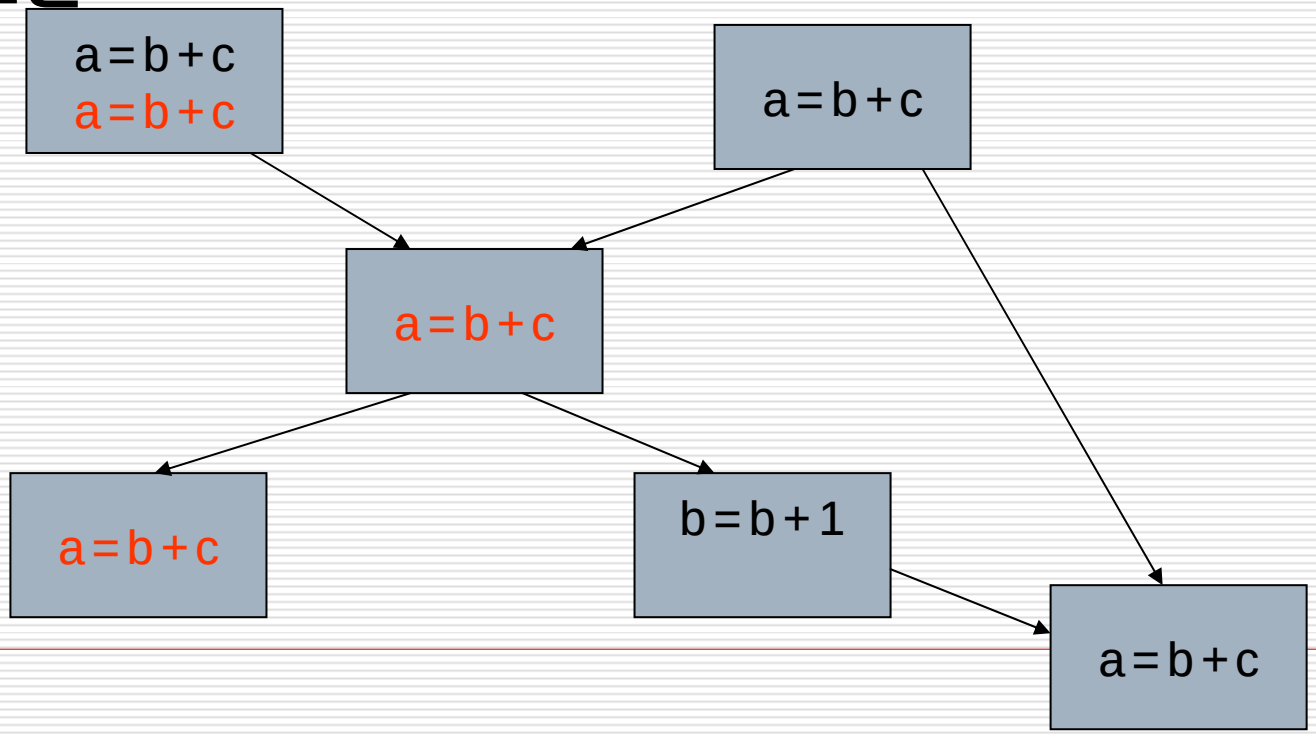
プログラミング言語処理系論 (9)
Design and Implementation of Programming Language Processors

佐藤周行
（情報基盤センター/電気系専攻融合情報学コース）

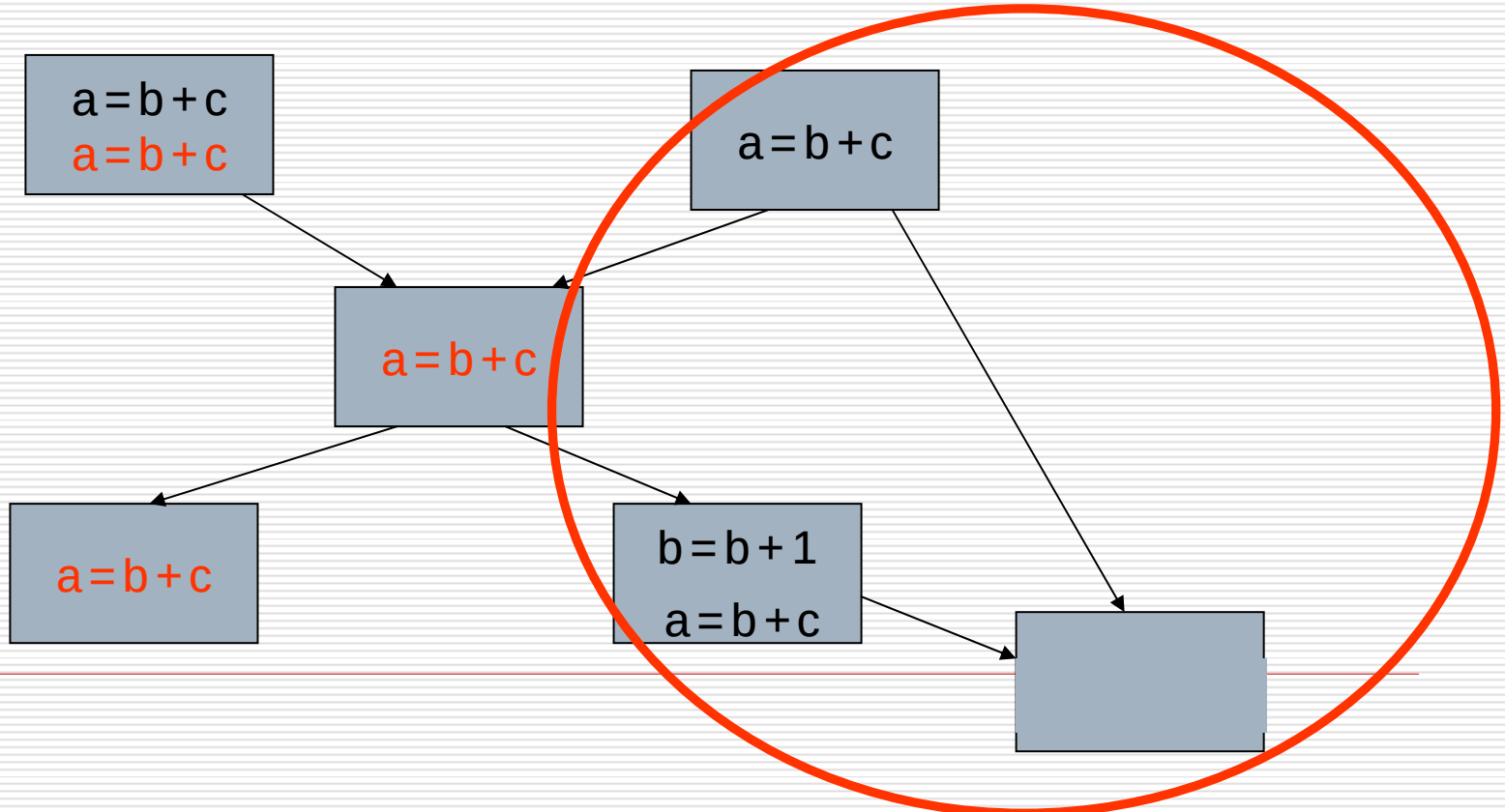
(Partial) Redundancy Elimination

- Def. Redundancy Elimination
  計算のパスにおいて、重複する計算を取り除くこと



- Def. Partially Redundant
  式がプログラムポイントでpartially redundantであるとは、そこにいたるパスの中でredundant expressionな式であるものが存在すること(すべてのパスである必要はない)
- たとえば、ループ不変式はpartially redundantである

Partial Redundancy



- Def. Lazy Code Motion
  Partially Redundant Eliminationのためにコードを移動すること

- Knoop, et.al., Lazy Code Motion, Programming Language Design and Implementation ’92, 1992.

- Def. Anticipable Expressions
  式(x op y)がanticipableとは、その計算が後続のパスのどこかでなされ、しかもそのパス中にx, yの定義が入らないこと(計算を先に実行してもよい)。
- $ANTout[n] = \bigcap_{s \in n\text{のsuccessor}} ANTin[s]$

  $ANTin[n] = Defed[n] \cup (ANTout[n] - kill[n])$

Lazy Code Motion

- Def. Earliest(i,j)　(i,j – ブロック)
  (1)ブロックjの先頭まで移動でき、
  (2)ブロックiの終端ではavailableでなく(よって、挿入してもredundantにならない)、
  (3)ブロックi中で無効になるか、iから出る他のパスのせいで、先頭に移動できない式の全体　→　エッジ(i,j)に挿入できる一番早いステージ

- Earliest(i,j) = ANTin(j) – AVAILout(i)
  ∩ (Kill(i) ∪¬ ANTout(i))

- Def. e ∈LaterIn(k)
  kに到達するパスすべてについて、そのどこかでeがEarliestであり、そこからkまで、eを評価していない。(eの計算はkから前に移すことができるけど、Earliestよりは後ろで計算できる)
- Def. Later(i,j)
  Earliestであるか、またはiから後ろに移動でき、かつiで計算され、iの先頭に移動できない式の全体
- Later(i,j) = Earliest(i,j)∪(LaterIn(i)–Defed’(i))
- $LaterIn(j) = \bigcap_{i \in predecessor(j)} Later(i,j)$

- INSERT(i,j) = Later(i,j)– LaterIn(j)
- Delete(k) = Defed(k) – LaterIn(k)

- (課題13) KnoopらのLazy Code Motion の論文を読んでサマリーを書け。

ここで反省

- データフロー方程式で集める情報の基本はDefinitionとUseに関するもの。
- Def Def/Use Chain
  - 変数のDefinitionとUse関係
- 似たような事を何回もやっている。
- Def/Use Chainを効率的に表現できる中間表現があれば、この種の最適化も効率的にできるはずである。

今回の予定

- SSAとSSAを用いた最適化

データフロー解析で困ったこと…

- Reaching definitionってそもそも方程式をたてなければわからないものか？
- 同じ変数にとっかえひっかえ値を代入すると、冗長性の解析でも無駄に複雑度が増す

- 中間言語ならば、変数の数は任意に取れるから、使い回しを気にする必要はない
- 関数型言語は、ここらへんの苦労がないから最適かも楽だよなぁ…

- IBMが中間言語としてSSA (Static Single Assignment)を提唱
  - Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, Oct 1991.
  - 開発は1980年代といわれている
  - ある人が評していわく(出典未詳。うそかもしれない)「SSAはIBMが作った関数型言語だ！」

定義

- Def.　Static Single Assignment Form
  プログラムの文面上、各変数に対して定義がひとつしか存在しないもの

- 注意：プログラムの文面上の話だから、同じ文が何回も実行されて、定義が複数回行なわれることは禁止しない

例

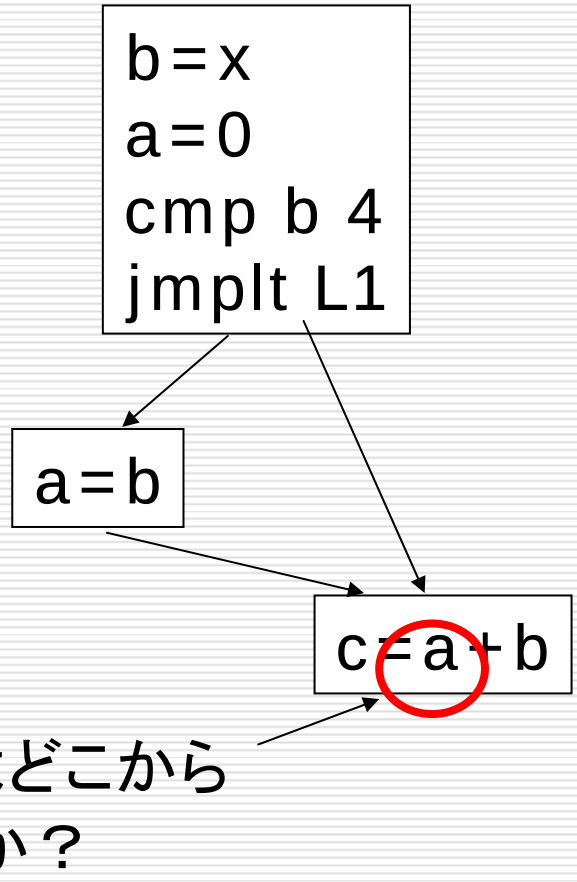
```
a=x+y
b=a-1
a=y+b
b=x*4
a=a+b
```

```
a1=x+y
b1=a1-1
a2=y+b1
b2=x*4
a3=a2+b2
```

(変数の名前の書き換え)

どうしても対応するSSAがない例

```
  b=x
  a=0
  cmp b 4
  jmplt L1
  a=b
L1:
  c=a+b
```



言語の拡張

- 今までの言語に以下を追加する

INSN : : = …

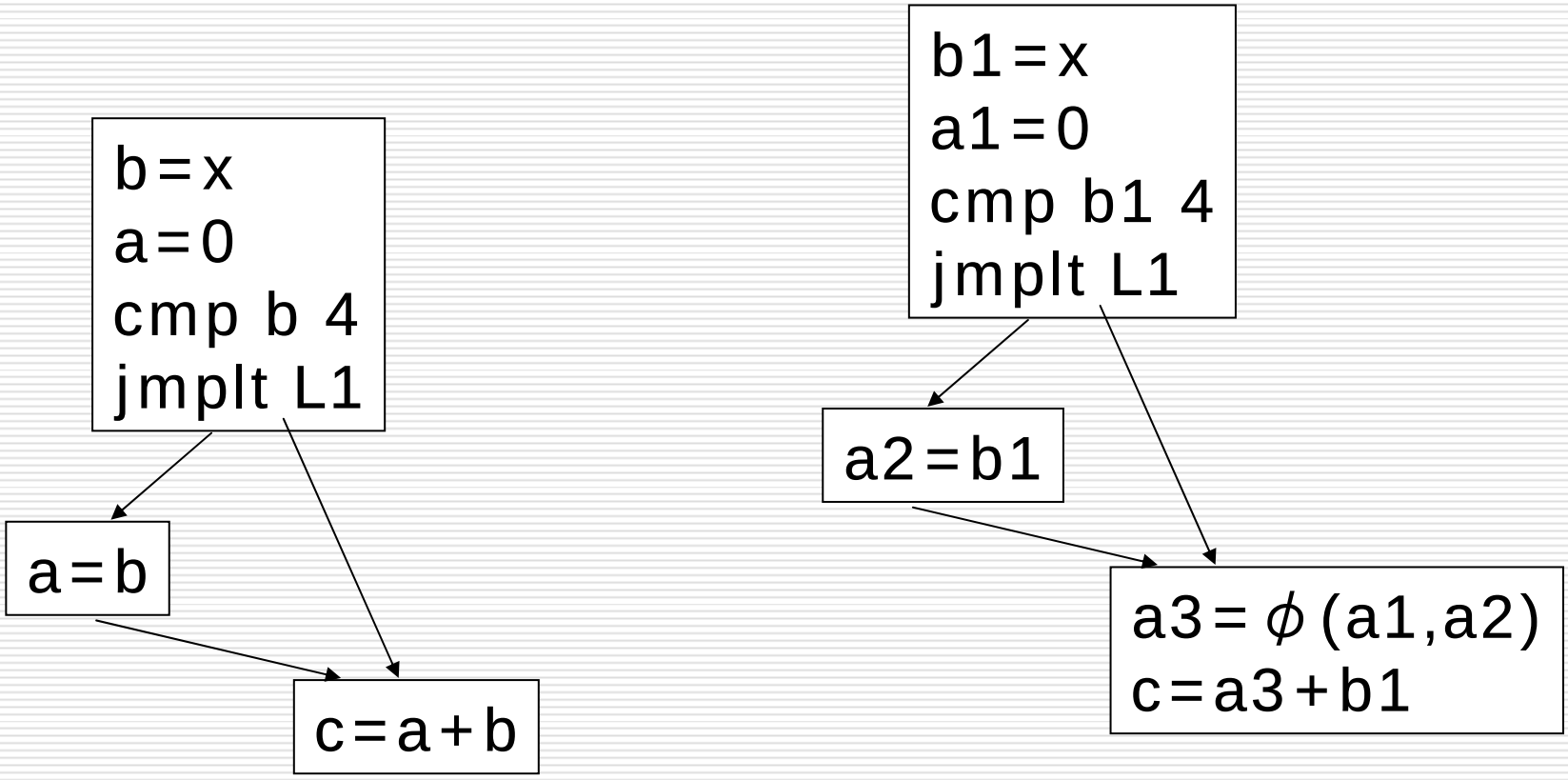
ϕ (var ‘,’ var+)

Φ（pseudo）関数は、以下の意味をもつとする。

x = ϕ (x1,x2)

制御がx1の定義を通ってこの場所に来た場合はx1を、
そうでない場合はx2を表す。

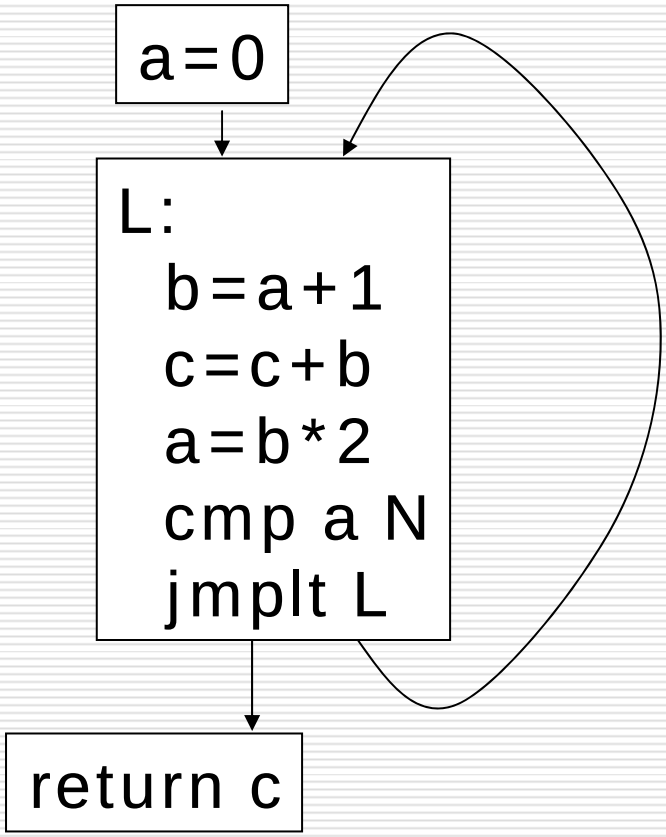
Φを使っての解決

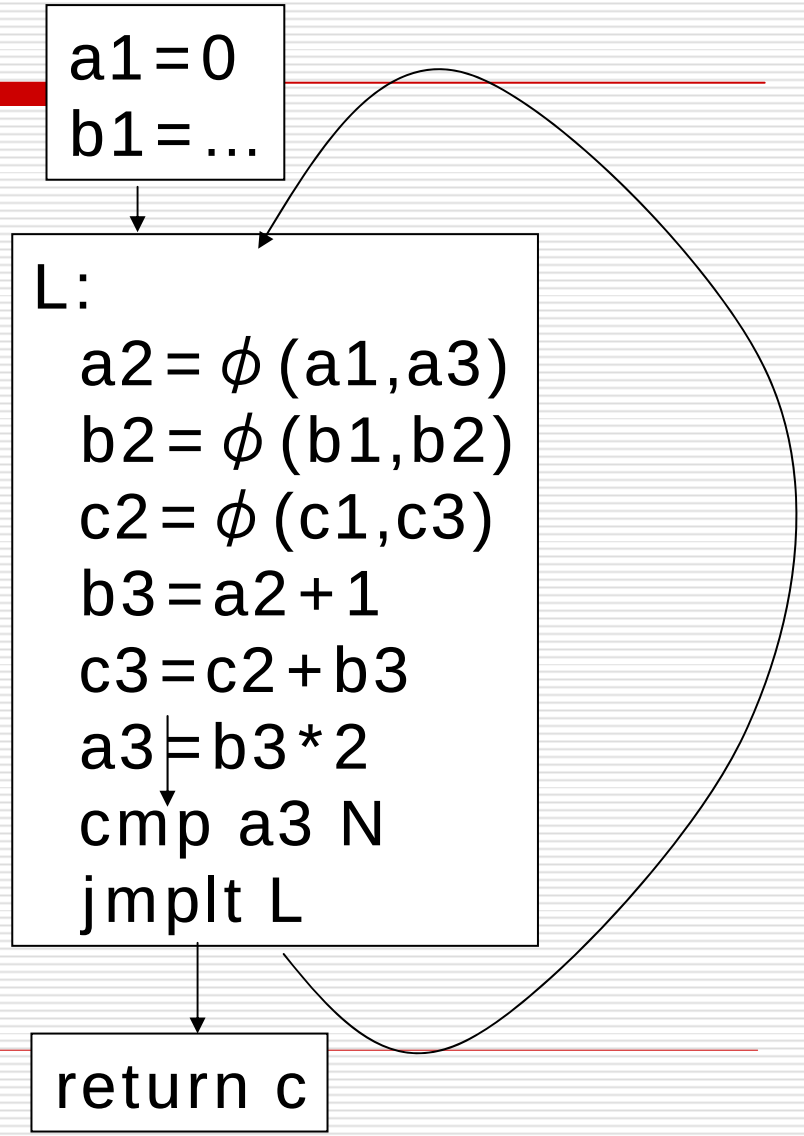

b=x
a=0
cmp b 4
jmplt L1
a=b
c=a+b
b1=x
a1=0
cmp b1 4
jmplt L1
a2=b1
a3=φ(a1,a2)
c=a3+b1

SSAの利点

- Def/Use関係の簡潔な表現 → さまざまな解析の簡潔化
- データフロー解析の大域化（killされるものがないので、ローカルなものとグローバルなものを分ける必要がなくなる） → ローカルな最適化の大域的なところでの適用（e.g. global value numbering）

特にループを考える


a=0
L:
b=a+1
c=c+b
a=b*2
cmp a N
jmplt L
return c


a1=0
b1=…
L:
a2=φ(a1,a3)
b2=φ(b1,b2)
c2=φ(c1,c3)
b3=a2+1
c3=c2+b3
a3=b3*2
cmp a3 N
jmplt L
return c

- 基本的な要請：
  （1）プログラムのフローは両者で同じでなければならない
  （2）計算の終了後、両者の計算の結果は、変数名を除いて同一でなければならない
  （3）任意のSSA形式は、φを持たないプログラムで、同一の効果を持つものを持たなければならない

SSA変換・SSA逆変換

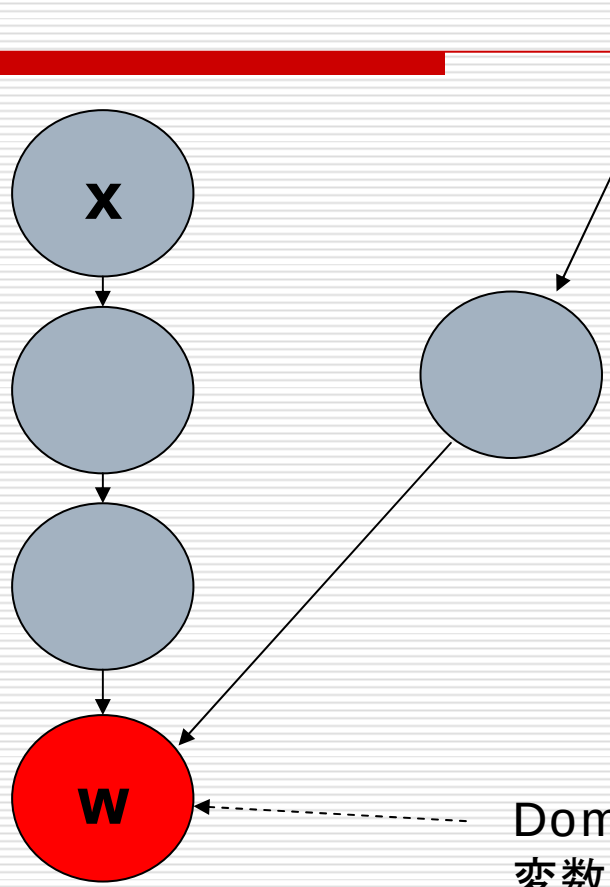
- SSA形式に変換するとは、φをどこに挿入するかに（だいたい）帰着される。

- 変数が（データフロー上）合流する点にφを挿入すればよい。
- 変数aのφ関数をブロックzに挿入するための方針：
  （1）zにおいて、変数aのreaching definitionsが複数あり、
  （2）zにおいてそれら複数の定義が初めて合流する

効率的な方法

- Def. x strictly dominates w if x dominates w and $x \neq w$
- Def. Dominance Frontier of a node x set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.
- $DF[x] = \{z \mid \exists y1,y2 \in \text{predecessor}(z).$ x dominates y1, and x does not dominate y2.$\}$

Illustrated



Dominance frontier
変数のデータフローの合流点になる

Dominatorの性質について

- x dominates y という関係はtreeを作る（x dominates y, and z dominates y → x dominates z or z dominates x)

（証明）x dom y , z dom y, not (x dom z), not (z dom x)とする。zに行くパスでxを通らないものp1が存在する。xに行くパスでzを通らないものp2が存在する。yに行くパスqを任意に取る。xとzの出現はそれぞれ1回として一般性を失わない。Q中xがあとに出現したら、そこまでのパスをp2に(反対の場合はp1に)置き換えると、z(x)を通らずにyに行くパスができるので矛盾する。

- Dominance Frontierの計算
- Def. DF[n]　　n– node
- DF[n] = {z|zはnのsuccessor, かつnにdominateされていない}
  ∪
  ∪ DF[c]
  idom(c) = n

- DFは、次のようにして効率的に計算できる

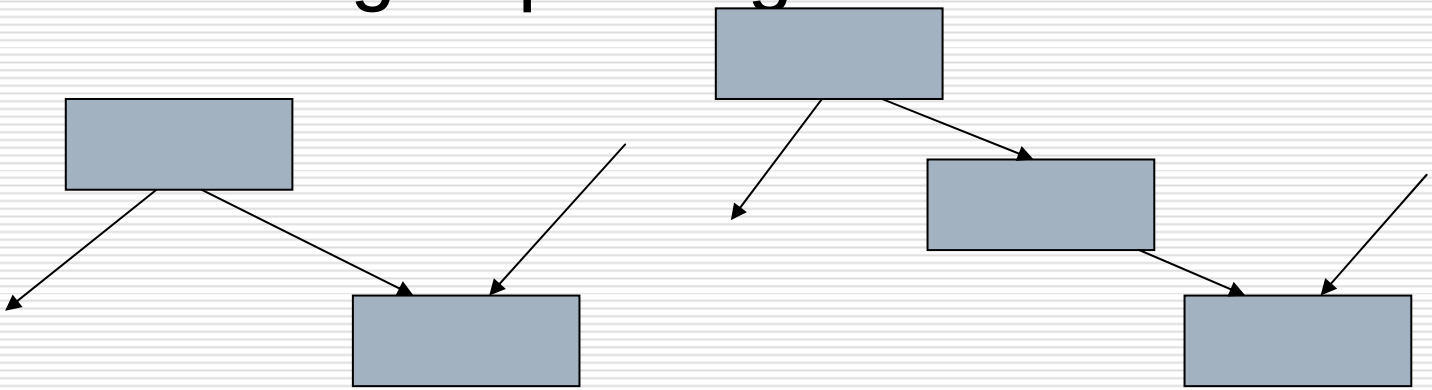
```
S = φ
for each node y in successor[n]
    if (idom(y) ≠ n)
            S = S ∪{y}
for each child c imm. dominated by n
    compute DF[c]
    for each element w ∈ DF[c]
            if n does not dominate w
                    S = S ∪{w}
DF[n] = S
```

ϕの挿入(Last Stage)

```
for each node n
    for each variable a (defined in n)
        defsite(a) = defsite(a) + n
for each variable a s.t. defsite(a) ≠ φ
    for each n ∈ defsite(a)
        for each Y ∈ DF[n]
            insert a=φ(a,a,...)
```

残りの仕事

- Renaming variables.
  - Dominator treeを上から下に下りていって、変数名をx → xnの形にしていく(どのdefがreaching definitionであるかは簡単にわかる)。
- Critical Edge splitting.

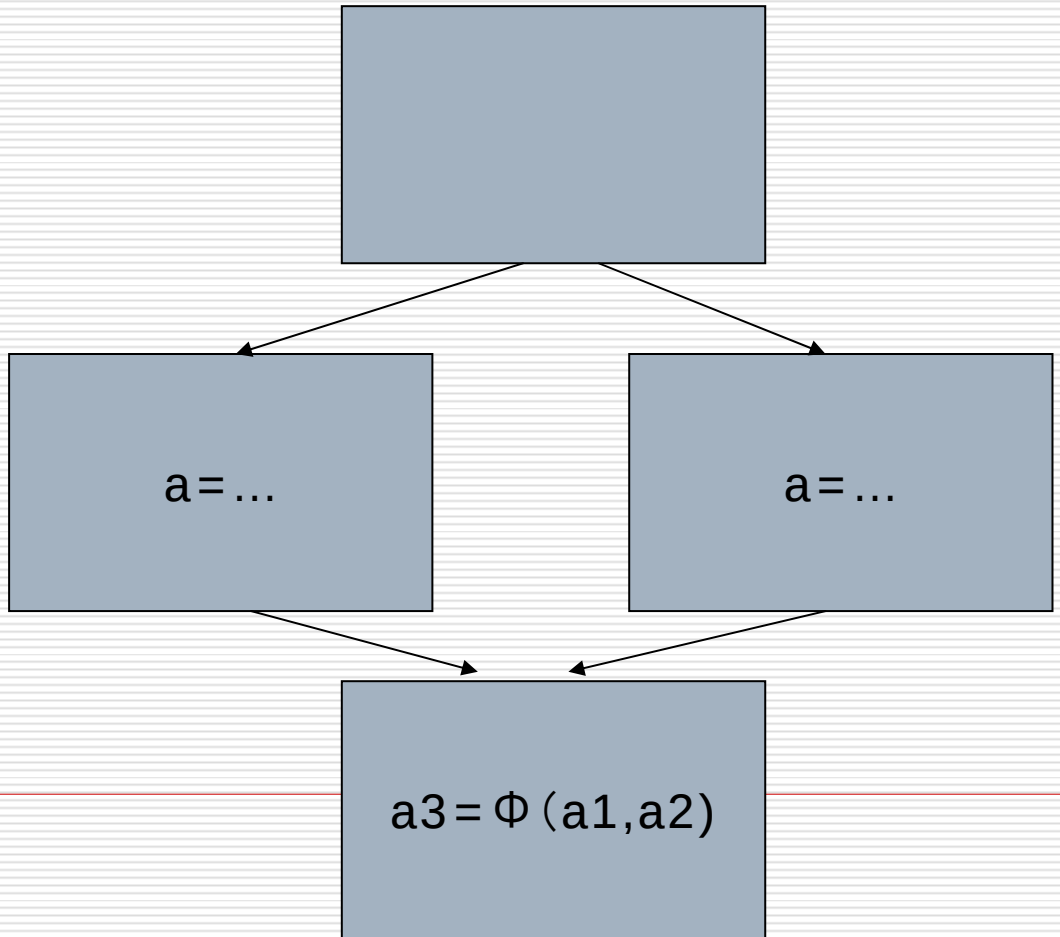


- Critical edge
  - これがあると、SSA逆変換がとたんに難しくなる
  - これがあると、Lazy Code Motionができない

より簡単なSSA変換

- 構造化されたプログラムでは、より簡単な方法でSSA変換ができる。
  - Loop (do loop, while loop, ...), If, compoundからなるプログラミング言語を考える（Javaの大部分のもの、よく整理されたCプログラムはこの範疇に入る）。

例えばif文



- (課題14) 自分で適当にloop, if, compoundを持つ簡単なプログラミング言語を想定し、フローグラフのパターンを決め、ϕの挿入場所を特定せよ。

SSAを用いた最適化

- われわれは、SSAを作るときに以下のものを同時につくることができる。
  - DEF/USE chain and USE/DEF chainを、DEFひとつに対するUSEの集合として管理できる。

- Dead Code Elimination
  - v= x op y において、vのUSEが0であれば、この文は削除できる。
- Constant Propagation
  - v = x において、xのDEFが定数 x = cであればこれをv = cに置き換えることができる。
  - v = φ(c1,c2,…,cn)で、c1,…cnがすべて同じであればv=c1に置き換えることができる。

- Copy Propagation
- Constant Folding
- Constant Conditions によるjump文の最適化
- Unreachable Codeの除去による unreachable blockの除去