

$$USE, DEF: V \to L_G(V)$$

Knoten v verwendet Variable:

$$USE(\mathrm{v}) = \begin{cases} \{x' \mid x' \text{ kommt in } e \text{ vor}\}, & \text{falls } v \in \{ \boldsymbol{x = e}, \boldsymbol{e}\} \\ \emptyset, & \text{sonst} \end{cases}$$

x' kann insbesondere x sein

Knoten v definiert Variable:

$$DEF(\mathrm{v}) = \begin{cases} \{x\}, & \text{falls } v = \boldsymbol{x = e} \\ \emptyset, & \text{sonst} \end{cases}$$

Lebendige Variablen

Algorithmus

forall $v \in V$ **do**
  $IN(v) = USE(v)$
od

while there are changes **do**
  forall $v \in V$ **do**
    $OUT(v) = \bigcup_{(v,l,s) \in E} IN(s)$
    $IN(v) = \left(OUT(v) \setminus DEF(v)\right) \cup USE(v)$
  od
od

Eine Variable ist nach einem Programmpunkt nur dann lebendig, wenn sie vor einem der nachfolgenden Programmpunkte lebendig ist.

Eine Variable ist vor einem Programmpunkt lebendig, wenn sie danach lebendig ist und nicht neudefiniert wurde …

… oder wenn sie im Programmpunkt verwendet wird.

Beispiel: Lebendige Variablen

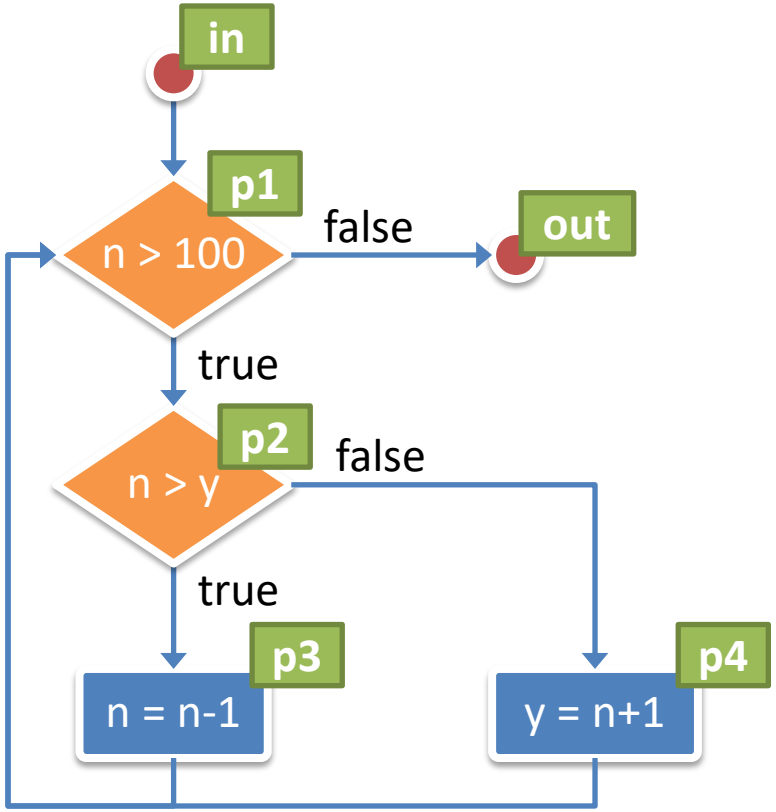
$$USE, DEF: V \to L_G(V)$$

$$USE(\mathrm{v}) = \begin{cases} \{x' \mid x' \text{ kommt in } e \text{ vor}\}, & \text{falls } v \in \{x = e, e\} \\ \emptyset, & \text{sonst} \end{cases}$$

$$DEF(\mathrm{v}) = \begin{cases} \{x\}, & \text{falls } v = x = e \\ \emptyset, & \text{sonst} \end{cases}$$

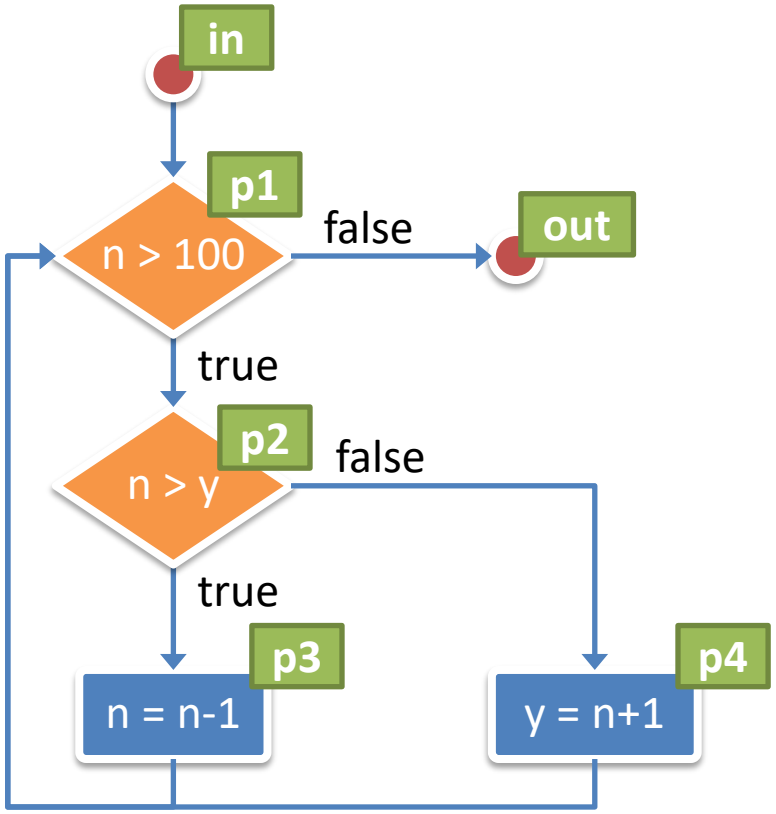
forall $v \in V$ **do**
  $IN(v) = USE(v)$
od

while there are changes **do**
  forall $v \in V$ **do**
    $OUT(v) = \bigcup_{(v,l,s) \in E} IN(s)$
    $IN(v) = \left(OUT(v) \backslash DEF(v)\right) \cup USE(v)$
  od
od



Beispiel: Lebendige Variablen

v	USE(v) =	DEF(v) =
in	Ø	Ø
p1	{n}	Ø
p2	{n, y}	Ø
p3	{n}	{n}
p4	{n}	{y}
out	Ø	Ø



Initialisierung:

$IN = USE$

Erste Iteration:

v	IN(v) =	OUT(v) =
in	{n}	{n}
p1	{n, y}	{n, y}
p2	{n, y}	{n}
p3	{n, y}	{n, y}
p4	{n}	{n, y}
out	Ø	Ø

Zweite Iteration:

v	IN(v) =	OUT(v) =
in	{n, y}	{n, y}
p1	{n, y}	{n, y}
p2	{n, y}	{n, y}
p3	{n, y}	{n, y}
p4	{n}	{n, y}
out	Ø	Ø

Dritte Iteration:

Keine Änderung → Fixpunkt

Vielgefragte Ausdrücke (Very Busy Expressions)

- Ein Ausdruck e ist **vielgefragt** an einem Programmpunkt p, wenn alle von p ausgehenden Pfade dessen Auswertung benötigen, bevor der Wert von e verändert wird.

- **Anwendungsbeispiel:** Programmoptimierungen
  - Vorberechnen eines vielgefragten Ausdrucks

- Rückwärtsflussproblem

Erreichbare Verwendungen

(Reached Uses)

- Eine Verwendung U von x ist von einer Zuweisung A mit x = e aus **erreichbar**, wenn es einen Pfad von A nach U gibt und x auf diesem nicht (erneut) definiert wurde.

- **Anwendungsbeispiel:** Programmoptimierungen
  - Finden von nicht verwendeten Zuweisungen

- Rückwärtsflussproblem