# Optimierungen (nicht notwendigerweise auf SSA)

- Fortschreiben v. Konstanten u. Kopien, intra- und interprozedural
- Konstanten Falten (Auswerten konstanter Ausdrücke)
- Beseitigen von totem/unerreichbarem Code
- Operatorvereinfachung (Beseitigen von Induktionsvariablen)
- Verschieben von schleifeninvariantem Code
- Eliminieren partieller Redundanzen
- Codeverschiebung, -plazierung
- Spezialisieren und Klonen von Grundblöcken und Prozeduren
- Eliminieren Indexgrenzenprüfung
- Offener Einbau von Prozeduren (inlining)



# Optimierungen (nicht notwendigerweise auf SSA) Teil 2

- Ablauf-Vereinfachung
- Ausrollen/Verschmelzen/Teilen von Schleifen; Software-Fließband
- Umordnen von Reihungen und -zugriffen (D-Cache-Optimierungen)
- Vorladen von Daten (*prefetching*)
- Optimieren von Blatt-Prozeduren
- Beseitigen von End-Aufrufen und Endrekursionen
- Registerzuteilung



# Stand der Forschung

- Offenes Problem:
  - Welche Optimierungen auszuwählen?
  - Optimale Reihenfolge der Optimierungen?
- Es gibt keine aktuellen wissenschaftliche Aussagen zu einer solchen Taxonomie!
- Faustregel:
  - Abgesehen von Cacheoptimierung und Operatorvereinfachung bringt die 1. durchgeführte Optimierung 15% alle weiteren $<$ 5%.
  - Das ist weitgehend unabhängig von der Wahl der Optimierungen und ihrer Reihenfolge.
  - Viele Optimierungen bewirken ähnliches bzw. sind ineinander enthalten.
  - Bei numerischen Programmen bringt Operatorvereinfachung Faktoren $>$ 2; Cacheoptimierung zwischen 2 und 5.



# Das Vollbeschäftigungstheorem für Übersetzerbauer

- Satz (Rice, 1953): Zu jedem algorithmisch arbeitenden Übersetzer $U$ gibt es einen Übersetzer $U'$, der für bestimmte Programme kürzeren Code erzeugt.
- Korollar (Vollbeschäftigungstheorem für Übersetzerbauer): Zu jedem optimierenden Übersetzer gibt es einen besseren.
- Beweis des Satzes:
  Annahme: es gibt einen Übersetzer $U$, der jedes Programm $\pi$ mit algorithmischen Methoden in das absolut kürzeste Programm $Opt(\pi)$ mit gleichem Ein-/Ausgabeverhalten übersetzt. Sei $\pi$ ein nicht haltendes Programm ohne E/A. Dann wird $\pi$ von $U$ übersetzt in:

  $$Opt(\pi) \mapsto m : \text{goto } m$$

  Um zu prüfen, ob $\pi$ nicht hält, muß man nur $Opt(\pi)$ berechnen und das Ergebnis inspizieren. Damit löst man also das Halteproblem. Da das Halteproblem unentscheidbar ist, kann es also ein solches $U$ nicht geben. □



# Prinzip von Kostenmodellen bei Optimierungen

Kostenmodell: Laufzeit eines Programms, eventuell kombiniert mit Energieverbrauch

- Speicherbedarf kann in Laufzeit umgerechnet werden
  - daher ist oft auch Verkürzung des Codes Laufzeitoptimierung
- Statisch nur konservativ abschätzbar wegen Unkenntnis der
  - Anzahl Wiederholungen von Schleifen
  - Sprungbedingungen
- Selbst bei linearem Code nicht statisch bekannt
  - Befehlsanordnung durch Prozessor
  - Pufferspeicher (Cache): Zugriffe auf Speicher sind datenabhängig
  - Fließbandverarbeitung im Prozessor

Laufzeit gewöhnlich $\neq$ Summe der Laufzeiten der einzelnen Befehle!



# Optimierungen auf SSA

Optimierungen

- sind auf SSA mit wenig Analyseaufwand durchführbar
- können oft mit SSA-Aufbau verschränkt werden
- können von weiteren Programmanalysen profitieren (mehr dazu in folgenden Vorlesungen)

Zwei Arten von Transformationen:

- Normalisierende Transformationen
  - bringen SSA-Graph in definierte Form, um unterschiedlich geschriebene Ausdrücke (syntaktisch) vergleichbar zu machen
  - ermöglichen optimierende Transformationen
  - nutzen z.B. algebraische Identitäten (Äquivalenzoperationen)
    - Assoziativgesetz
    - Distributivgesetz
  - sind eingeschränkt durch
    - Auswertungsreihenfolge (Java)
    - Ausnahmen (Java, Eiffel)
    - Gleitpunktarithmetik (nicht assoziativ, nicht distributiv)
- Optimierende Transformationen



# Betrachtete Optimierungen

- Operatorvereinfachung (strength reduction)
  - Idee: Ersetze teuere Operationen durch billigere, semantisch äquivalente Operationen
  - Hauptanwendung: Vereinfache Multiplikationen mit Indexvariable in Schleifen zu Additionen (Induktionsanalyse, Lineare Adreßfortschaltung)
- Eliminieren gemeinsamer Teilausdrücke (GTE)
  (common subexpression elimination, CSE)
  - Idee: Eliminiere Mehrfachberechnungen von identischen Werten
  - Hauptanwendung: Adreßrechnung
- Eliminieren partieller Redundanzen (EPR)
  (partial redundancy elimination, PRE)
  schwächere Varianten:Code-Plazierung, Code-Verschiebung
  - Ziel: Vermeide Berechnung von Werten die nicht auf allen Ausführungspfaden gebraucht werden
  - Anwendung: Verschieben von schleifeninvarianten Berechnungen aus Schleifen



# Inhalt

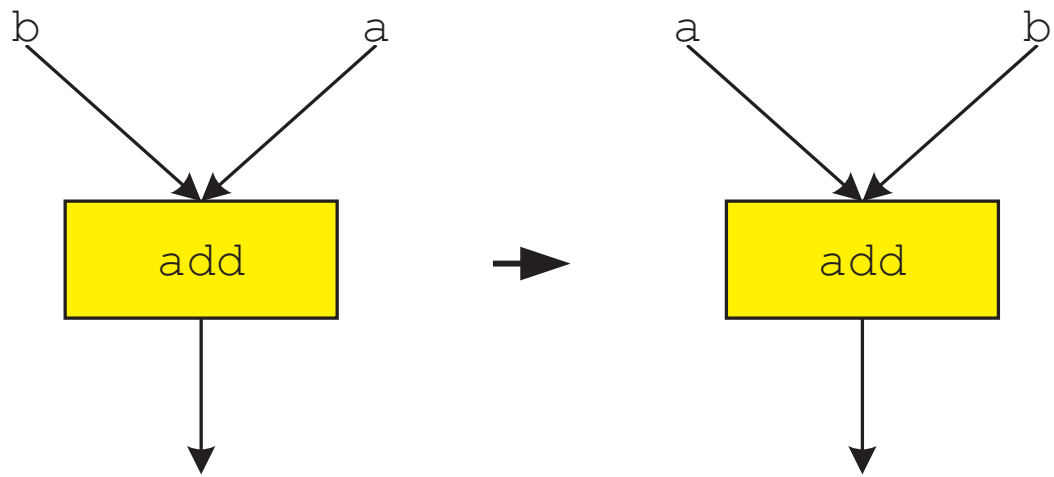




# Normalisierung - Kommutativgesetz

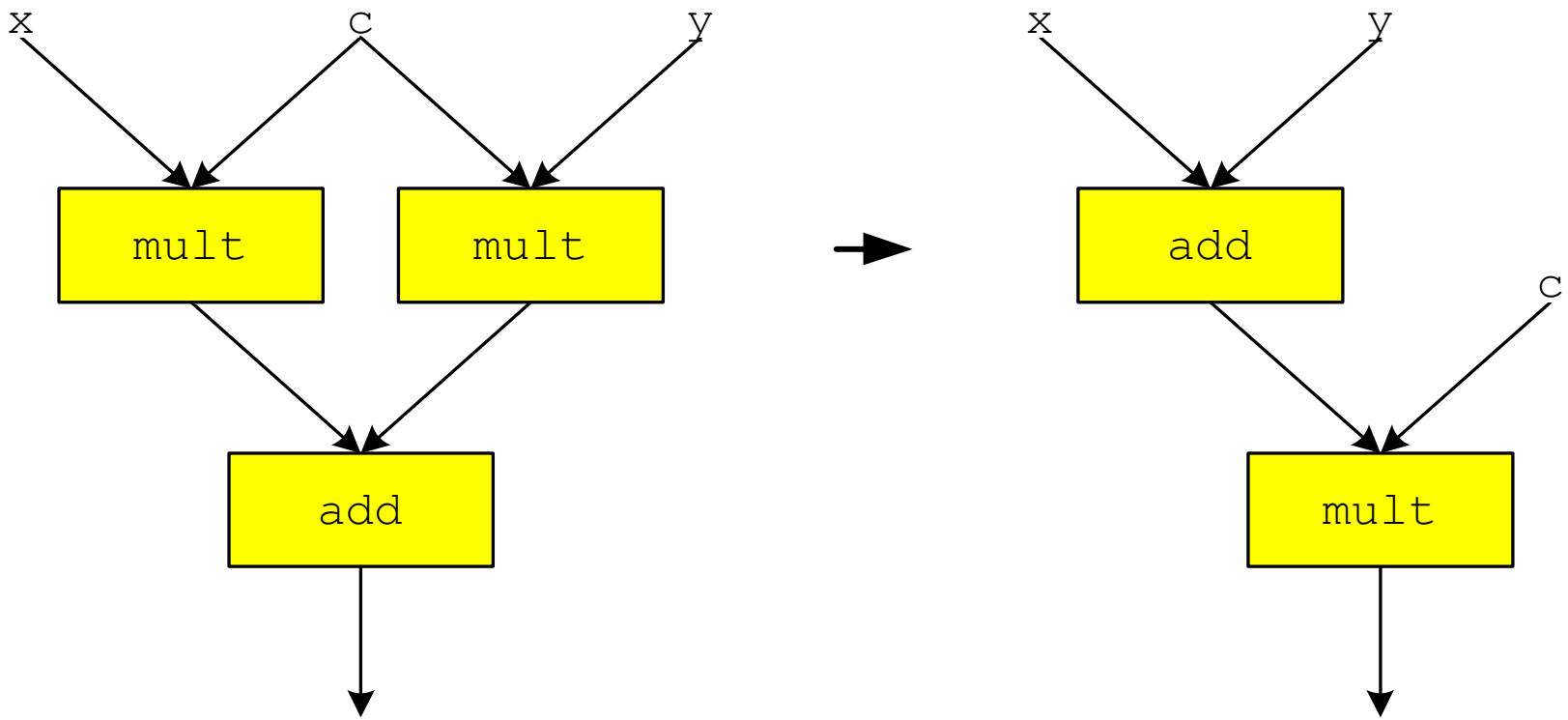
Ziel der Normalisierung: arithmetische Ausdrücke vergleichbar machen, um gemeinsame Teilausdrücke zu finden (Problem: keine kanonische Normalform arithmetischer Ausdrücke)

- Definiere Ordnung B auf SSA-Knoten (z. B. Reihenfolge des Aufbaus)
- Ordne kommutative Operation $\tau(a, b)$ um, so daß $B(a) > B(b)$.



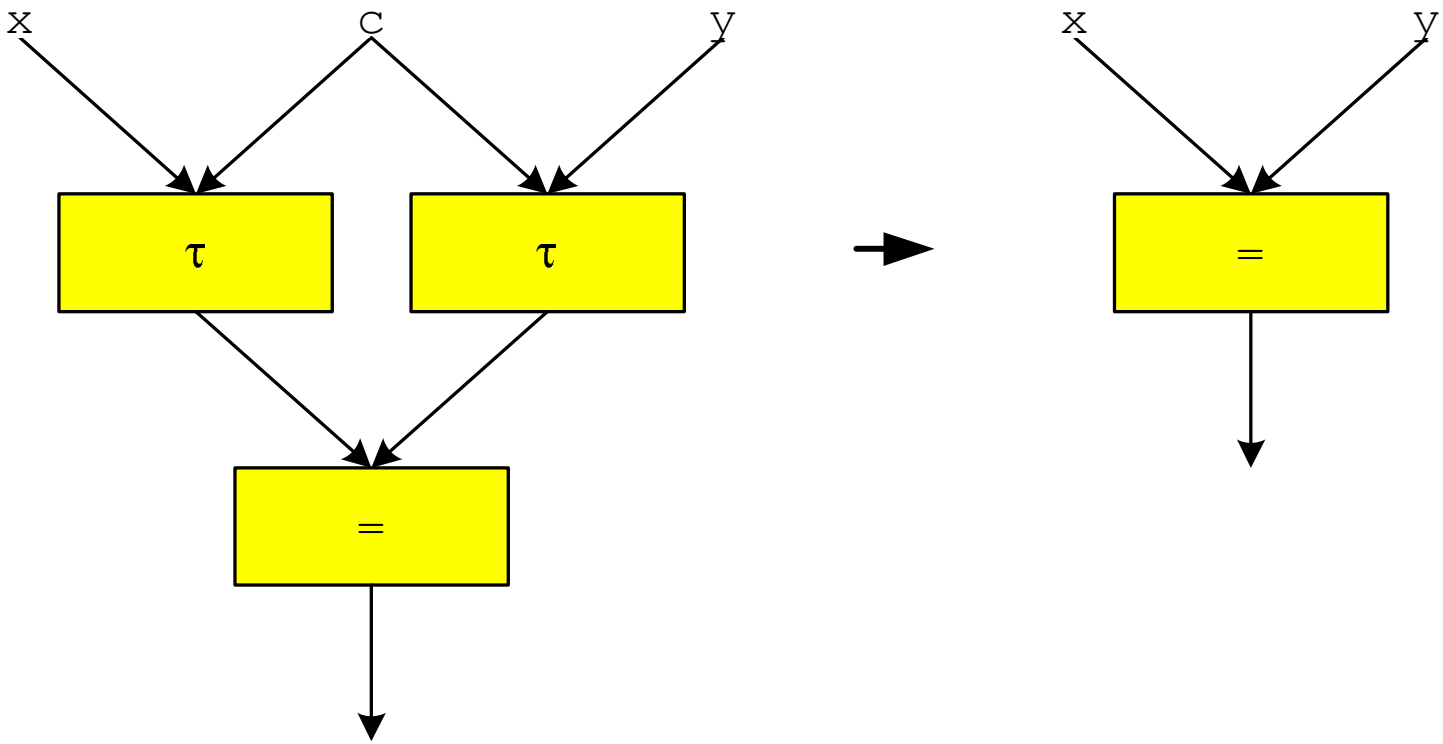
# Normalisierung - Distributivgesetz

- Definiere Ordnung $B'$ auf Operationen
- Ordne Operationen $\tau_1 \circ \tau_2$ um, so daß $B'(\tau_1) > B'(\tau_2)$



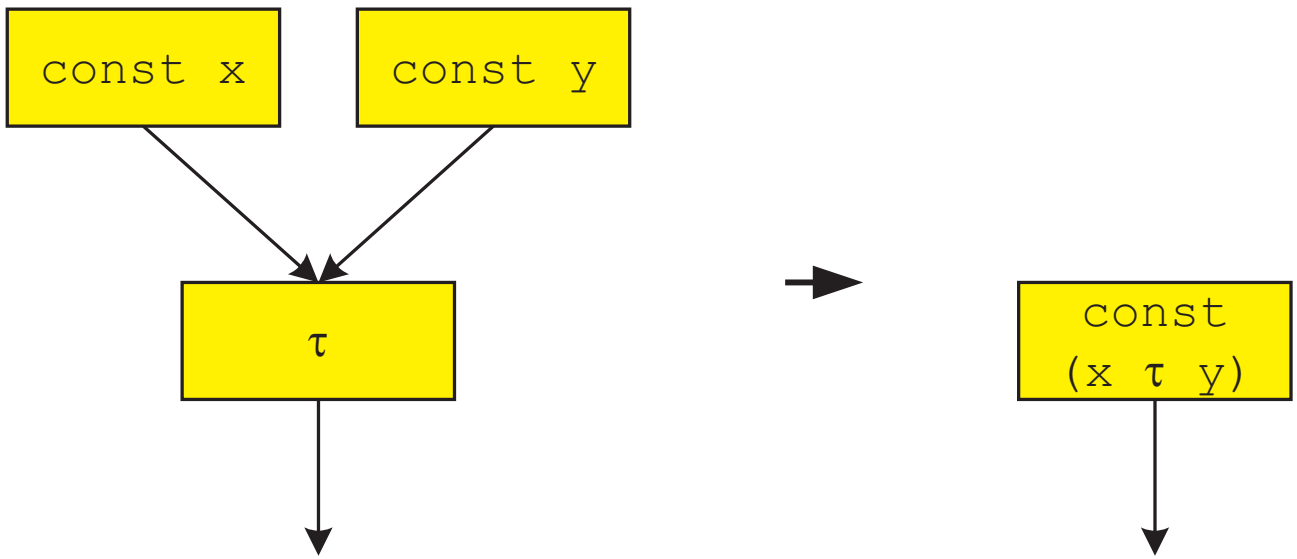
# Normalisierung - Vergleiche ändern

- Wenn die Ergebnisse der Operationen $\tau(x, c)$ und $\tau(c, y)$ nur zum Vergleichen benötigt werden, können die Operationen u.U. wegfallen.



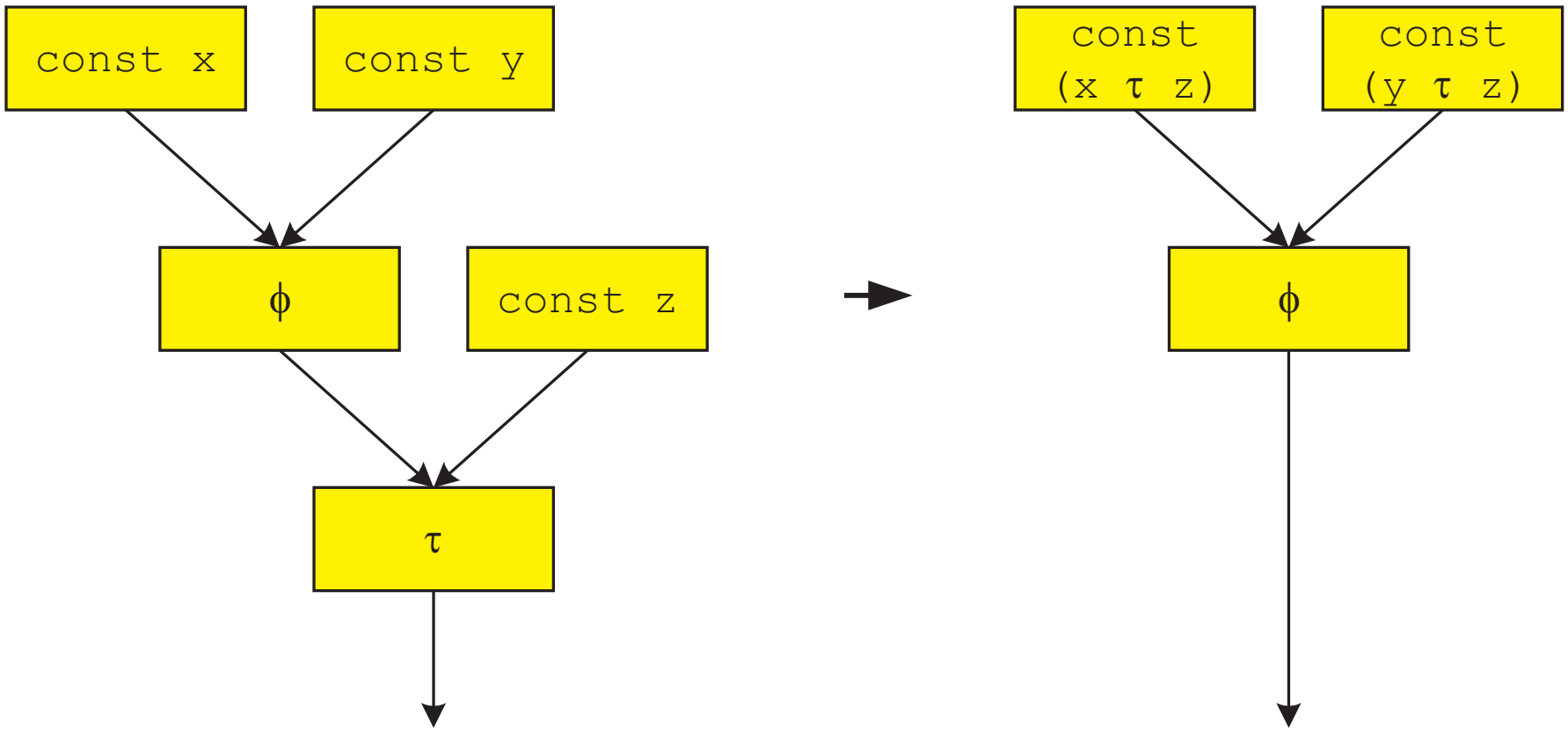
# Optimierungen - Konstantenfaltung I

- Berechne (Teil-)Ausdrücke mit konstanten Operanden
- Beachte Arithmetik der Sprache
- Beachte Arithmetik der Zielmaschine
- Zu faltende Konstanten werden zu 80 bis 90% bei der Adreßrechnung erzeugt.



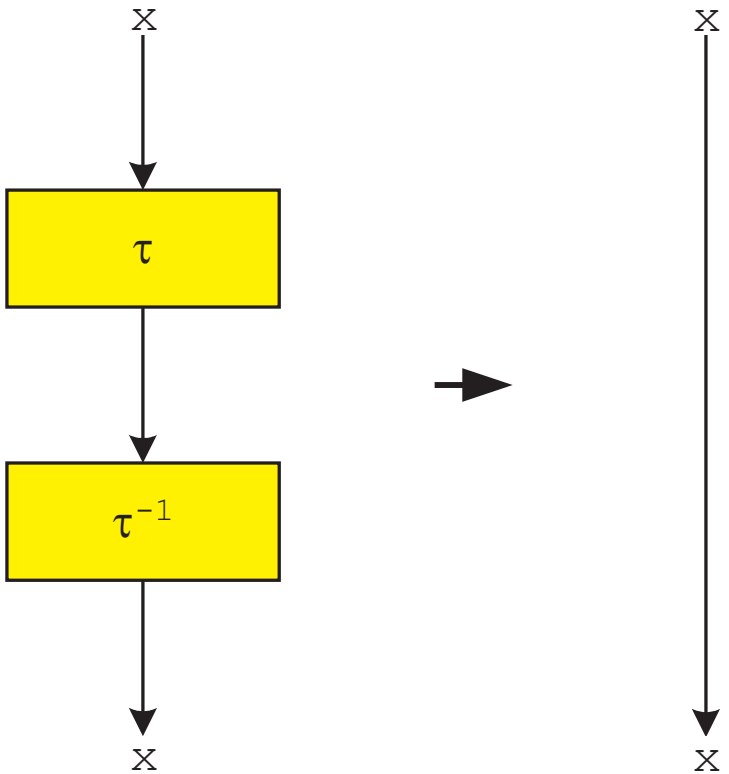
# Optimierungen - Konstantenfaltung II

- auch über $\phi$-Operationen hinweg möglich!



# Optimierung - Inverse Operationen löschen

- oft nach Anwendung anderer Transformationen nötig.



# Operatorvereinfachung

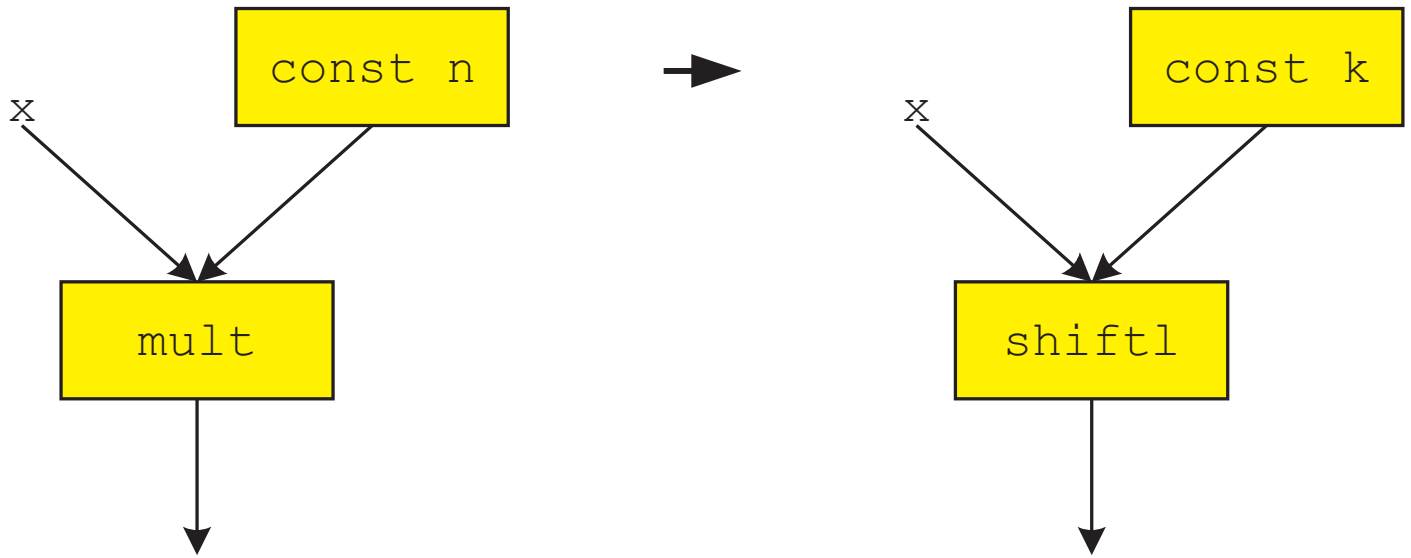
Operatorvereinfachung (strength reduction)

- Idee: Ersetze teuere Operationen durch billigere, semantisch äquivalente Operationen
- Hauptanwendung: Vereinfache Multiplikationen mit Indexvariablen in Schleifen zu Additionen (Induktionsanalyse, lineare Adreßfortschaltung)
  - besonders wichtig auf Rechnern mit Multiplikationszeit $>>$ Additionszeit
  - dort bei numerischen Programmen Beschleunigung bis Faktor 3 erzielbar.



# Operatorvereinfachung Beispiel I

- elementares Beispiel: Berechnung von $x \times n, n = 2^k$



# Operatorvereinfachung Beispiel II

```
for (i=0; i<n; i++)
for (j=0; j<m; j++)
... a[i,j] ...;
```

Adreßberechnung:
⟩a[i,j]⟨=
⟩a[0,0]⟨+ i*m*d + j*d
Kosten pro Iteration:
3 Multiplikationen,
2 Additionen

Ideen eines
Maschinensprachprogrammierers:

- initialisiere Adreßvariable adr vor den Schleifen mit ⟩a[0,0]⟨
- in der inneren Schleife:
  adr = adr + d
- dann gilt für eine
  $[0 : (n-1), 0 : (m-1)$ Reihung in zeilenweiser Speicherung jeweils
  ⟨adr⟩ == ⟩a[i,j]⟨
- dabei benutzt:
  ⟩a[i+1,0]⟨==⟩a[i,m-1]⟨+d
- Kosten pro Iteration:
  1 Addition

# Operatorvereinfachung Beispiel IIa

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    ... a[i,j] ...;
```

Adreßberechnung:
⟩a[i,j]⟨=
⟩a[0,0]⟨ + i*m*d + j*d
Kosten pro Iteration:
3 Multiplikationen,
2 Additionen

```
adr = ⟩a[0,0]⟨;

for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {

...⟨adr⟩...;

    adr = adr + d;
  }
}
```

Kosten pro Iteration:
1 Addition

# Induktionsanalyse (nicht im SSA-Kontext)

Operatorvereinfachung für Induktionsvariable
erfordert Induktionsanalyse

Definition: Induktionsvariable (IV):

- Variable $i$ ist Induktionsvariable, wenn in der Schleife nur Zuweisungen der Form $i := i + c'$ vorkommen
- Variable $i'$ ist Induktionsvariable, wenn $i'$ nur Werte $i' := c * i + c''$ annimmt, wobei $i$ eine Induktionsvariable ist.
- Dabei sind $c, c', c''$ während Ausführung der Schleife konstant

Transformation:

- Sei $i_0$ die Vorbesetzung von $i$ vor Schleifenbeginn
  - Neue Variable $i_a$ mit Initialisierung $i_a := c * i_0 + c''$ einführen
  - Zuweisung $i_a := i_a + c * c'$ am Ende der Schleife
  - Ersetze $i$ durch $(i_a - c'')$ **div** $c$ und $i'$ durch $i_a$



# Induktionsanalyse (nicht im SSA-Kontext)

- Einfaches Beispiel:

```
s := 0;
for (i:=0; i<100; i++)
    s := s + a[i];
```

- $>a[i]<$ wird in jedem Schleifendurchlauf berechnet als $>a[0]<+\,i \times d$
  mit Konstante $d$: Umfang des Typs von $<a>$



# Bestimmung von Induktionsvariablen I (im SSA-Kontext)

- Gesucht: Werte (d.h., SSA-Ecken), die linear mit Anzahl der Schleifendurchläufe wachsen
- Optimistischer Ansatz:
  - Betrachte alle Werte als IV-Kandidaten
  - Bis zum Erreichen des Fixpunkts: Eliminiere Werte $t$ aus der Menge der IV-Kandidaten, falls sie nicht eine der folgenden Formen haben:
    - $t := t' \pm c'$, wobei $t'$ IV-Kandidat ist und $c'$ außerhalb der Schleife berechnet wird (*direkte IV*)
    - $t := c \cdot t' \pm c''$, wobei $t'$ IV-Kandidat ist und $c$, $c''$ außerhalb der Schleife berechnet werden (*indirekte IV*)
    - $t := \Phi(t_{i_1}, t_{i_2}, \ldots, t_{i_n})$, wobei alle Operanden des $\Phi$-Operators entweder direkte IV-Kandidaten sind oder außerhalb der Schleife berechnet werden.
  - Beim Erreichen des Fixpunkts bleiben von IV-Kandidaten gerade die Induktionsvariablen übrig.
- In der Praxis auf ganze Zahlen beschränkt

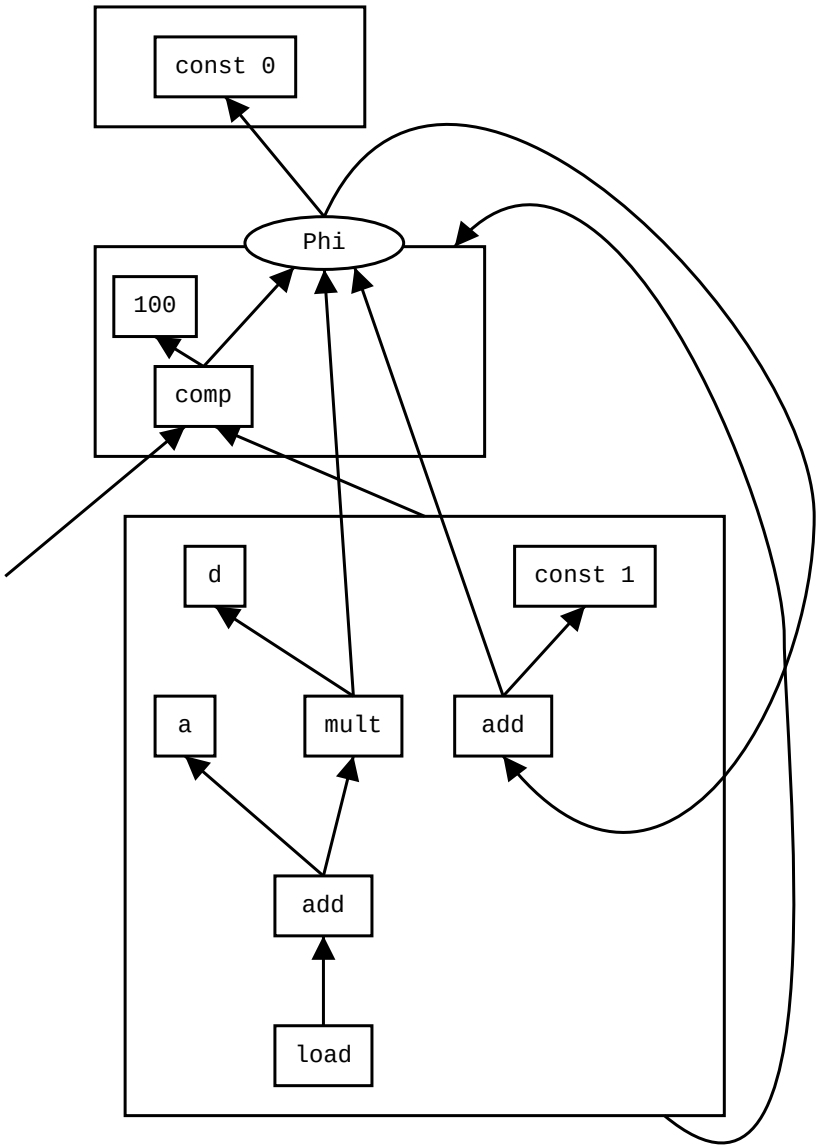


# Bestimmung von Induktionsvariablen II (im SSA-Kontext)

- Definition stark zusammenhängende Komponente (SZK) eines gerichteten Graphen $G = (N, E)$: $N' \subseteq N$ ist SZK von $G$, wenn für alle $n_i, n_j \in N'$ gilt: Es gibt einen Pfad von $n_i$ nach $n_j$.
- Beobachtung:
  - Im SSA-Graph wird für Induktionsvariable ein SZK aufgebaut
  - Die SZK ist mit dem Restgraphen über eine Φ-Ecke verbunden
- Vorgehen:
  - Bestimme SZK's des SSA-Graphen (Tarjan-Algorithmus)
  - Überprüfe SZK's auf IV-Kandidaten
  - Für jede Benutzung eines IV-Kandidaten:
    - Vereinfache Berechnung des IV-Kandidaten (Φ-Zyklus)
    - Ändere Benutzung des IV-Kandidaten

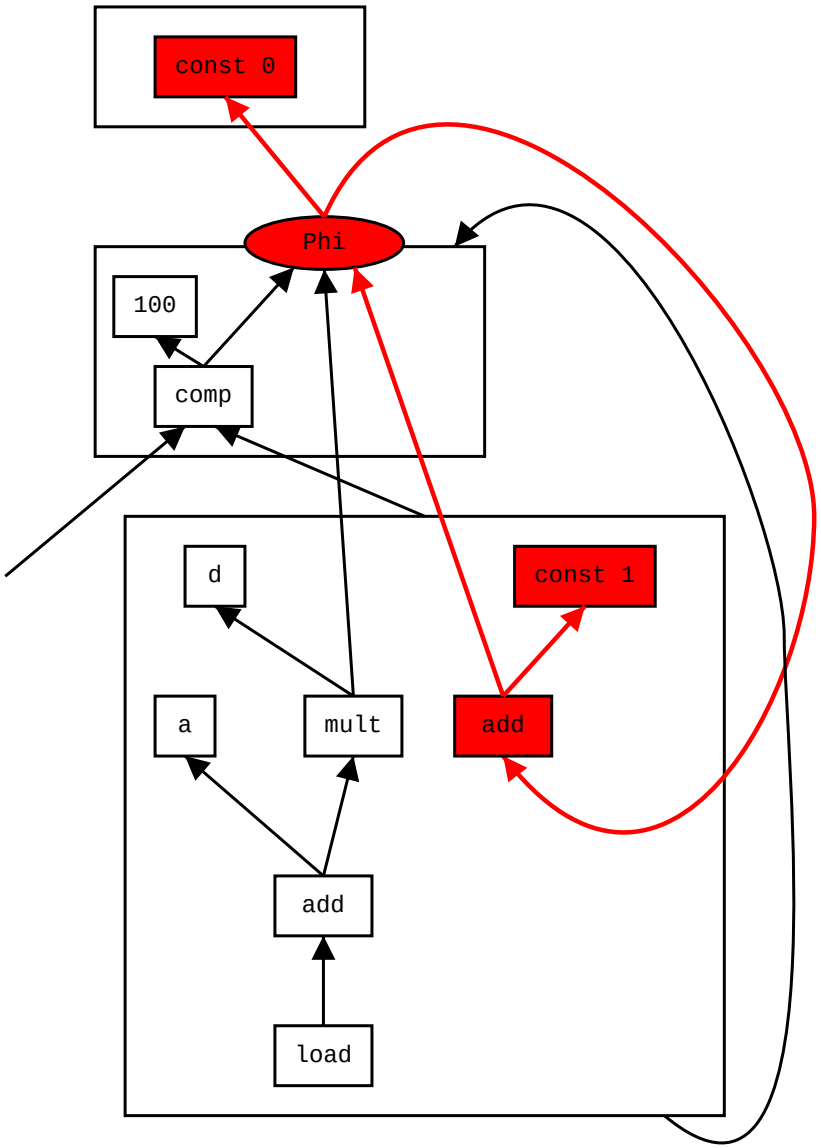


# Beispiel: Partieller SSA-Graph



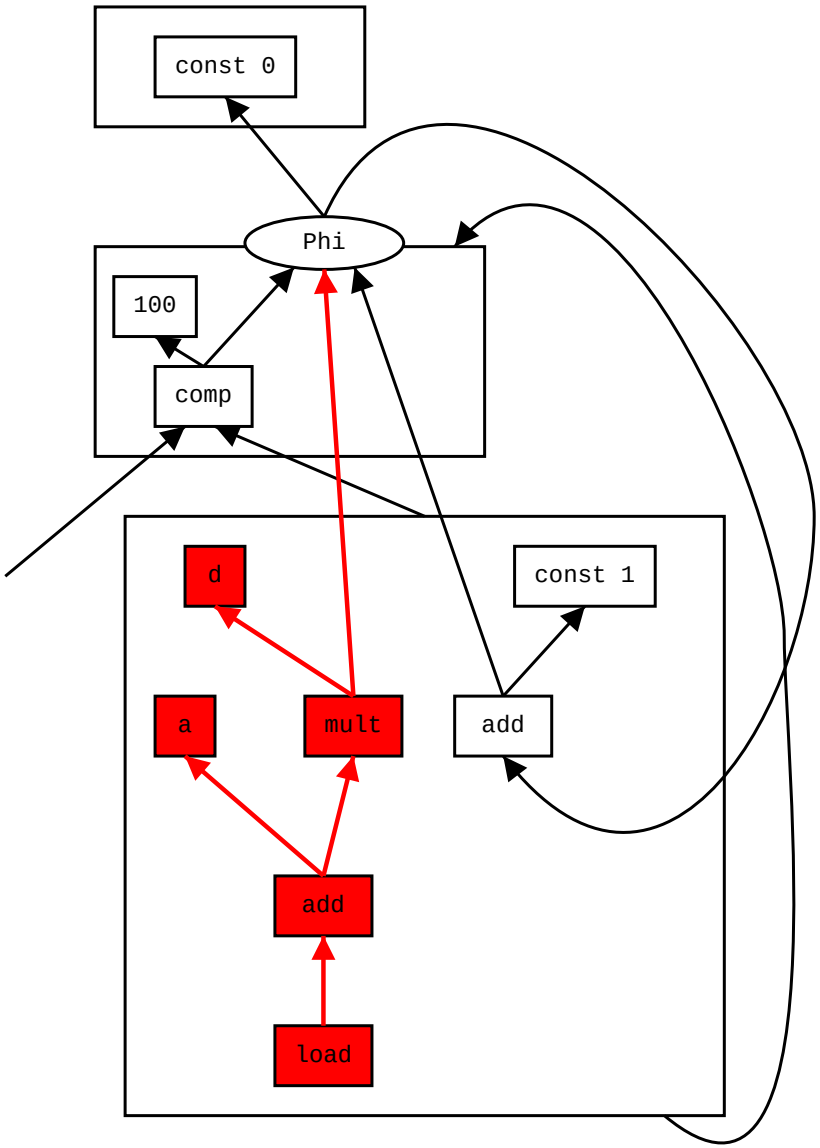

- Gezeigt:
  - Initialisierung und
  - Fortschaltung der Induktionsvariable
  - Berechnung der Lade-Adresse
- Nicht gezeigt:
  - Summierung der geladenen Werte

# Beispiel: SZK im SSA-Graph



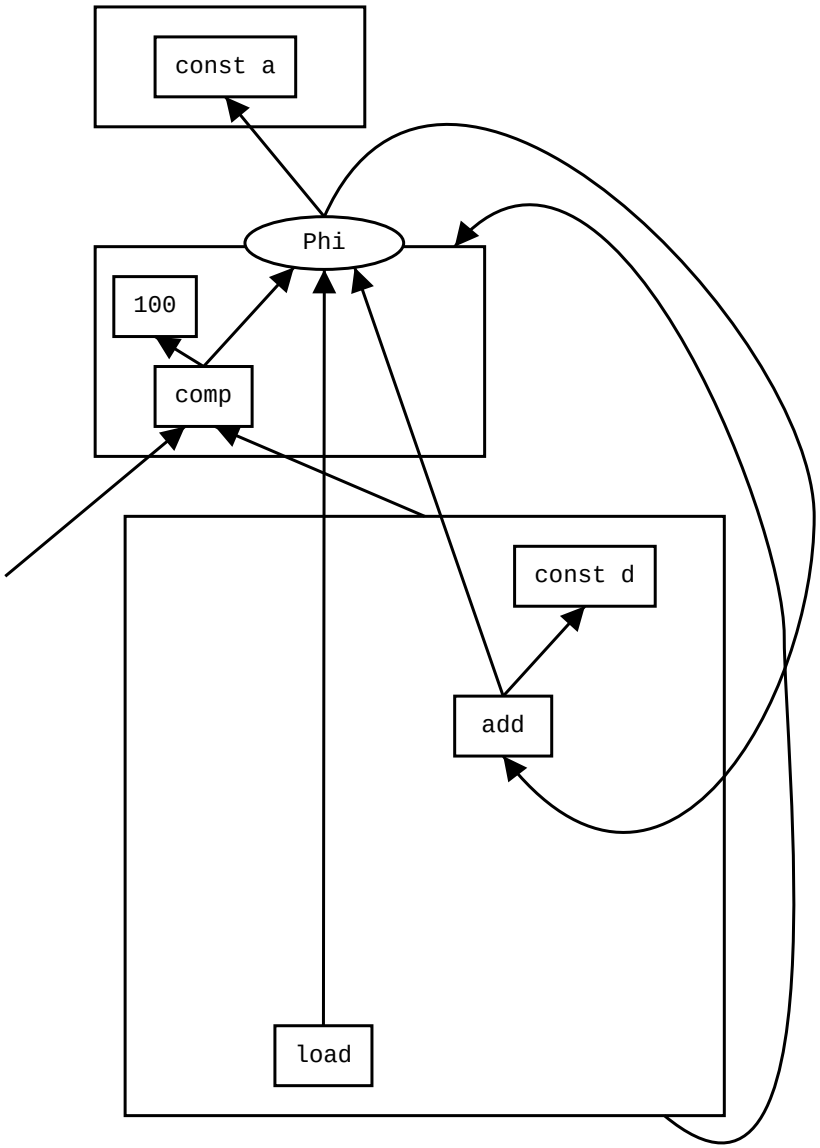

- Initialisierung und
- Fortschaltung der Induktionsvariable
- Φ-Ecke faßt zusammen:
  - Initialisierung
  - Konstante Erhöhung der Induktionsvariable
- bildet *stark zusammenhängende Komponente*
- Folge: Φ-Ecke ist *direkte* Induktionsvariable

# Beispiel: Benutzung der Induktionsvariable




- Benutzung der Induktionsvariable ($\Phi$-Ecken) für Adreß-Arithmetik
- Lineare Funktion $a_i := a_0 + d \cdot \Phi$
- Deshalb *indirekte* Induktionsvariable

# Beispiel: Umgeformter SSA-Graph




- Bereits geändert:
  - Initialisierung und
  - Fortschaltung der Induktionsvariable
- Noch zu ändern:
  - Vergleich zum Schleifenabbruch



# Inhalt





# Eliminieren gemeinsamer Teilausdrücke

Ziel:

- Vermeide wiederholte Berechnung von Teilausdrücken

Anwendung:

- Berechnung von Speicheradressen bei Zugriffen auf Felder eines Verbundes (Studie: In PL/1-Programmen 60% aller Berechnungen sind Adreß-Arithmetik)
- Generierter, d. h. nicht von Hand geschriebener Code

Beispiel: Sei `offset_x` die relative Position von Feld `x` in Struktur `A`:

```
A a;
a.x = a.x + 1;

Code:
tmp1 = a + offset_x;
tmp2 = a + offset_x;
>tmp2< = <tmp1> + 1;
```



# Eliminieren gemeinsamer Teilausdrücke

Definition Semantische Gleichheit: Wenn zwei Operationen das gleiche Ergebnis liefern, sind sie semantisch gleich. Beobachtung in SSA:

- Sind zwei Ausdrücke $k, k'$ syntaktisch gleich, so sind sie auch semantisch gleich.
  - Syntaktische Gleichheit heißt: Gleiche Operanden, gleiche Operation
  - in Nicht-SSA-Darstellungen gilt dies nicht!

Folgerung:

- Sind $k$ und $k'$ syntaktisch gleich, dann kann $k'$ wegfallen,
  - wenn $k$ $k'$ dominiert

Vorberechnung:

- Normalisierung, Kommutativgesetz anwenden
- Kann mit Eliminierungsalgorithmus verschränkt werden



# Eliminieren gemeinsamer Teilausdrücke - Implementierung

- Aufgabe: Gegeben Ecken $k$ mit Operation $\tau(op_1, op_2)$, finde alle Ecken $j$ mit Operation $\tau(op_1, op_2)$.
- Durchführung:
  - Unterhalte Haschtabelle für SSA-Ecken;
  - Berechnung des Hasch-Schlüssels aus $\tau, op_1$ und $op_2$
  - Für jede Ecke $k = \tau(op_1, op_2)$:

    Suche in Haschtabelle nach $k$
    Wenn Eintrag $j$ vorhanden und $j$'s Block dominiert $k$'s Block:
     verwende $j$ statt $k$

    sonst:
     verwende $k$ und trage $k$ in Tabelle ein

- Algorithmus kann in SSA-Aufbau integriert werden
  - SSA-Aufbau liefert alle Dominatoren von $b$ vor dem Block $b$ selbst
  - geringerer Speicherbedarf des SSA-Graphen



# Inhalt





# Eliminieren partieller Redundanzen

- Problem:
  Operation $p$ (d.h. die Berechnung eines Wertes) wird auf manchen Pfaden mehrfach ausgeführt
- Idee:
  Reduziere die Anzahl der Berechnungen durch Umplazieren der entsprechenden Operationen $p$



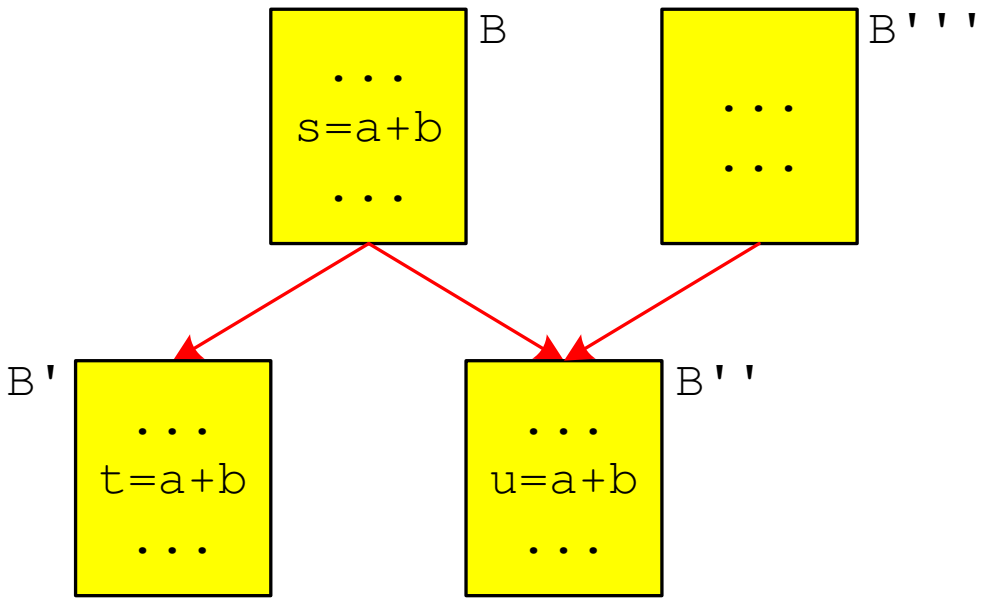
# Wiederholung: Sicher verfügbarer Ausdruck

- $a + b$ ist ein sicher/partiell verfügbarer Ausdruck an einer Stelle $l$, wenn er auf allen Pfaden/einem Pfad vom Funktionsbeginn berechnet wurde, und sich seine Operanden nicht geändert haben
- Bemerkung: Berechenbar mit Datenflußattribut *available expressions*.
  - Partiell verfügbare Ausdrücke ist eine möglich-Analyse
  - Sicher verfügbare Ausdrücke ist eine sicher-Analyse



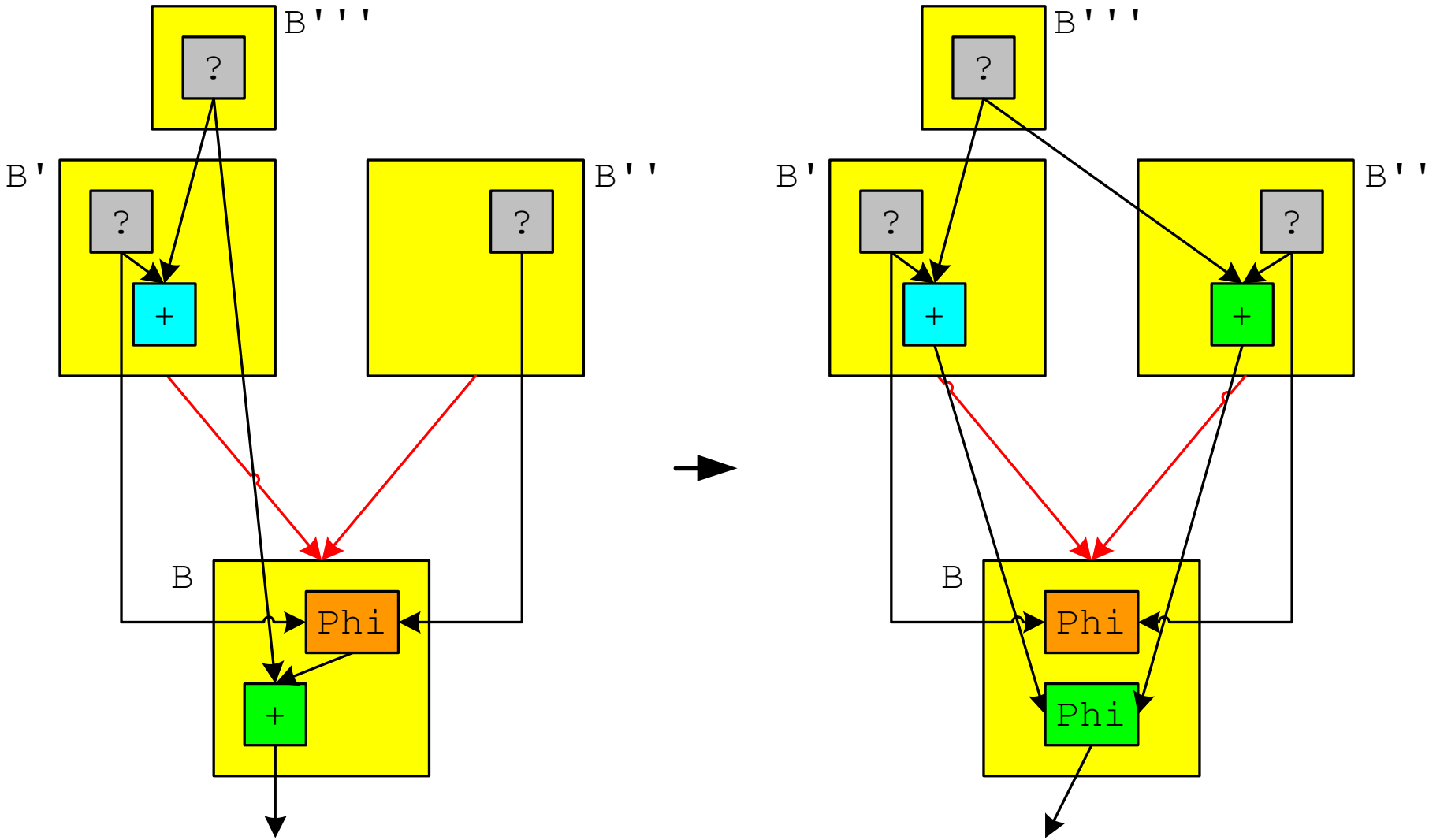
# Partielle Redundanz (nicht im SSA-Kontext)

- Eine Berechnung $a + b$ ist sicher/partiell redundant, wenn $a + b$ unmittelbar vor dieser Berechnung sicher/partiell verfügbar ist.
- Beispiel:




Ausdruck $a + b$ ist in $B'$ redundant und in $B''$ partiell redundant, da es in $B'''$ nicht berechnet wird.

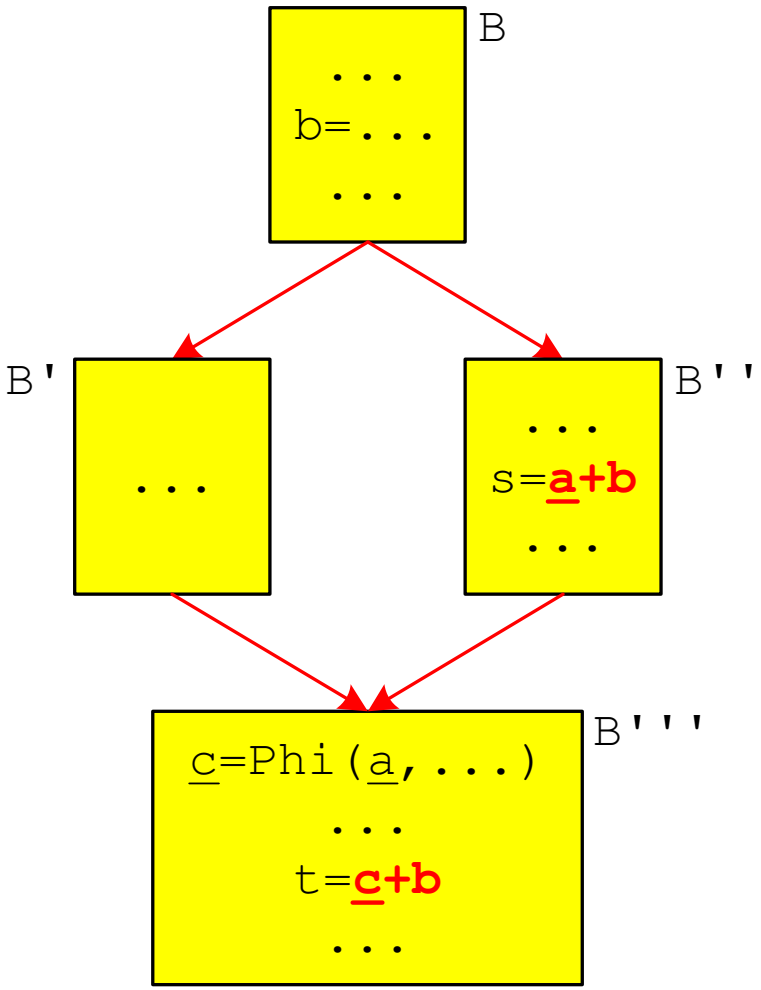
# Eliminieren partieller Redundanzen – Beispiel in SSA-Form



# Eliminieren partieller Redundanzen

## Probleme bei SSA:

- In SSA haben Variable mehrere Versionen $a \longrightarrow a_1, a_2, \ldots$.
- $\Phi$s vermischen den Wert einer Variable mit dem Ausführungspfad.
- Dadurch können gleiche Berechnungen auch unterschiedliche Versionen haben, z.B. sind $a + b$ sowie $c + b$ die gleichen Berechnungen, wenn der Ausführungspfad $B, B'', B'''$ ist.



# PRE: Entwicklung im Überblick

- Standard PRE-Algorithmen verwenden bis zu sechs Datenflußgleichungen parallel und arbeiten nicht auf SSA.
- Standard-Verfahren sind:
  - Morel, Renvoise, *Global optimization by suppression of partial redundancies,* 1979
  - Drechsler, Stadel, *A Solution to a Problem with Morel's and Renvoise's Global Optimization,* 1988
  - Knoop, Steffen, Rüthing, *Lazy Code Motion,* 1993
  - Cooper, Briggs, *Effective PRE,* 1994
- Datenflußgleichungen auf SSA bedeutungslos
- SSA wenig geeignet, da Φs gleiche Berechnungen verstecken
- PRE (auf SSA) ist aktuelles Forschungsthema
  - Chow, Kennedy, et al. *Partial Redundancy Elimination in SSA Form*, 1997
  - Bodík, Gupta, Soffa, *Complete Removal of Redundant Expressions*, 1998
  - VanDrunen, *Partial Redundancy Elimination for Global Value Numbering*, 2002



# Inhalt





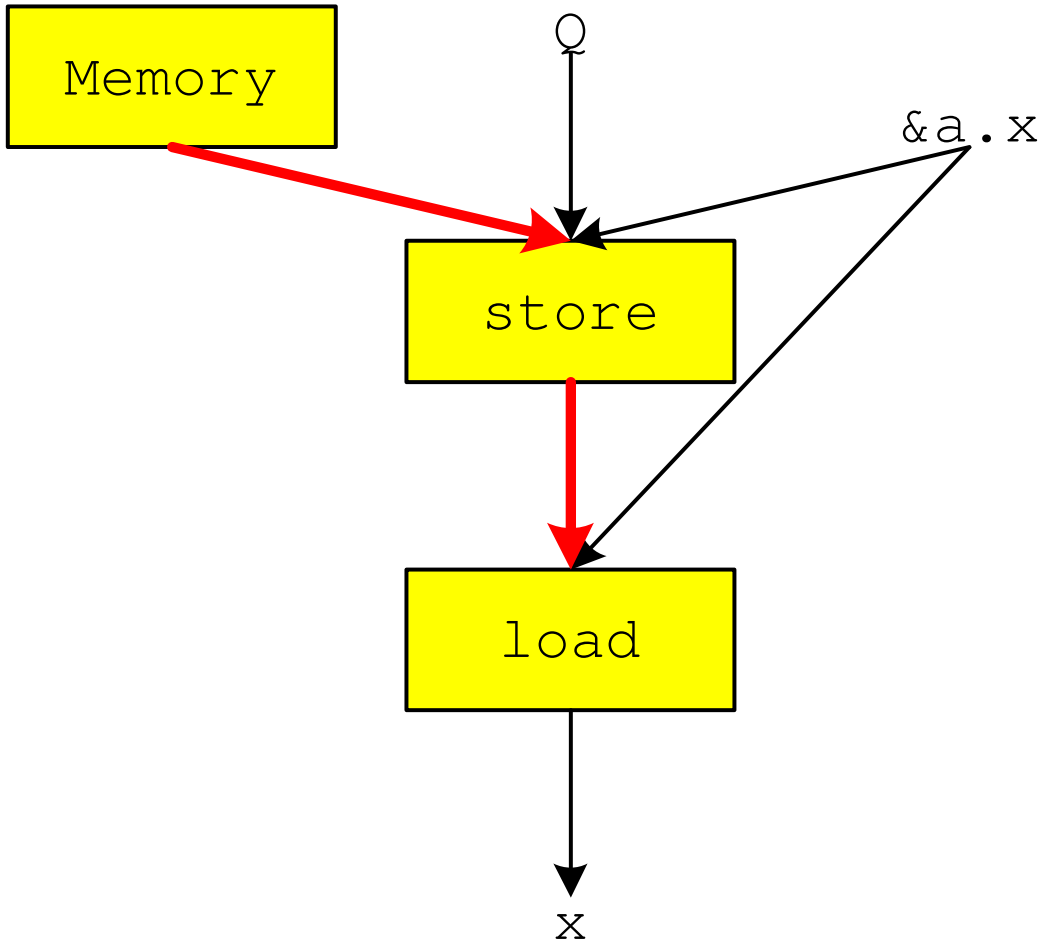
# Einfache Speicheroptimierungen – Alias Problematik

- Problem: Namen bezeichnen Zugriffspfade, keine (absoluten) Adressen
- Definition „Alias“:
  - unterschiedliche (Quelltext-)Namen, aber
  - gleiche (physikalische) Ressource (Adresse)
- Ursache:
  - Reihungen (`a[i]`, `a[j]`)
  - Referenzparameter
  - Zeiger, Referenzen, *address-of*-Operator `&`
  - *Call-by-Name*
- Folge:
  - Zugriffe auf Reihungen/Verbunde sind geordnet; Ordnung *nicht* mit bisherigem SSA-Aufbau darstellbar
  - Pessimistischer Ansatz: Totale Ordnung bei Speicherzugriffen

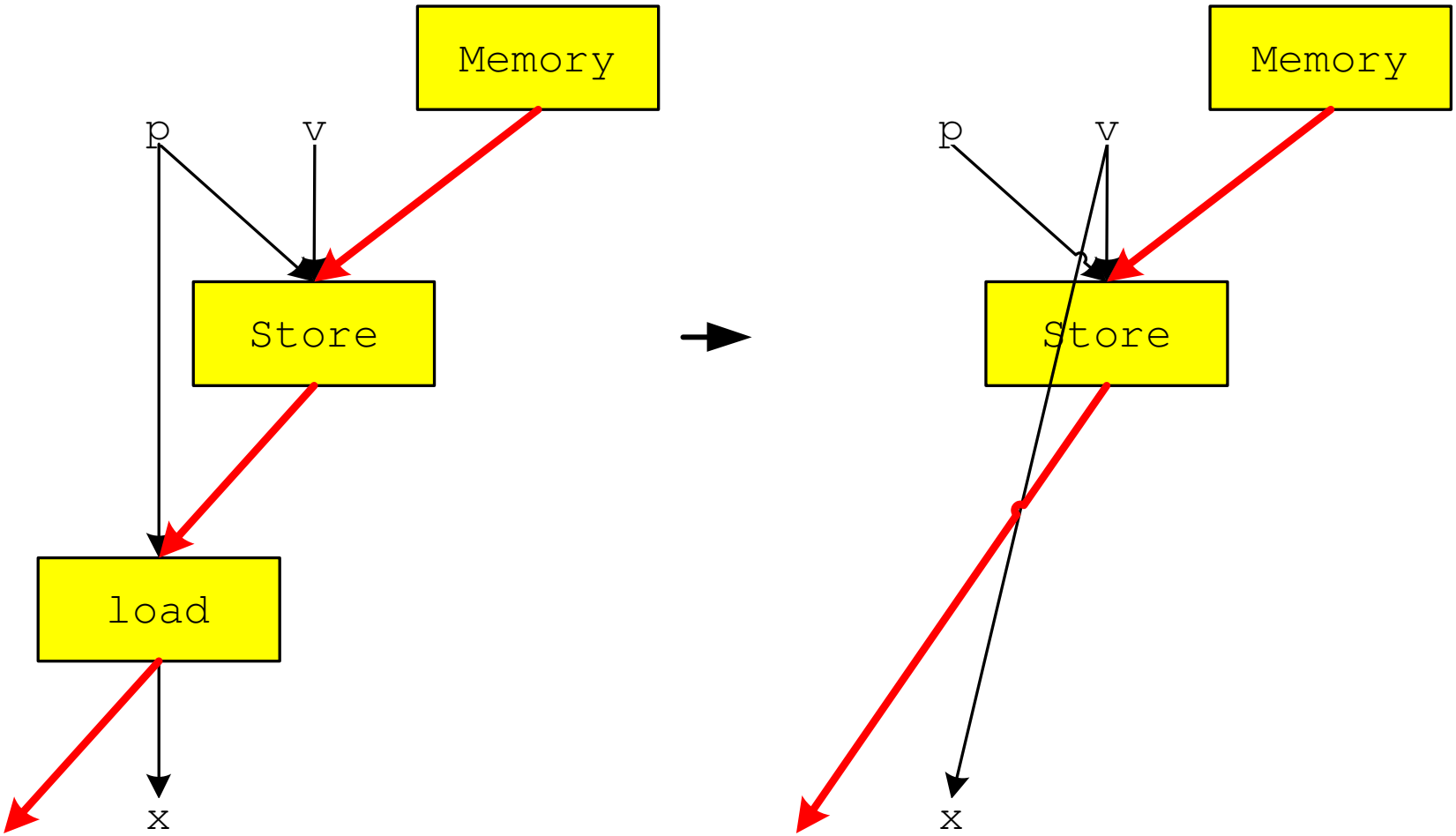


# Beispiel: SSA-Aufbau für Speicherzugriffe

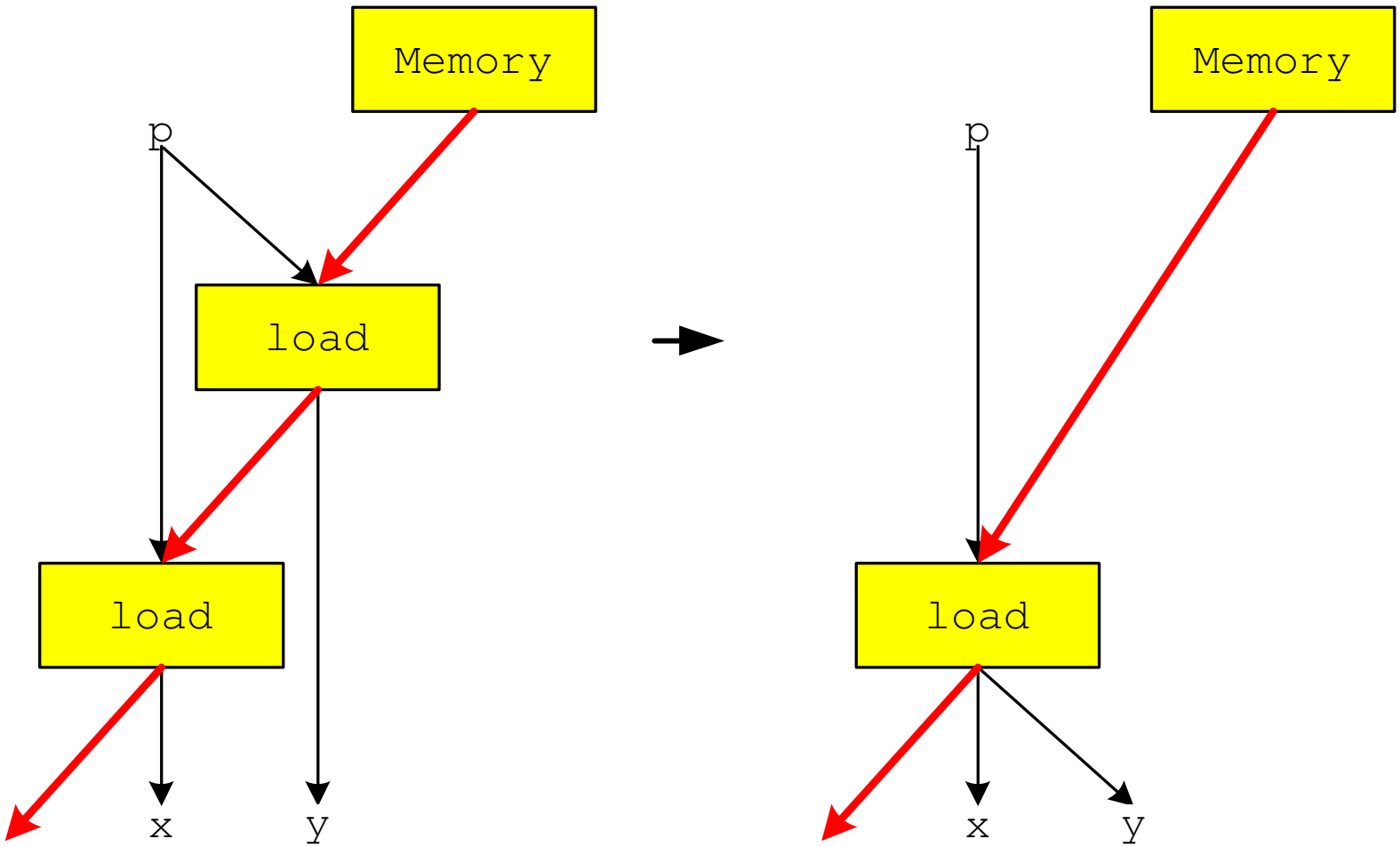
```
foo (A a, A b)
{
  int x;
  a.x = Q;

  x = a.x;
}
```



# Optimierungen - Lade-Speichere entfernen



# Optimierungen - Lade-Lade entfernen

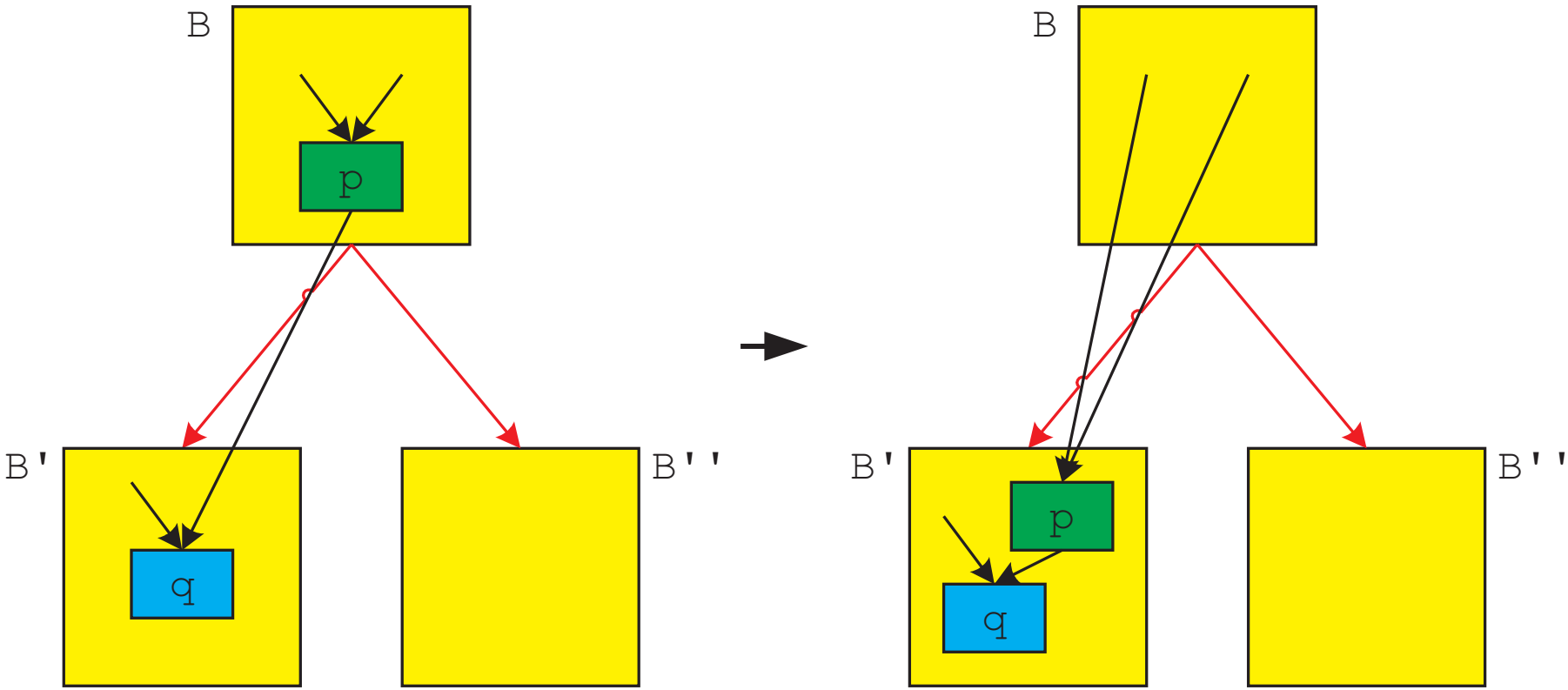


# Eliminieren unnötiger Berechnungen

- Problem:
  - Operation $p$ in Block $B$
  - Block $B$ dominiert Blöcke $B'$ und $B''$
    keine Dominanz-Relation zwischen $B'$ und $B''$
  - Wert von $p$ wird nur in $B'$ benutzt
  - Folge: Wert von $p$ wird auch berechnet, wenn später $B''$ ausgeführt wird
- Transformation:
  - Verschiebe $p$, so daß $p$'s Block keine Blöcke $B'''$ dominiert, in denen $p$ nicht benutzt wird
- Hinweis:
  - Eliminieren partieller Redundanzen entfernt Berechnungen die auf manchen Pfaden doppelt oder mehrfach stattfinden. Diese Optimierung hingegen entfernt Berechnungen, die auf manchen Pfaden gar nicht benötigt werden.
  - Es wird nicht beachtet ob Code in/aus Schleifen verschoben wird. Dies bietet weiteres Potential, kompliziert aber die Optimierung.



# Eliminieren unnötiger Berechnungen



# Eliminieren unnötiger Berechnungen

- Problem: Es gibt einen Ablaufpfad, auf dem $p$ berechnet, aber nicht benutzt wird.
- Wiederholung Nachdominanz: Block $B'$ wird von Block $B$ nachdominiert genau dann, wenn jeder Ablaufpfad, auf dem $B'$ liegt, auch $B$ enthält.
- Beobachtung:
  - Operation $p$ in Block $B'$, Operation in Block $B$ benutzt Ergebnis $q$ von $p$.
  - Wird Block $B'$ von $B$ nachdominiert, dann wird $p$ immer verwendet!
- Algorithmus zur Erkennung unnötiger Operationen:
  - Markiere alle SSA-Ecken als unnötig.
  - Für jede SSA-Ecke $p$ in Block $B$, betrachte deren Operanden $q_1, \ldots, q_n$ aus Blöcken $B_1, \ldots, B_n$:
    Wenn für ein $i \in [1 \ldots n]$ Block $B_i$ von Block $B$ nachdominiert wird, markiere $q_i$ als nötig (d. h. *nicht* unnötig).



# Eliminieren unnötiger Berechnungen

- Transformation unnötiger Berechnungen:
  - Für jede Benutzung $q_i$ einer unnötigen Operation $p$, erstelle Kopie $p_i$ von $p$, die nur von $q_i$ verwendet wird
  - Verschiebe $p_i$ entlang Dominatorbaum „zu $q_i$ hin“
  - Danach gemeinsame Teilausdrücke eliminieren
- Problem: Operationen könnten in Schleifen *hineinverschoben* werden
- Ausweg: Stoppe vor Block $C$, wenn $C$ Schleifenkopf ist (Verwende stark zusammenhängende Komponenten aus Induktionsanalyse)
- Ergebnis:
  - Operationen sind entweder nötig (also nicht unnötig), oder
  - Operationen sind unnötig, aber stehen nicht in Schleifen, innerhalb derer sie nicht neu berechnet werden
  - Vorsicht: Anwachsen des Codes möglich!



# Offener Einbau von Prozeduren - Idee

- In Prozedur $p$, ersetze Prozeduraufruf an $q$ durch den Rumpf von $q$ unter Ersetzung der formalen Parameter durch die Argumente des Aufrufs.
- Verwaltungsaufwand für Prozeduraufruf:
  - Ablage der Argumente durch den Aufrufer
  - Sichern des Kontextes des Aufrufers
  - Erstellen der Schachtel der aufgerufenen Prozedur
  - Abbau der Schachtel der aufgerufenen Prozedur
  - Wiederherstellen des Kontextes des Aufrufers
  - Rückgabe des Ergebnisses an den Aufrufer
- Wird $q$ an allen Aufrufstellen offen eingebaut, dann ist $q$ redundant
- Optimierung des offen eingebauten Codes im Kontext der Aufrufer



# Offener Einbau von Prozeduren - Vorgehen

- Gieriges Einfügen nicht sinnvoll
  - Auch ohne Rekursion exponentielles Anwachsen des Codes
  - Mit Rekusion keine Terminierung
- Profitabel für
  - Prozeduren mit einem oder nur wenigen Aufrufern
  - Prozeduren, die oft mit konstanten Argumenten aufgerufen werden
  - Blattprozeduren
  - Aufrufe in Schleifen
  - (Rekursive Prozeduren mit geringer Rekursionstiefe)
- Entscheidung:
  - Definiere Kostenmaß $I$ auf Prozeduraufrufen (Heuristik)
  - Definiere Maximalwert $I_{max}$
  - Wenn $I(p, q) < I_{max}$ ist, dann wird $q$ offen in $p$ eingebaut
- Wenn offener Einbau nicht möglich, dann wenigstens *Tail-call*- oder eingeschränkt *Tail-cecursion*-Optimierung (in SSA schwierig)



# Zusammenfassung

- Viele Optimierungen, jedoch auch mit einer kleinen Auswahl kann viel erreicht werden (SSA-Auf/-Abbbau, Operatorvereinfachung, CSE, PRE).
- Durch SSA-Form werden viel Analysen (Datenfluß) überflüssig bzw. trivial. Optimierungen lassen sich leichter ausführen.
- Bei der Operatorvereinfachung werden teuere (z.B. `mult`) duch billigere (z.B. `add`) Operationen ersetzt.
- Beim gemeinsame Teilausdrücke Eliminieren (CSE) werden die Werte von Berechnungen wiederverwendet.
- Das Eliminieren partieller Redundanzen (PRE) verschiebt Berechungen in Ablaufvorgänger. Dabei sollen die Berechnungen nur genau einmal erfolgen, und ihr Ergebnis auf allen Pfaden gebraucht werden.
- Hauptvorteil der SSA-Form: Nicht nur Ausgangspunkt, sondern auch einheitliches Ziel der Optimierungen