Last-Def-Analyse (Reaching Definitions)

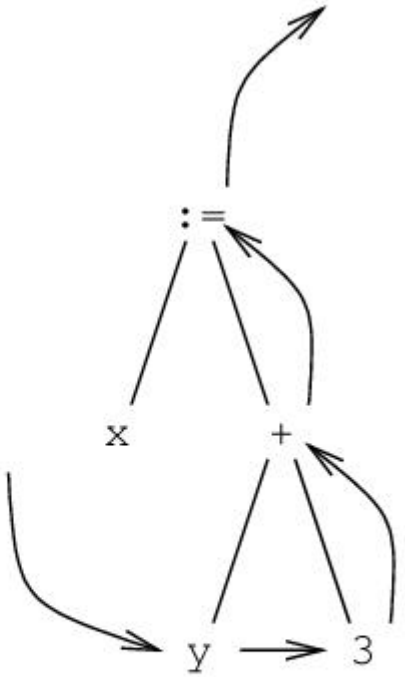
- Idee:
  - Jedes Vorkommen einer Variablen v wird mit Zeigern auf die Programmstellen annotiert, an denen der letzte Wert entstanden sein kann.
- Kriterien:
  - Die Programmstelle muss den Wert von v ändern.
  - Auf jedem Kontrollpfad von der Stelle zur betrachteten Benutzung von v darf der Wert von v nicht weiter verändert werden.
- Anwendungen:
  - Registervergabe und Speicherverwaltung
  - Parallelisierung
- Anmerkung:
  - Last-Def-Analyse braucht volle symbolische Interpretation!
  - Verletzung von Bedingung 4: in Schleifen kann eine Last-Def aus der vorherigen Iteration vorkommen.

Analyse mit Datenflussgleichungen

- Unterschiede zur symbolischen Interpretation:
  - formellere Beschreibung der Semantik eines Knotens: Informationsverarbeitung mit Hilfe von vier Mengen (statt eines Stacks)
  - Teile des Kontrollflussgraphen können zu einem Knoten zusammengefasst werden (z.B. eine Zuweisung)
  - Baumdurchlauf muss nicht dem Kontrollfluss folgen (keine Interpretation)
  - halbautomatisch (statt manuell)
  - keine Fehlerüberprüfung (erfordert zusätzliche Durchläufe)
- Beispiele für Information: am Knoten N
  - `x=1`
  - keine entfernter Programmaufruf seit Routinenbeginn
  - Wert von `y` wird am Knoten N_1 und N_2 definiert
  - globale Variable `count` seit Routinenbeginn verändert

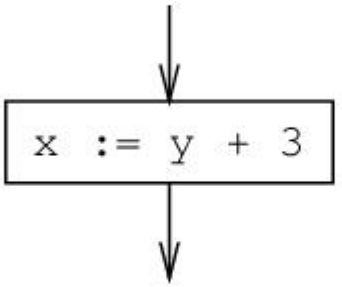
Graphknoten für Zuweisungen

oft sogar: ein Knoten für einen gesamten Basisblock



(a)

Menge *IN* von Bedingungen



Menge *OUT* von Bedingungen

(b)

Figure 3.48 An assignment as a full control flow graph and as a single node.

IN, *OUT*, *KILL* und *GEN*

- Mit einem Knoten N assoziierte Informationen:
  - $IN(N)$: Eingangsinformation
  - $OUT(N)$: Ausgangsinformation
  - $KILL(N)$: Information, die durch N ungültig wird
  - $GEN(N)$: Information, die durch N hergestellt wird
- Zusammenhang (Semantik)
  - Kanten (Vorwärtsanalyse): $IN(N) = \bigcup_{\{M \mid N \text{ dynamischer Nachfolger von } M\}} OUT(M)$
  - Knoten: $OUT(N) = (IN(N) \setminus KILL(N)) \cup GEN(N)$

  Bsp.: `x:=y`

$IN(N)$	$KILL(N)$	$GEN(N)$	$OUT(N)$
`{'x=0'}`	`{'x=*'}`	`{'x=y'}`	`{'x=y'}`

- Anmerkungen:
  - Umsetzung von \ und ∪ hängen von der Repräsentation der Information ab (\ = Informationsdifferenz, ∪ = Informationsvereinigung)
  - beide lassen sich oft als Mengenoperation implementieren

Routinenrümpfe mit *einem* Ausgang

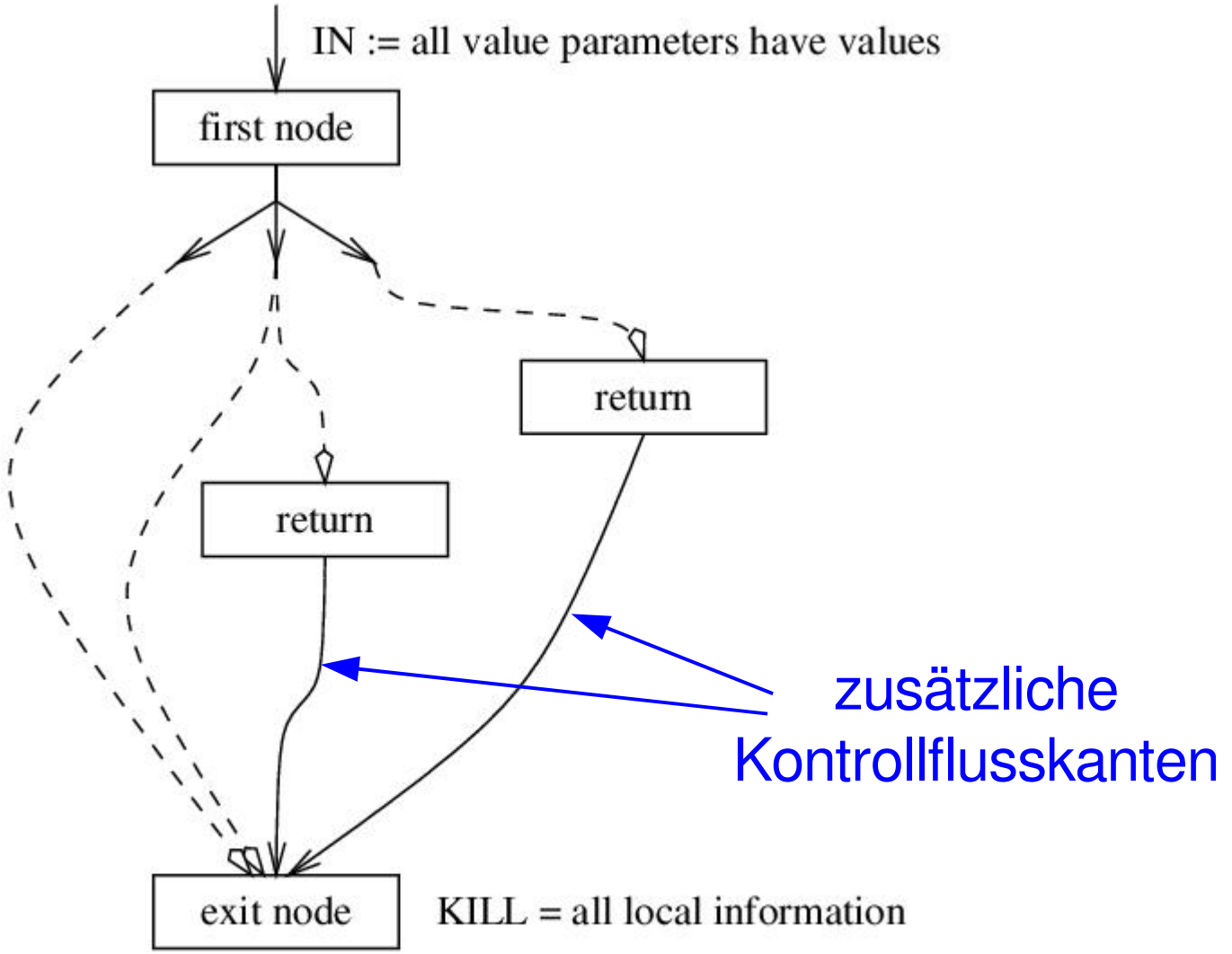


Figure 3.50 Data-flow details at routine entry and exit.

Lösen der Datenflussgleichungen (1)

- Idee
  - Hüllenalgorithmus zur Informationspropagierung (Abb. 3.51)
  - erfordert wiederholten Durchlauf des Kontrollflussgraphen
  - Durchläufe müssen nicht immer dem Kontrollfluss folgen
  - Termination bei Erreichen des Fixpunkts

Hüllenalgorithmus

Data definitions:

1. Constant *KILL* and *GEN* sets for each node.
2. Variable *IN* and *OUT* sets for each node.

Initializations:

1. The *IN* set of the top node is initialized with information established externally.
2. For all other nodes N, $IN(N)$ and $OUT(N)$ are set to empty.

Inference rules:

1. For any node N, $IN(N)$ should contain

$$\bigcup_{M=\text{dynamic predecessor of } N} OUT(M).$$

2. For any node N, $OUT(N)$ should contain

$$(IN(N) \setminus KILL(N)) \cup GEN(N).$$

Figure 3.51 Closure algorithm for solving the data-flow equations.

Lösen der Datenflussgleichungen (2)

- Idee
  - Hüllenalgorithmus zur Informationspropagierung (Abb. 3.51)
  - erfordert wiederholten Durchlauf des Kontrollflussgraphen
  - Durchläufe müssen nicht immer dem Kontrollfluss folgen
  - Termination bei Erreichen des Fixpunkts
- Bsp.: Propagierung von Initialisierungsinformation durch ein `if`
  - Statement: `if y>0 then x:=y else y:=0`
  - Annahme: `y` initialisiert, `x` nicht

Beispiel: Initialisierungsinformation

- Drei Möglichkeiten:
  1. Variable ist sicher nicht initialisiert
  2. Initialisierungszustand unbekannt
  3. Variable ist sicher initialisiert
- Problem:
  - Formuliere Bedingungen so, dass Ihre Vereinigung dem bitweisen logischen Oder entspricht
- Lösung:

  Bedingungen lauten: *es kann sein ...*

  (a) *dass Variable initialisiert ist*

  (b) *dass Variable nicht initialisiert ist*

  Bedingung nicht erfüllt: *es kann nicht sein ...* (sicher!)

Lösen der Datenflussgleichungen (3)

- Idee
  - Hüllenalgorithmus zur Informationspropagierung (Abb. 3.51)
  - erfordert wiederholten Durchlauf des Kontrollflussgraphen
  - Durchläufe müssen nicht immer dem Kontrollfluss folgen
  - Termination bei Erreichen des Fixpunkts
- Bsp.: Propagierung von Initialisierungsinformation durch ein `if`
  - Statement: `if y>0 then x:=y else y:=0`
  - Annahme: `y` initialisiert, `x` nicht
  - Kodierung: dreiwertige Logik (Abb. 3.52-53)

00	01	10	11	$A \wedge \neg B$	$A \vee B$
Fehler	`Init`	`Uninit`	`Maybeinit`	$A \setminus B$	$A \cup B$

  - Informationspropagierung: (Abb. 3.54)

Beispiel: Initialisierungsinformation

Kodierung durch Bitmuster

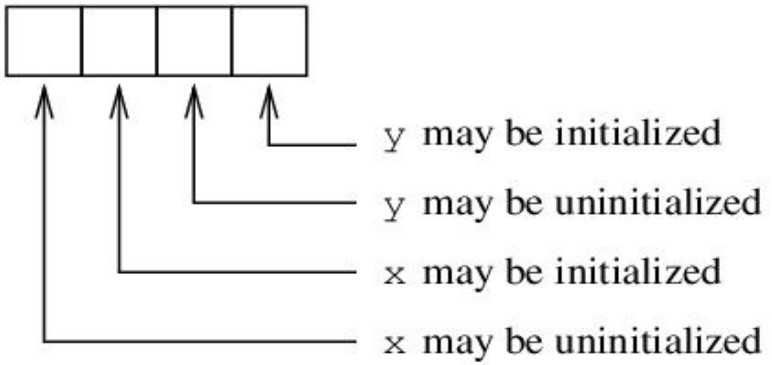


Figure 3.52 Bit patterns for properties of the variables x and y.

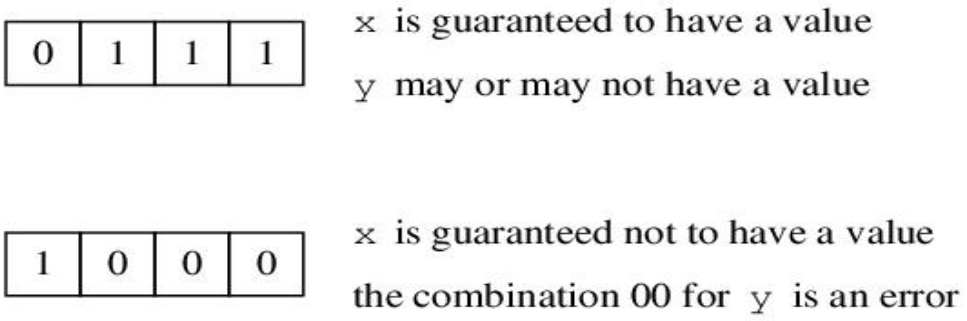
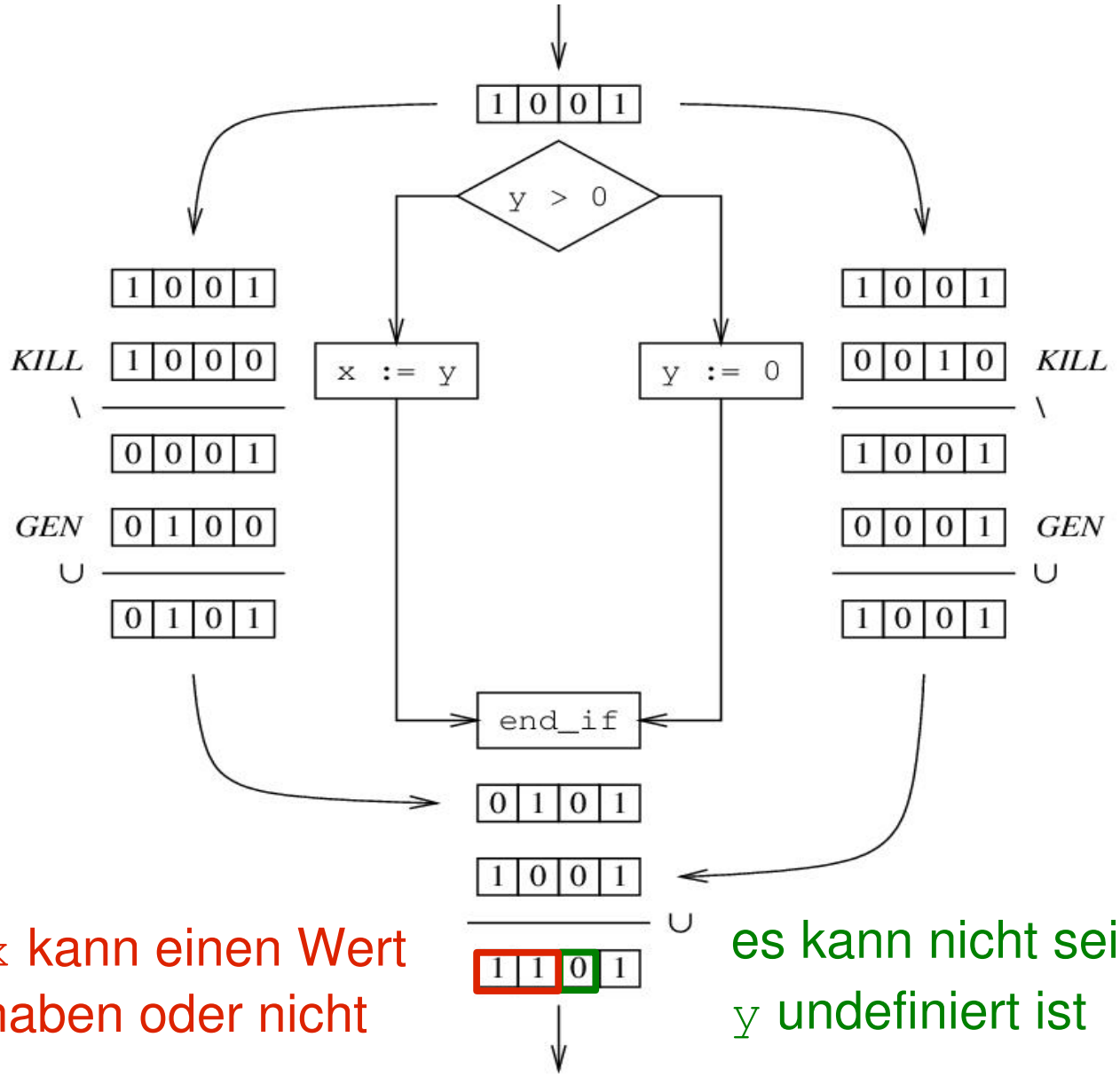


Figure 3.53 Examples of bit patterns for properties of the variables x and y.

Datenflussberechnung (Beispiel)

Programmteil:

```
if(y>0)
then x:=y;
else y:=0;
```



x kann einen Wert haben oder nicht

es kann nicht sein, dass y undefiniert ist

Figure 3.54 Data-flow propagation through an if-statement.

Interprozedurale Datenflussanalyse (1a)

zwei Richtungen der Informationsübertragung:

- von der rufenden in die aufgerufene Prozedur
  - Information über den Kontext
  - Bsp.: `pow(x,3)` kann optimiert werden zu `x*x*x`
- bei der Rückkehr in die rufende Prozedur
  - Information über das Verhalten der Prozedur
  - Bsp.: eine Funktion hat keine Seiteneffekte

Interprozedurale Datenflussanalyse (1b)

- zwei Richtungen der Informationsübertragung
- anwendbare Methoden:
  - symbolische Interpretation
    - umkopieren der aktuellen/formalen Parameter
  - Lösen von Datenflussgleichungen
    - Informationsaustausch über Entry- und Exit-Knoten

Interprozedurale Datenflussanalyse (2)

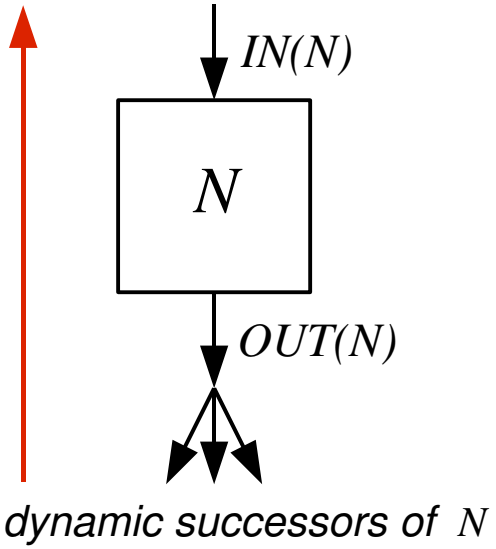
- Problem 1: tatsächlich aufgerufene Prozedur ist unbekannt
  - Grund: Funktionszeiger oder virtuelle Methoden
  - Lösung: Analyse aller in Frage kommenden Prozeduren (candidate set)
- Problem 2: nicht alle Flussgraphen sind verfügbar
  - Grund: separate Übersetzung von Modulen, Bibliothekscode
  - Lösung: Compiler erzeugt Datei mit Informationen über den Kontrollflussgraphen, zusätzlich zum Objektcode

Analyserichtung

- Vorwärtsanalyse (bisher betrachtet)
  - Daten-getrieben: folgt der Kantenrichtung im Kontollflussgraphen
  - Datenflussgleichungen
    - mergen von Ausgangs- zu Eingangsbedingungen
    - *KILL/GEN*: Eingangsbedingungen $\rightarrow$ Ausgangsbedingungen
- Rückwärtsanalyse (analog)
  - anforderungsgetrieben: entgegen der Kantenrichtung
  - Datenflussgleichungen
    - mergen von Eingangs- zu Ausgangsbedingungen
    - *KILL/GEN*: Ausgangsbedingungen $\rightarrow$ Eingangsbedingungen

Rückwärtsanalyse

- anforderungsgetrieben: entgegen der Richtung der Kanten des Datenflussgraphen
- Datenflussgleichungen
  - mergen von Eingangs- zu Ausgangsbedingungen
  - *KILL/GEN*: Ausgangsbedingungen → Eingangsbedingungen



$$OUT(N) = \bigcup_{M=\text{dynamic successor of } N} IN(M)$$

$$IN(N) = (OUT(N) \setminus KILL(N)) \cup GEN(N)$$

Figure 3.59 Backwards data-flow equations for a node *N*.

Liveness-Analyse

- Beispiel für Rückwärtsanalyse
- Definition (Liveness/Lebendigkeit):
  - eine Variable heisst *lebendig* an einem Knoten N im Kontrollflussgraphen gdw. sein Wert in N auf mindestens einem von N ausgehenden Pfad benutzt wird
  - eine Variable kann mehrere Lebenszeitintervalle haben
    - beginnend mit der Zuweisung eines Wertes
    - endend mit der letzten Benutzung dieses Wertes
- Motivation: (Aktionen am Ende eines Lebenszeitintervalls)
  - Speicherfreigabe für im Heap gespeicherte Datenobjekte
  - anderweitige Verwendung eines belegten Registers

Liveness-Analyse (Beispielcode)

```
{    int x;
     x = ...;                /* code fragment  1, does not use x */
     if (...) {
          ...                /* code fragment  2, does not use x */
          print(x);          /* code fragment  3, uses x */
          ...                /* code fragment  4, does not use x */
     } else {
          int y;
          ...                /* code fragment  5, does not use x,y */
          print(x);          /* code fragment  6, uses x, but not y */
          ...                /* code fragment  7, does not use x,y */
          y = ...;           /* code fragment  8, does not use x,y */
          ...                /* code fragment  9, does not use x,y */
          print(y);          /* code fragment 10, uses y but not x */
          ...                /* code fragment 11, does not use x,y */
     }
     x = ...;                /* code fragment 12, does not use x */
     ...                     /* code fragment 13, does not use x */
     print(x);               /* code fragment 14, uses x */
     ...                     /* code fragment 15, does not use x */
}
```

Figure 3.55 A segment of C code to demonstrate live analysis.

Liveness-Analyse / symb. Interpret. (1)

- Symbolische Interpretation
  - Vorwärtsdurchlauf durch den Kontrollflussgraphen
  - Rückpropagation von Information durch *Backpatching*
- Backpatching
  - merke die Stellen, wo die Information gebraucht wird
  - setze die Information ein, sobald sie verfügbar ist
  - Bsp. aus der Zwischencodegenerierung: Einsetzen von Sprungzielen
- speziell bei der Liveness-Analyse
  - halte Liste der letzten Verwendungen der Variablen

Liveness-Analyse / symb. Interpret. (2)

- Verwendung einer Variablen
  - setze Flag der letzten Verwendung
    - am aktuellen Knoten auf TRUE
    - an den Knoten aus der Backpatch-Liste auf FALSE
  - Backpatch-Liste für diese Variable := { aktuellen Knoten }
- Zuweisung einer Variablen (verwendet oder nicht)
  - setze Flag der letzten Verwendung
    - am aktuellen Knoten auf TRUE (Initialisierung / Konsistenz)
  - Backpatch-Liste für diese Variable := { aktuellen Knoten }

Liveness-Analyse / symb. Interpret. (3a)

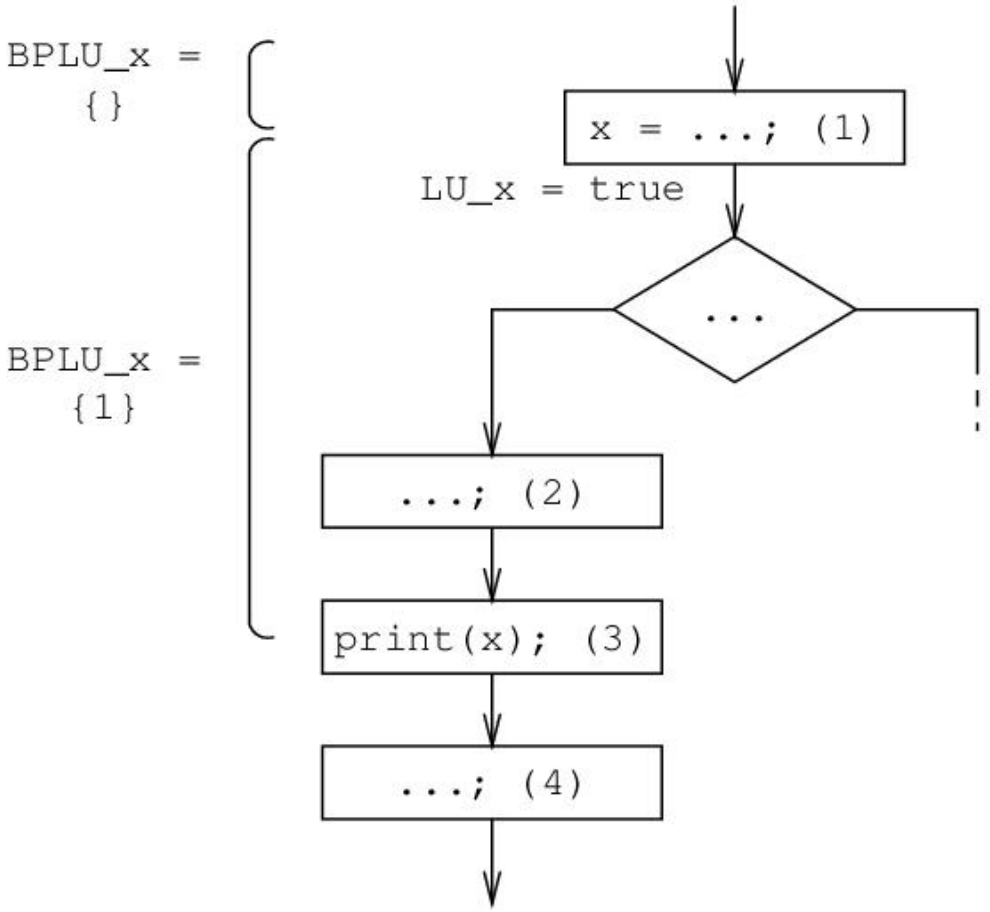


Figure 3.56 The first few steps in live analysis for Figure 3.55 using backpatch lists.

Liveness-Analyse / symb. Interpret. (3b)

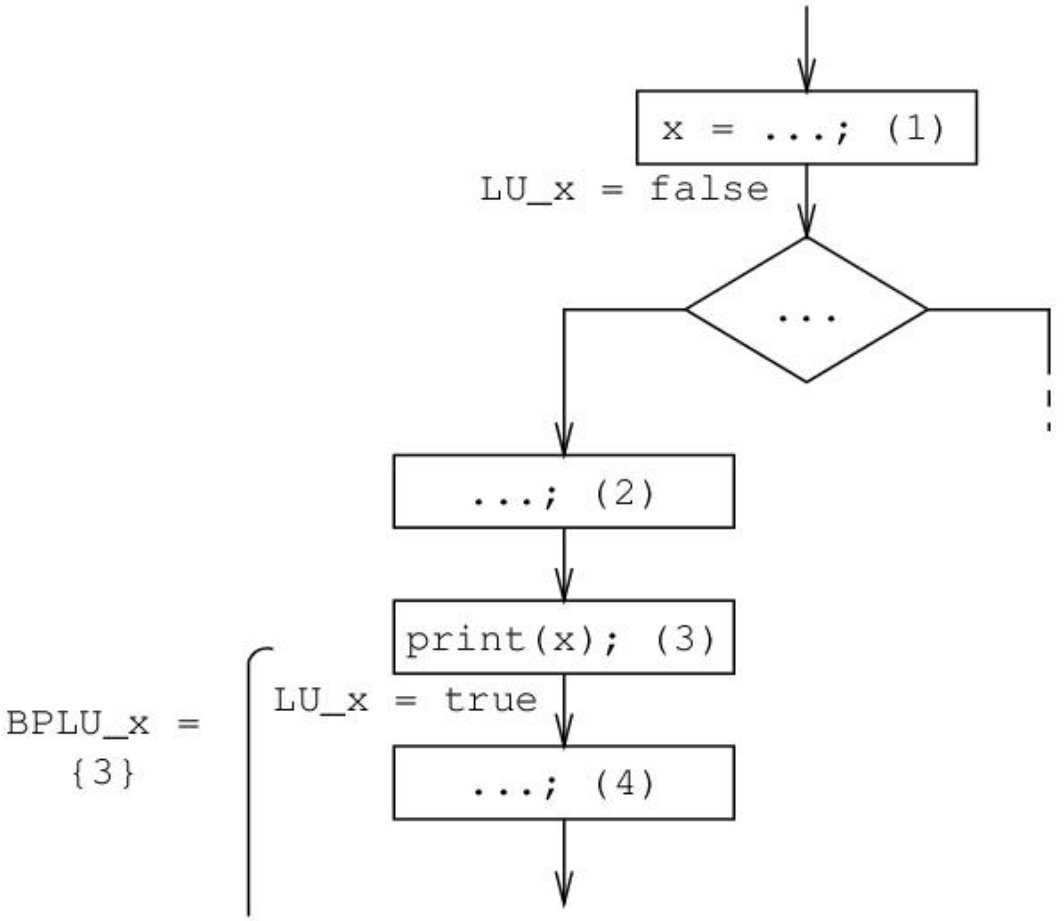


Figure 3.57 Live analysis for Figure 3.55, after a few steps.

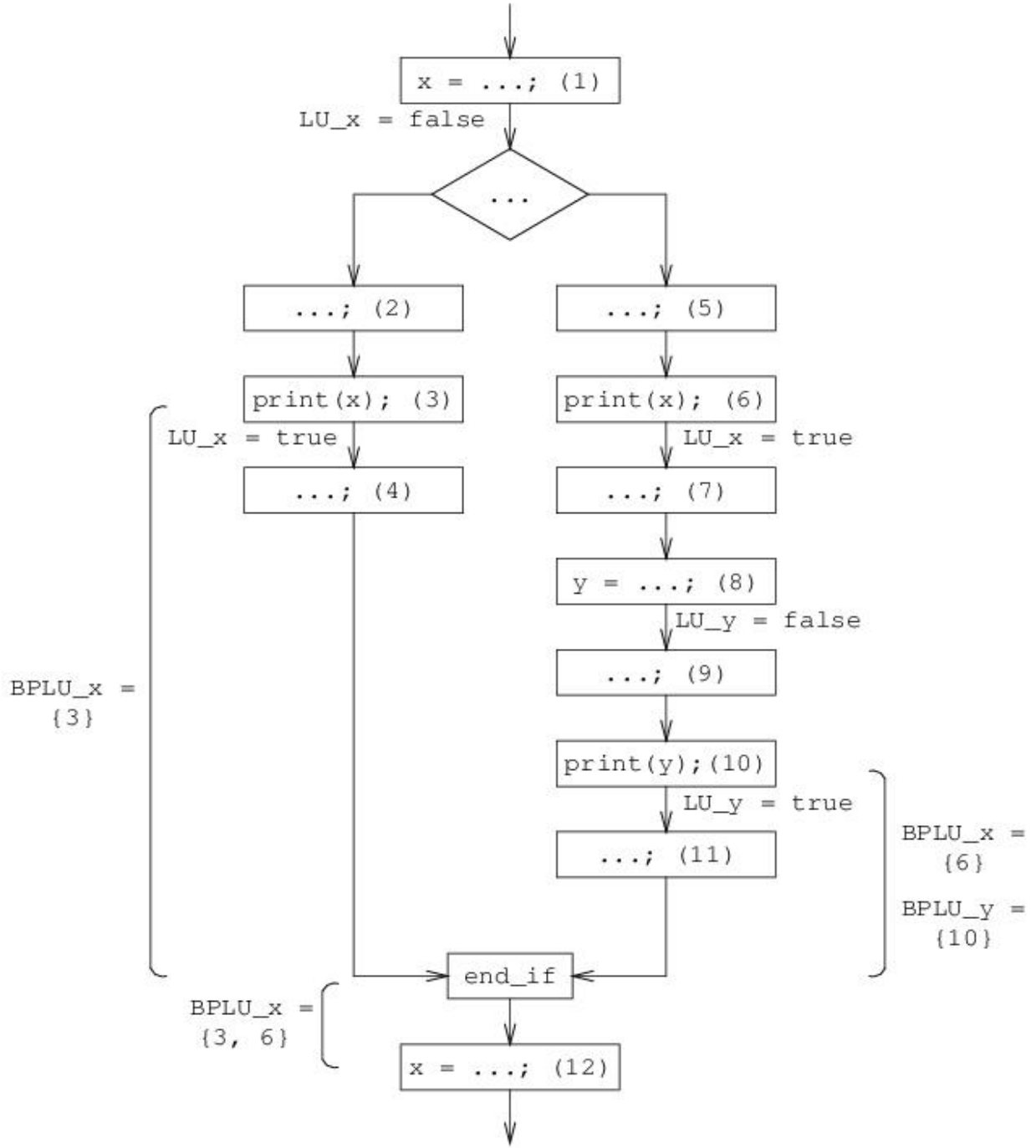


Figure 3.58 Live analysis for Figure 3.55, merging at the end-if node.

Liveness-Analyse / Datenflussgleichungen (1)

- Rückwärtsanalyse systematisch

$$OUT(N) = \bigcup_{M=\text{dynamic successor of } N} IN(M)$$

$$IN(N) = (OUT(N) \setminus KILL(N)) \cup GEN(N)$$

Figure 3.59 Backwards data-flow equations for a node *N*.

- Wertzuweisungen an Variable V
  - KILL={ “V ist lebendig hier” }, GEN={ }
- Verwendung der Variablen V
  - KILL={ } , GEN={ “V ist lebendig hier” }
- beides (Vereinigung)
  - KILL={ “V ist lebendig hier” }, GEN={ “V ist lebendig hier” }

Liveness-Analyse / Datenflussgleichungen (2)

- Bitmuster xy
  - `x=1` gdw. `x` ist lebendig
  - `y=1` gdw. `y` ist lebendig
- Merge ist Vereinigung
  - Lebendigkeit ist existenziell quantifiziert (es gibt einen Pfad)

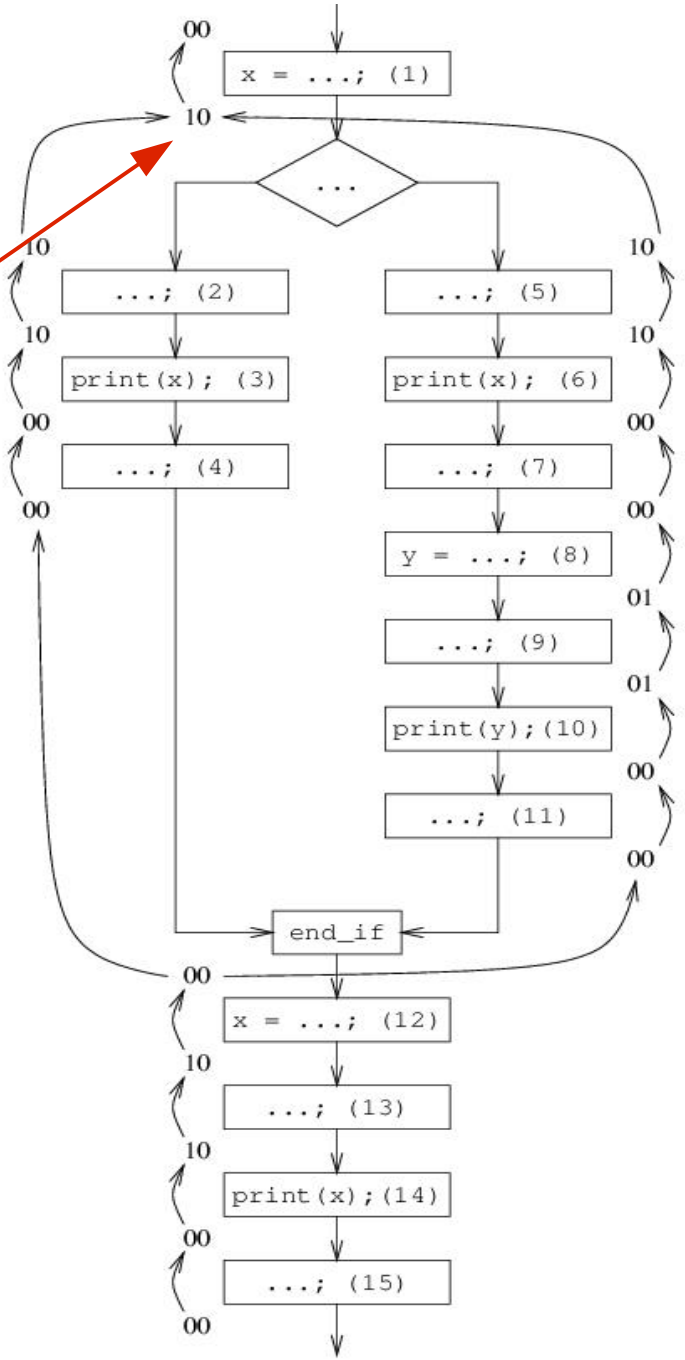


Figure 3.60 Live analysis for Figure 3.55 using backward data-flow equations.

Liveness-Analyse / Datenflussgleichungen (3)

- Interpretation der Ergebnisse
  - Knoten der letzten Verwendung von V
    - Liveness(V)=1 in der IN-Menge
    - Liveness(V)=0 in der OUT-Menge
  - Zuweisung an V und Liveness(V)=0 in IN und OUT
    - Zuweisung kann entfernt werden
    - rechte Seite kann entfernt werden, falls sie keine Seiteneffekte hat

Vergleich: Symbolische Interpretation und Datenflussgleichungen

- symbolische Interpretation
  - intuitiv
  - ideal für narrow Compiler
  - unterstützt Informationsfluss in Ausdrücken
- Datenflussgleichungen
  - für jede Art von Kontrollfluss geeignet
  - globale Information notwendig