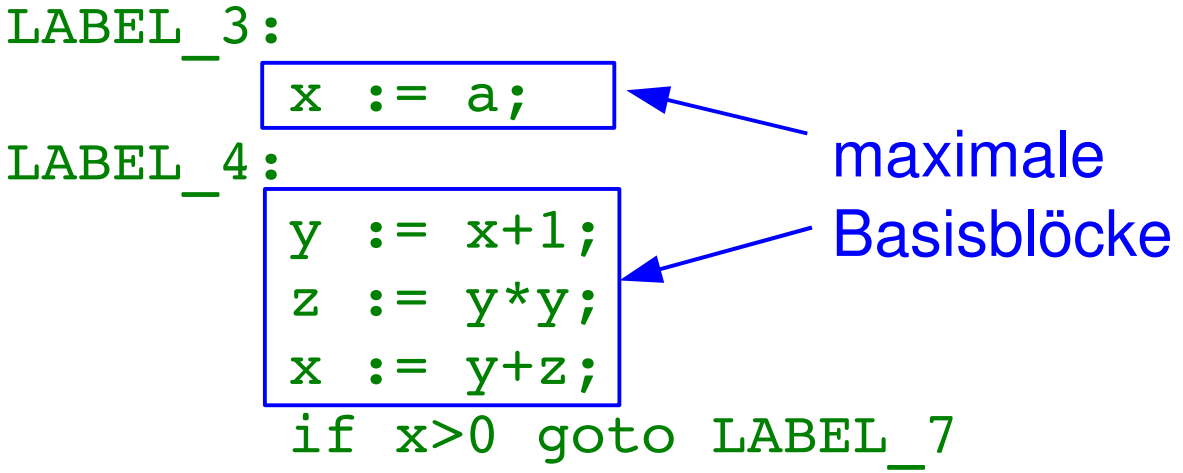
Codegenerierung für Basisblöcke (1)

- Basisblock (Grundblock):
  - Teil des Kontrollflussgraphen, der keine Verzweigungen (Sprünge) oder Vereinigungen (Label) enthält
  - keine Zyklen: bessere Optimierung möglich
  - einzige Bestandteile: Ausdrücke und Zuweisungen
- maximaler Basisblock:
  - Basisblock, der nicht (am Anfang/Ende) vergrößert werden kann



Codegenerierung für Basisblöcke (2)

- Informationen mittels Analyse
  - Eingangsvariablen (ausserhalb belegt, innerhalb verwendet)
    - berechnet durch Last-Def (Reaching-Definitions) Analyse
  - Ausgangsvariablen (innerhalb belegt, ausserhalb verwendet)
    - berechnet durch Lebendigkeits (Liveness) - Analyse
- Informationen durch Scope-Regeln:

```
{    int n;

     n = a + 1;
     x = b + n*n + c;
     n = n + 1;
     y = d * n;
}
```

`n` lokal, daher weder Eingangs- noch Ausgangsvariable

Eingangsvariablen: `a,b,c,d`

Ausgangsvariablen: `x,y`

Figure 4.41 Sample basic block in C.

Codegenerierung für Basisblöcke (3)

Zwei Phasen:

1. Erzeugung eines Datenflussgraphen
   - aus abstraktem Syntaxbaum
   - Kontrollflussgraph eines Basisblocks azyklisch, kann einfach implizit berücksichtigt werden
   - Semikolonknoten im AST: Codesequenz
   - Datenflussgraph ist ein gerichteter, kreisfreier Graph (DAG)
2. Umsetzung in eine Codesequenz
   - Elimination temporärer Variablen
   - Elimination gemeinsamer Teilausdrücke
   - Umwandlung von Pfaden im Datenflussgraph in Folgen von Assemblerbefehlen

Abstrakter Syntaxbaum (AST) für Basisblock

```
{    int n;

     n = a + 1;
     x = b + n*n + c;
     n = n + 1;
     y = d * n;
}
```

Figure 4.41 Sample basic block in C.

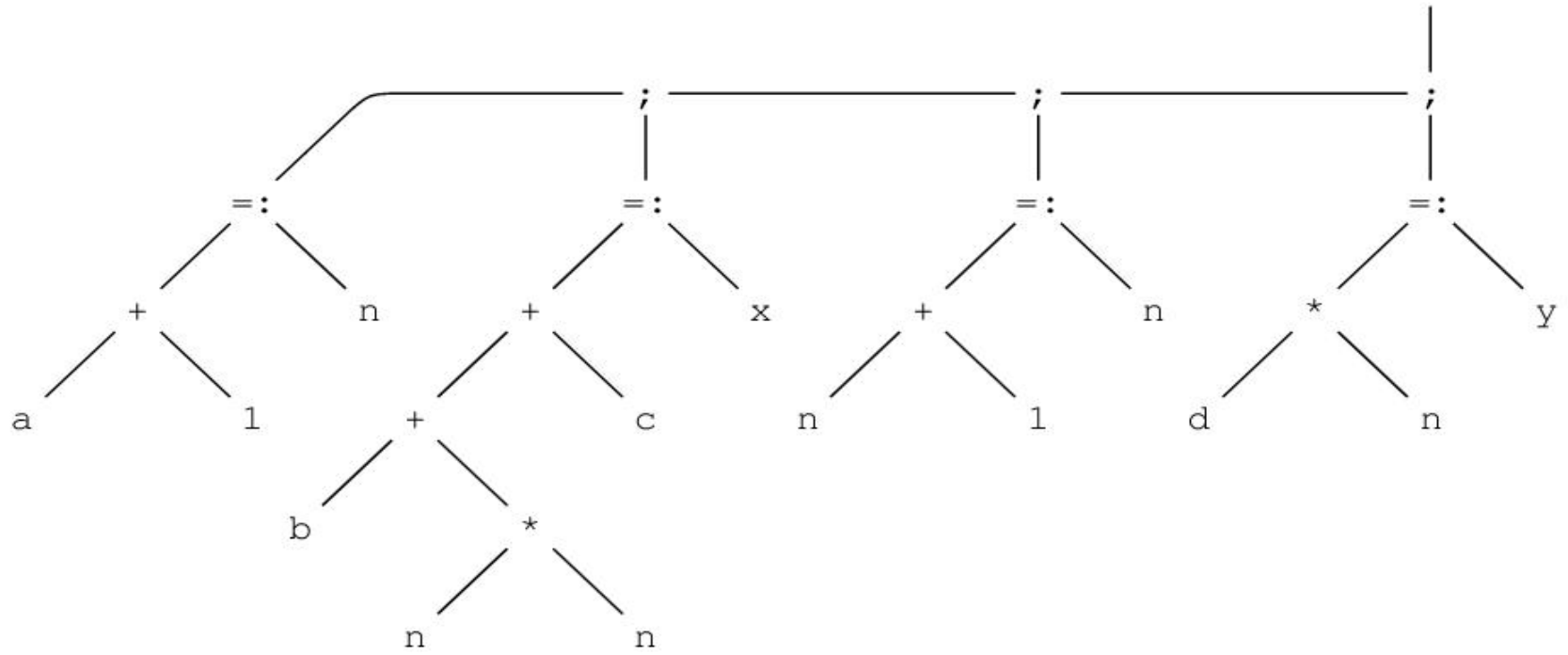


Figure 4.42 AST of the sample basic block.

Typen von Datenabhängigkeiten

Nutzungen und Zuweisungen beziehen sich auf eine feste Variable

- True-Dependence: Zuweisung → nachfolgende Nutzung
  - wichtigster Abhängigkeitstyp: *muss* berücksichtigt werden
  - rein-funktionale Programme haben nur diesen Typ
- Anti-Dependence: Nutzung → nächste Zuweisung
  - Constraint zur Beschreibung des Wertwechsels einer Variablen
  - imperative Programme: Einschränkung der Statementreihenfolge
- Output-Dependence: Zuweisung → nächste Zuweisung
  - bestimmt, welcher Wert am Ende in einer Variablen steht
- Input-Dependence: Nutzung → nächste Nutzung
  - Umkehrung der Richtung ändert Programmsemantik nicht
  - modelliert Unmöglichkeit zeitgleichen Lesens (Parallelität)

Behandlung von Datenabhängigkeiten

- Vorteil hier: Einschränkung auf Basisblock
  - Abhängigkeitsgraph leicht abzuleiten, da Kontrollfluss azyklisch
- Gewinnung aller Datenabhängigkeiten aus dem Programm
  - in Ausdrücken: (True-Abh.)
    - von einem Zuweisungsoperator zur Zielvariablen
    - von Operanden zu Operator
  - von der Definition einer Variablen zu seiner Nutzung (True-Abh.)
  - von einer Nutzung zum nächsten Vorkommen als Zuweisungsziel (Anti-Abh., verbietet Vertauschung)
  - von einer Zuweisung einer Variablen zur nächsten (Output-Abh.)
  - Behandlung von Aliasing (Zeiger) später
- Auswertungsreihenfolge beliebig, falls keine Abhängigkeit existiert:
  - insbesondere: Zuweisungen an verschiedene Variablen
- Abhängigkeitsgraph: Kanten gerichtet von Wirkung zu Ursache

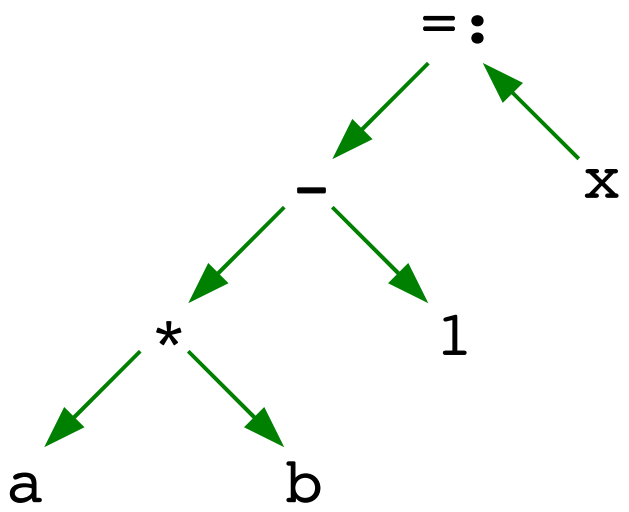
Vom AST zum Datenabhängigkeitsgraphen (1)

- Algorithmus: (vgl. Abb. 4.43)
  - AST-Kanten werden gerichtet (True-Abh.)
    - von einer Zielvariablen zu ihrem Zuweisungsoperator
    - von einem Operator zu seinen Operanden
  - zusätzliche Kanten: von der Nutzung einer Variablen (True-Abh.)
    - zu ihrem letzten Vorkommen als Zuweisungsziel im Block, oder
    - zum Blockanfang (wenn keine vorherige Zuweisung im Block)
  - weitere Kanten:
    - von der Zuweisung einer Variablen auf die vorhergehende Zuweisung derselben Variablen (Output-Abh.)
    - von der Zielvariablen einer Zuweisung zu ihren vorherigen Zuweisung auf dieselbe Variable (Anti-Abh.)
  - markiere Knoten, die Ausgabewerte repräsentieren, als “Wurzel”
  - entferne “`;`”-Knoten und ihre Kanten (Kontrollfluss irrelevant)

Vom AST zum Datenabhängigkeitsgraphen (2)

True-Abhängigkeiten in Ausdrücken

Bsp.: `x := (a*b)-1`



Vom AST zum Datenabhängigkeitsgraphen (3)

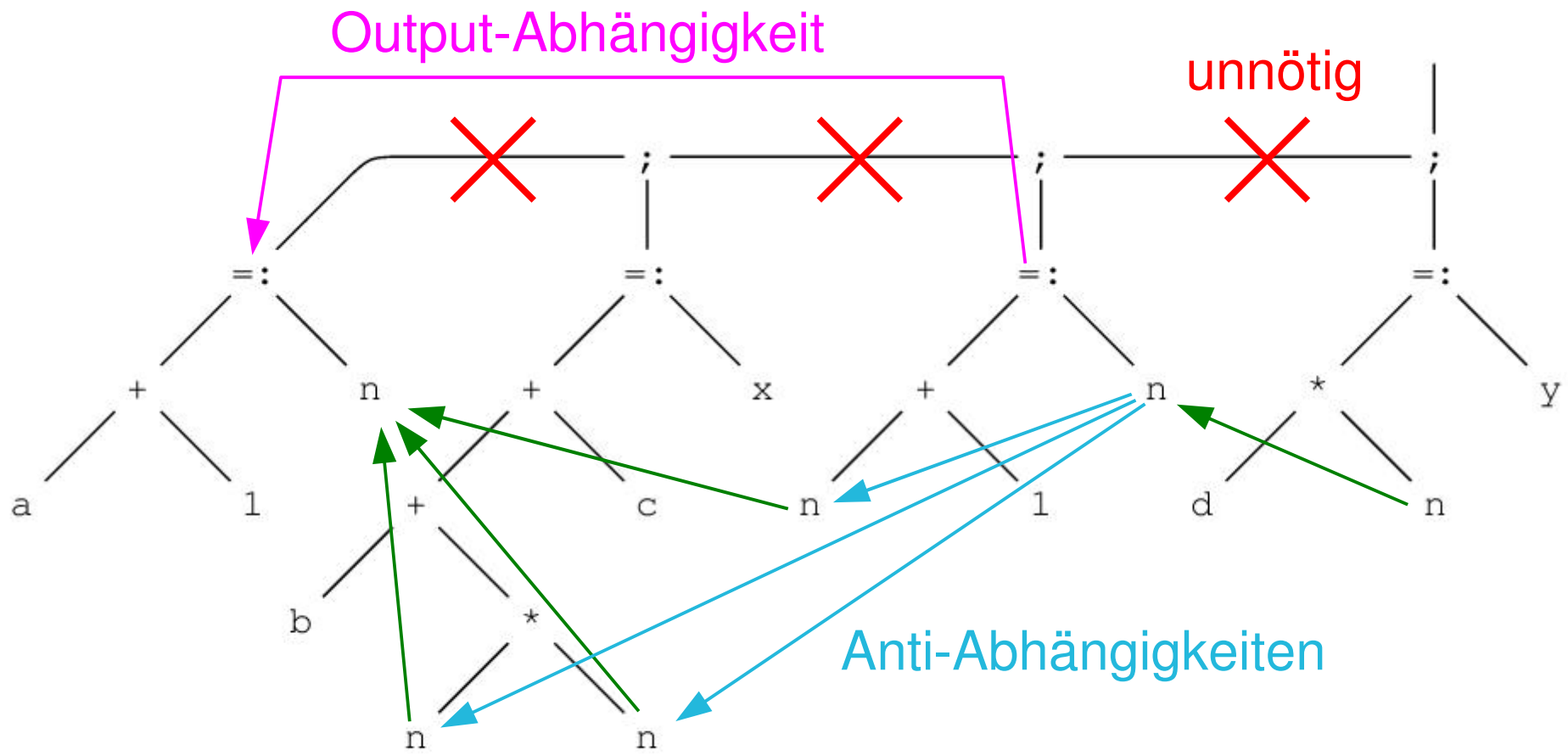


Figure 4.42 AST of the sample basic block.

Vom AST zum Datenabhängigkeitsgraphen (4)

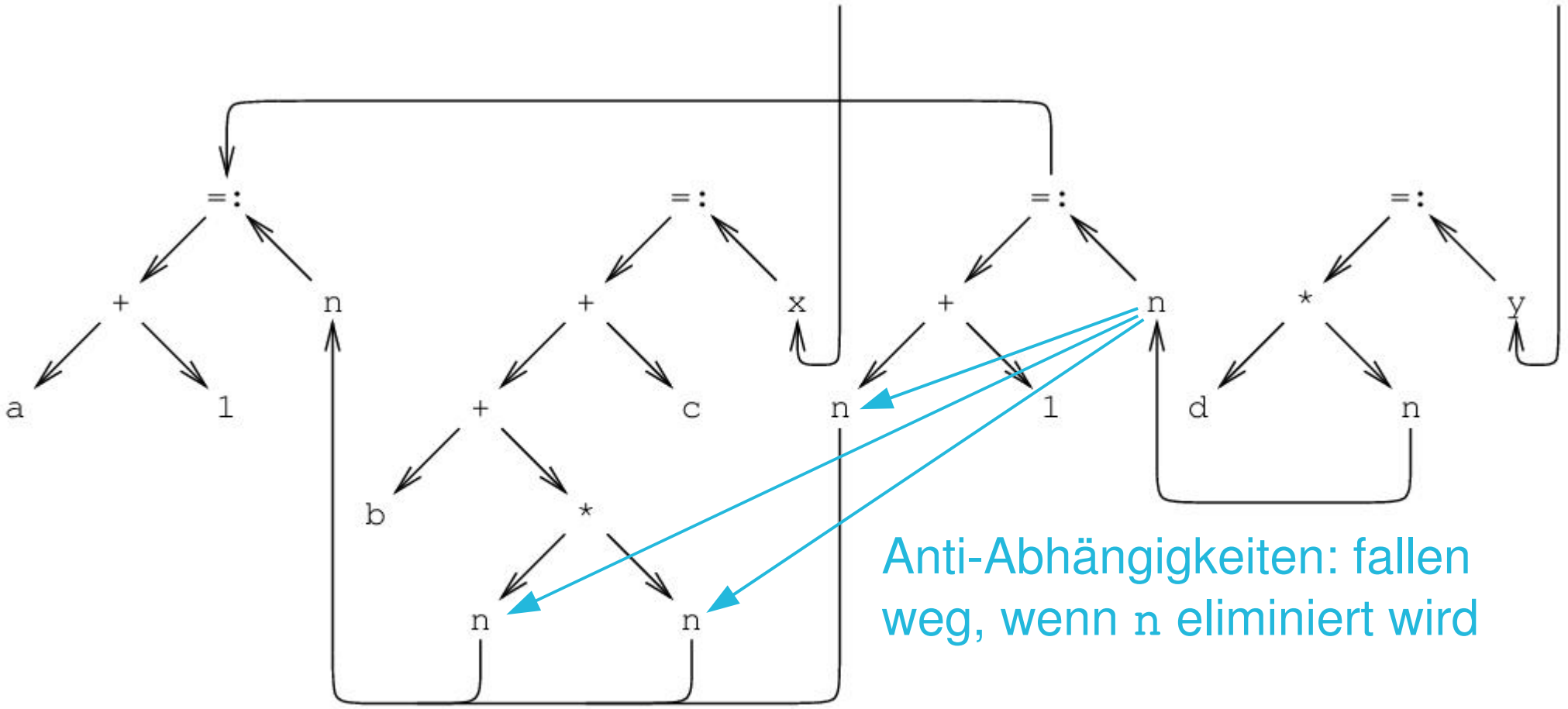


Figure 4.43 Data dependency graph for the sample basic block.

Elimination temporärer Variablen (hier: `n`)

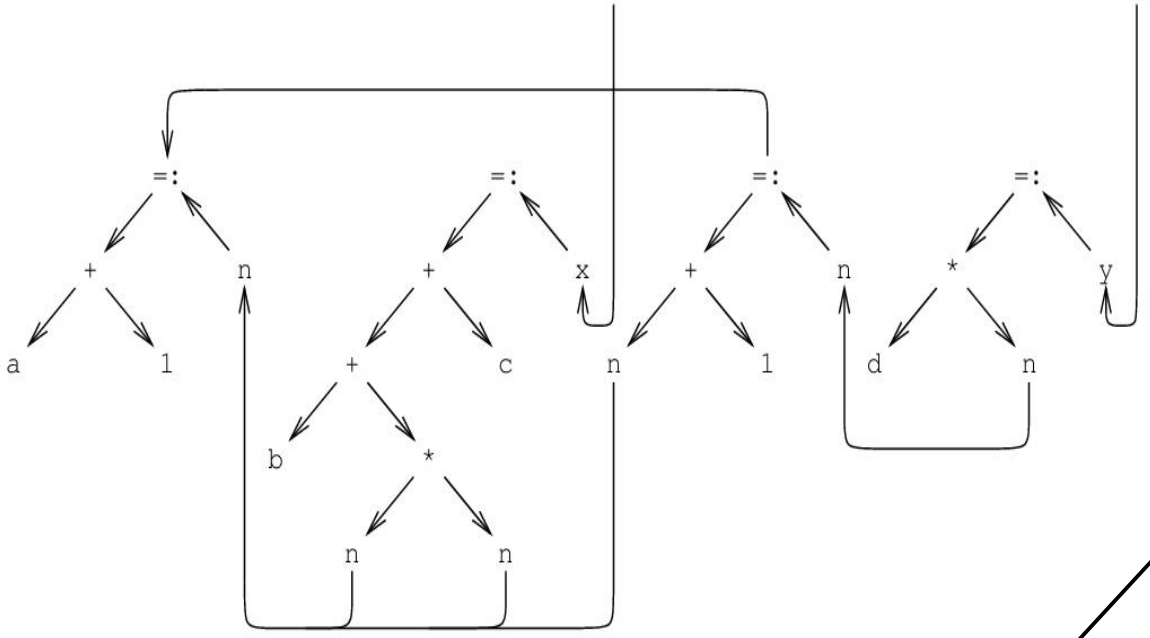


Figure 4.43 Data dependency graph for the sample basic block.

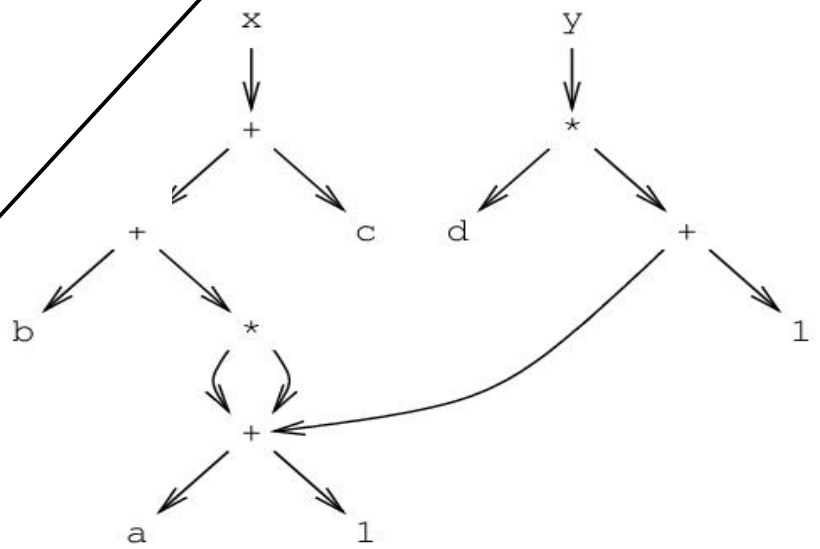


Figure 4.44 Cleaned-up data dependency graph for the sample basic block.

Gemeinsame Teilausdrücke

```
{    int n;

     n = a + 1;
     x = b + n*n + c;      /* subexpression n*n ... */
     n = n*n + 1;          /* ... in common */
     y = d * n;
}
```

Figure 4.45 Basic block in C with common subexpression.

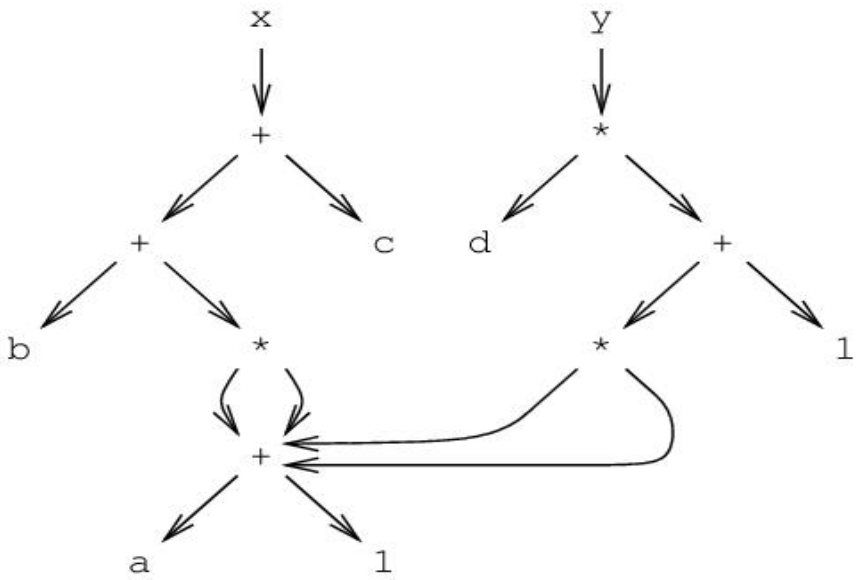


Figure 4.46 Data dependency graph with common subexpression.

“Elimination” gemeinsamer Teilausdrücke

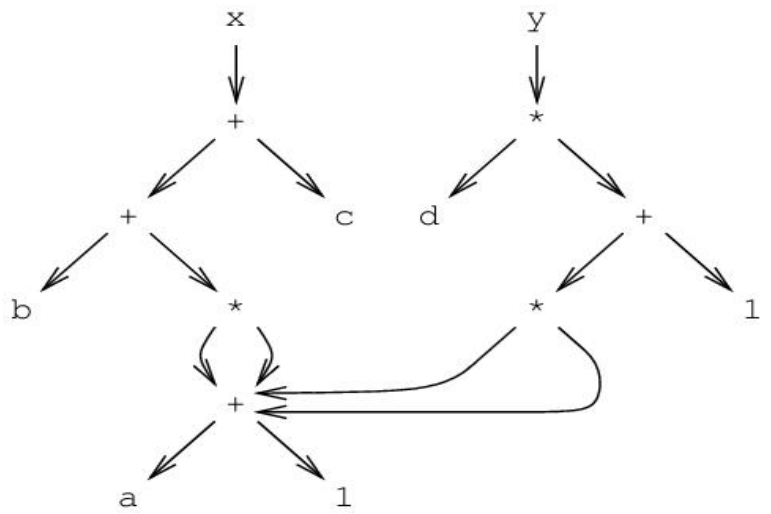


Figure 4.46 Data dependency graph with common subexpression.

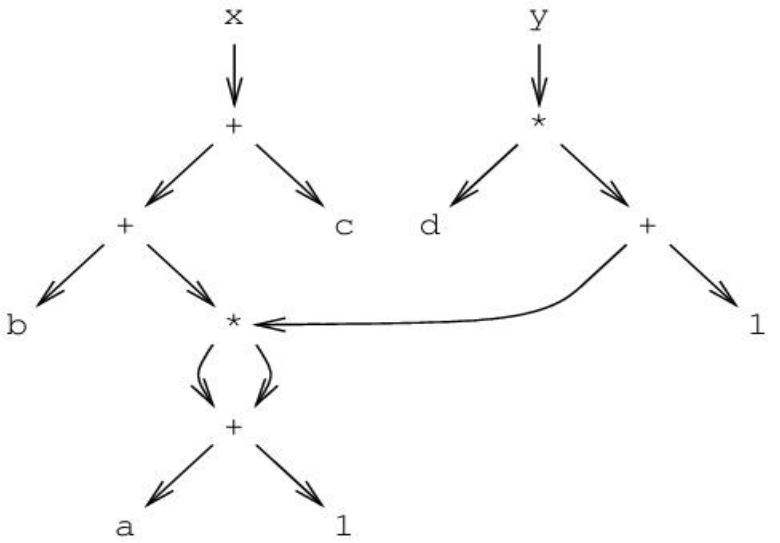


Figure 4.47 Cleaned-up data dependency graph with common subexpression eliminated.

Umsetzung in eine Codesequenz (1)

beachte: Tripeldarstellung von Datenabhängigkeitsgraphen

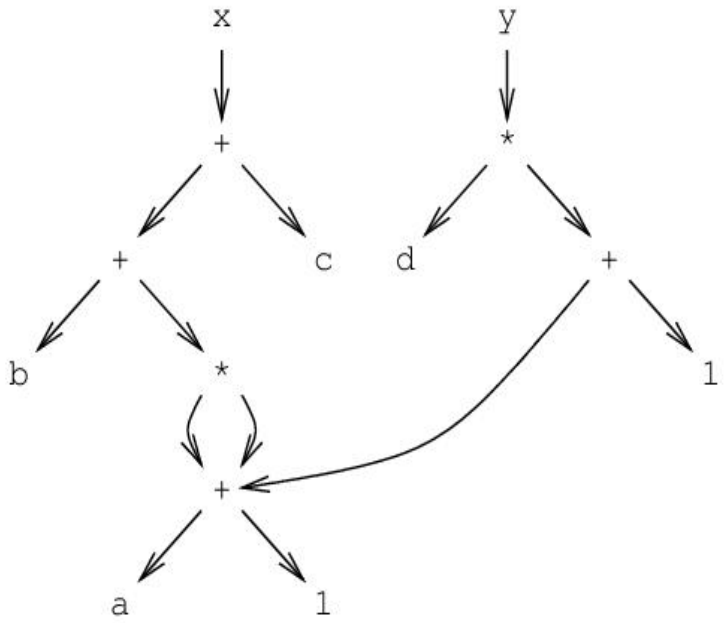


Figure 4.44 Cleaned-up data dependency graph for the sample basic block.

position	triple
1	a + 1
2	@1 * @1
3	b + @2
4	@3 + c
5	@4 =: x
6	@1 + 1
7	d * @6
8	@7 =: y

Figure 4.48 The data dependency graph of Figure 4.44 as an array of triples.

Umsetzung in eine Codesequenz (2)

- Topologische Sortierung, mögliche Scheduling-Strategien:
  - früh: Auswertung zugelassen, sobald Operanden verfügbar
  - spät: Auswertung erst dann, wenn gebraucht (spart Register)
- Leitersequenzen: spezielle Teilgraphen des Datenflussgraphen
  - Sinn: Ausnutzung von Register/Speicher-Befehlen
    - Bsp.: Code für `(((a+b)*c)-d)`

```
Load_Mem   a, R1
Add_Mem    b, R1
Mult_Mem   c, R1
Subtr_Mem  d, R1
```

  - Definition:
    - jede Wurzel (Output-Variable) ist eine Leitersequenz
    - sei S Leitersequenz, deren Endknoten ein Operator N ist. S erweitert um den linken Operanden von N ist eine Leitersequenz
    - falls N kommutativ: statt linkem auch rechter Operand möglich

Leitersequenzen (1)

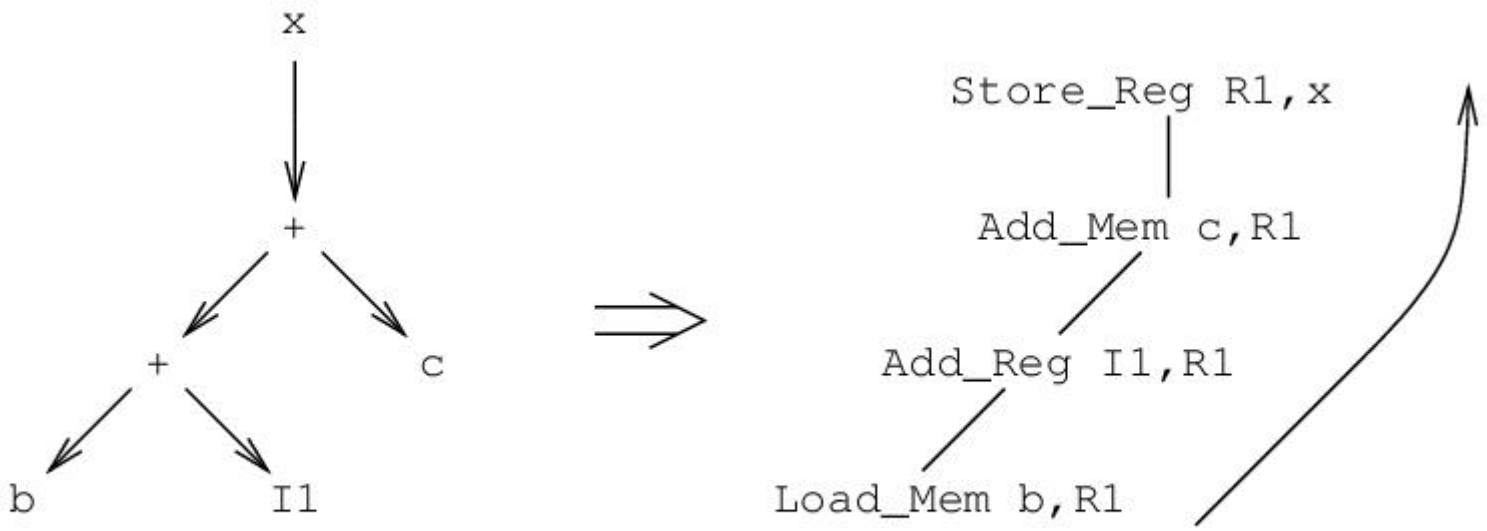


Figure 4.49 Rewriting and ordering a ladder sequence.

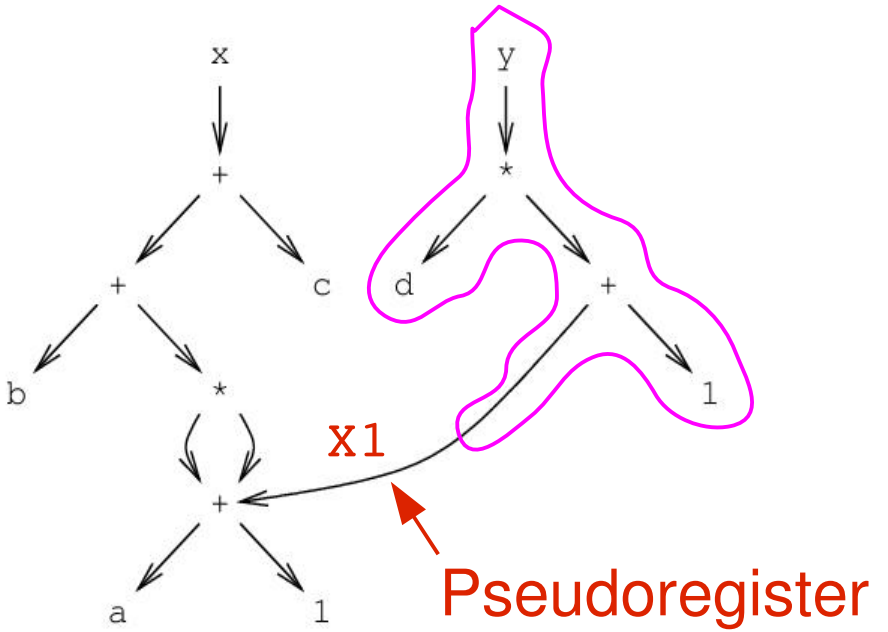
Leitersequenzen (2)

- linearisiert die Instruktionen in umgekehrter Reihenfolge
  - benutzt beliebig viele Temporaries (Speicher oder Register)
  - braucht ein eigenes Register (im folgenden R1 genannt)
- Heuristik / Schrittfolge

  wiederhole folgende Schritte bis Graph leer:

  1. finde Leitersequenz S, deren Knoten jeweils höchstens eine eingehende Datenabhängigkeit haben
  2. für jeden Operanden M, der weder Blatt ist noch Element von S
     (a) weise M ein neues Temporary zu (falls M noch keines hat)
     (b) mache M zu einer neuen Wurzel (einer Leitersequenz)
  3. generiere Code für S
  4. entferne S aus dem Datenabhängigkeitsgraphen

Leitersequenzen (3a)




```
Load_Reg    X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg   R1,y
```

Figure 4.50 Cleaned-up data dependency graph for the sample basic block.

Leitersequenzen (3b)

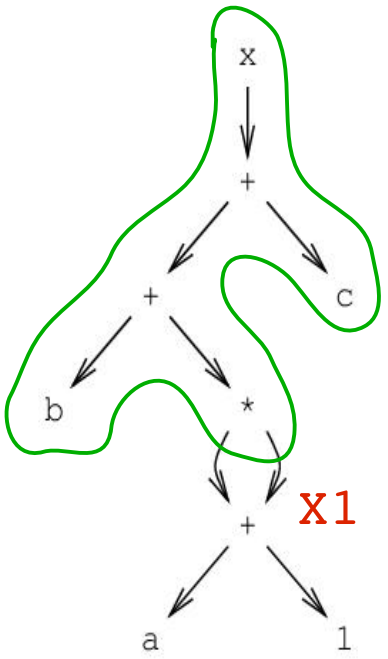


Figure 4.51 Data dependency graph after removal of the first ladder sequence.

```
Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg   R1,y
```

Leitersequenzen (3c)

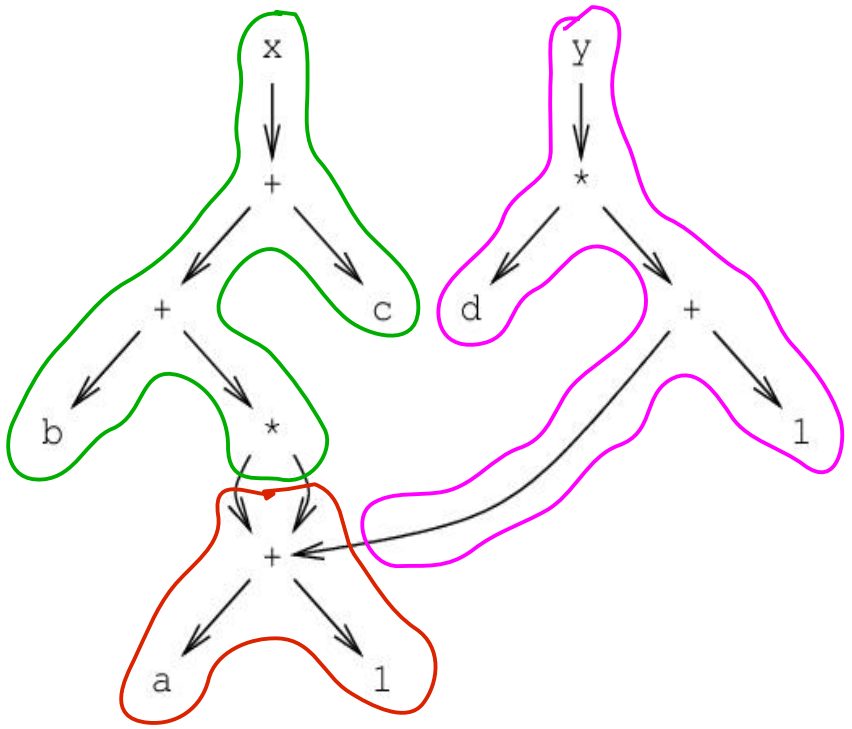


Figure 4.50 Cleaned-up data dependency graph for the sample basic block.

```
Load_Mem     a,R1
Add_Const    1,R1
Load_Reg    R1,X1

Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem      b,R1
Add_Mem      c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const    1,R1
Mult_Mem     d,R1
Store_Reg   R1,y
```

Registerzuweisung

X1: Temporary → R2: Register

```
Load_Mem     a,R1          Load_Mem     a,R1
Add_Const    1,R1          Add_Const    1,R1
Load_Reg    R1,X1          Load_Reg    R1,R2
Load_Reg    X1,R1          Load_Reg    R2,R1
Mult_Reg    X1,R1          Mult_Reg    R2,R1
Add_Mem      b,R1          Add_Mem      b,R1
Add_Mem      c,R1          Add_Mem      c,R1
Store_Reg   R1,x           Store_Reg   R1,x
Load_Reg    X1,R1          Load_Reg    R2,R1
Add_Const    1,R1          Add_Const    1,R1
Mult_Mem     d,R1          Mult_Mem     d,R1
Store_Reg   R1,y           Store_Reg   R1,y
```

Optimierte Registerzuweisung

Wegfall redundanter Befehle und Registervertauschung

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2
Load_Reg    R2,R1
Mult_Reg    R2,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x
Load_Reg    R2,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2

Mult_Reg    R1,R2
Add_Mem     b,R2
Add_Mem     c,R2
Store_Reg   R2,x

Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Methoden: Register-Tracking oder Peephole-Optimization

Gemeinsame Teilausdrücken und Aliasing (1)

```
 a = x * y;
*p = 3;
 b = x * y;
```

keine gemeinsamen Teilausdrücke,
`p` kann auf `x` oder `y` zeigen

```
 a = *p * q;
 b = 3;
 c = *p * q;
```

keine gemeinsamen Teilausdrücke,
`b` kann Alias für `*p` sein

Gemeinsame Teilausdrücken und Aliasing (2)

```
 n = a + 1;
*x = b + n*n + c;
 n = n + 1;
 y = d * n
```

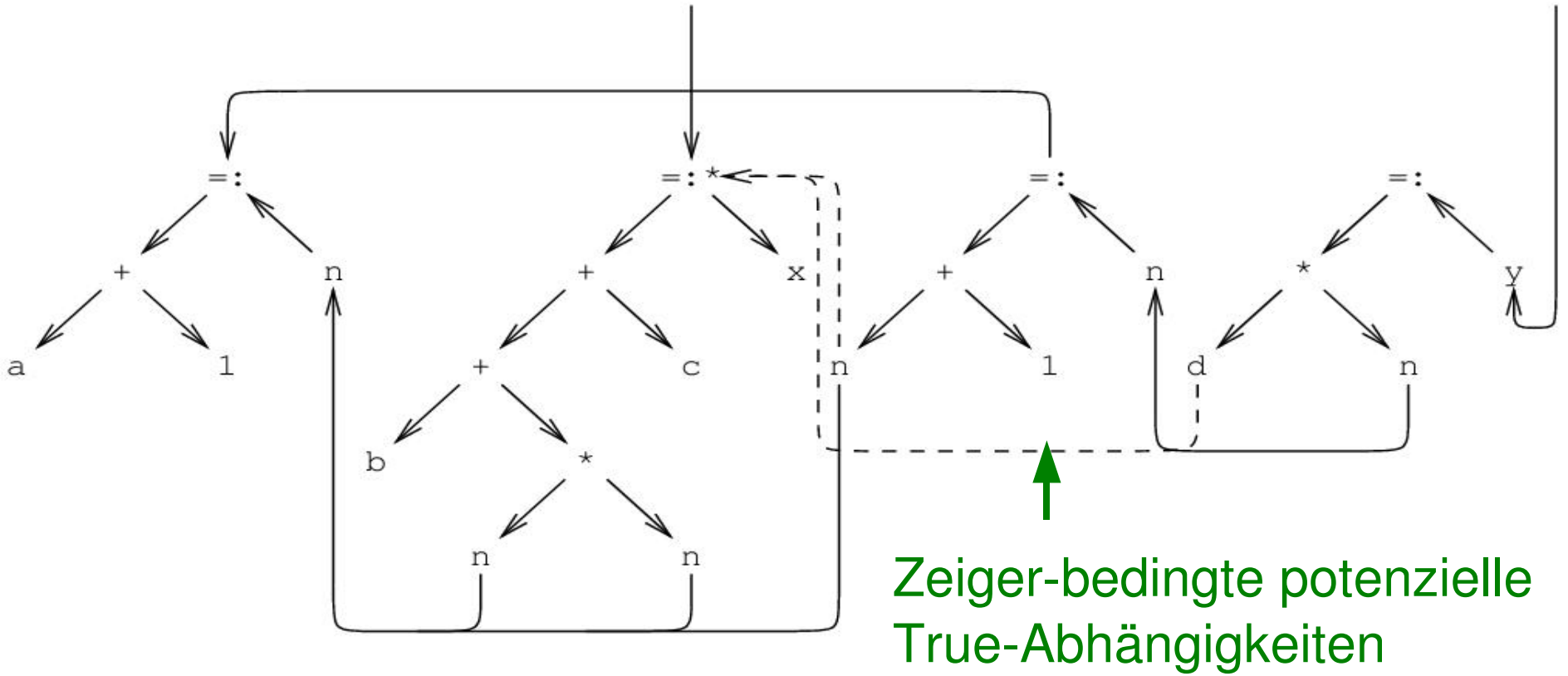


Figure 4.57 Data dependency graph with an assignment under a pointer.

Gemeinsame Teilausdrücken und Aliasing (3)

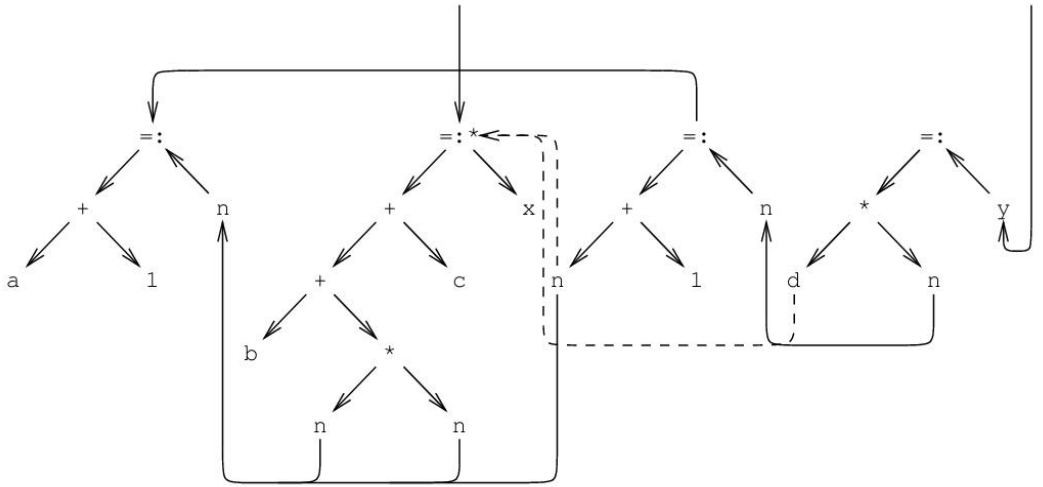


Figure 4.57 Data dependency graph with an assignment under a pointer.

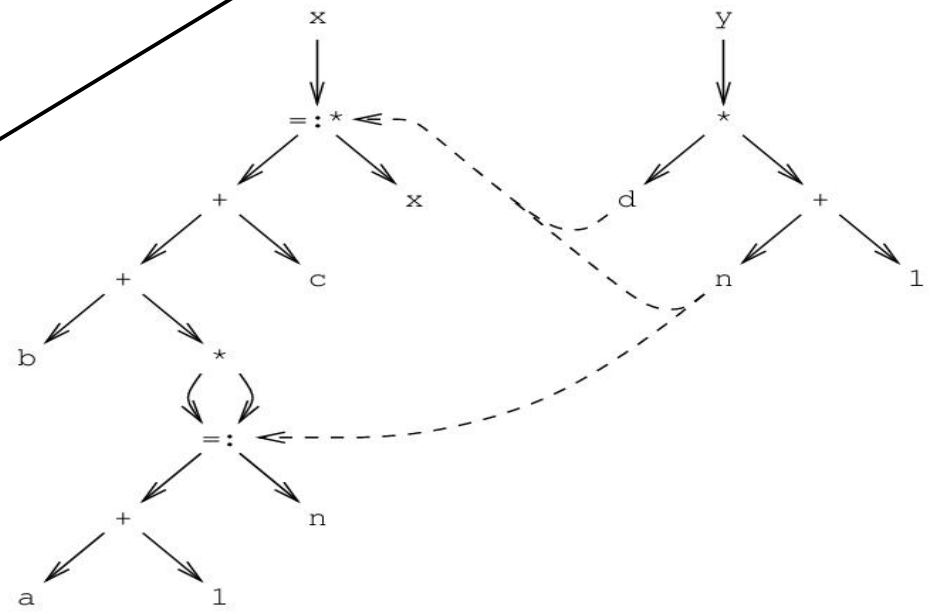


Figure 4.58 Cleaned-up data dependency graph with an assignment under a pointer.

Gemeinsame Teilausdrücken und Aliasing (4)

beachte: (i) Pfeilrichtung bei "=:*", (ii) Pseudo-Maschinenbefehl

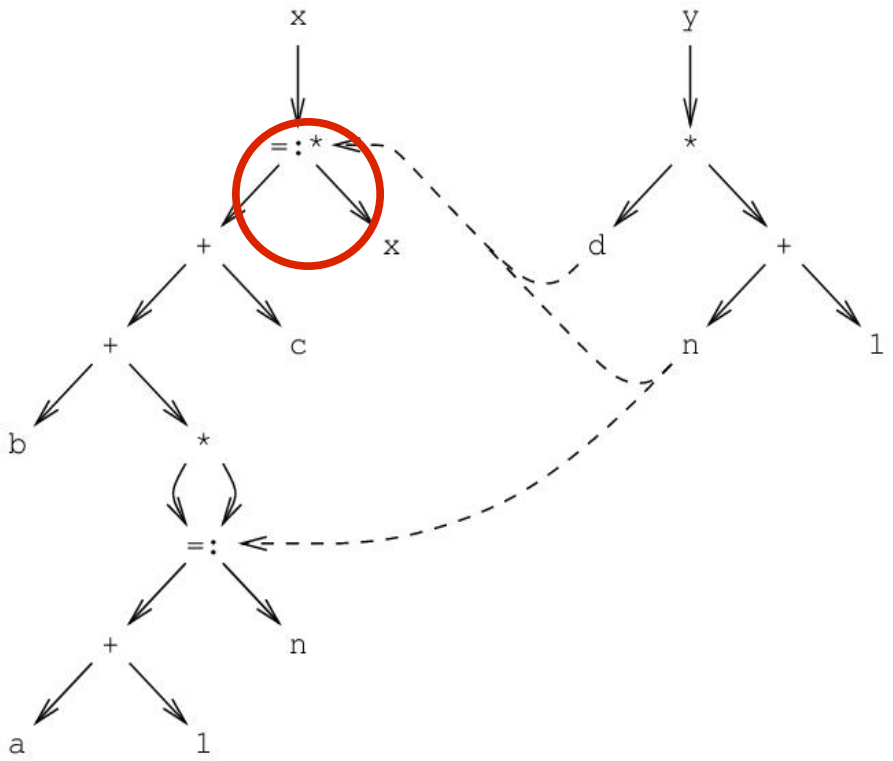


Figure 4.58 Cleaned-up data dependency graph with an assignment under a pointer.

```
Load_Mem              a,R1
Add_Const             1,R1
Store_Reg             R1,n

Load_Mem              n,R1
Mult_Mem              n,R1
Add_Mem               b,R1
Add_Mem               c,R1
Store_Indirect_Mem    R1,x

Load_Mem              n,R1
Add_Const             1,R1
Mult_Mem              d,R1
Store_Reg             R1,y
```

Figure 4.59 Target code generated for the basic block of Figure 4.56.