Kontrollabhängigkeit VI

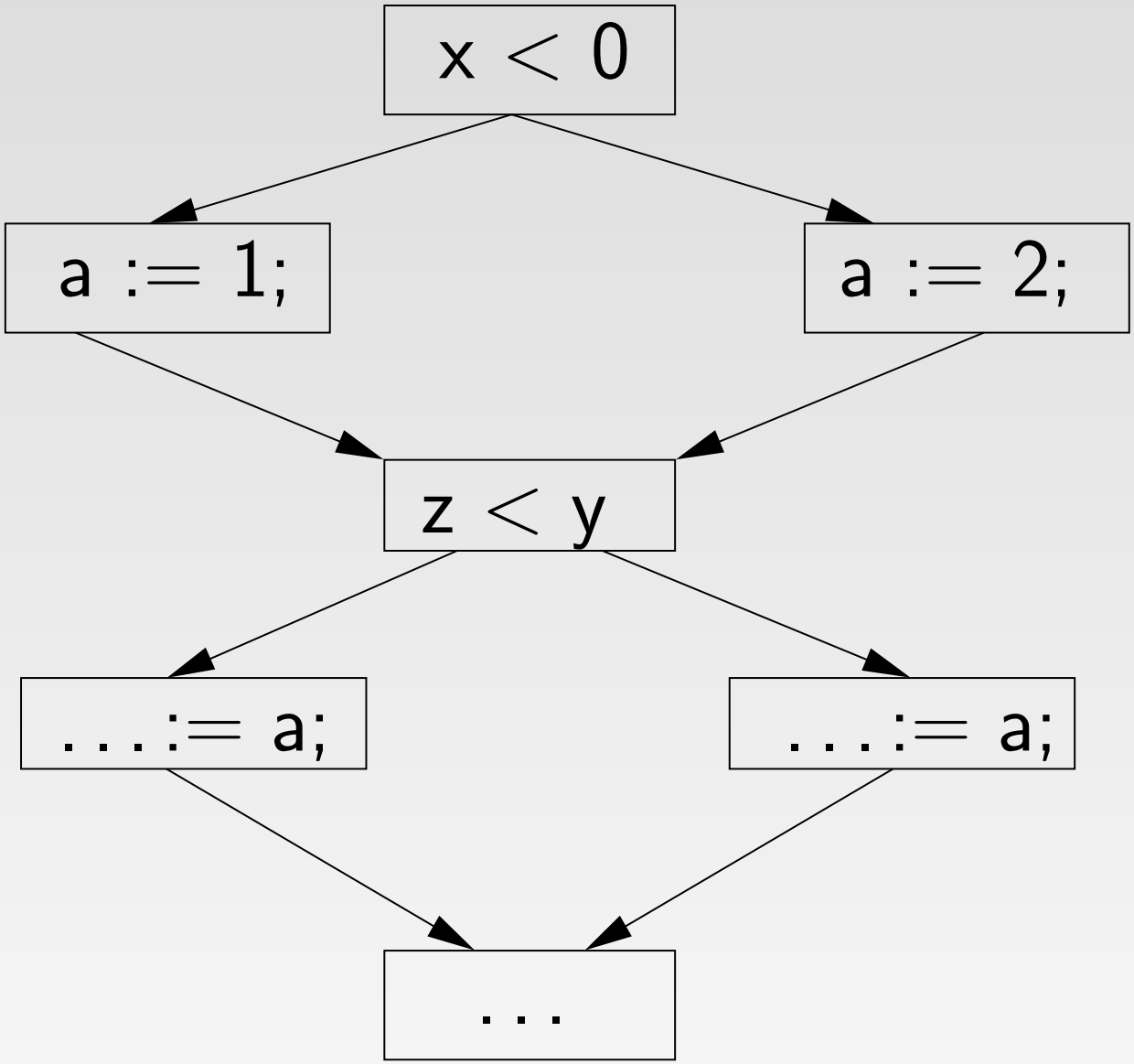
Für strukturierte Programme:

- eine Anweisung ist kontrollabhängig von der Bedingung der nächstumgebenden Schleife oder bedingten Anweisung.

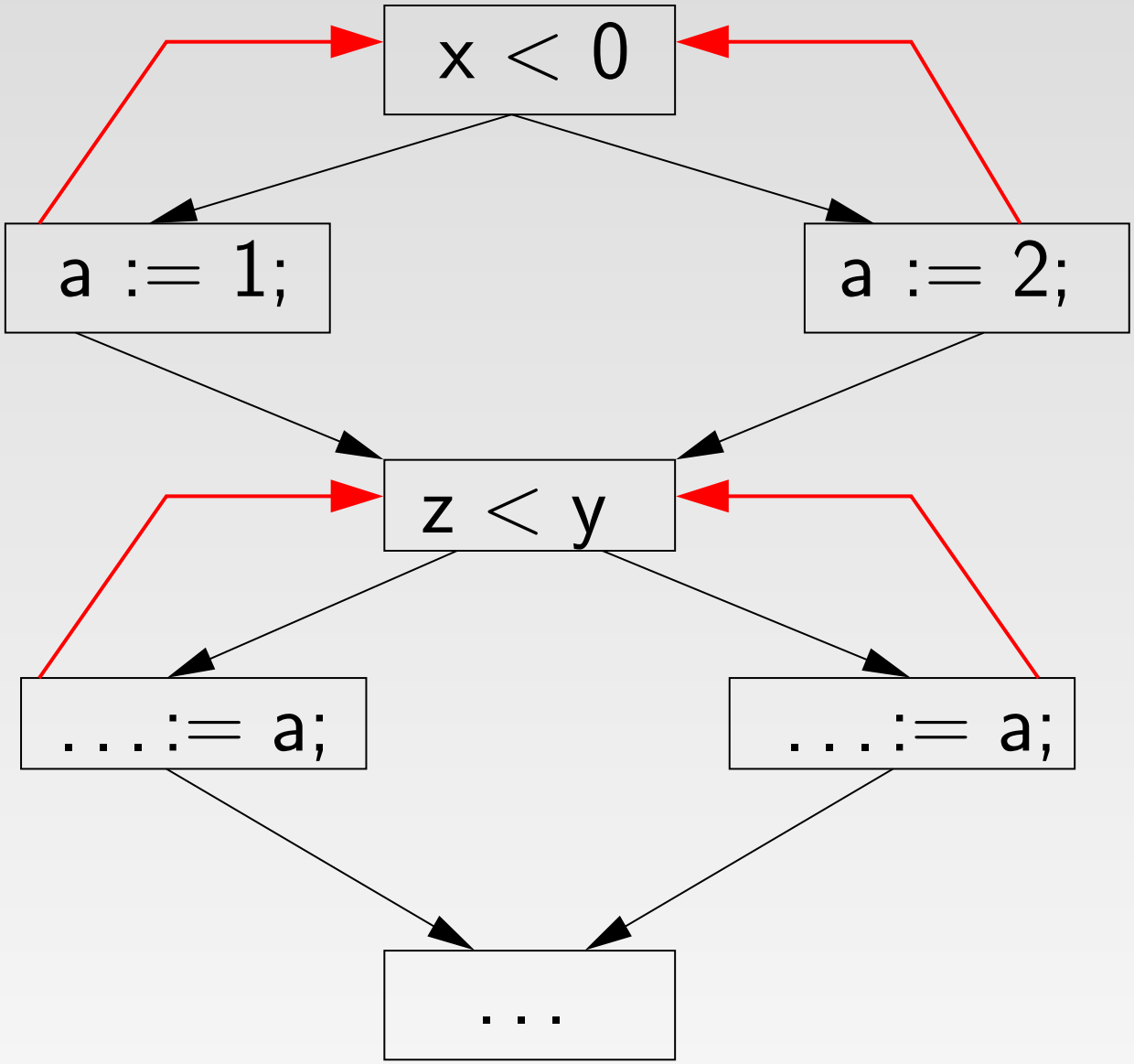
```
while a loop
  if b then
     x := y;   -- kontrollabhaengig von b
  end if;
  z := 1;      -- kontrollabhaengig von a
end loop;
```

N.B.: Bedingungen in repeat-Schleifen sind von sich und dem umgebenden Konstrukt abhängig (siehe voriges Beispiel).

Repräsentation der Kontrollabhängigkeit



Repräsentation der Kontrollabhängigkeit



Datenabhängigkeitsanalyse

- Set = Setzen eines Wertes
- Use = Verwendung eines Wertes

Datenabhängigkeitsanalyse

- Set = Setzen eines Wertes
- Use = Verwendung eines Wertes

Daraus ergeben sich folgende relevanten Beziehungen zwischen zwei Anweisungen (resp. Knoten) A und B:

- Set-Use Beziehung (Datenabhängigkeit)
  A setzt den Wert, der von B verwendet wird

Datenabhängigkeitsanalyse

- Set = Setzen eines Wertes
- Use = Verwendung eines Wertes

Daraus ergeben sich folgende relevanten Beziehungen zwischen zwei Anweisungen (resp. Knoten) A und B:

- Set-Use Beziehung (Datenabhängigkeit)
  A setzt den Wert, der von B verwendet wird
- Use-Set Beziehung (Anti-Dependency)
  A liest den Wert und B überschreibt ihn danach

Datenabhängigkeitsanalyse

- Set = Setzen eines Wertes
- Use = Verwendung eines Wertes

Daraus ergeben sich folgende relevanten Beziehungen zwischen zwei Anweisungen (resp. Knoten) A und B:

- Set-Use Beziehung (Datenabhängigkeit)
  A setzt den Wert, der von B verwendet wird
- Use-Set Beziehung (Anti-Dependency)
  A liest den Wert und B überschreibt ihn danach
- Set-Set Beziehung (Output-Dependency)
  der von A gesetzte Wert wird von B überschrieben

Datenabhängigkeitsanalyse

- Set = Setzen eines Wertes
- Use = Verwendung eines Wertes

Daraus ergeben sich folgende relevanten Beziehungen zwischen zwei Anweisungen (resp. Knoten) A und B:

- Set-Use Beziehung (Datenabhängigkeit)
  A setzt den Wert, der von B verwendet wird
- Use-Set Beziehung (Anti-Dependency)
  A liest den Wert und B überschreibt ihn danach
- Set-Set Beziehung (Output-Dependency)
  der von A gesetzte Wert wird von B überschrieben

Codetransformationen müssen diese Beziehungen erhalten.

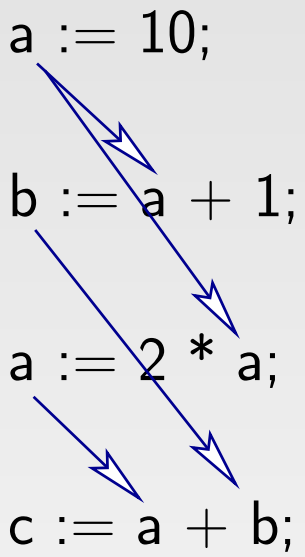
Datenabhängigkeit (Set-Use)

Für die Zwecke der Analyse ist die Set-Use Beziehung die wichtigste Beziehung und wird intern oft explizit repräsentiert (meist in der umgekehrten Richtung).

```
a := 10;

b := a + 1;

a := 2 * a;

c := a + b;
```

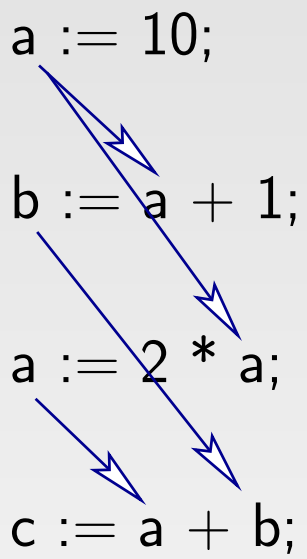
Datenabhängigkeit (Set-Use)

Für die Zwecke der Analyse ist die Set-Use Beziehung die wichtigste Beziehung und wird intern oft explizit repräsentiert (meist in der umgekehrten Richtung).



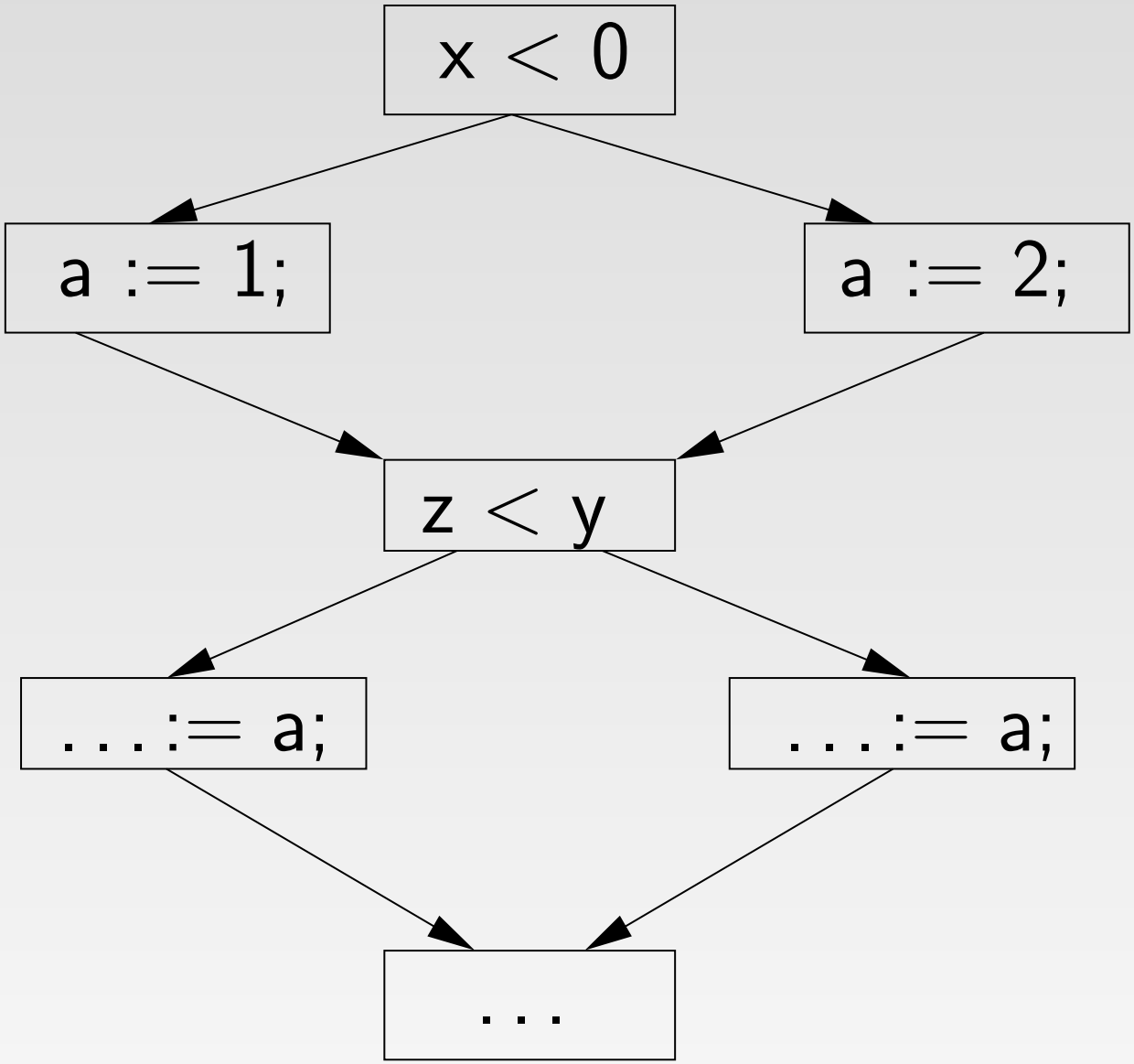
Datenabhängigkeit (Set-Use)

Für die Zwecke der Analyse ist die Set-Use Beziehung die wichtigste Beziehung und wird intern oft explizit repräsentiert (meist in der umgekehrten Richtung).

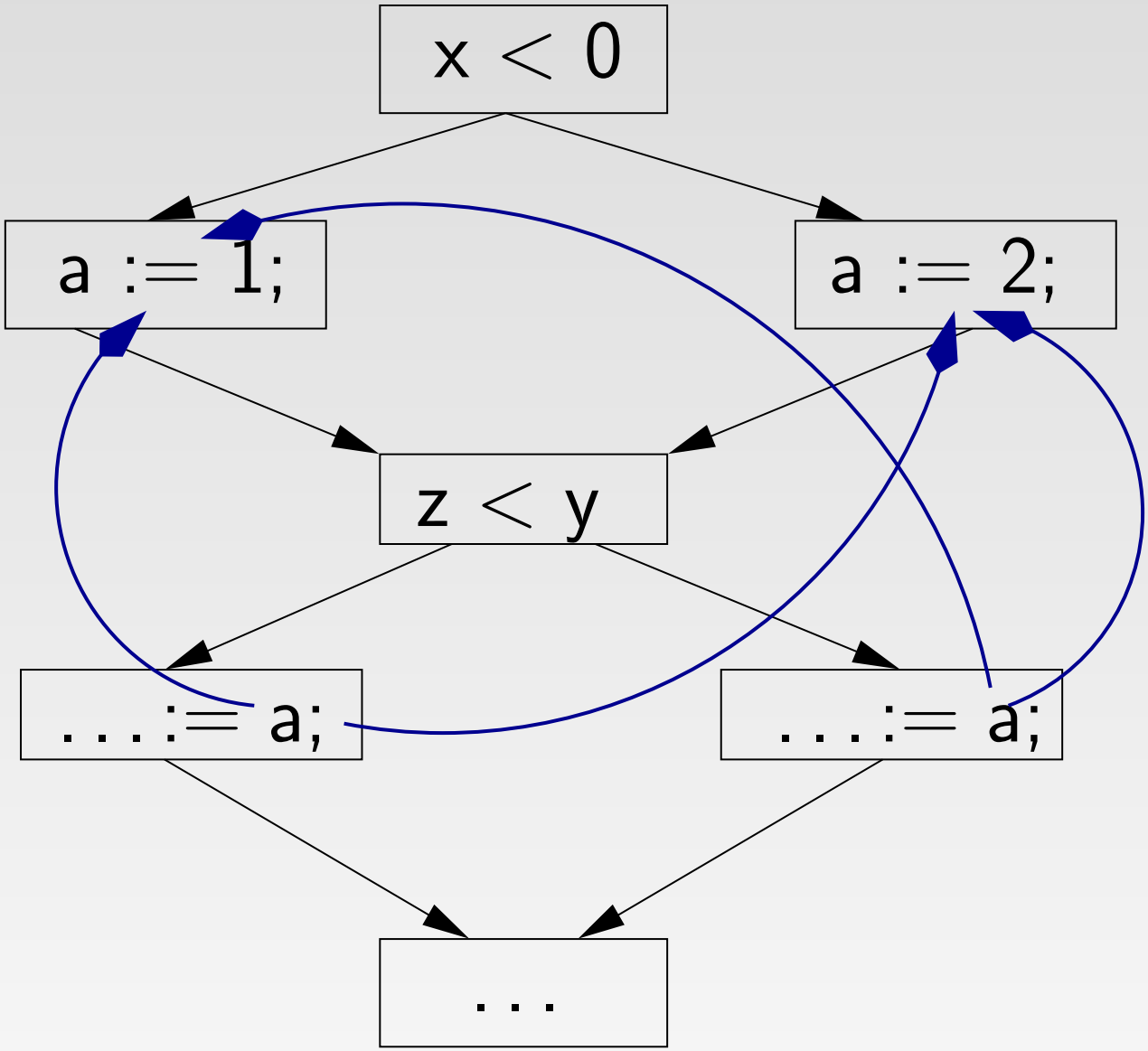


Zwischen der 2. und 3. Anweisung besteht eine "'Use-Set"', zwischen der 1. und 3. Anweisung eine "'Set-Set"'-Beziehung.

Repräsentation der Datenabhängigkeit



Repräsentation der Datenabhängigkeit

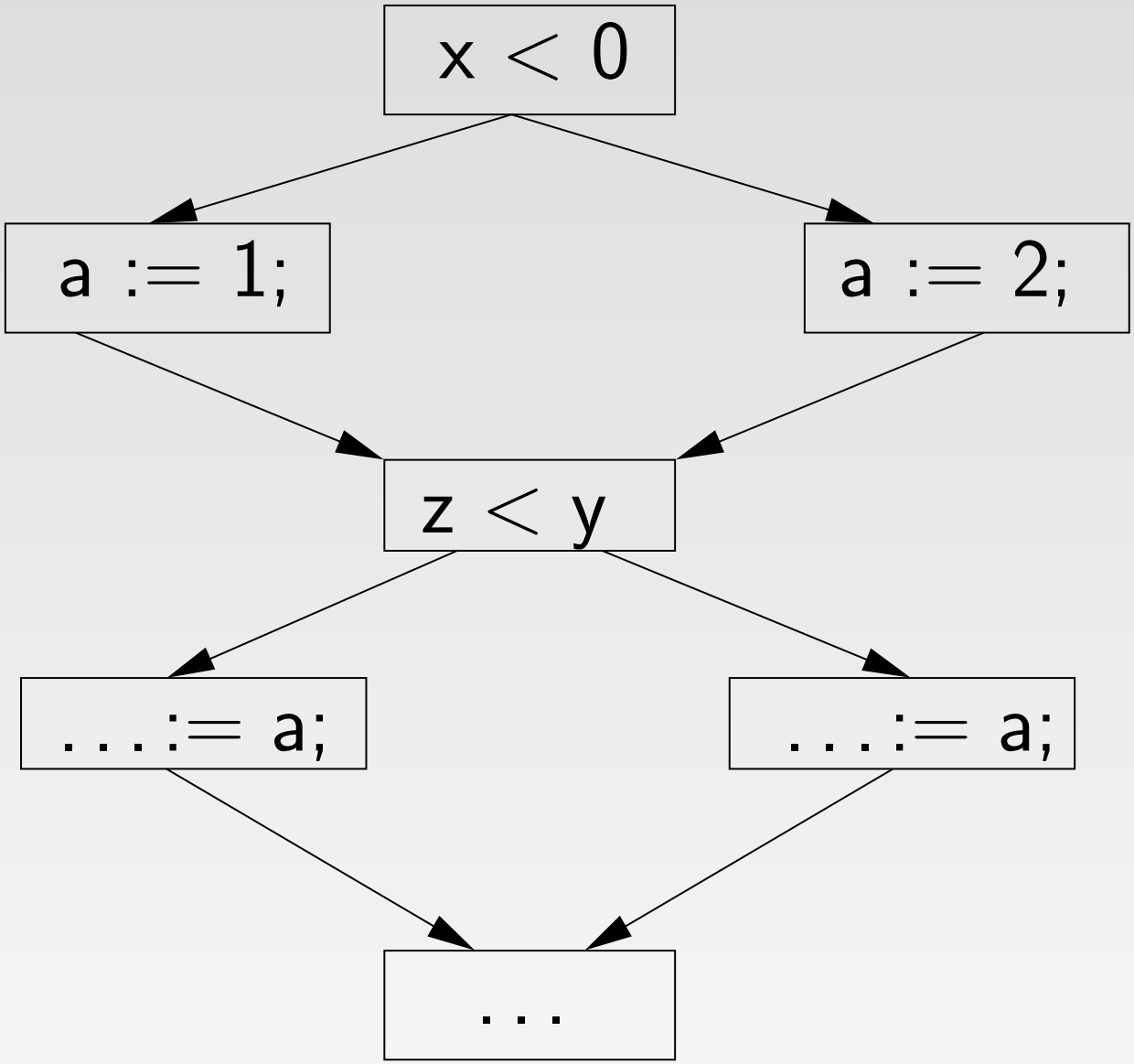


Repräsentation der Datenabhängigkeit

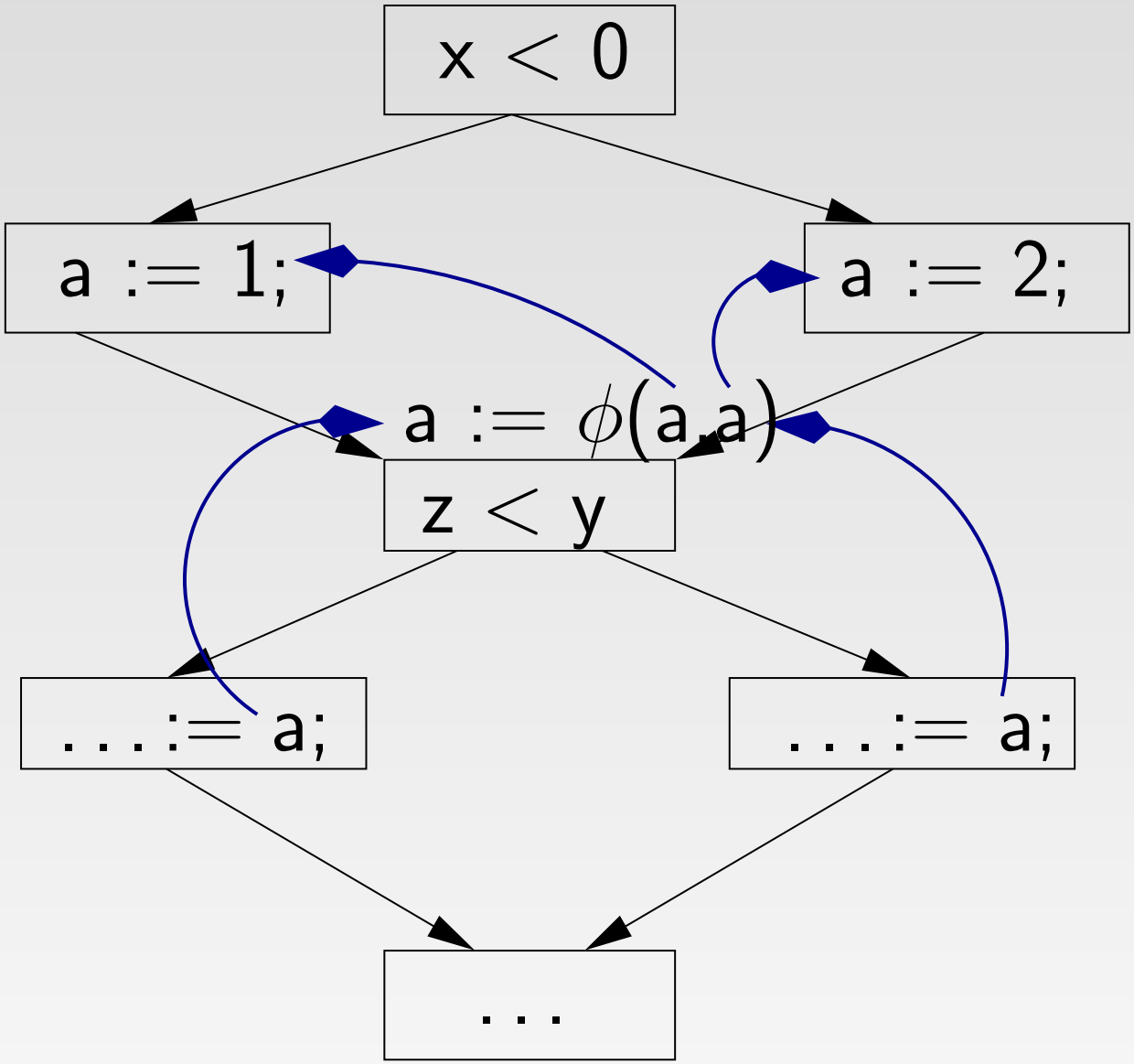
Wunsch nach einer kompakteren Darstellung: Nur eine Definition für jede Verwendung.

- Static Single Assignment Form (SSA)
- künstliche Zuweisungen, sogenannte ϕ-Knoten, werden eingefügt an den Knoten, an denen Kontrollflüsse zusammenlaufen, auf denen verschiedene Definitionen der gleichen Variablen liegen.
- die entsprechenden Stellen lassen sich effizient identifizieren (**?**)

Static Single Assignment Form



Static Single Assignment Form



Beispiel der SSA

```
x, y, z : T;
if cond1 then
  y := 4;
  x := 5;
else
  y := 3;
end if;

while cond2 loop
  if y > 0 then
    z := x + z ;
  end if;

  if y <= 0 then
    z := y ;
  end if;

end loop;
... = z ;
```

Beispiel der SSA

```
x, y, z : T; x0 := init; y0 := init; z0 := init;
if cond1 then
  y1 := 4;
  x1 := 5;
else
  y2 := 3;
end if;
x2 := φ(x0,x1); y3 := φ(y1,y2);
while z1 := φ(z0,z5); cond2 loop
  if y3 > 0 then
    z2 := x2 + z1;
  end if;
  z3 := φ(z1,z2)
  if y3 <= 0 then
    z4 := y3;
  end if;
  z5 := φ(z3,z4)
end loop;
... = z1;
```

Beispielanwendung

Finde alle potenziell lokal uninitialisierten Variablen

- (füge am Entry-Knoten eine Pseudodefinition für jede lokale Variable ein)
- für jede Pseudo-Definition D:
  - propagiere D längs der Use-Kanten und markiere alle dabei erreichten Verwendungen von Variablen
  - jede markierte Verwendung einer Variablen ist potenziell undefiniert

Beispielanwendung der SSA

```
x, y, z : T; x0 := init; y0 := init; z0 := init;
if cond1 then
  y1 := 4;
  x1 := 5;
else
  y2 := 3;
end if;
x2 := φ(x0,x1); y3 := φ(y1,y2);
while z1 := φ(z0,z5); cond2 loop
  if y3 > 0 then
    z2 := x2 + z1;
  end if;
  z3 := φ(z1,z2)
  if y3 <= 0 then
    z4 := y3;
  end if;
  z5 := φ(z3,z4)
end loop;
... = z1;
```

Beispielanwendung der SSA

```
x, y, z : T; x0 := init; y0 := init; z0 := init;
if cond1 then
  y1 := 4;
  x1 := 5;
else
  y2 := 3;
end if;
x2 := φ(x0,x1); y3 := φ(y1,y2);
while z1 := φ(z0,z5); cond2 loop
  if y3 > 0 then
    z2 := x2 + z1;
  end if;
  z3 := φ(z1,z2)
  if y3 <= 0 then
    z4 := y3;
  end if;
  z5 := φ(z3,z4)
end loop;
... = z1;
```

Wichtige Mengen beim Aufbau von SSA

- MustMod (A):
  Menge von Variablen, die sicher von A verändert werden
  → identifiziert die Definitionen
- MayMod (A):
  Menge von Variablen, die möglicherweise von A verändert werden
  → hier muss ein ϕ-Knoten eingefügt werden
- MayUse (A):
  Menge von Variablen, die möglicherweise von A verwendet werden
  → bei allen Verwendungen müssen Verweise auf Definition eingetragen werden

```
a1 = 1; // MustMod = {a}
f(&a);      // MayMod = {a}
  // wenn auch MayUse = {a}, dann stellt dies auch ein Use dar
  a2 = φ(a1, af) // af repräsentiert den Wert aus f
... a2 ... // MayUse = {a}
```

N.B.: MayMod und MustMod sind disjunkt.

Probleme beim Aufbau von SSA

- Aliasing: Zwei Namen sind Aliase, wenn sie auf überlappende Speicherbereiche verweisen.
- Problem:
  `x := ...;` – falls Alias (x, y), dann ist Zuweisung an x auch Zuweisung an y

Probleme beim Aufbau von SSA

- Aliasing: Zwei Namen sind Aliase, wenn sie auf überlappende Speicherbereiche verweisen.
- Problem:
  `x := ...;` – falls Alias (x, y), dann ist Zuweisung an x auch Zuweisung an y

```
y := 5;

if ... then
  x := 6;
end if;
...
... = y;
```

Probleme beim Aufbau von SSA

- Aliasing: Zwei Namen sind Aliase, wenn sie auf überlappende Speicherbereiche verweisen.
- Problem:
  `x := ...;` – falls Alias (x, y), dann ist Zuweisung an x auch Zuweisung an y

```
y := 5;

if ... then
  x := 6; -- if Is_Alias (x,y) then y := x; end if;
end if;
...
... = y;
```

Probleme beim Aufbau von SSA

- Aliasing: Zwei Namen sind Aliase, wenn sie auf überlappende Speicherbereiche verweisen.
- Problem:
  `x := ...;` – falls Alias (x, y), dann ist Zuweisung an x auch Zuweisung an y

```
y := 5;

if ... then
  x := 6; -- if Is_Alias (x,y) then y := x; end if;
end if;
... y := φ(y,y);
... = y;
```

Probleme beim Aufbau von SSA

- Aliasing: Zwei Namen sind Aliase, wenn sie auf überlappende Speicherbereiche verweisen.
- Problem:
  `x := ...;` – falls Alias (x, y), dann ist Zuweisung an x auch Zuweisung an y

```
y1 := 5;

if ... then
  x1 := 6; -- if Is_Alias (x,y) then y2 := x1; end if;
end if;
... y3 := φ(y1,y2);
... = y3;
```

Aliasing-Varianten I

- parameter-induziertes Aliasing durch Referenzparameter
  - `p (x, x)` (bzw. `p(&x, &x)` in C und C++)
  - `p (g)`, wobei g eine globalere Variable ist und in p verwendet wird
  - Lösung: Propagierung der Alias-Info über den Call-Graphen
  - (Ansonsten konservative Annahme: zwei Referenzparameter bzw. globale Variable/Referenzparameter sind Aliase; eventuell kann Typinformation helfen)
- Arraykomponenten
  - `a [2 * i]` und `a [ j ]`
  - Lösungsansatz: Löse Gleichung 2*i = j
  - Ansonsten: Modelliere Zuweisungen an Teilkomponente als Zuweisung an gesamtes Array: Partial Update/Read m.a.W. konservative Annahme, dass jeder Index sich auf alle Array-Elemente beziehen kann.

Aliasing-Varianten II

- stack-gerichtete Zeiger
  - `p := &x;`
  - Lösungsansatz: Points-To-Analyse
  - Konservative Annahme: `*p` könnte jede statische Variable betreffen, deren Adresse genommen wird.
- heap-gerichtete Zeiger
  - `p := new T; q := new T; q.next = p;`
  - Lösungsansatz: "'Shape-Analyse"'
  - Konservative Annahme: `*p` kann jedes Element des Heaps betreffen.