

ENGINEERING A COMPILER

9.3.4 – 9.3.6

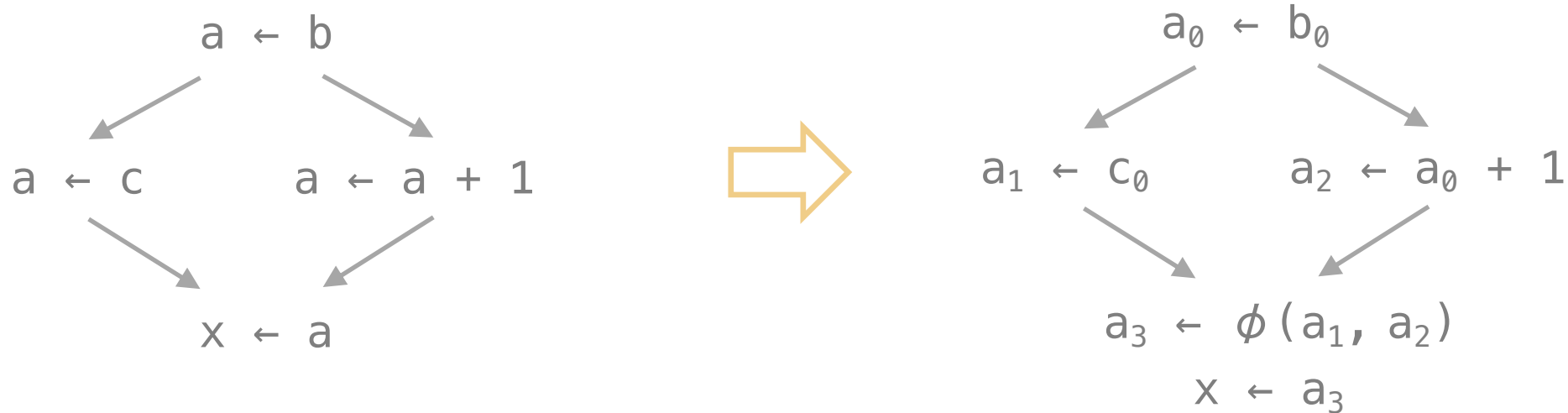
山本 航平

簡単な復習

SSA (Static Single-Assignment) 形式のルール :

1. 手続き内の各計算は一意の名前を定義する
2. 手続き内の各使用は単一の名前を参照する

同じ変数には静的に一度しか代入されない

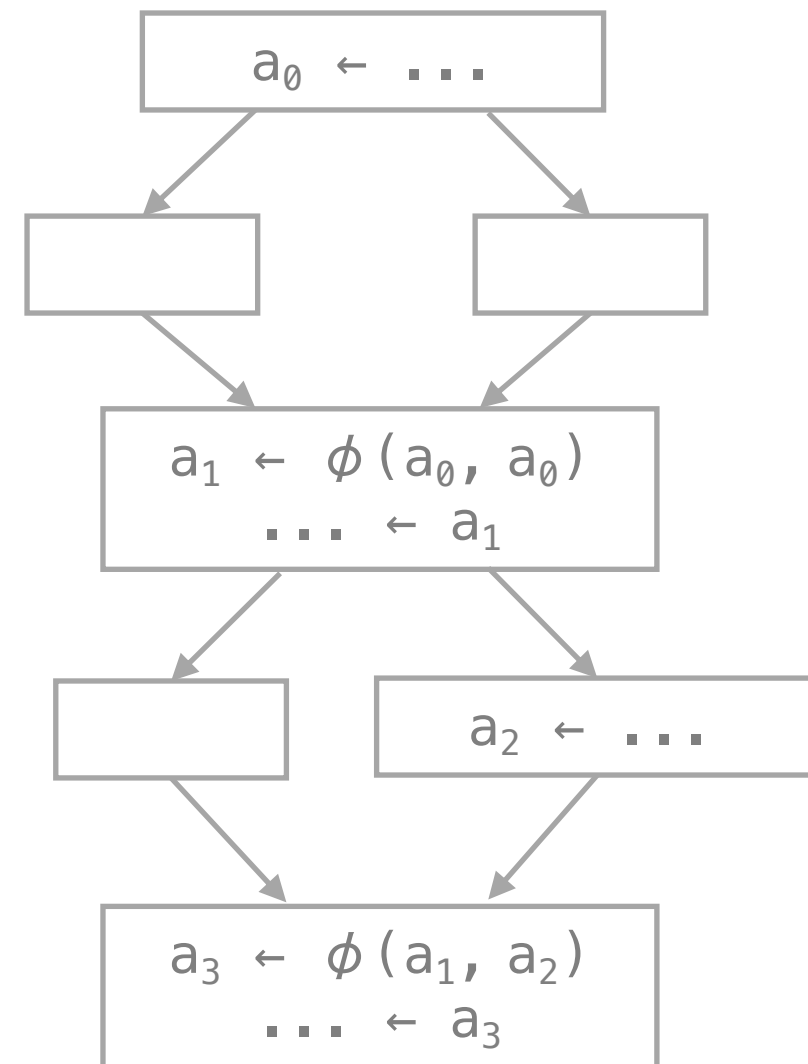


簡単な復習

9.3.1 : A Naive for Building SSA Form

- ・ CFG の合流点すべてに ϕ 関数を挿入する

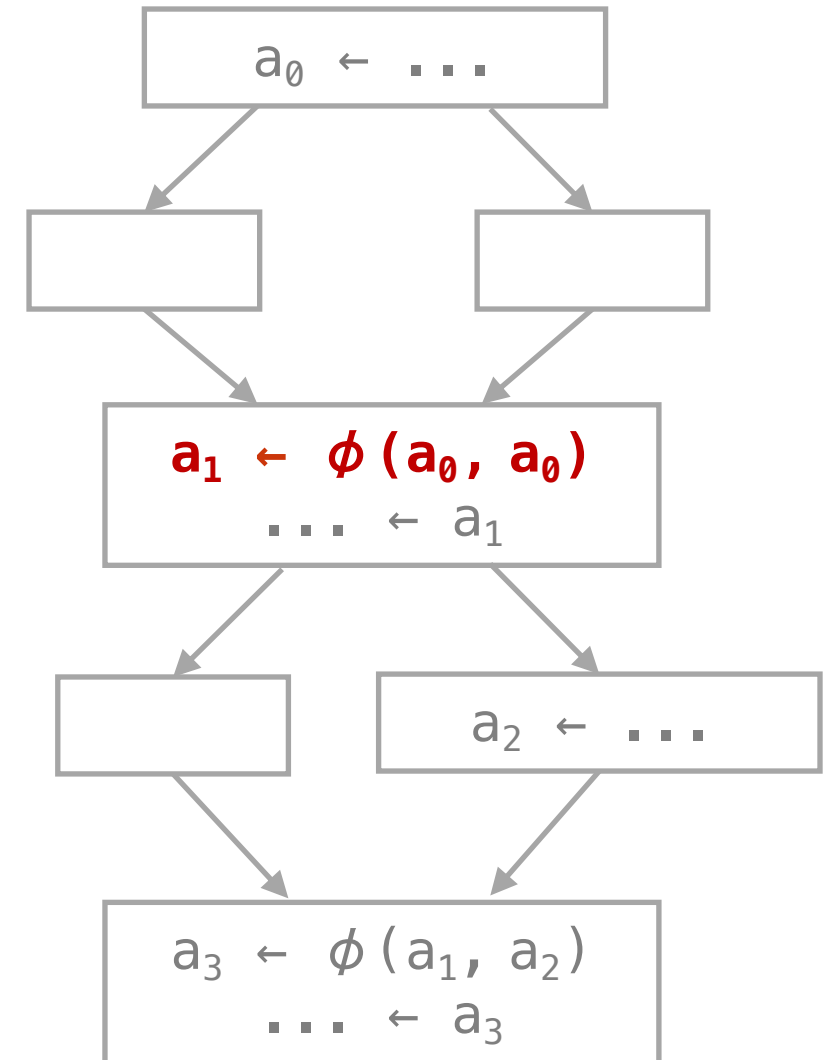
→ 無駄な ϕ 関数が多く挿入されるため
必要な箇所だけ挿入したい



簡単な復習

9.3.1 : A Naive for Building SSA Form

- CFG の合流点すべてに ϕ 関数を挿入する
→ 無駄な ϕ 関数が多く挿入されるため
必要な箇所だけ挿入したい

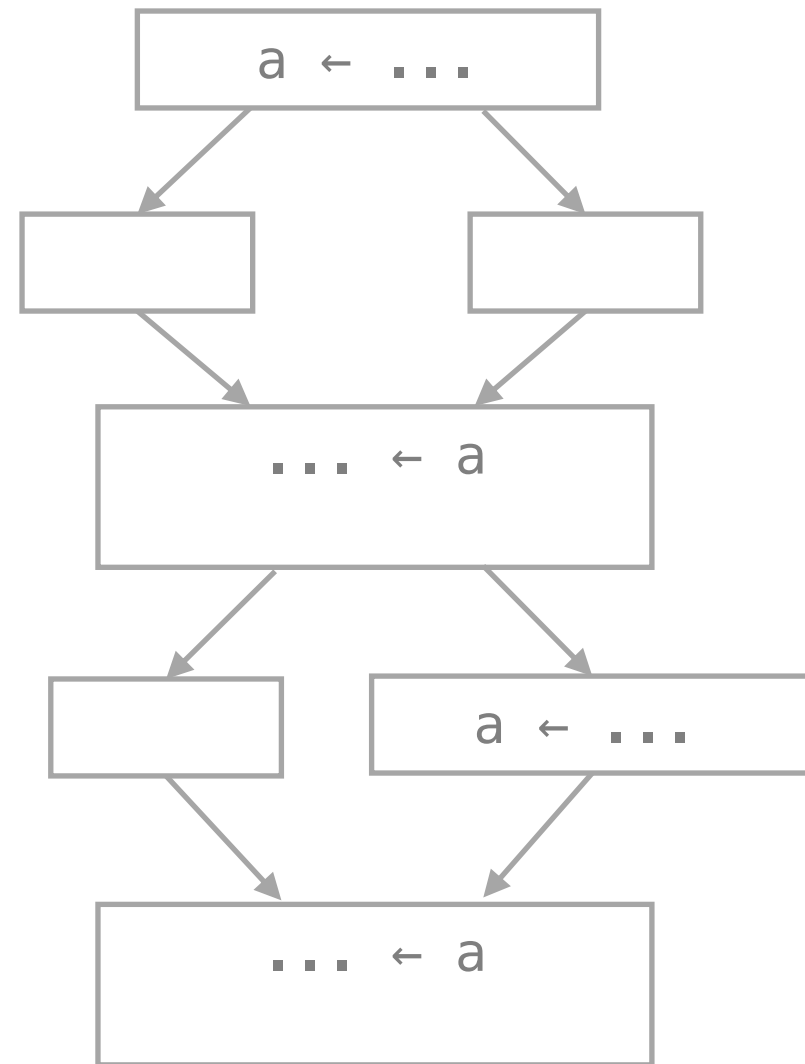


簡単な復習

支配関係を考慮した SSA を構築する

9.3.2 : Dominance Frontiers

- CFG の支配関係から, ϕ 関数が必要な基本ブロックを特定する





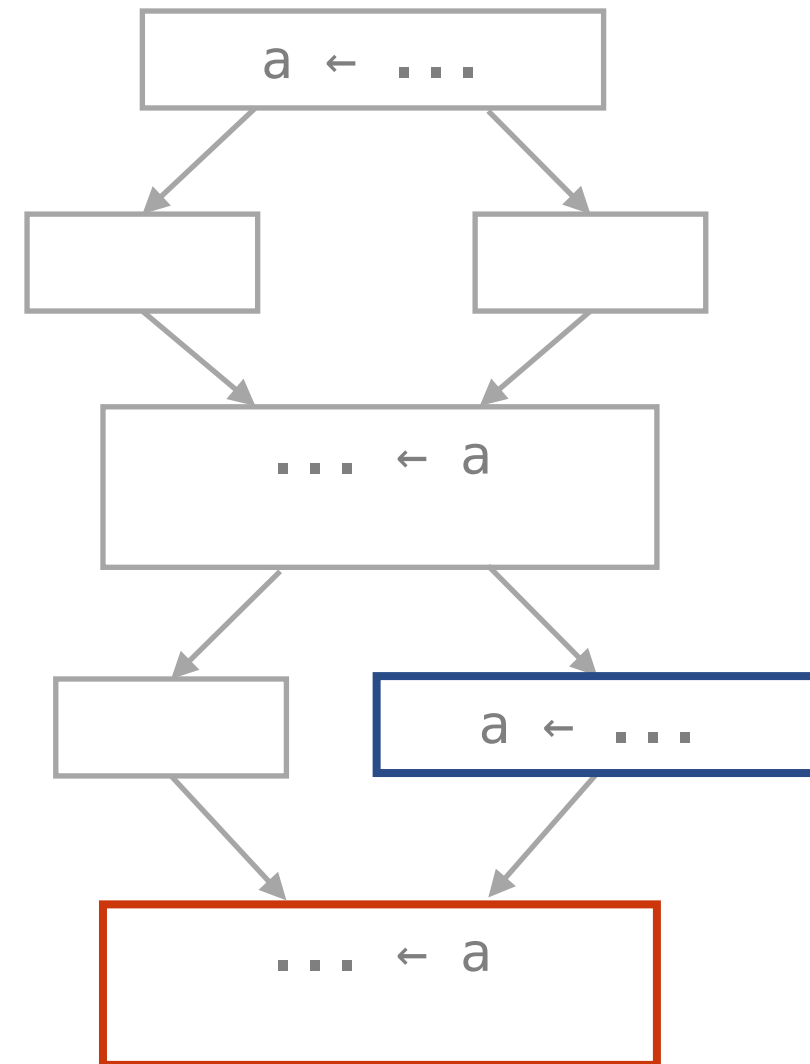
簡単な復習

支配関係を考慮した SSA を構築する

9.3.2 : Dominance Frontiers

- CFG の支配関係から, ϕ 関数が必要な基本ブロックを特定する

例 :  で定義された変数は
 で ϕ 関数が必要



簡単な復習

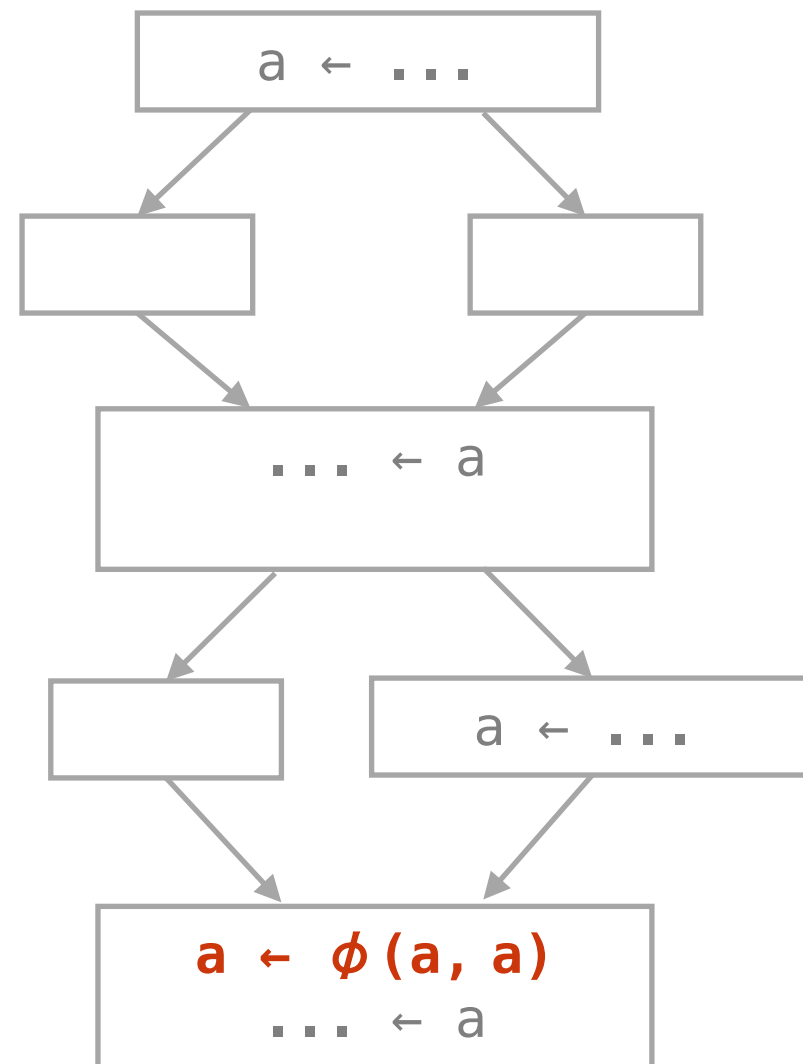
支配関係を考慮した SSA を構築する

9.3.2 : Dominance Frontiers

- CFG の支配関係から, ϕ 関数が必要な基本ブロックを特定する

9.3.3 : Placing ϕ -Functions

- DF集合に基づいて ϕ 関数を挿入

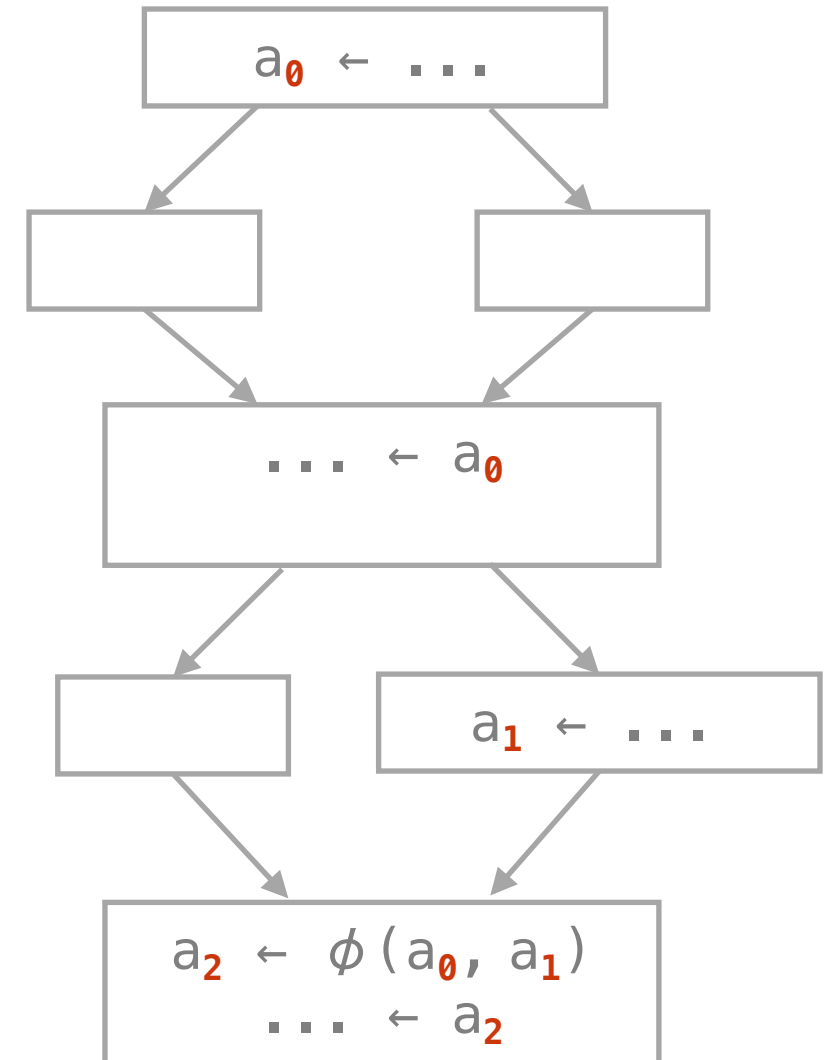


今回やること

支配関係を考慮した SSA を構築する

9.3.4 : Renaming

- ・ 変数名の変更



今回やること

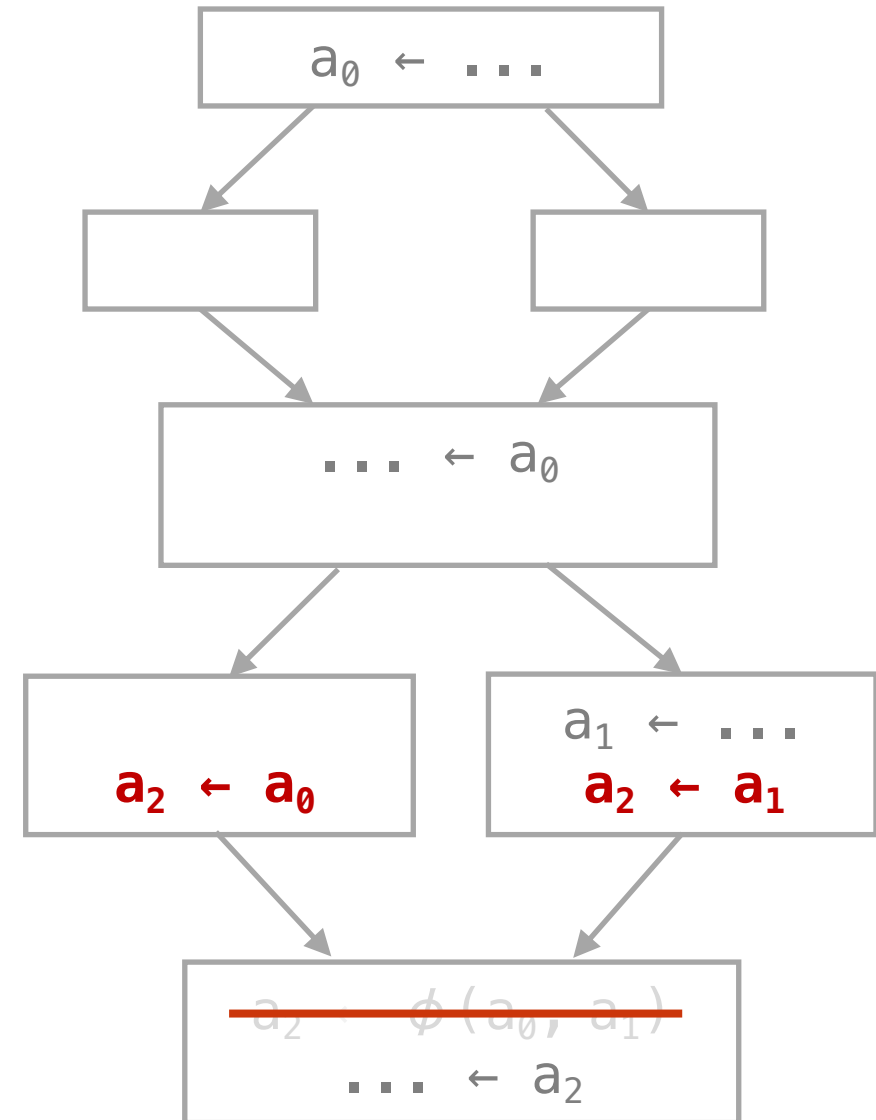
支配関係を考慮した SSA を構築する

9.3.4 : Renaming

- ・変数名の変更

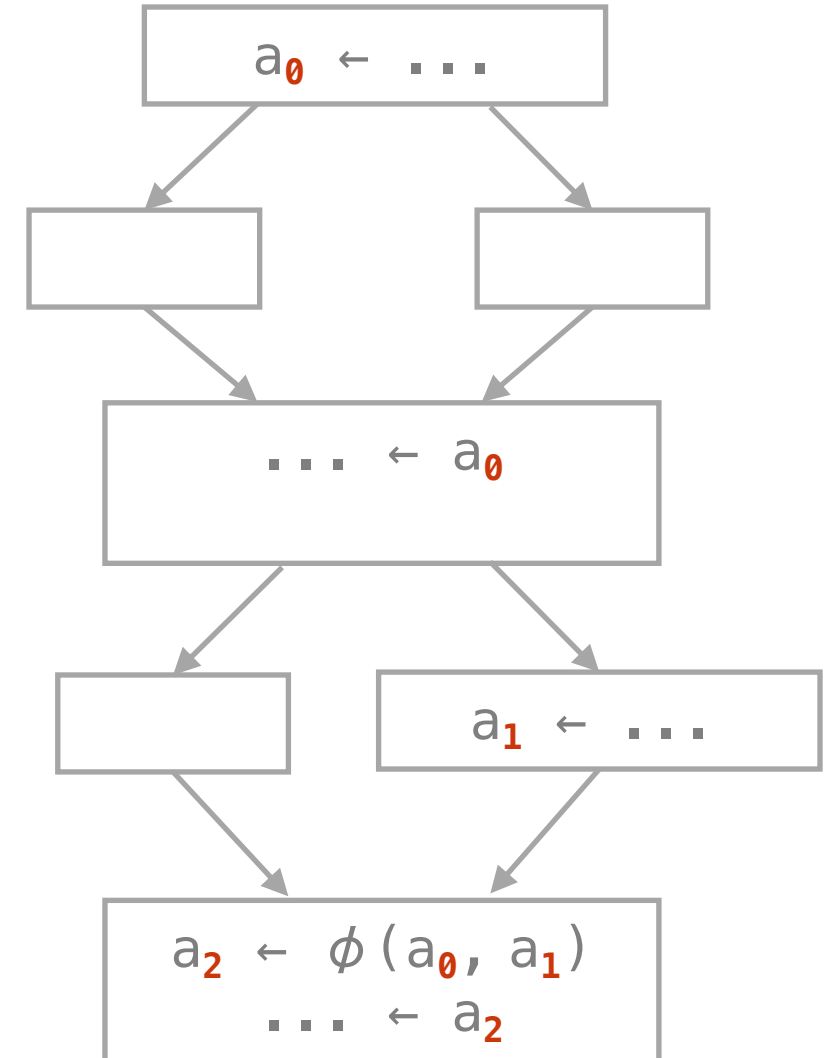
9.3.5 : Translation out of SSA Form

- ・SSA を通常の形式に変換
- ・ ϕ 関数のない形にコードを変換
(コンピュータは ϕ 関数をそのまま実行できない)



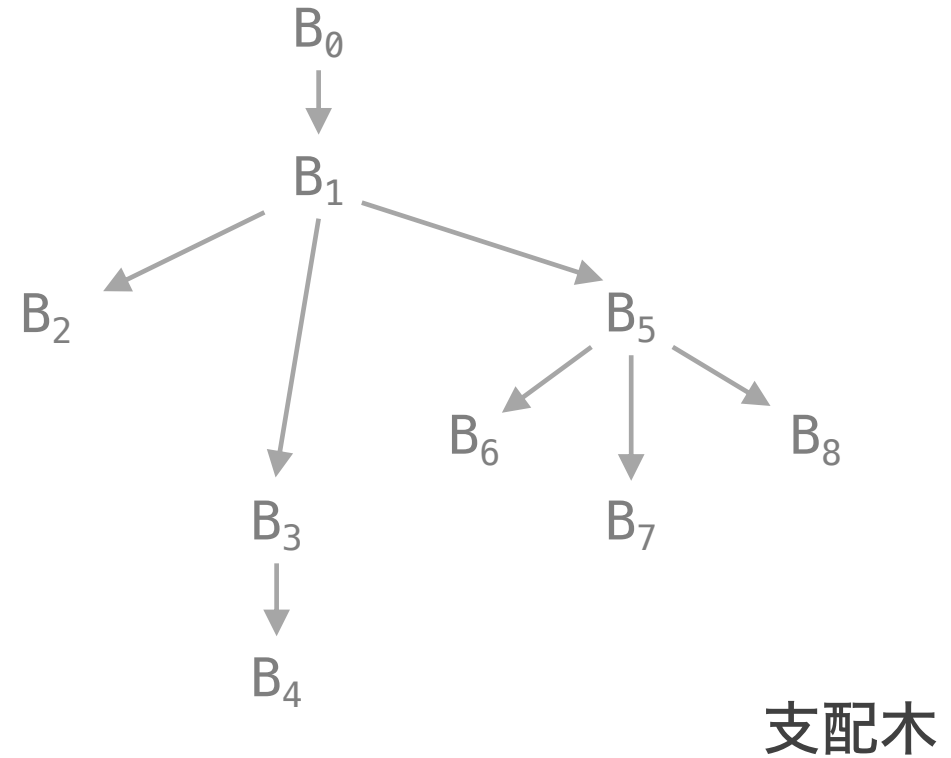
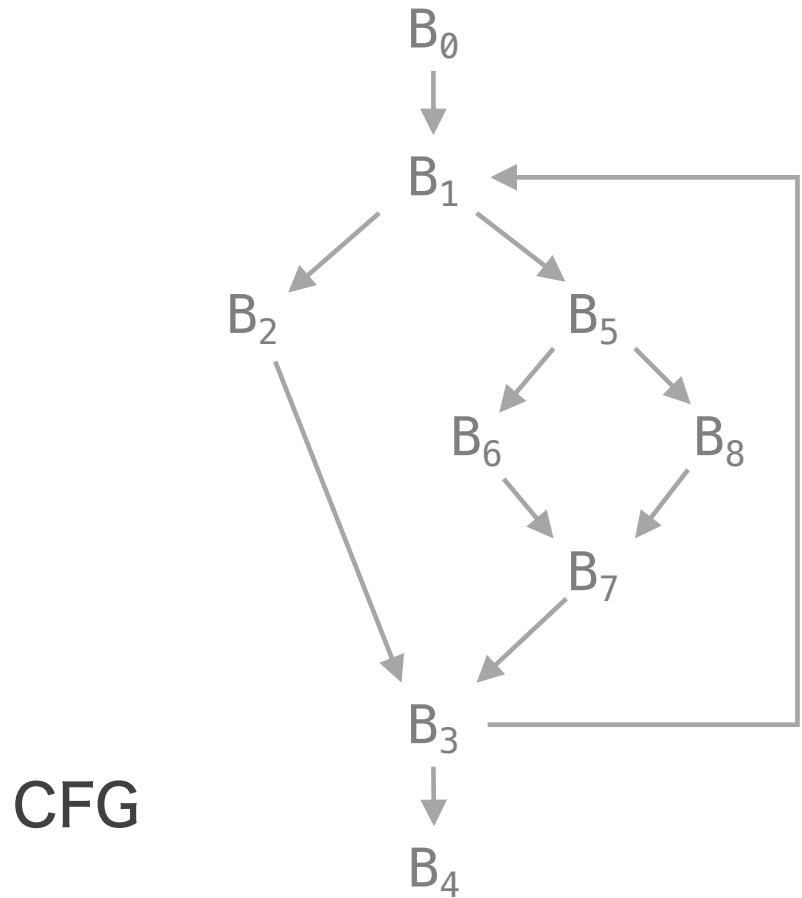
9.3.4 Renaming

- ・ 変数名の変更
- ・ 元々の変数名をベース名として
 $0 \rightarrow 1 \rightarrow \dots$ と添え字をつける



Renaming アルゴリズム

- 支配木を preorder で走査



Renaming アルゴリズム

- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加

Renaming アルゴリズム

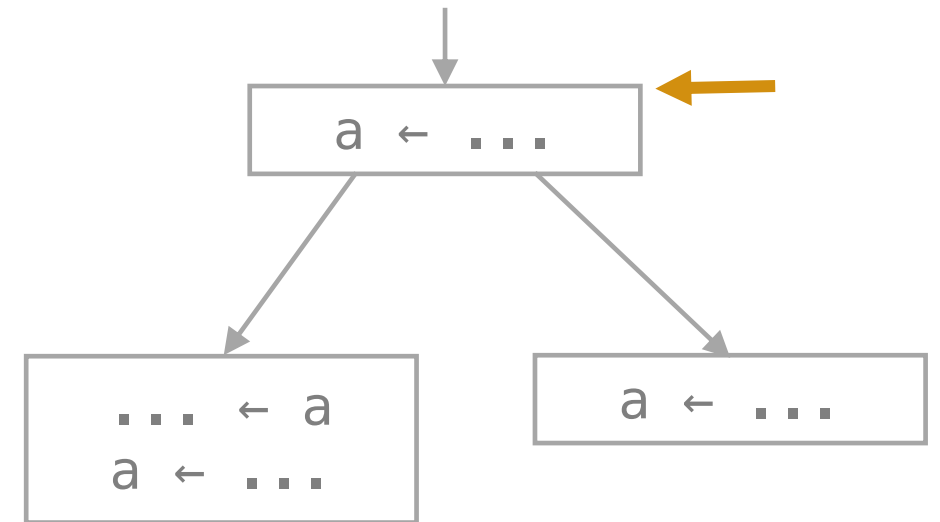
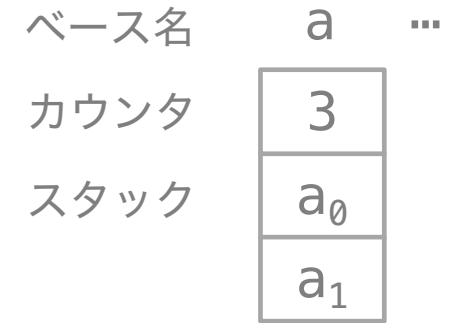
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



CFG

Renaming アルゴリズム

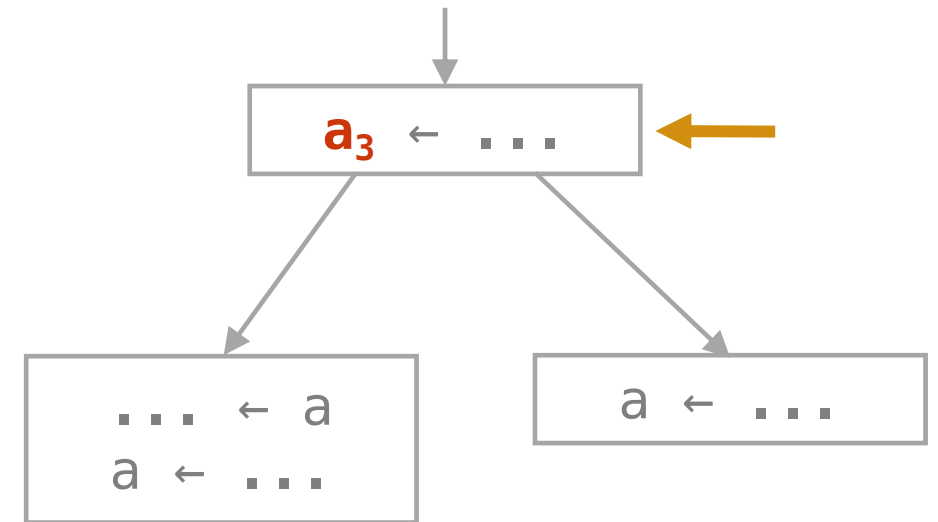
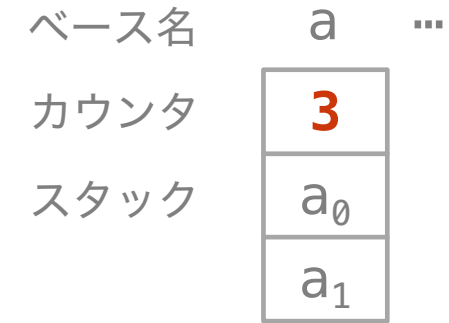
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



CFG

Renaming アルゴリズム

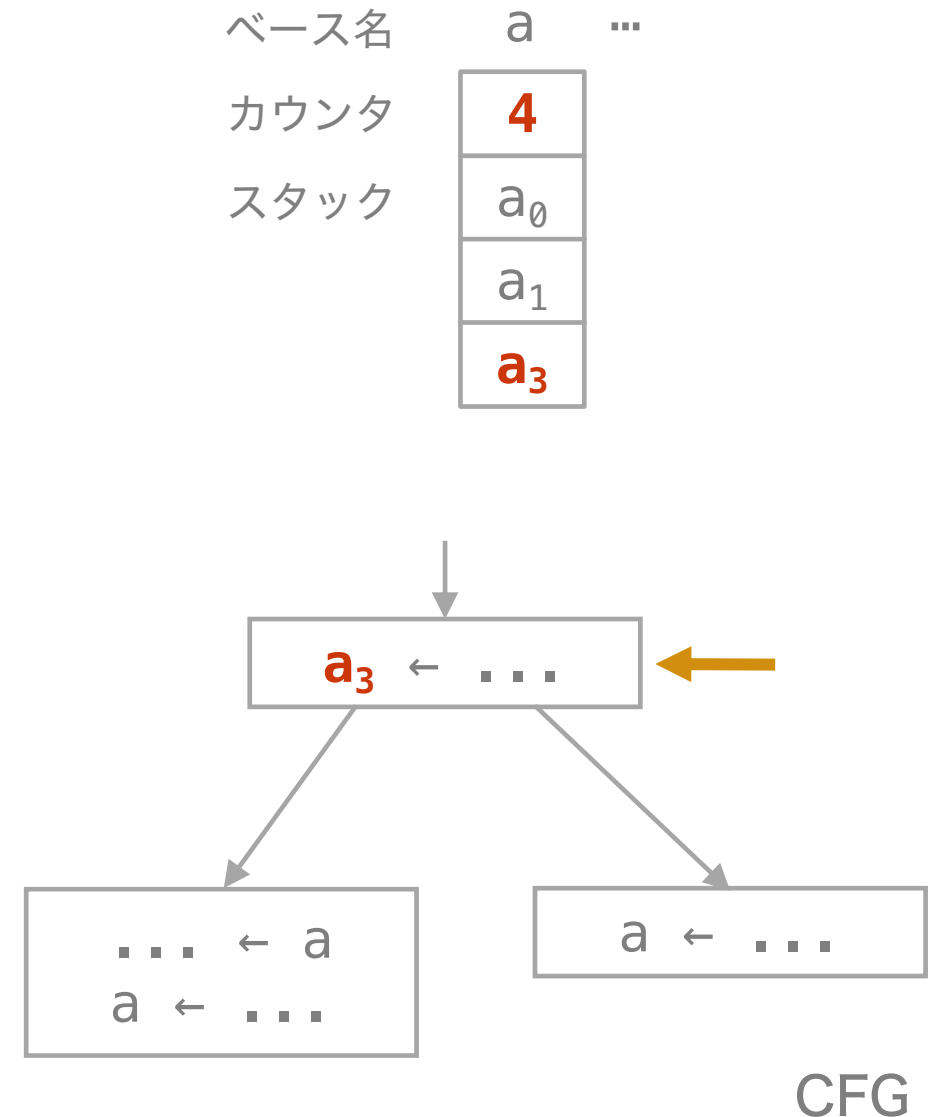
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



Renaming アルゴリズム

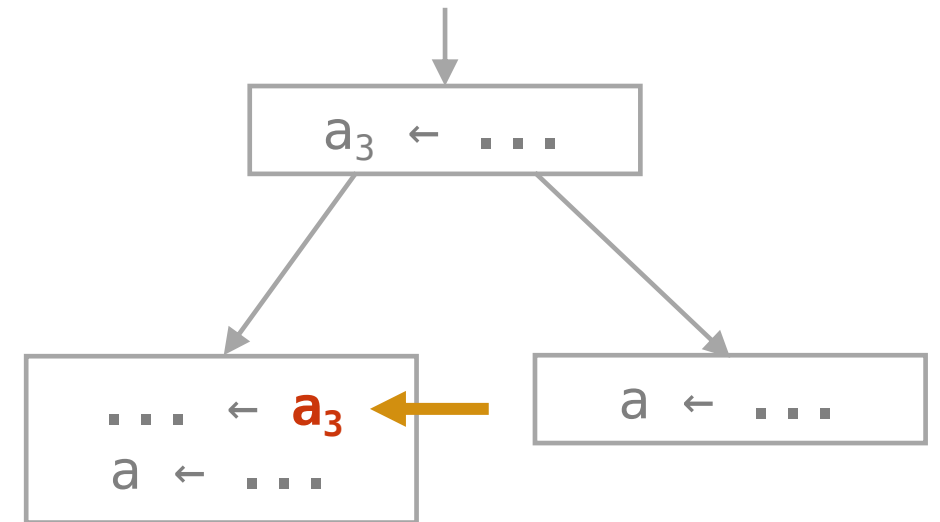
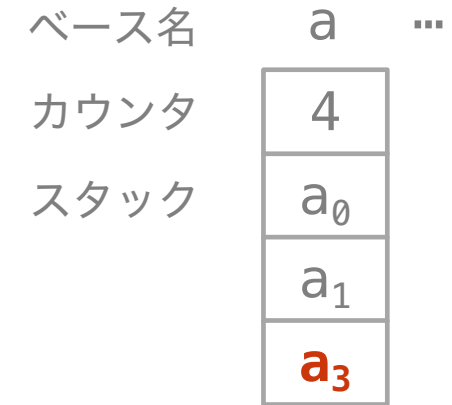
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



CFG

Renaming アルゴリズム

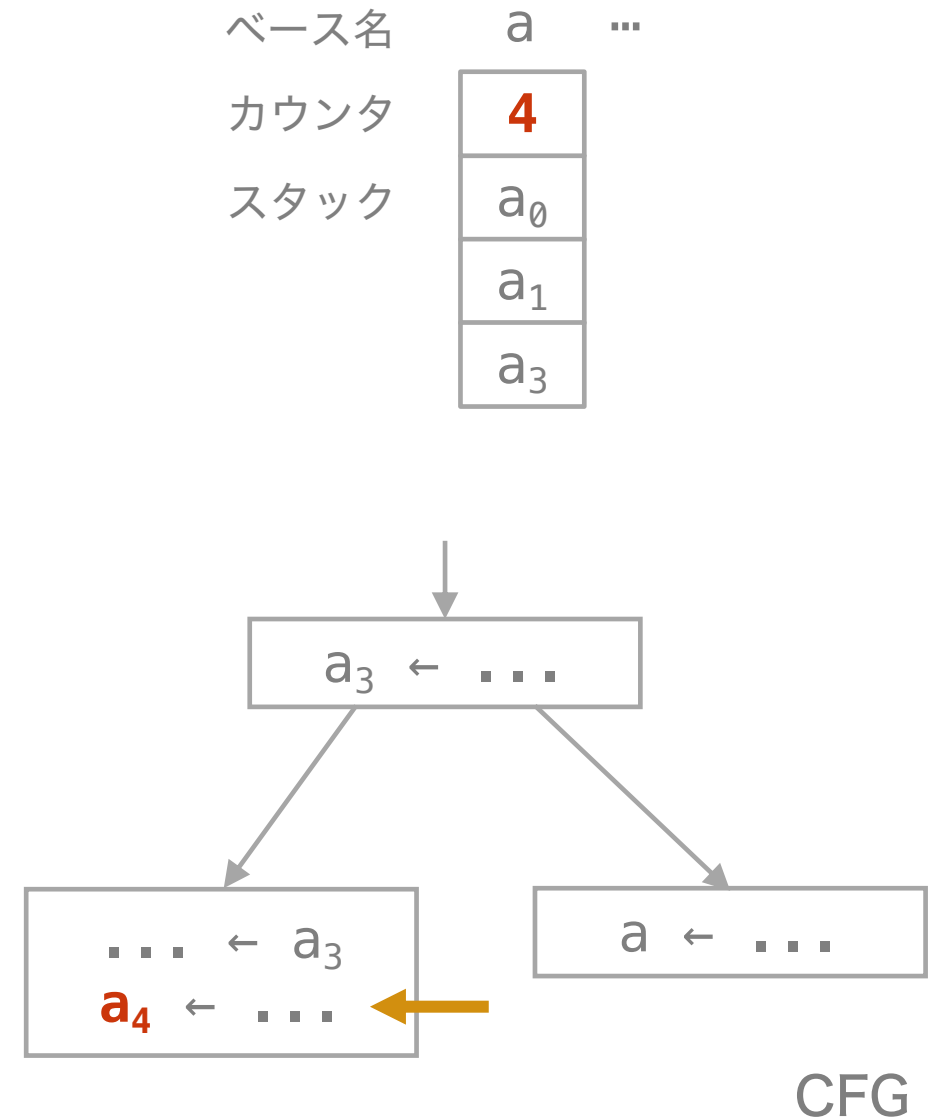
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



Renaming アルゴリズム

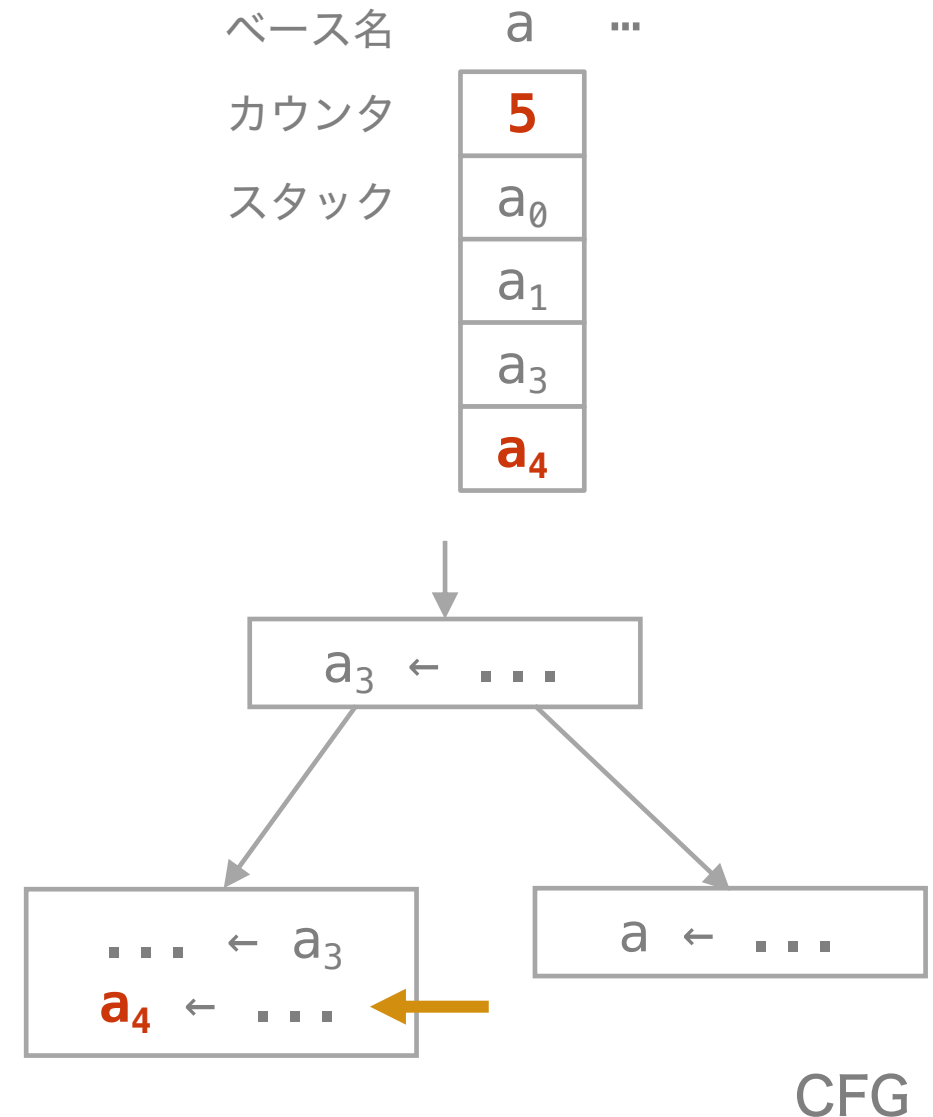
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

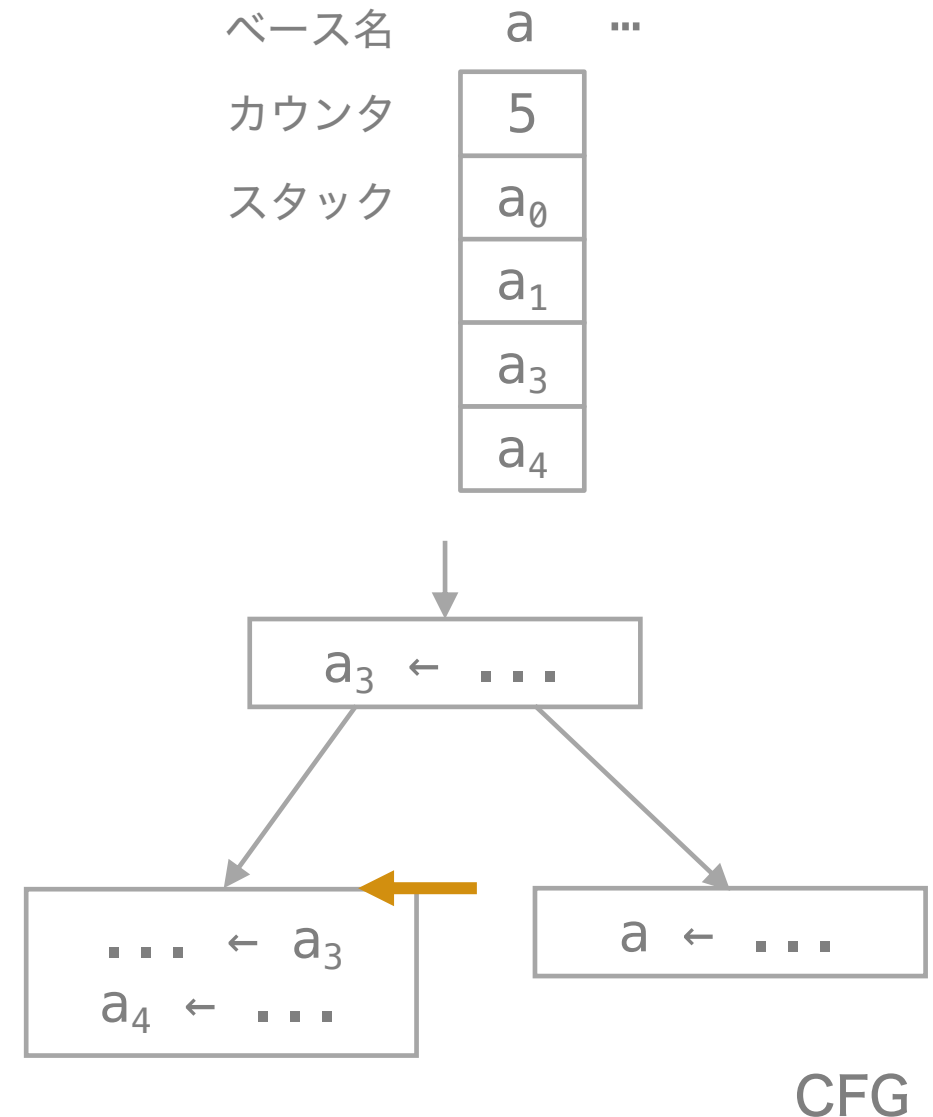
スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加





Renaming アルゴリズム

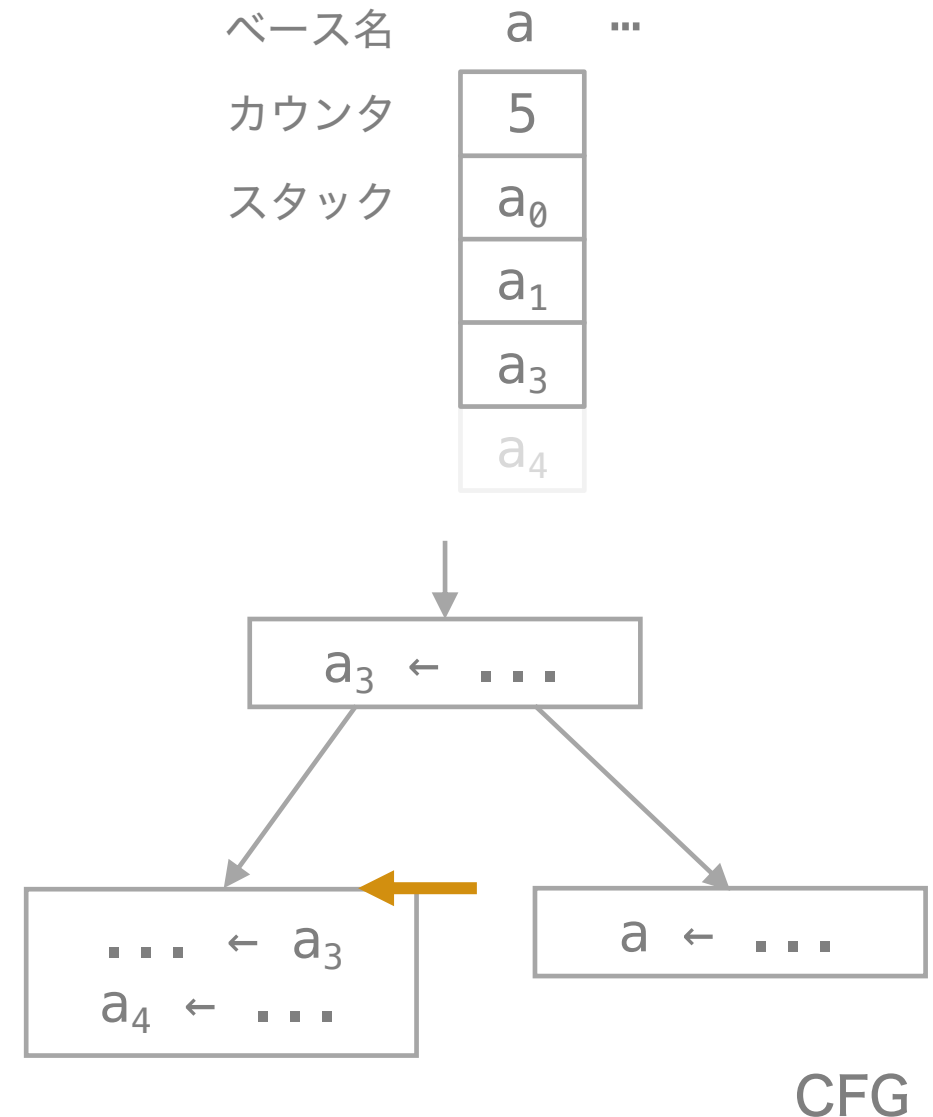
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



Renaming アルゴリズム

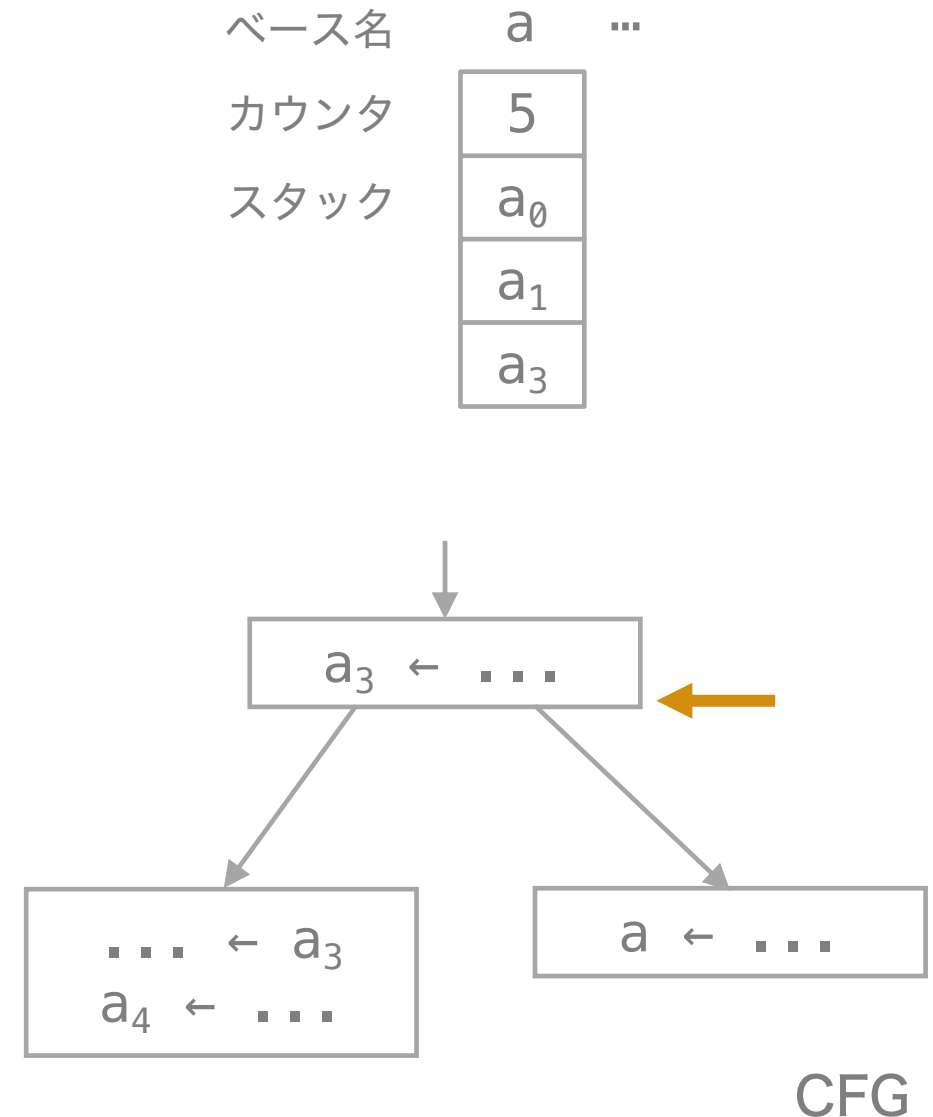
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加



Renaming アルゴリズム

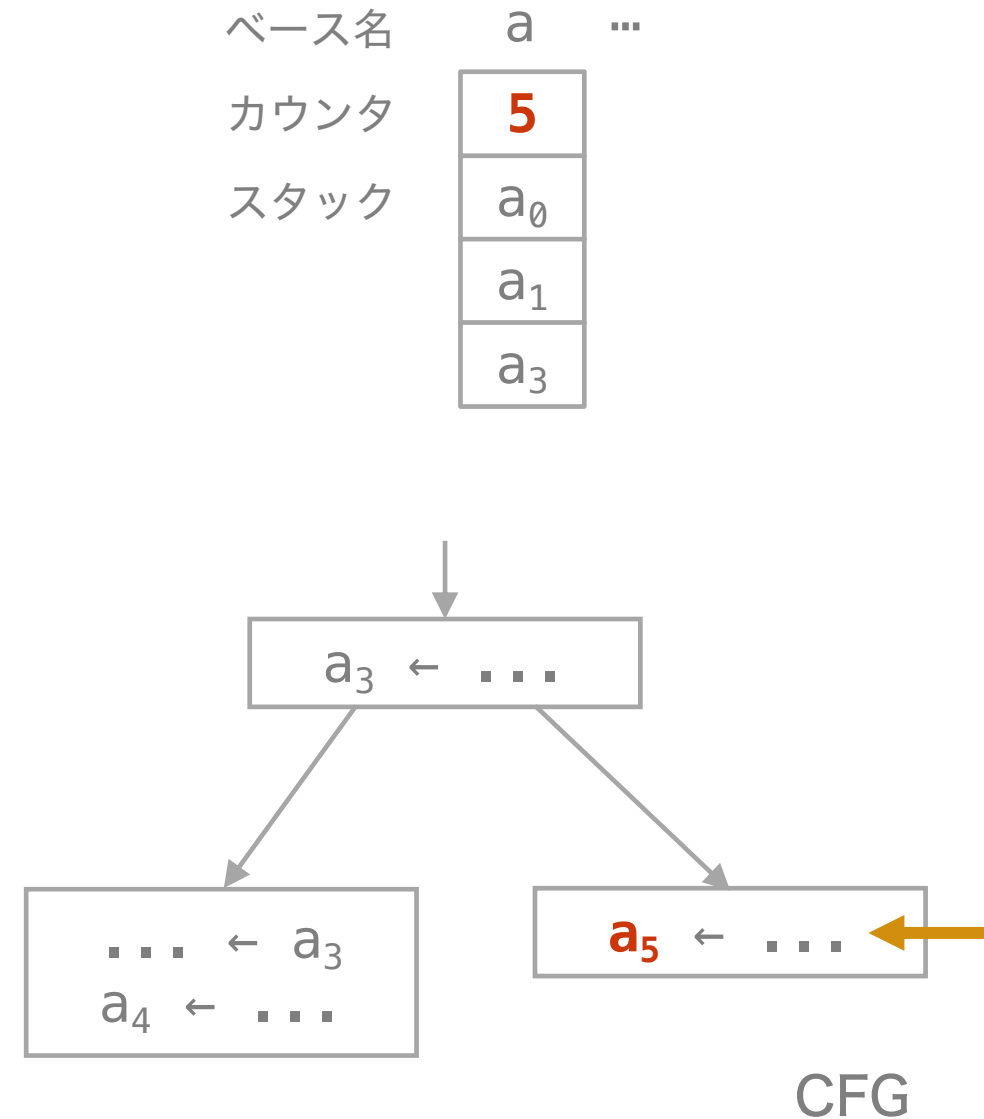
- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

スタック

- ・ top が現在使用可能な添え字を表す
- ・ 新しく定義されたら添え字を push
- ・ BB を探索し終わったら
BB 内で定義された添え字を pop

カウンタ

- ・ 定義された時にカウンタの値を添え字に
- ・ 値は単調増加

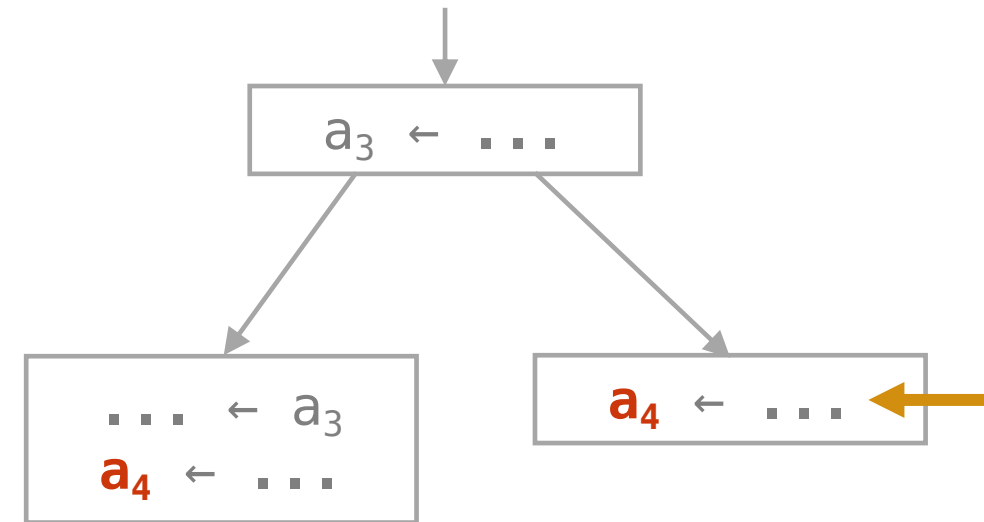
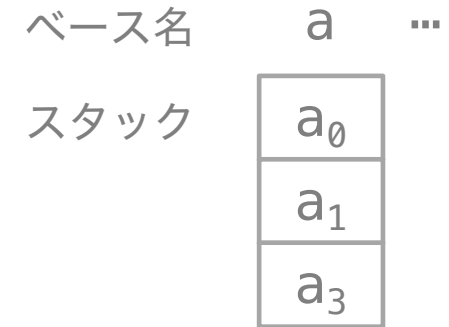


Renaming アルゴリズム

- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用

カウンタ

→ カウンタがなかったら同じ添え字で定義される可能性



CFG

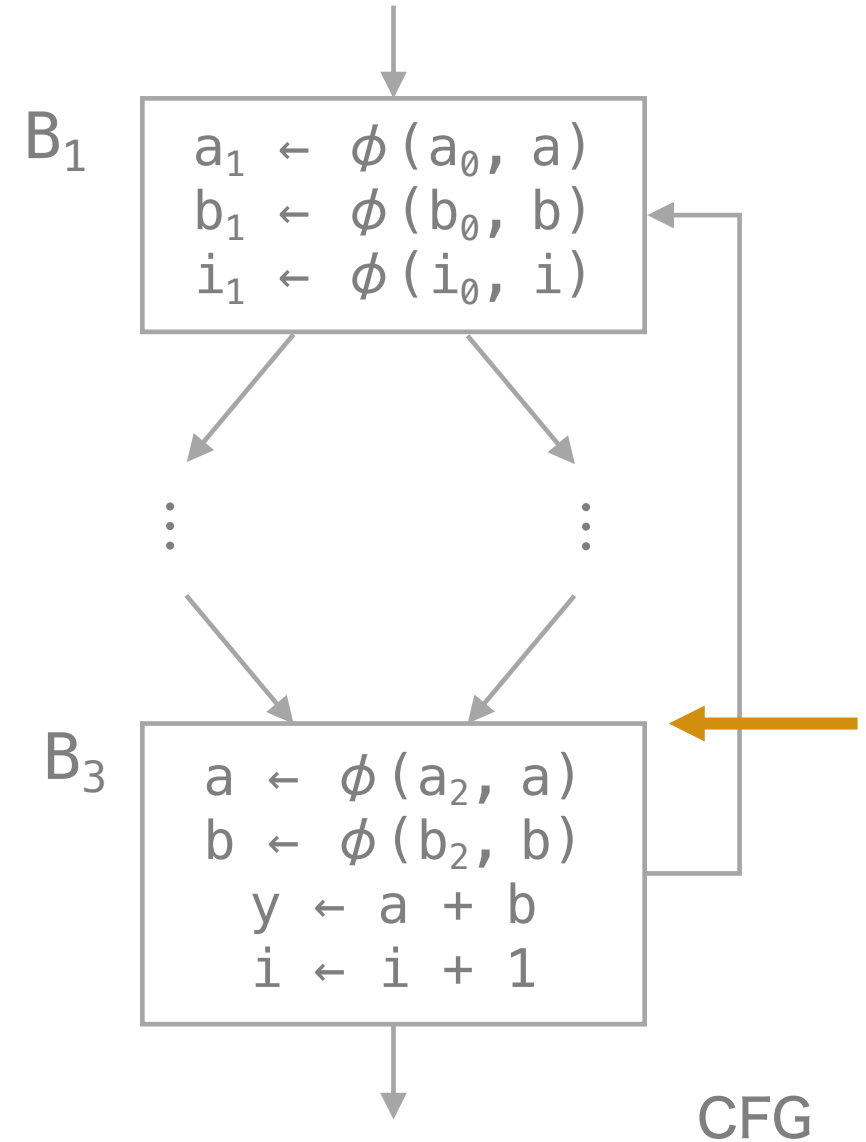
Renaming アルゴリズム

- ・ 支配木を preorder で走査
- ・ 各ベース名ごとにスタックとカウンタを使用
- ・ 各基本ブロックでの操作
 1. ϕ 関数の定義を SSA 名に
 2. BB 内の各命令の使用と定義を SSA 名に
 3. CFG 上で後続の BB の ϕ 関数のパラメータを現在の SSA 名に変更
 4. 再帰的に支配木上で後続の BB に移動
 5. BB 内で定義された SSA 名をスタックから pop

Renaming の例

Rename(B_3) :

ベース名	a	b	i
カウンタ	3	3	2
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2		



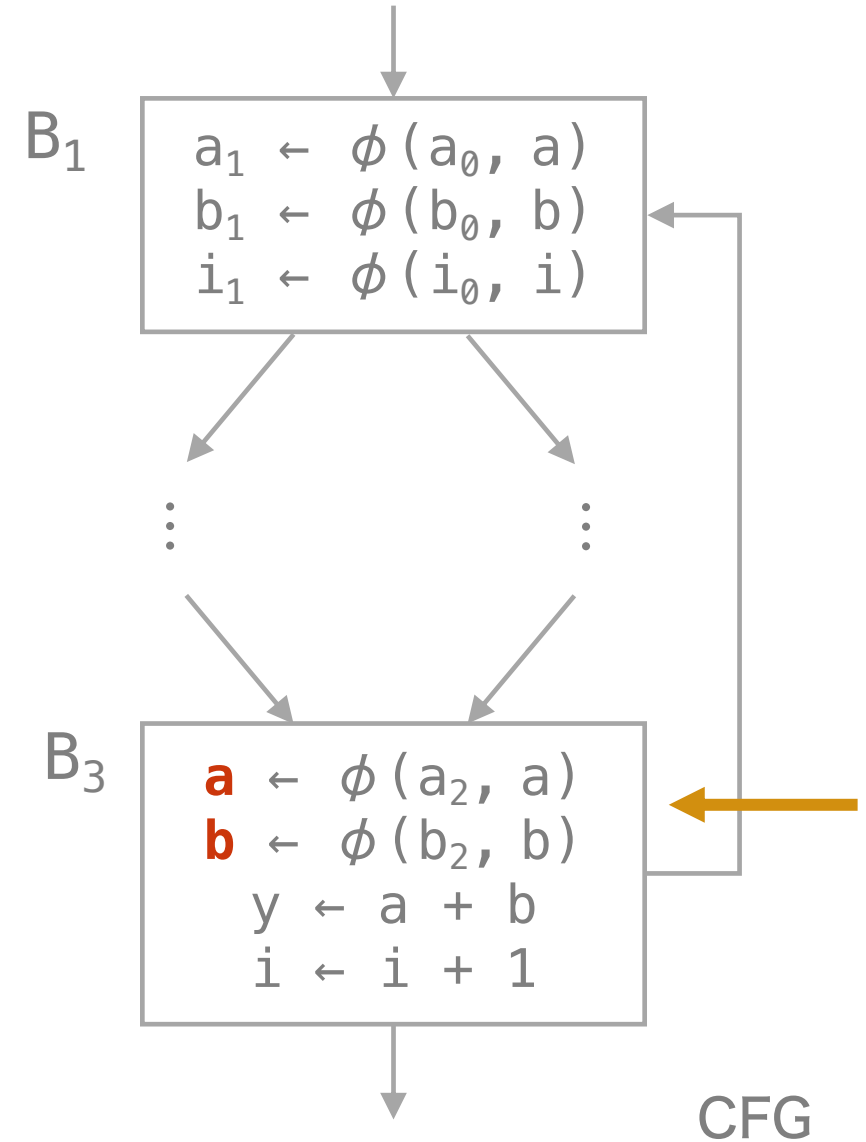
Renaming の例

Rename(B_3) :

1. ϕ 関数の定義を SSA 名に

Q. **a** と **b** の添え字は？

ベース名	a	b	i
カウンタ	3	3	2
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2		

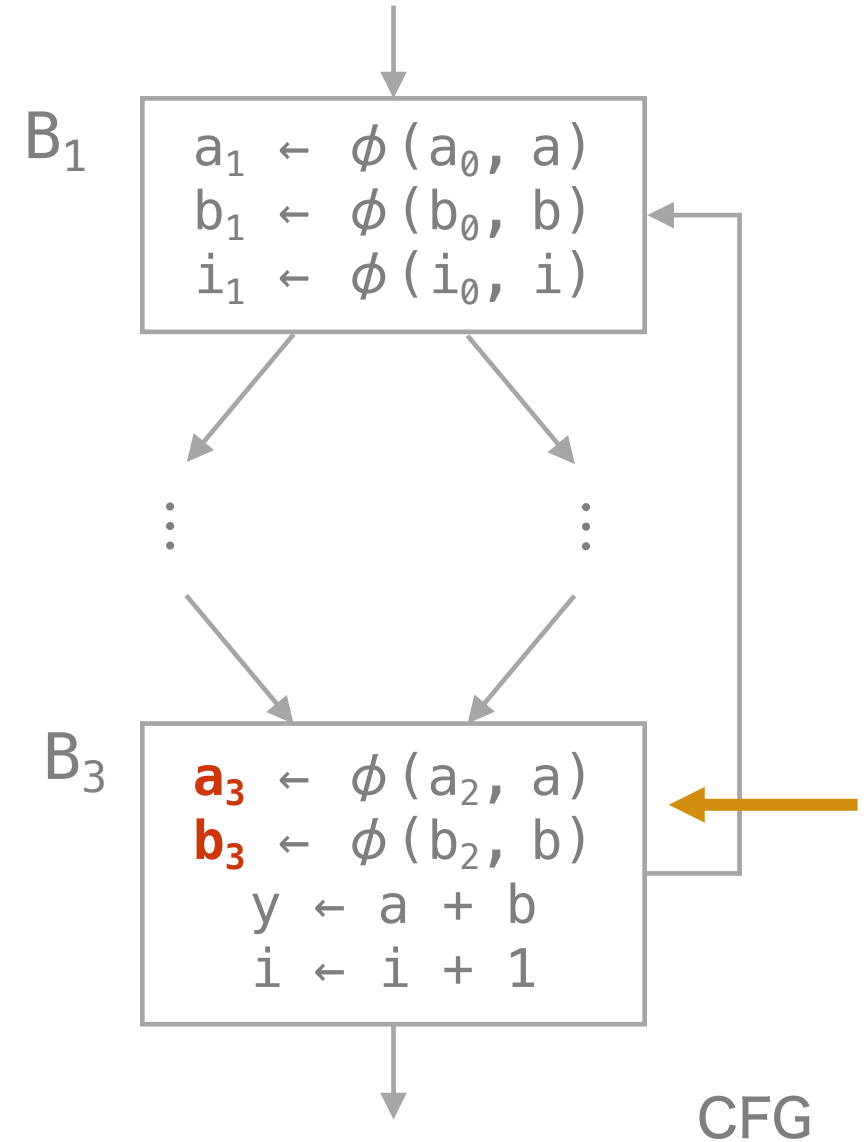


Renaming の例

Rename(B_3) :

1. ϕ 関数の定義を SSA 名に

ベース名	a	b	i
カウンタ	4	4	2
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	
	a_3		



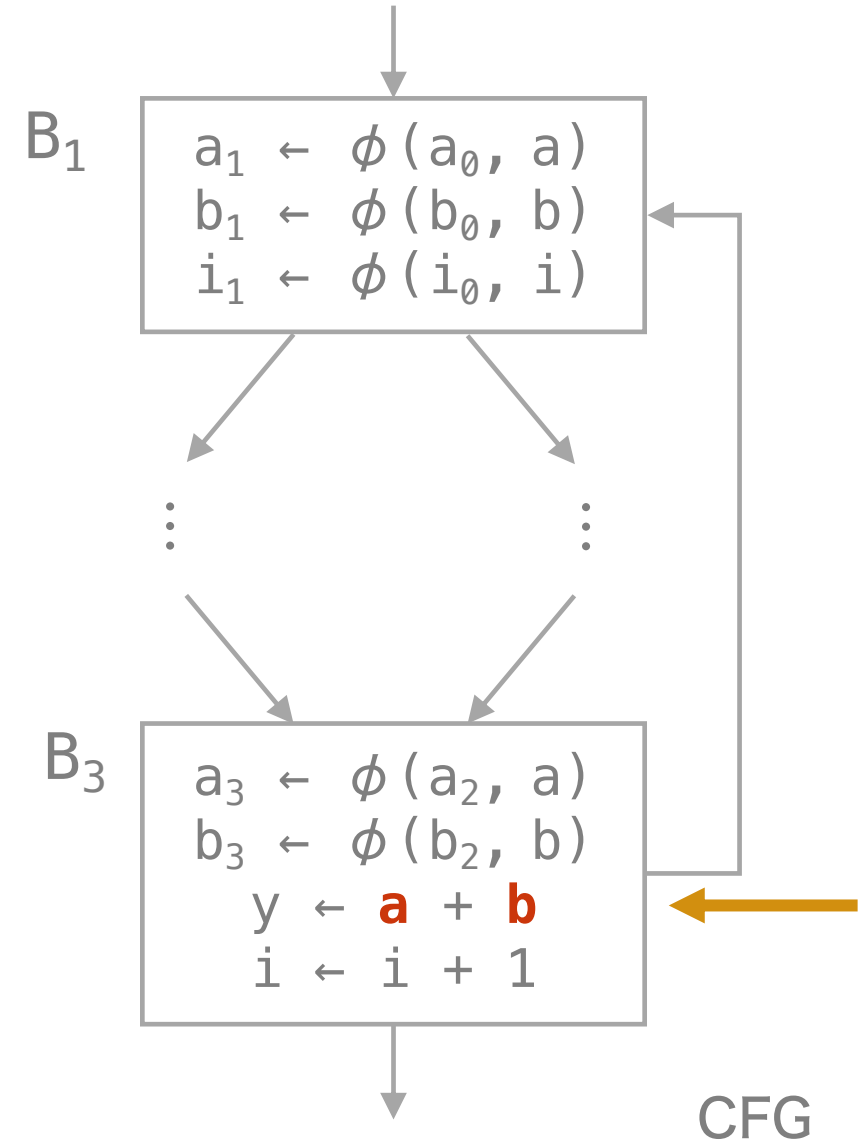
Renaming の例

Rename(B_3) :

2. BB 内の各命令の使用と定義を SSA 名に

Q. **a** と **b** の添え字は？

ベース名	a	b	i
カウンタ	4	4	2
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	
	a_3		

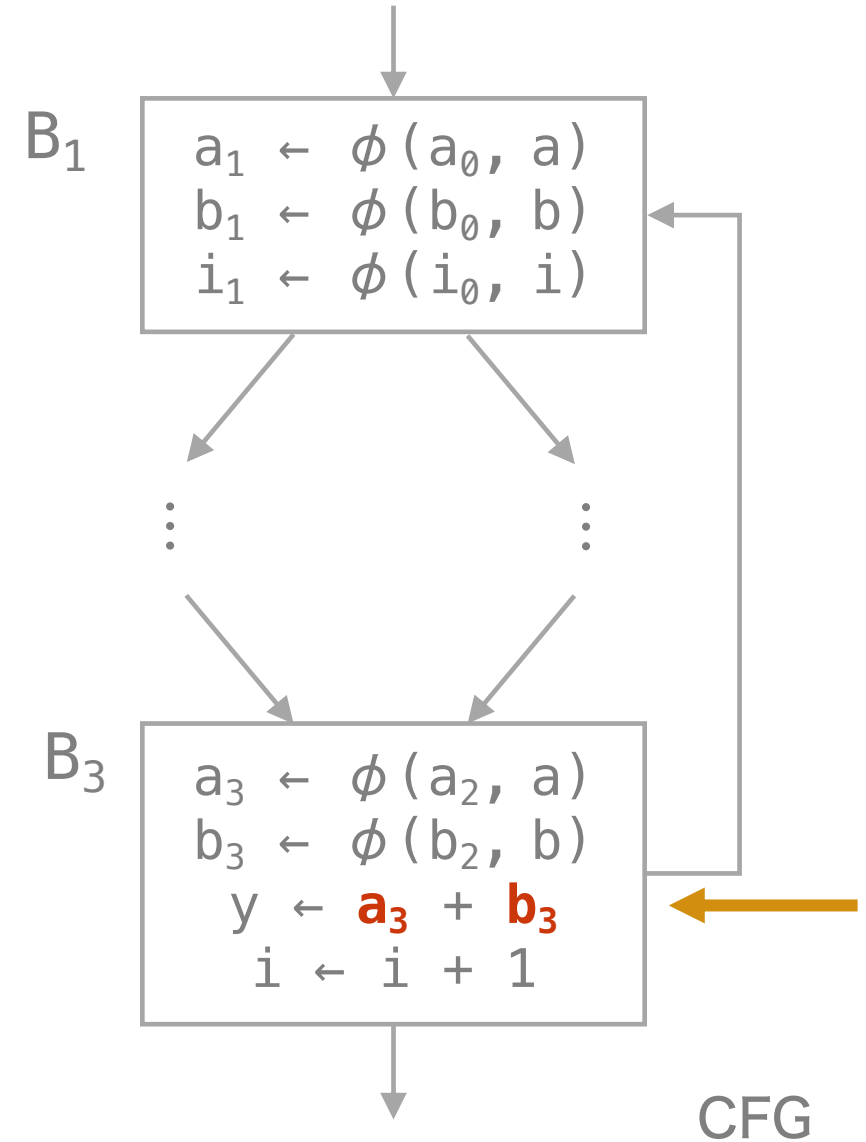


Renaming の例

Rename(B_3) :

2. BB 内の各命令の使用と定義を SSA 名に

ベース名	a	b	i
カウンタ	4	4	2
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	
	a_3		



Renaming の例

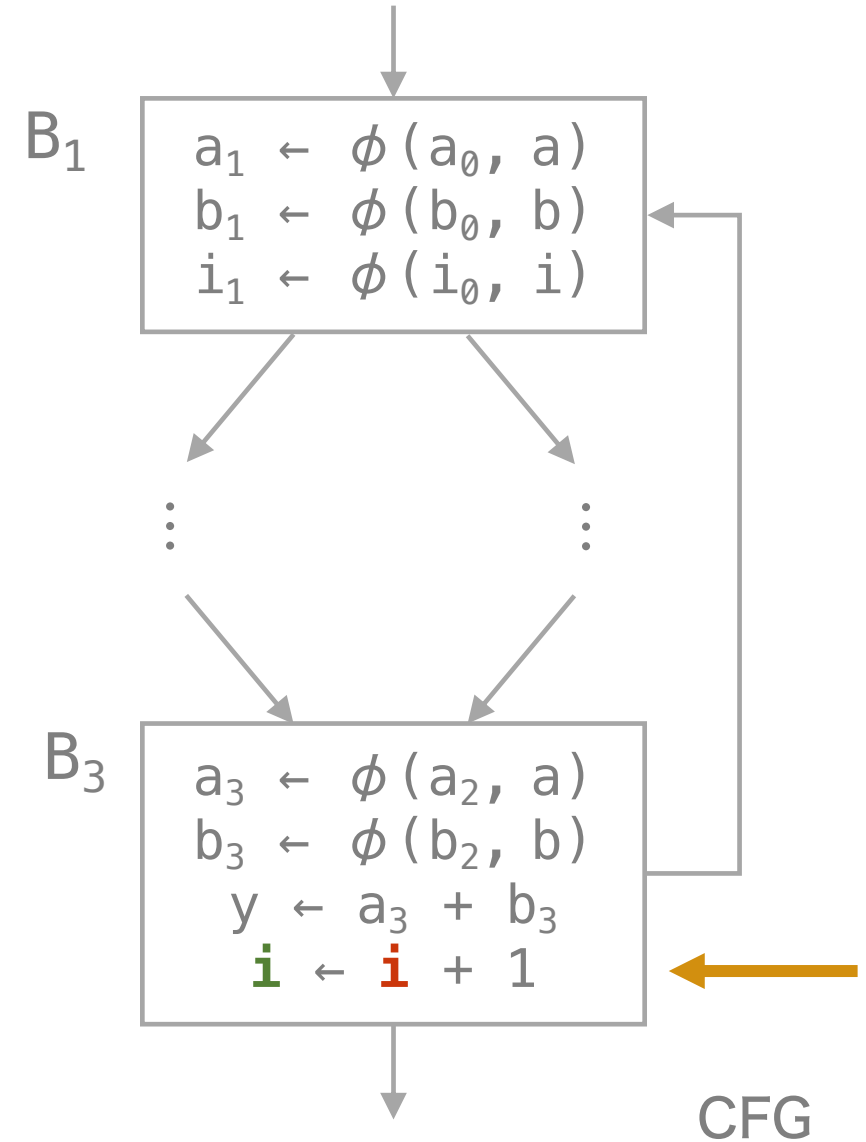
Rename(B_3) :

2. BB 内の各命令の使用と定義を SSA 名に

Q. **i** と **i** の添え字は？

(使用 → 定義の順で SSA 名にする)

ベース名	a	b	i
カウンタ	4	4	2
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	
	a_3		

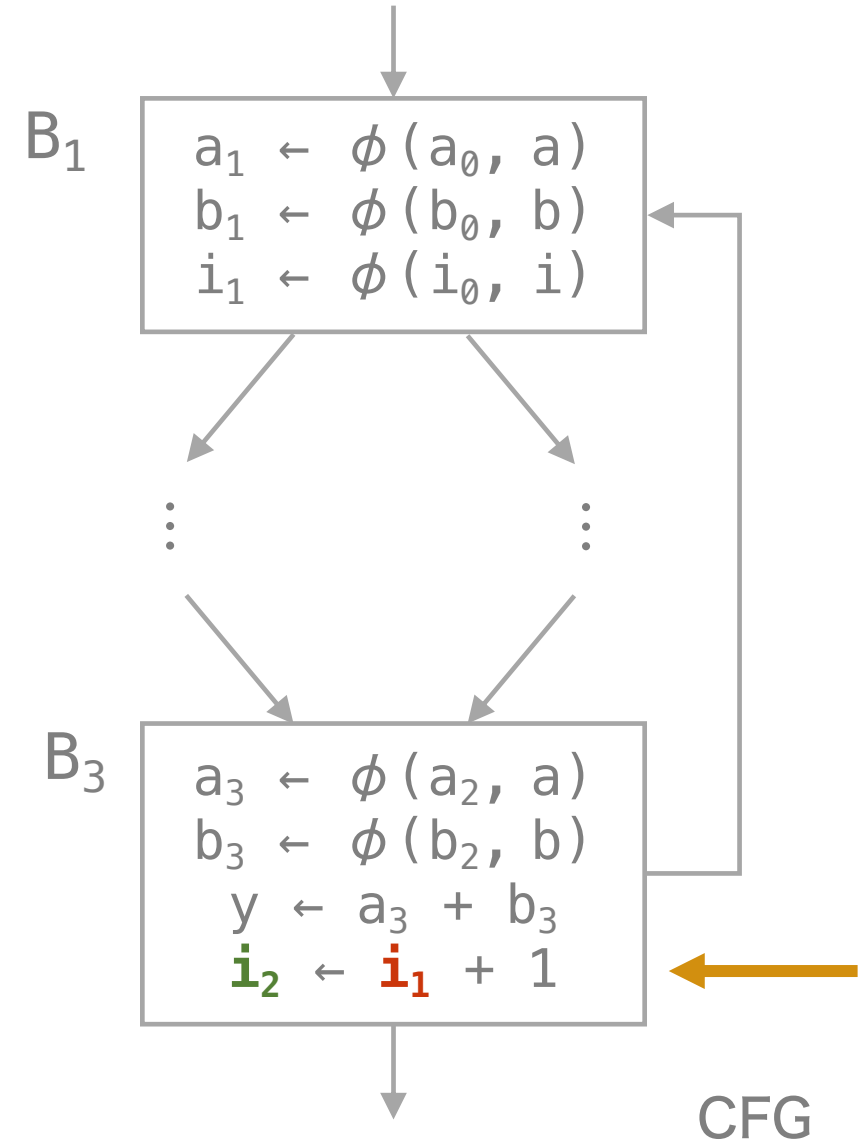


Renaming の例

Rename(B_3) :

2. BB 内の各命令の使用と定義を SSA 名に

ベース名	a	b	i
カウンタ	4	4	3
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	i_2
	a_3		



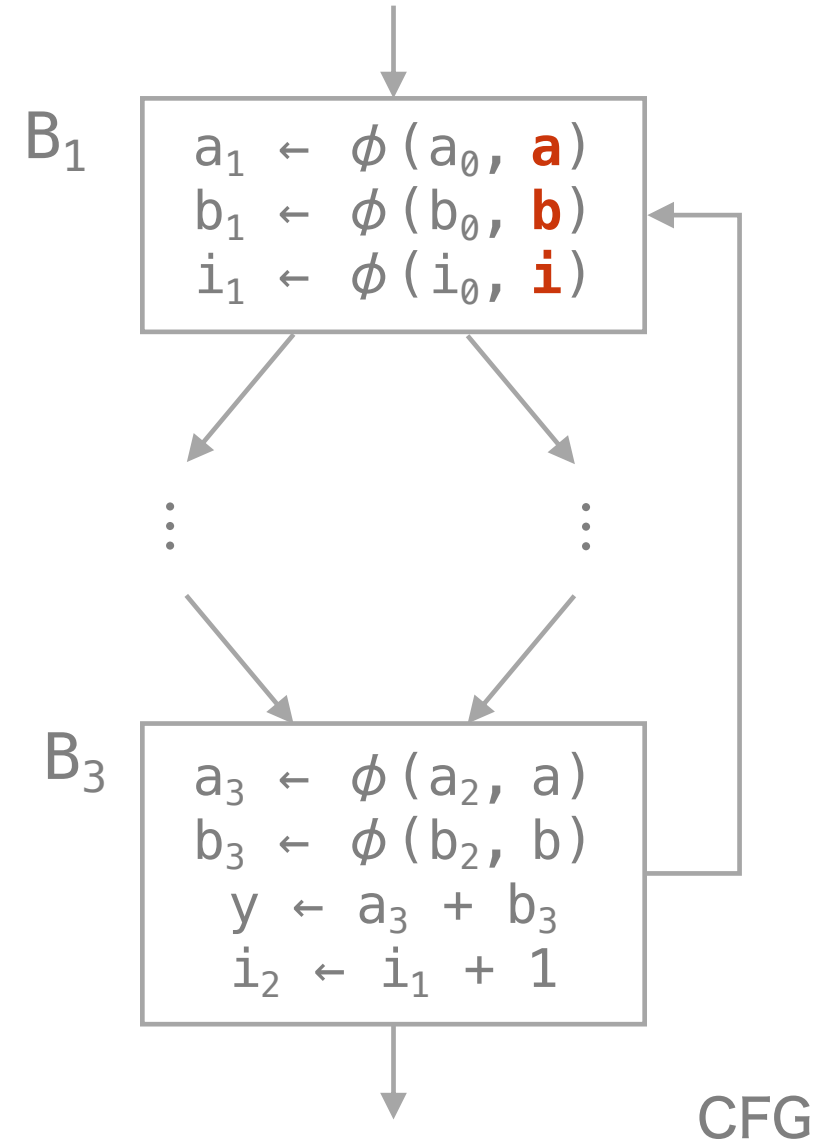
Renaming の例

Rename(B_3) :

- CFG 上で後続の BB の ϕ 関数のパラメータを現在の SSA 名に変更

Q. **a** と **b** と **i** の添え字は？

ベース名	a	b	i
カウンタ	4	4	3
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	i_2
	a_3		

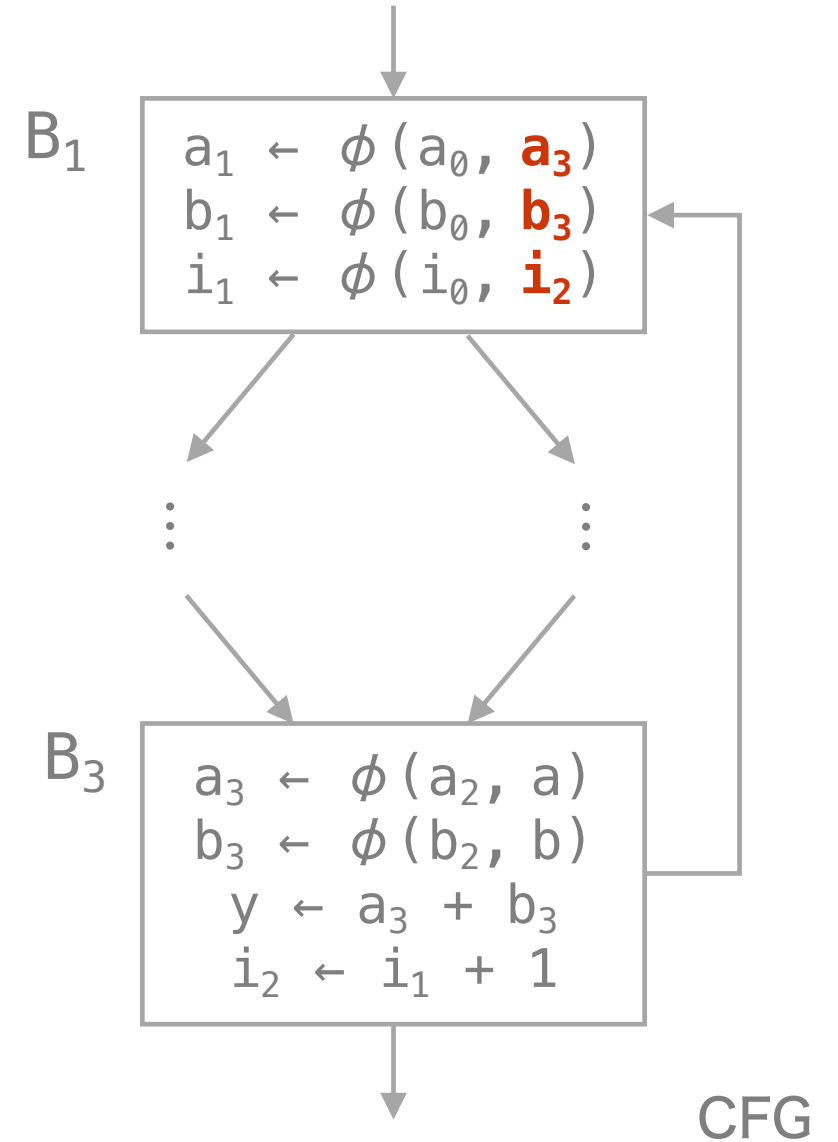


Renaming の例

Rename(B_3) :

- CFG 上で後続の BB の ϕ 関数のパラメータを現在の SSA 名に変更

ベース名	a	b	i
カウンタ	4	4	3
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	i_2
	a_3		

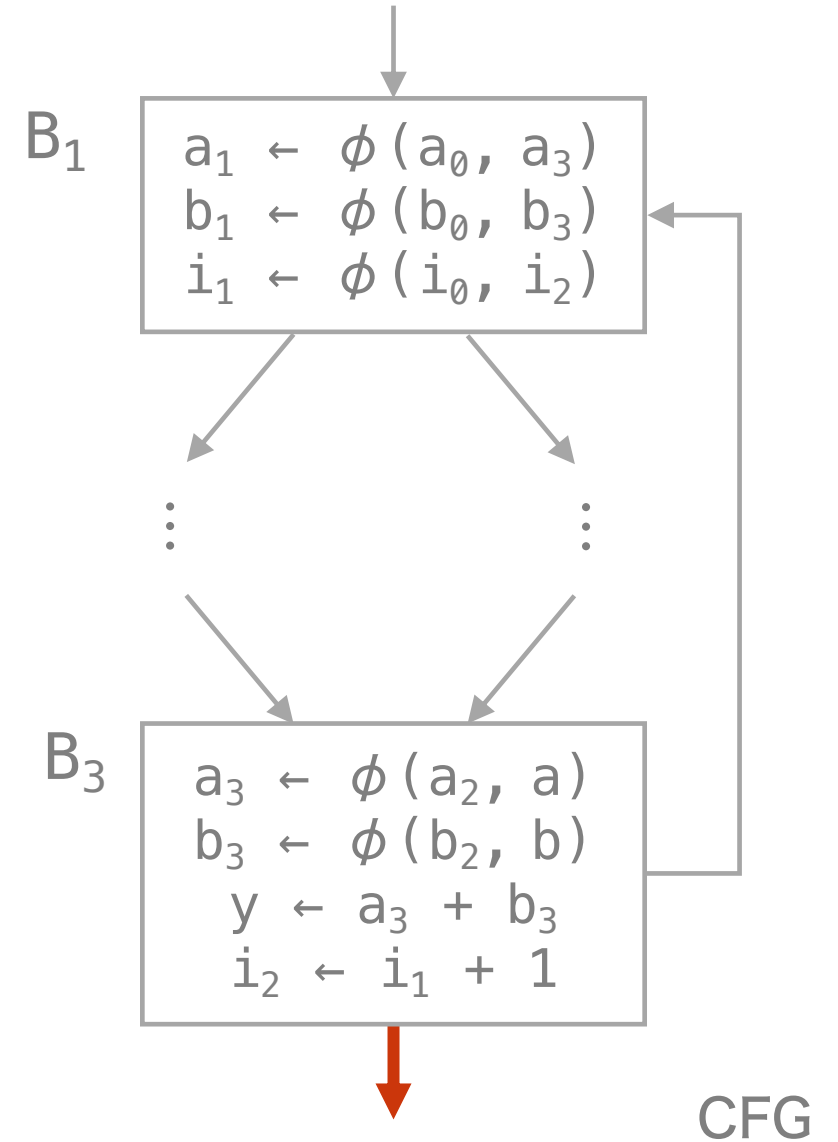


Renaming の例

Rename(B_3) :

4. 再帰的に支配木上で後続の BB に移動

ベース名	a	b	i
カウンタ	4	4	3
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	i_2
	a_3		



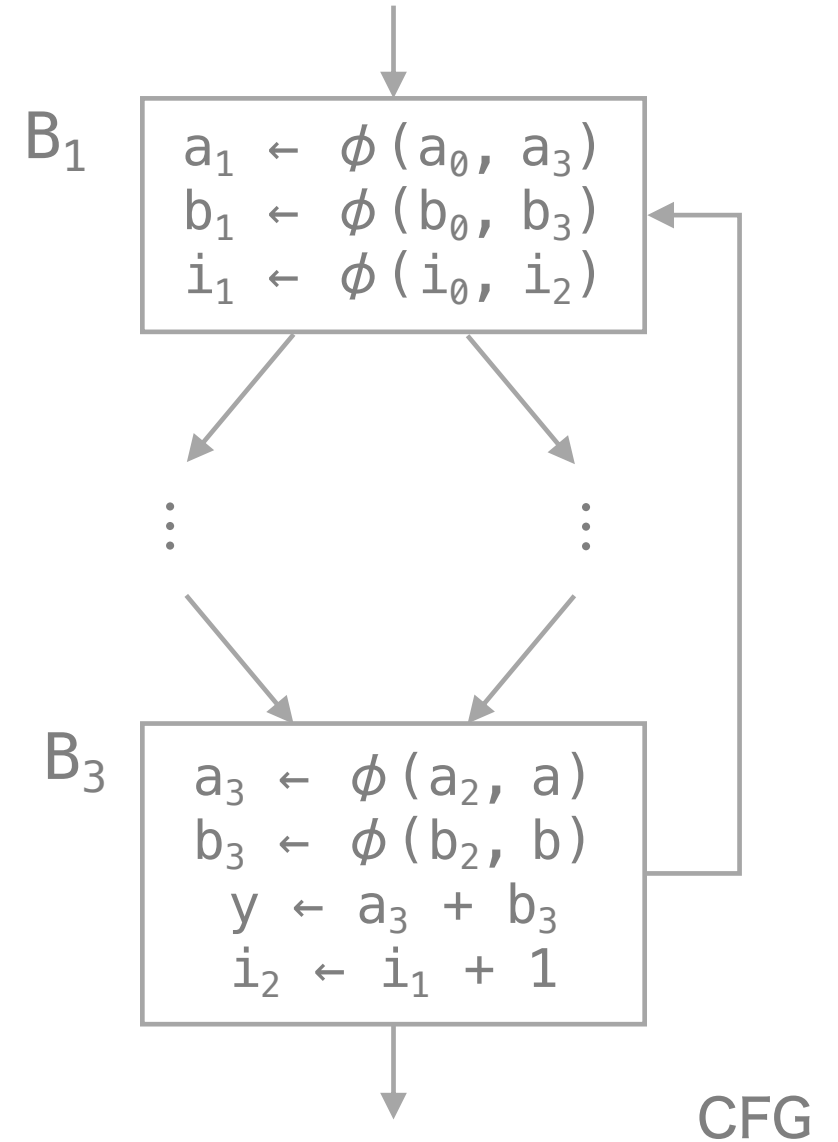
CFG

Renaming の例

Rename(B_3) :

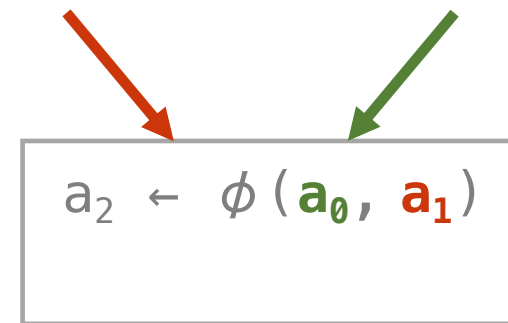
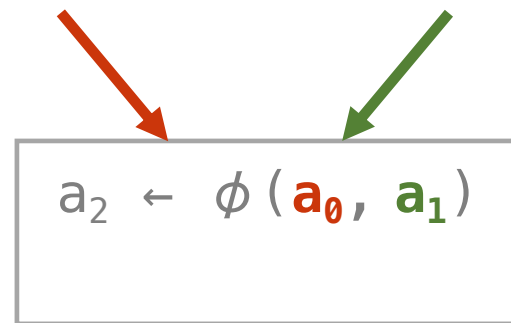
5. BB 内で定義された SSA 名をスタックから pop

ベース名	a	b	i
カウンタ	4	4	3
スタック	a_0	b_0	i_0
	a_1	b_1	i_1
	a_2	b_3	i_2
	a_3		



ϕ 関数のパラメータ

3. CFG 上で後続の BB の ϕ 関数のパラメータを現在の SSA 名に変更
→ どのパラメータを変更するか知る必要がある
- CFG の実装と SSA の構築で一貫したルールを定める
→ CFG エッジがリストで保存されているなら, その順序で引数を決定する
 - 教科書では図の CFG エッジの 左 → 右が, ϕ 関数の引数の 左 → 右 に対応



A Final Improvement

- ・スタックには最新の名前だけ push すれば良い

ex.) a_1 はスタックに乘せる必要なし

$$\begin{array}{lcl} a_1 & \leftarrow & \phi(a_0, a_3) \\ b_1 & \leftarrow & \phi(b_0, b_3) \\ i_1 & \leftarrow & \phi(i_0, i_2) \\ a_2 & \leftarrow & \dots \end{array}$$

- ・ブロック内で定義されたベース名1つにつき push と pop 1回ずつ

→ スタック操作の時間削減

→ スタックのスペース削減・オーバーフロー回避

(スタックの深さは必ず支配木の深さ以下になる)

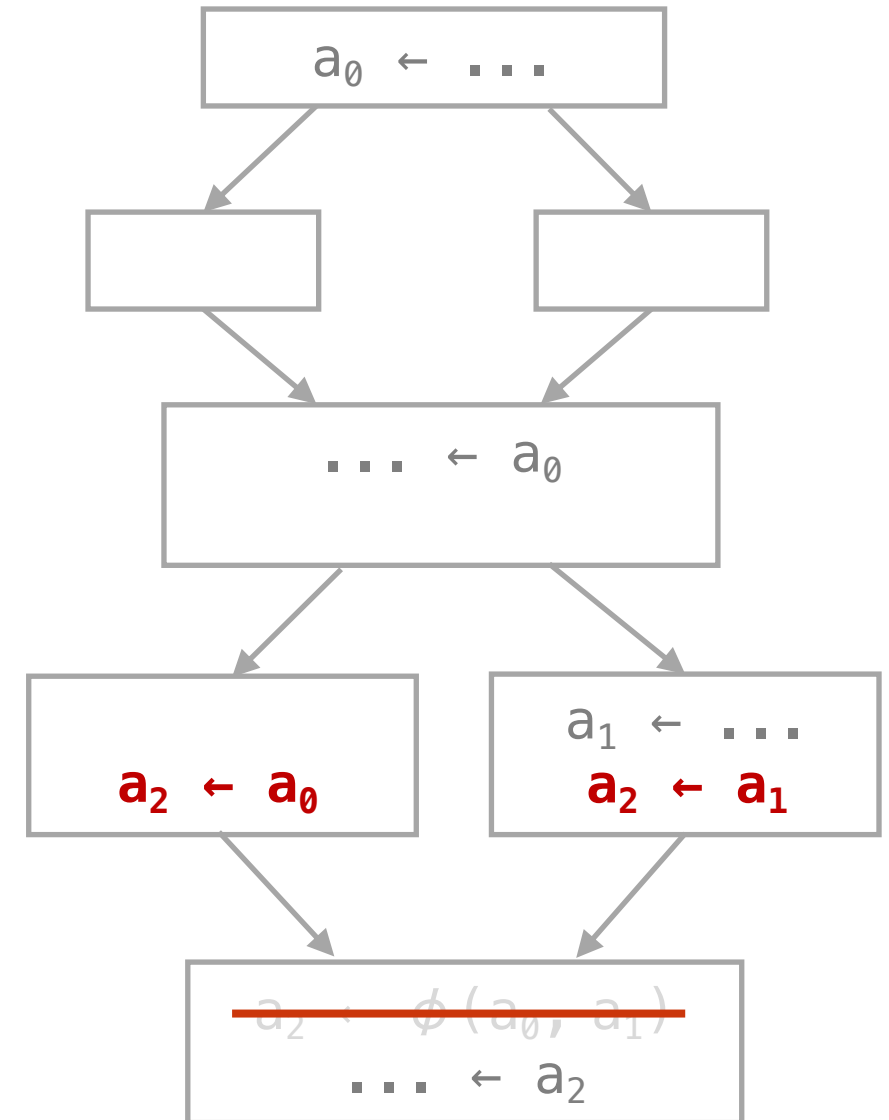


新潟県 青梅川駅

9.3.5 Translation out of SSA Form

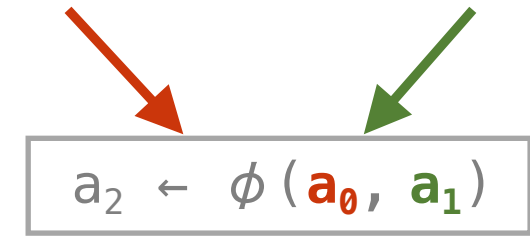
SSA 逆変換：

- ・ SSA を通常の形式に変換
 - ・ ϕ 関数のない形にコードを変換
(コンピュータは ϕ 関数をそのまま実行できない)
- ϕ 関数のセマンティクスを満たすように
 ϕ 関数をコピー演算に置き換えたい

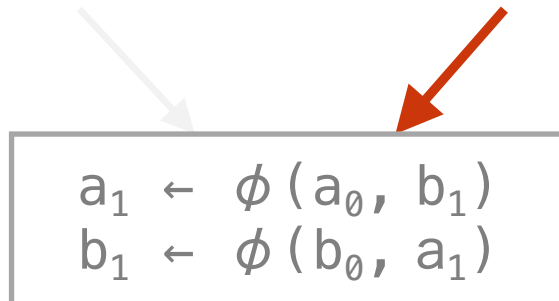


ϕ 関数のセマンティクス

1. どのエッジから来たかによって値が選択される



2. 同じブロック内の ϕ 関数は 並列に 計算される



b_1 の値を a_1 に
 元々の a_1 の値を b_1 に
 (a_1 と b_1 の値がスワップ)



b_1 の値を a_1 に
 その a_1 の値を b_1 に

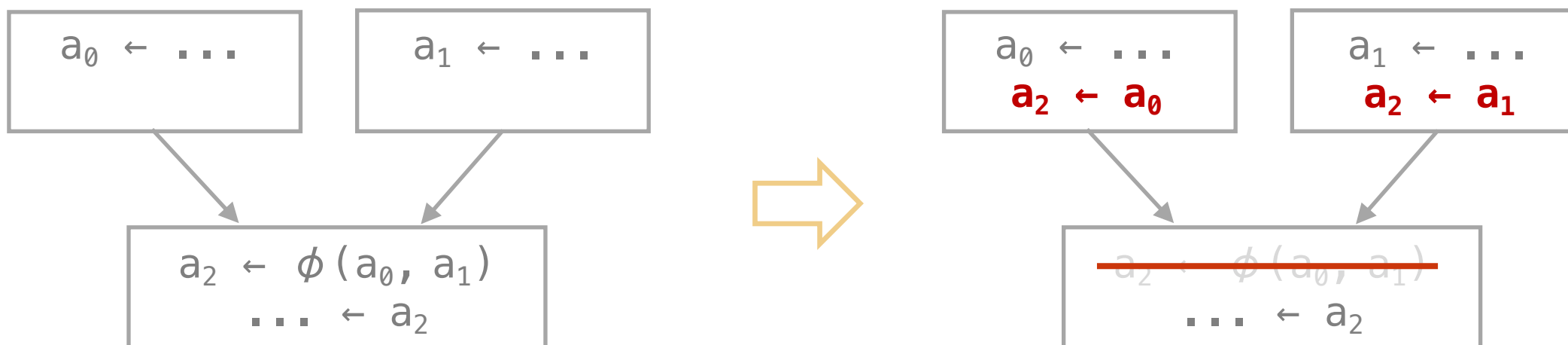
9.3.5 Translation out of SSA Form

説明の流れ：

1. 素朴な SSA 逆変換
2. 素朴な SSA 逆変換の 2つの問題点
3. 問題点を解決する SSA 逆変換

素朴な SSA 逆変換

- CFG の前のノードに、適切な ϕ 関数の引数を ϕ 関数で定義された変数にコピーする操作を挿入



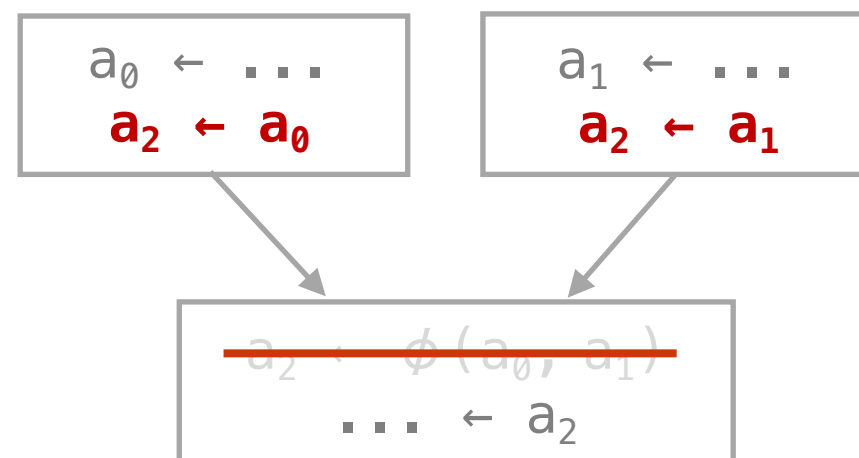
素朴な SSA 逆変換

- CFG の前のノードに、適切な ϕ 関数の引数を ϕ 関数で定義された変数にコピーする操作を挿入

問題点：

正しくないコードを生成する可能性がある

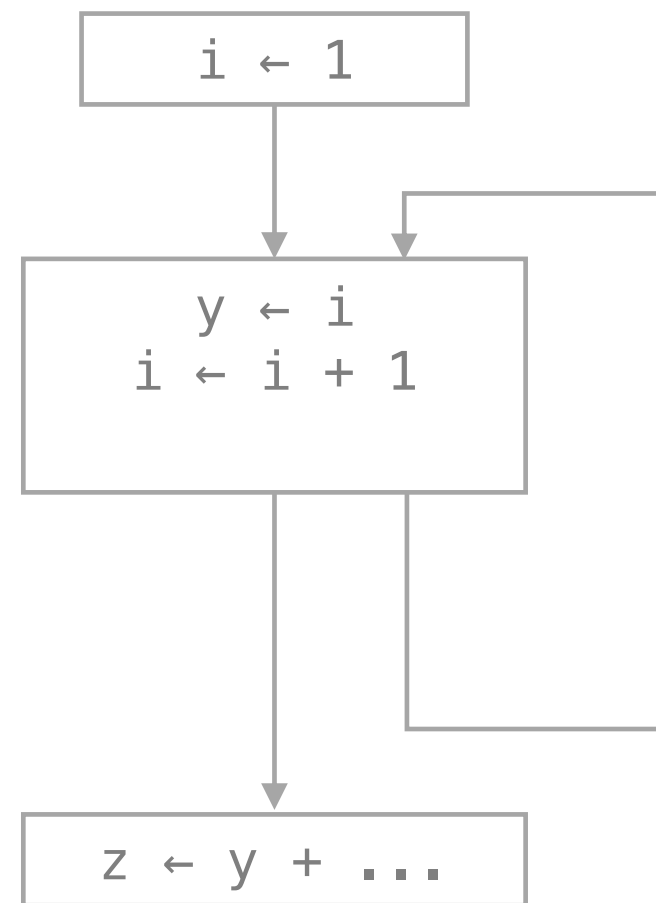
1. The Lost-Copy Problem
2. The Swap Problem



The Lost-Copy Problem の例

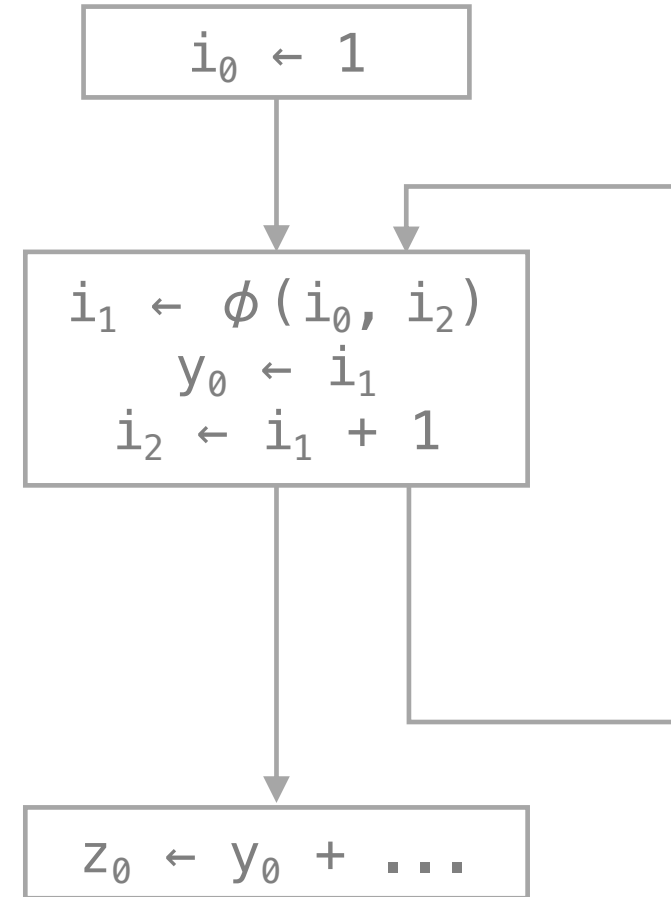
元々のコード

- ・ ループ内で i をインクリメントする
- ・ ループ後の z の計算には
 i の最後から2番目の値が使われる



The Lost-Copy Problem の例

Pruned SSA

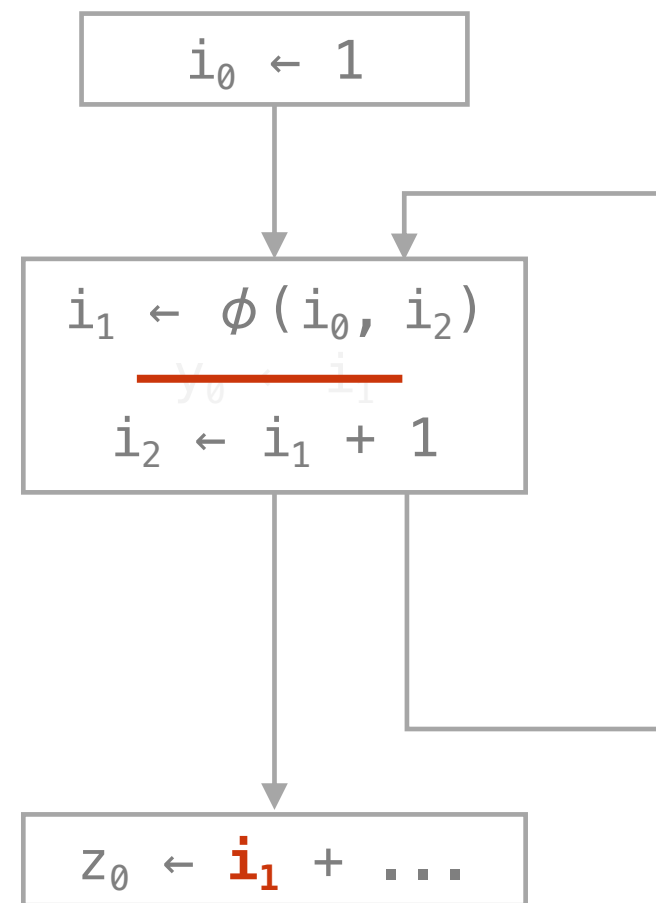


The Lost-Copy Problem の例

After Copy Folding

Copy Folding (コピー畳み込み) :

source と destination の名前を変更することで
不必要なコピー操作を削除する最適化

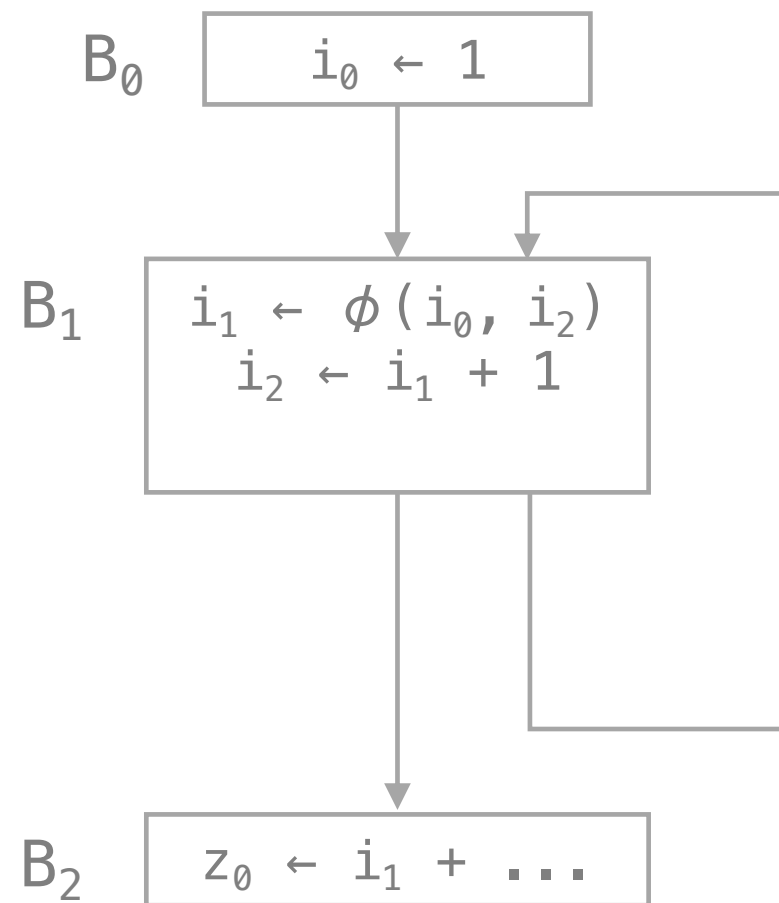


The Lost-Copy Problem の例

素朴な SSA 逆変換

ϕ 関数の前の BB に適切なコピー操作を挿入

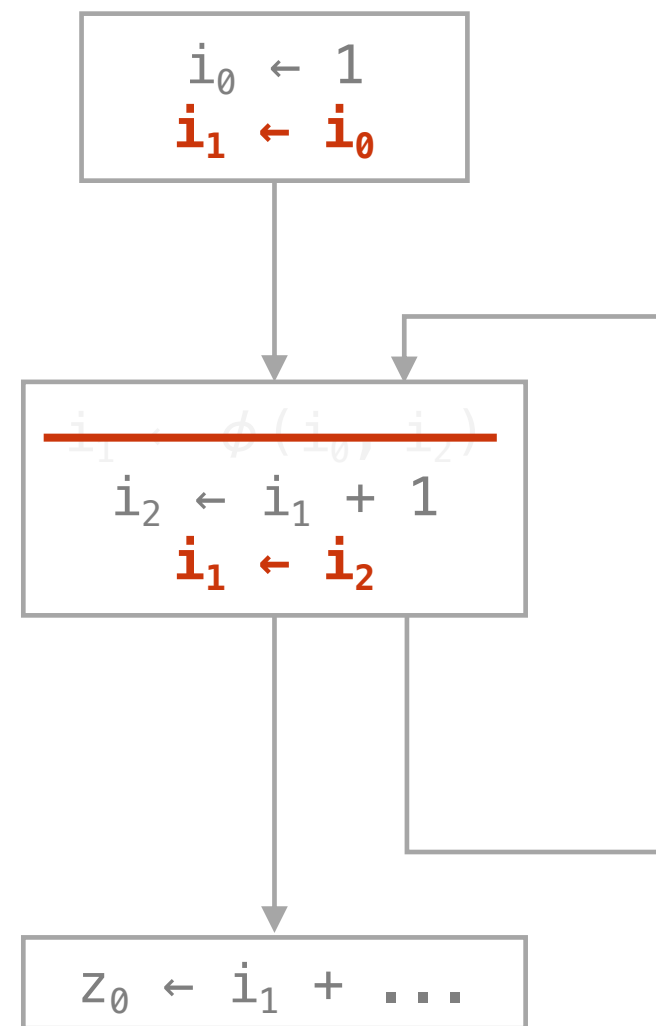
Q. どこに何が挿入される？



The Lost-Copy Problem の例

素朴な SSA 逆変換

ϕ 関数の前の BB に適切なコピー操作を挿入



The Lost-Copy Problem の例

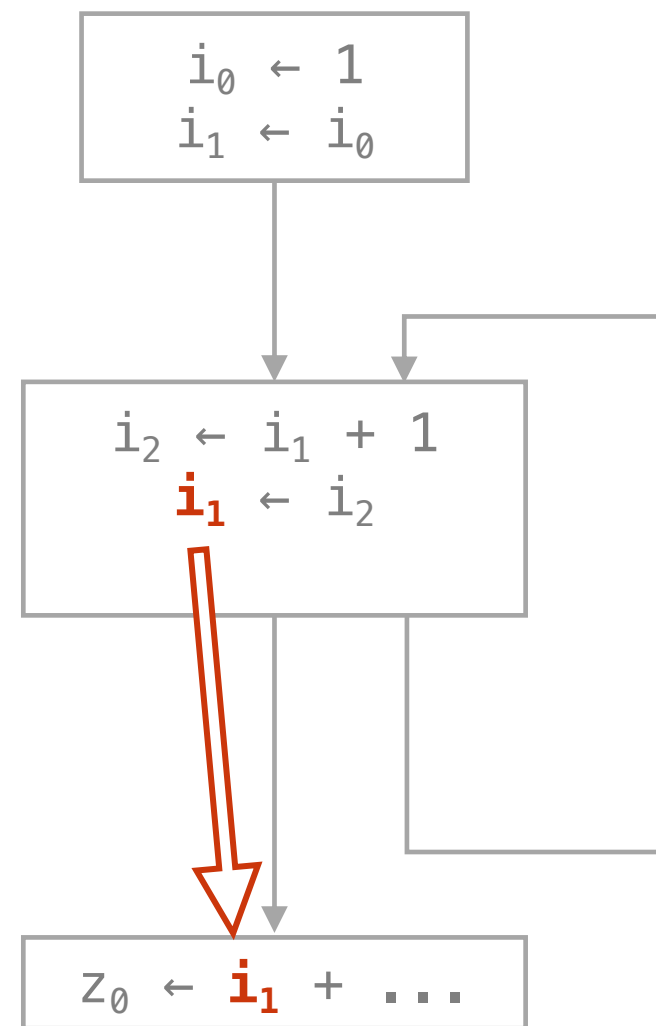
素朴な SSA 逆変換

φ 関数の前の BB に適切なコピー操作を挿入

元々のコード

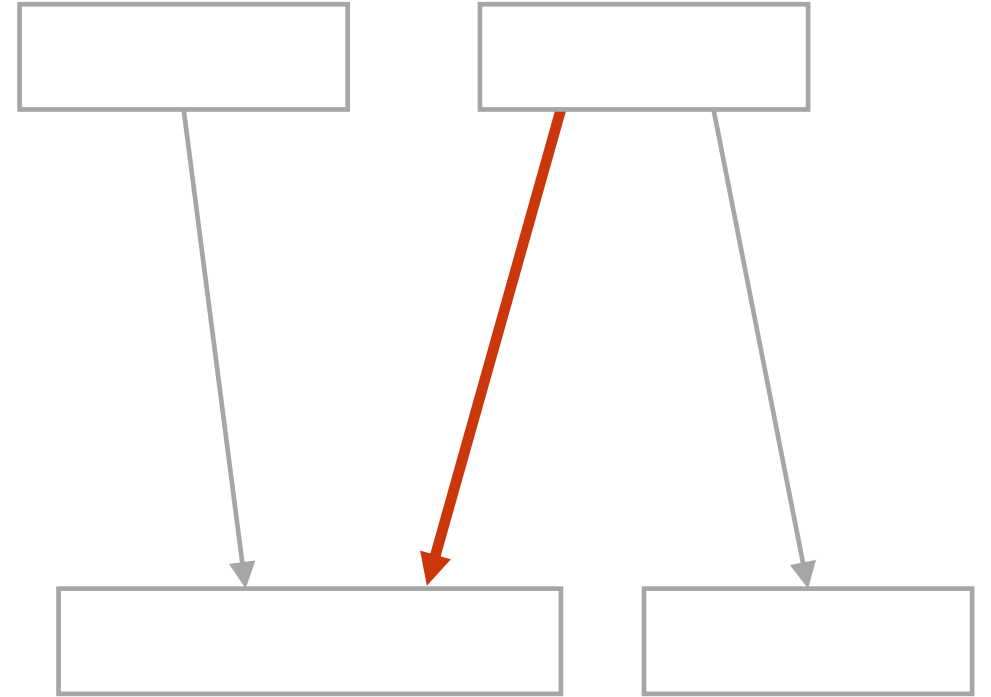
- ・ ループ内で i をインクリメントする
- ・ ループ後の z の計算には
 i の最後から2番目の値が使われる

→ 元々のコードとは違うコードが生成



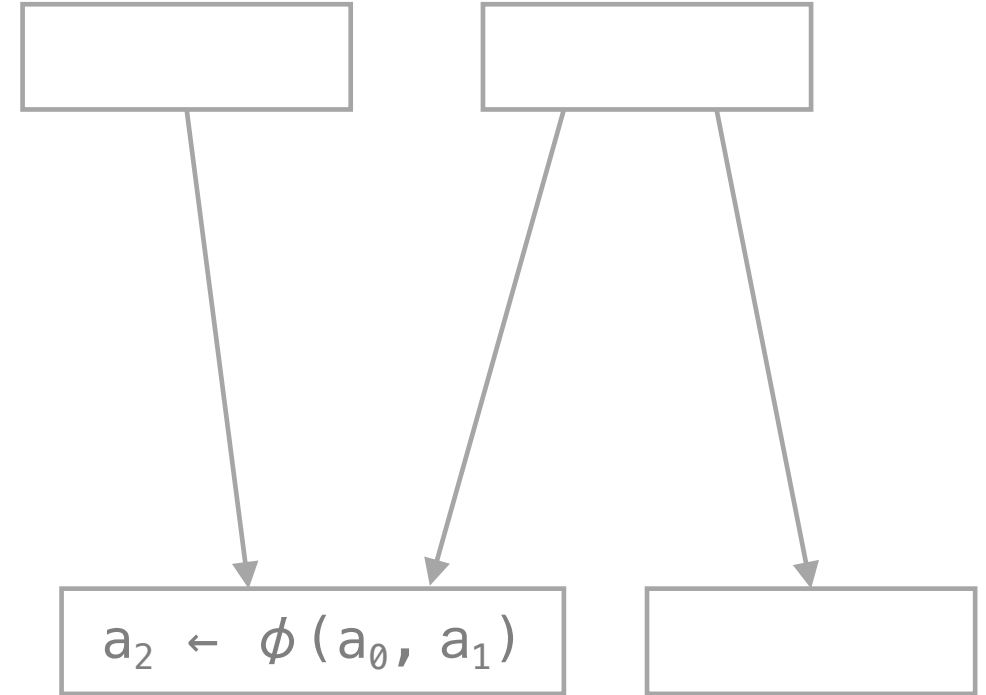
クリティカルエッジ

- ・ 始点のノードが複数の子を持ち、
終点のノードが複数の親を持つ辺



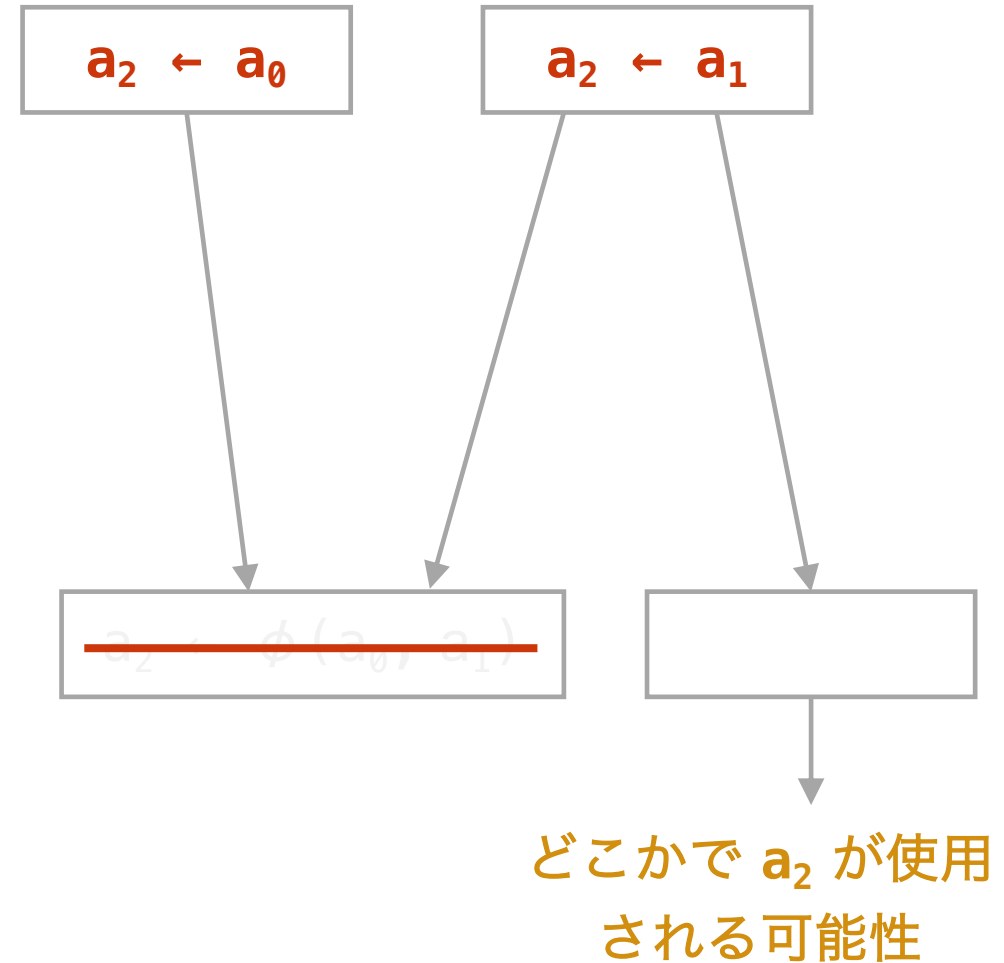
クリティカルエッジ

- ・ 始点のノードが複数の子を持ち、
終点のノードが複数の親を持つ辺
- ・ 始点のノードへのコピー操作の挿入は
live な変数の値を変更する可能性がある



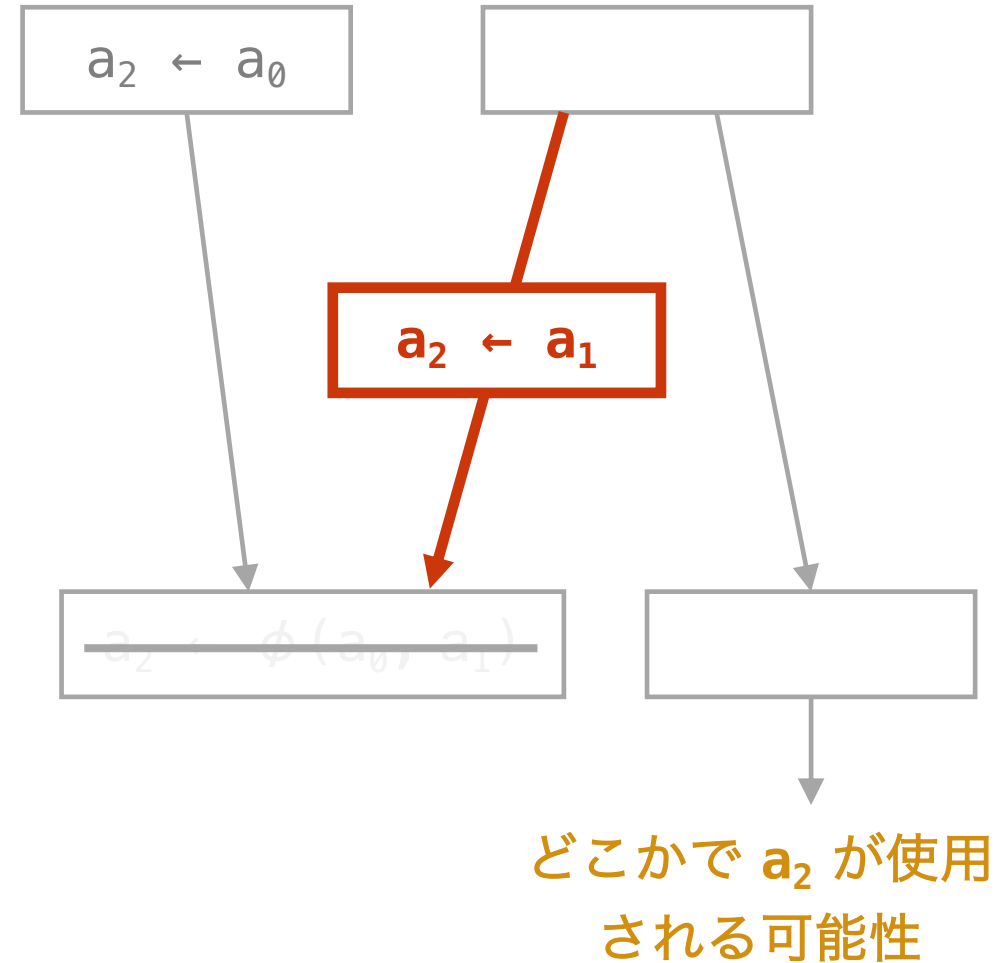
クリティカルエッジ

- ・ 始点のノードが複数の子を持ち、
終点のノードが複数の親を持つ辺
- ・ 始点のノードへのコピー操作の挿入は
live な変数の値を変更する可能性がある



クリティカルエッジ

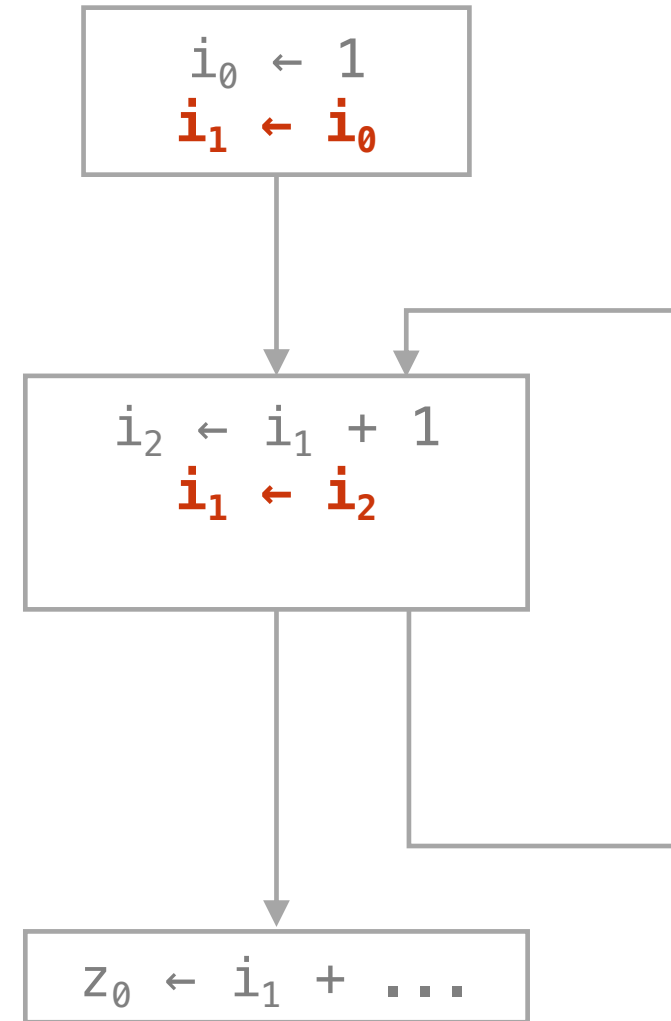
- ・ 始点のノードが複数の子を持ち、
終点のノードが複数の親を持つ辺
- ・ 始点のノードへのコピー操作の挿入は
live な変数の値を変更する可能性がある
- ・ クリティカルエッジを分割できれば
この問題は解決する



The Lost-Copy Problem の例

素朴な SSA 逆変換

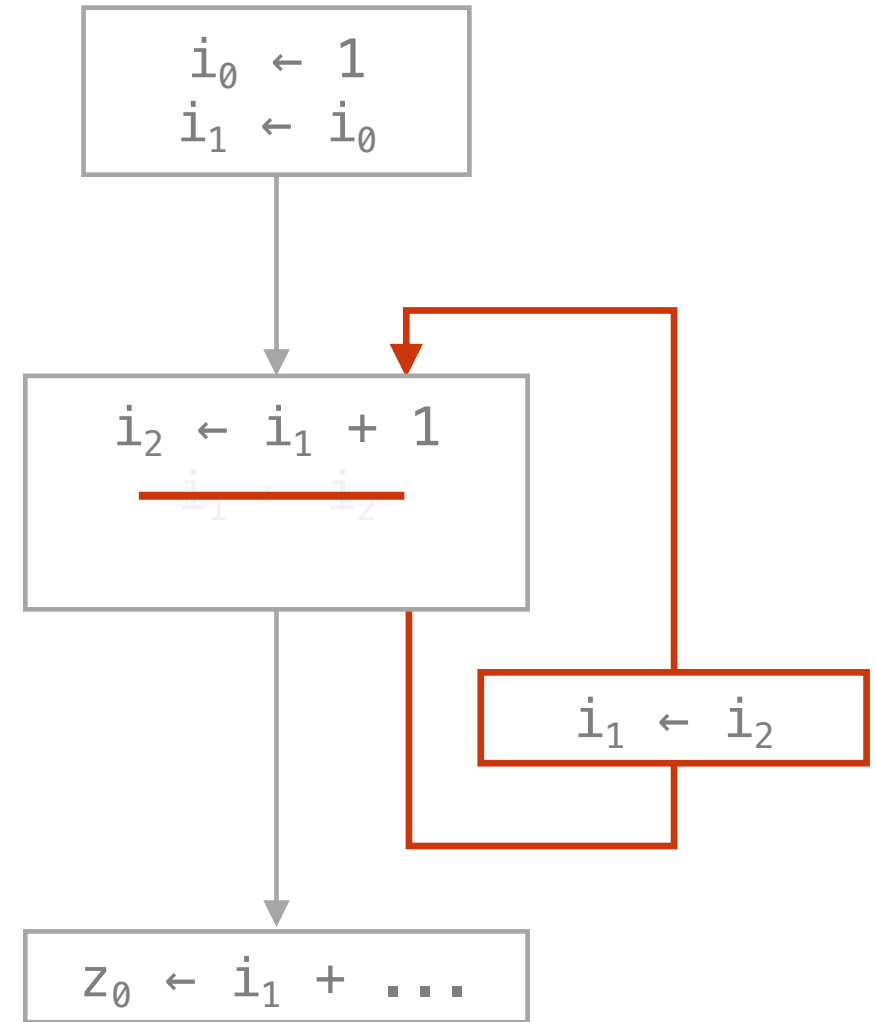
ϕ 関数の前の BB に適切なコピー操作を挿入



The Lost-Copy Problem の例

クリティカルエッジの分割

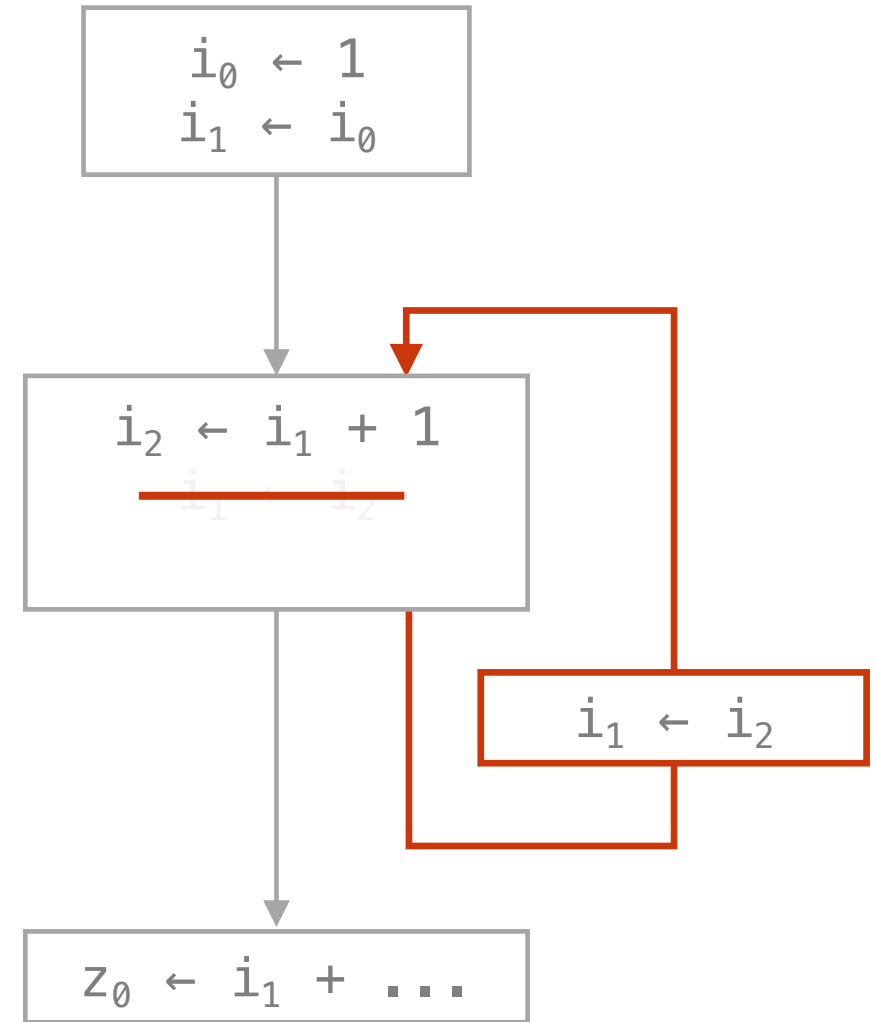
元々のコードと同じ意味のコードが生成



CFG

クリティカルエッジの分割

- すべてのクリティカルエッジを分割できれば、素朴な方法でも正しいコードを生成する
- CFG 上のクリティカルエッジを分割できない・すべきでない状況もある
 - 例えば, ジャンプ命令が一つ増えることによる影響



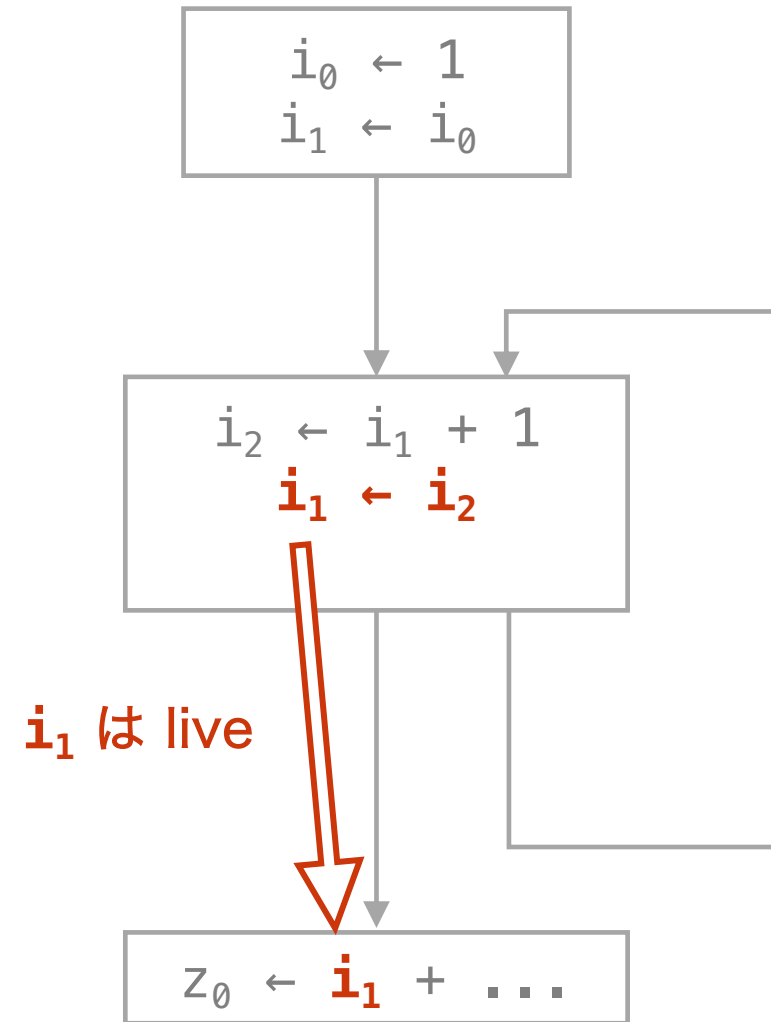
The Lost-Copy Problem の例

素朴な SSA 逆変換

φ 関数の前の BB に適切なコピー操作を挿入

クリティカルエッジを分割できない場合

- ・ 挿入ポイントでコピーのターゲットが
生きているか確認
- ・ 生きていれば新しい名前を導入



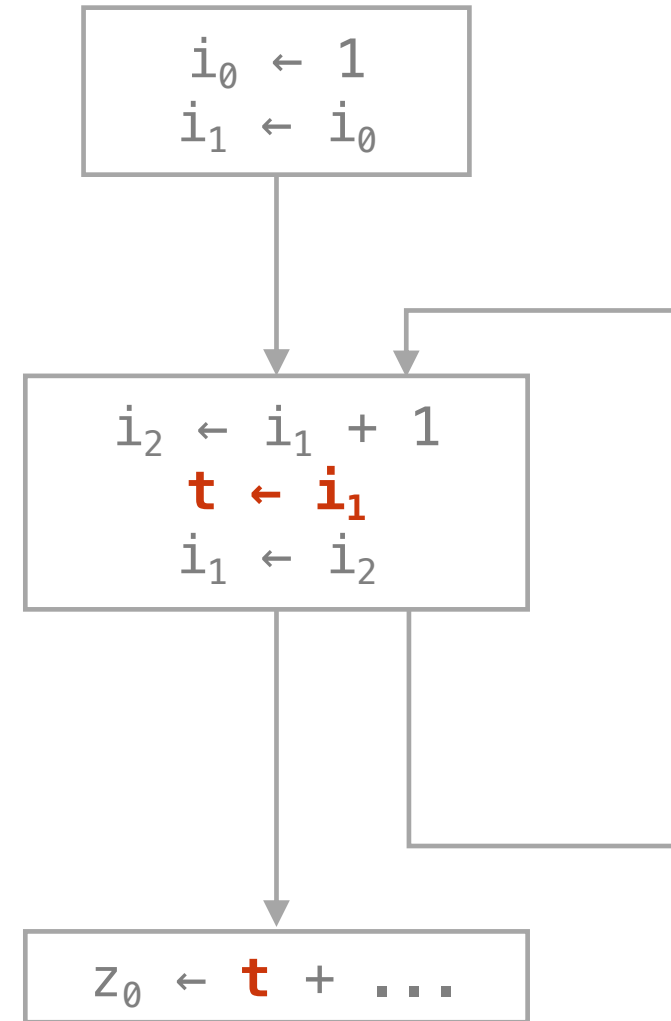
The Lost-Copy Problem の例

素朴な SSA 逆変換

φ 関数の前の BB に適切なコピー操作を挿入

クリティカルエッジを分割できない場合

- ・ 挿入ポイントでコピーのターゲットが
生きているか確認
- ・ 生きていれば新しい名前を導入



The Swap Problem

ϕ 関数のセマンティクス

1. どのエッジから来たかによって値が選択される
2. 同じブロック内の ϕ 関数は **並列に** 計算される

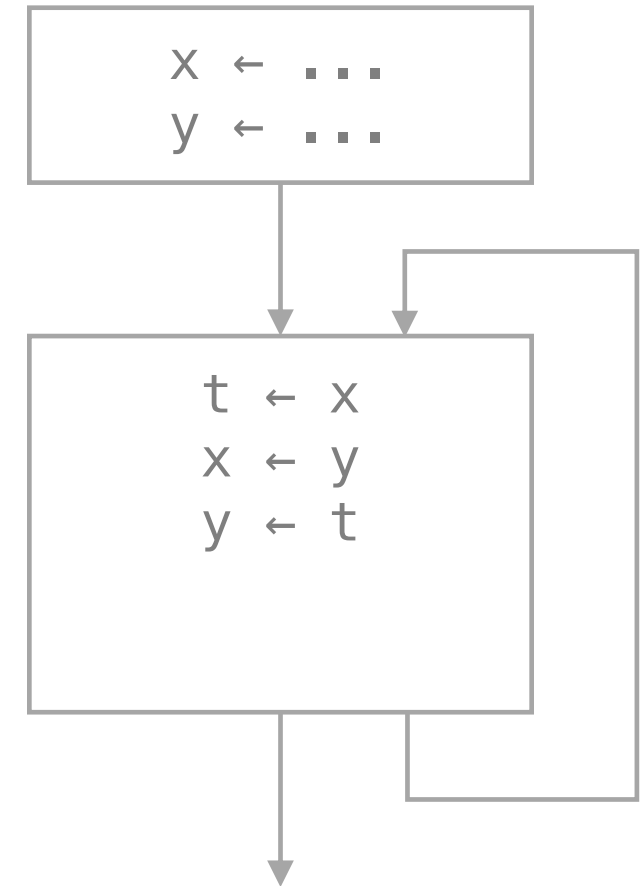
素朴な SSA 逆変換の問題点

素朴な SSA 逆変換は **並列な ϕ 関数** を **逐次コピー操作** に置き換える

The Swap Problem の例

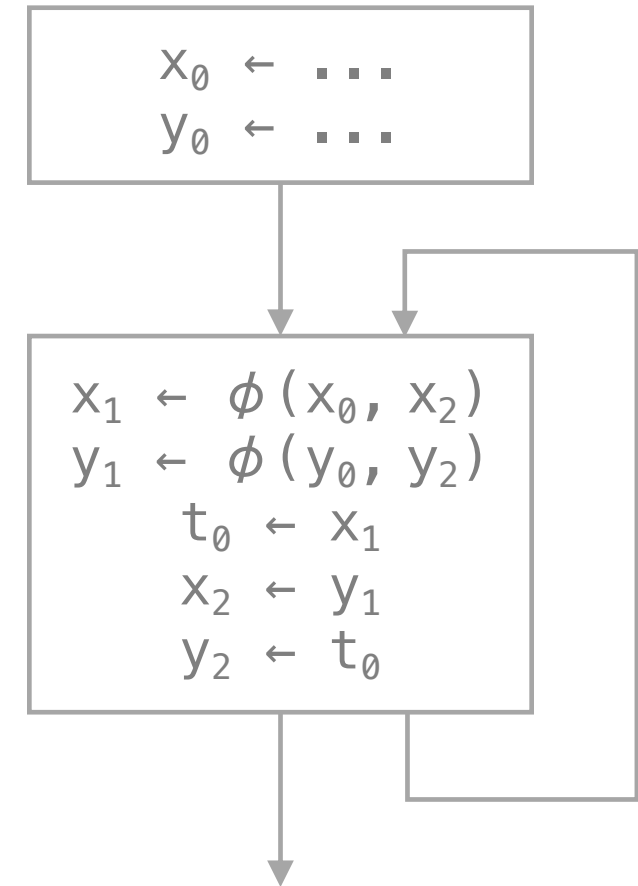
元々のコード

- ・ x と y の値をスワップする



The Swap Problem の例

Pruned SSA

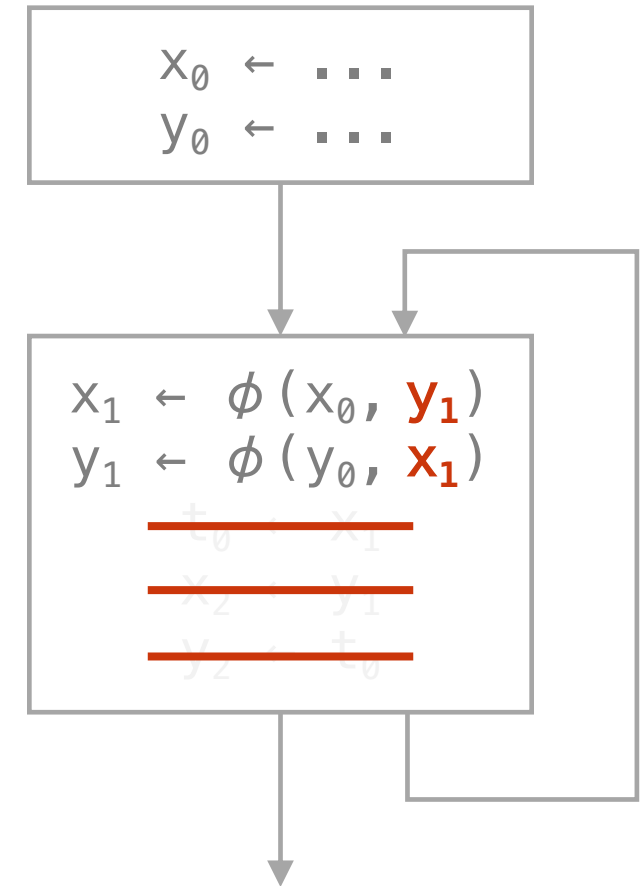


The Swap Problem の例

After Copy Folding

Copy Folding (コピー畳み込み) :

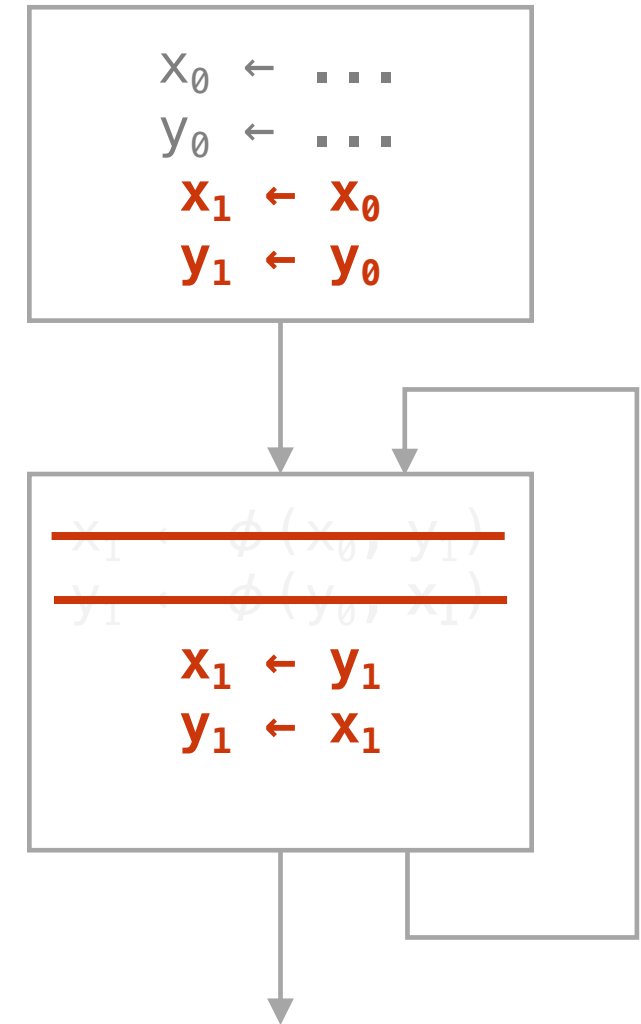
source と destination の名前を変更することで
不必要なコピー操作を削除する最適化



The Swap Problem の例

素朴な SSA 逆変換

ϕ 関数の前の BB に適切なコピー操作を挿入



The Swap Problem の例

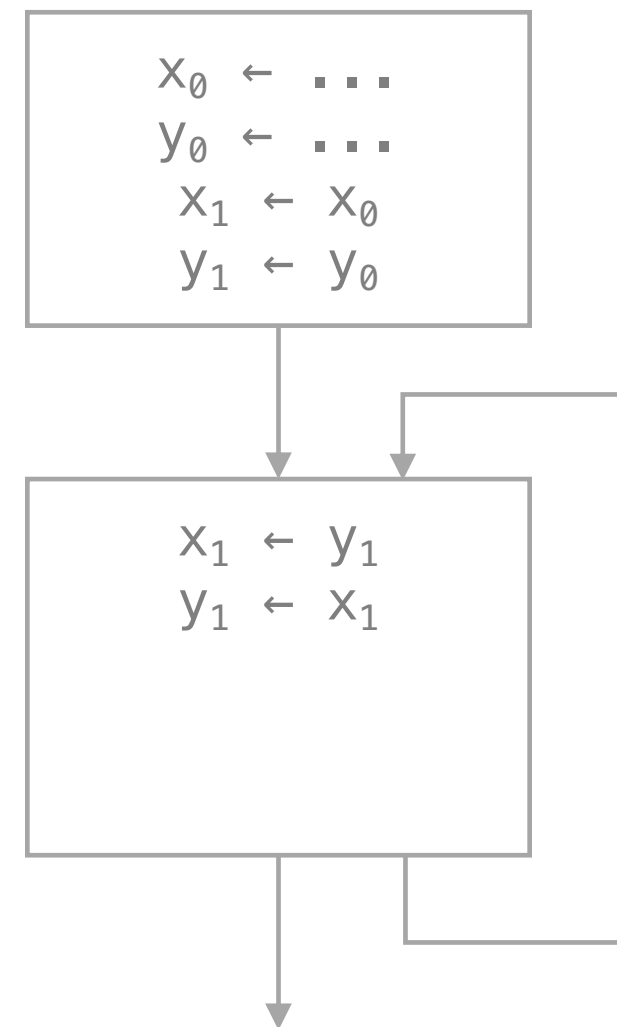
素朴な SSA 逆変換

ϕ 関数の前の BB に適切なコピー操作を挿入

元々のコード

・ x と y の値をスワップする

→ 元々のコードとは違うコードが生成

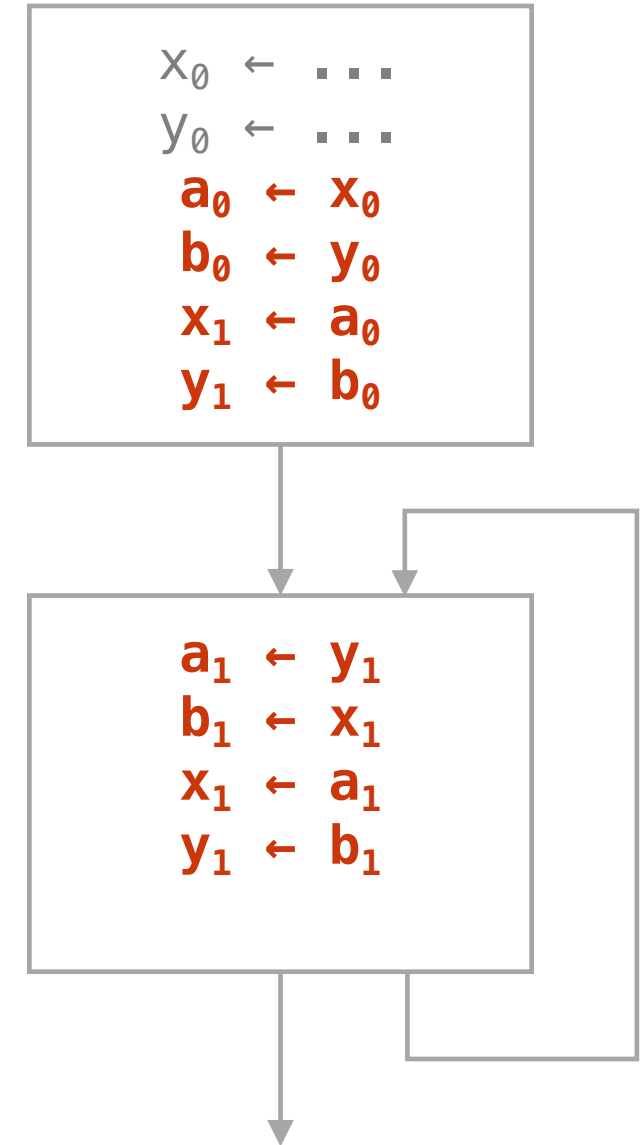


The Swap Problem の例

単純な解決法

ϕ 関数の各引数を一時的な変数にコピーする

→ 必要なコピー操作の数が2倍



The Swap Problem の例

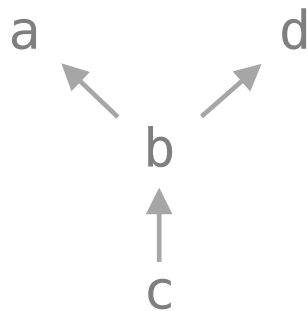
依存グラフを用いた解決法

- ・ 余分なコピーを減らすことができる

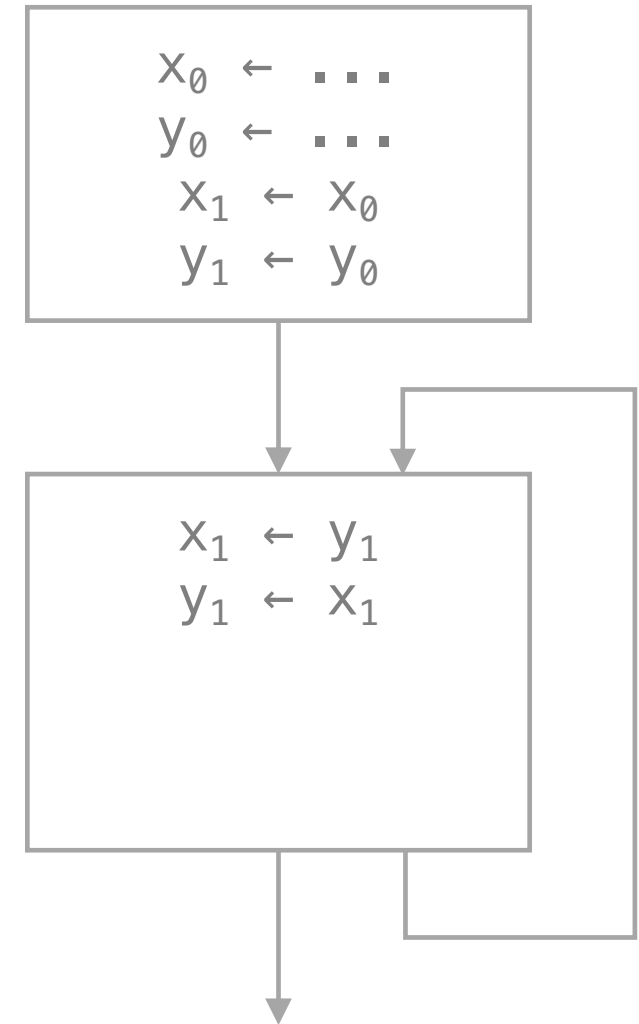
Dependence graph (依存グラフ) :

定義から使用に値の流れを表したグラフ

$a \leftarrow b$
 $d \leftarrow b$
 $b \leftarrow c$



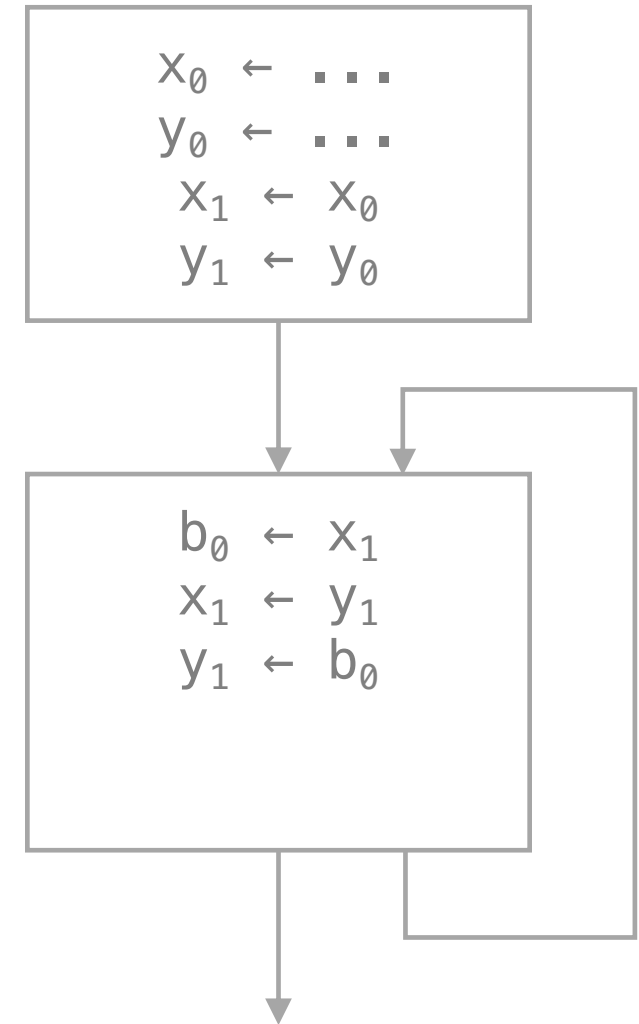
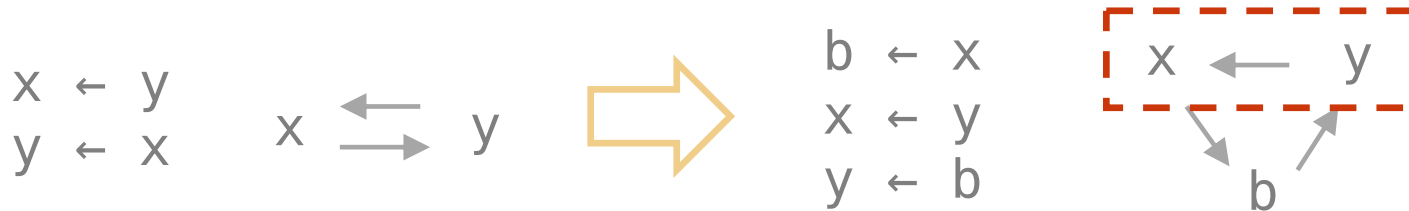
依存グラフ



The Swap Problem の例

依存グラフを用いた解決法

- ・ 余分なコピーを減らすことができる
- ・ 依存グラフにサイクルが含まれるとき
サイクルをなくすように新しい変数を導入



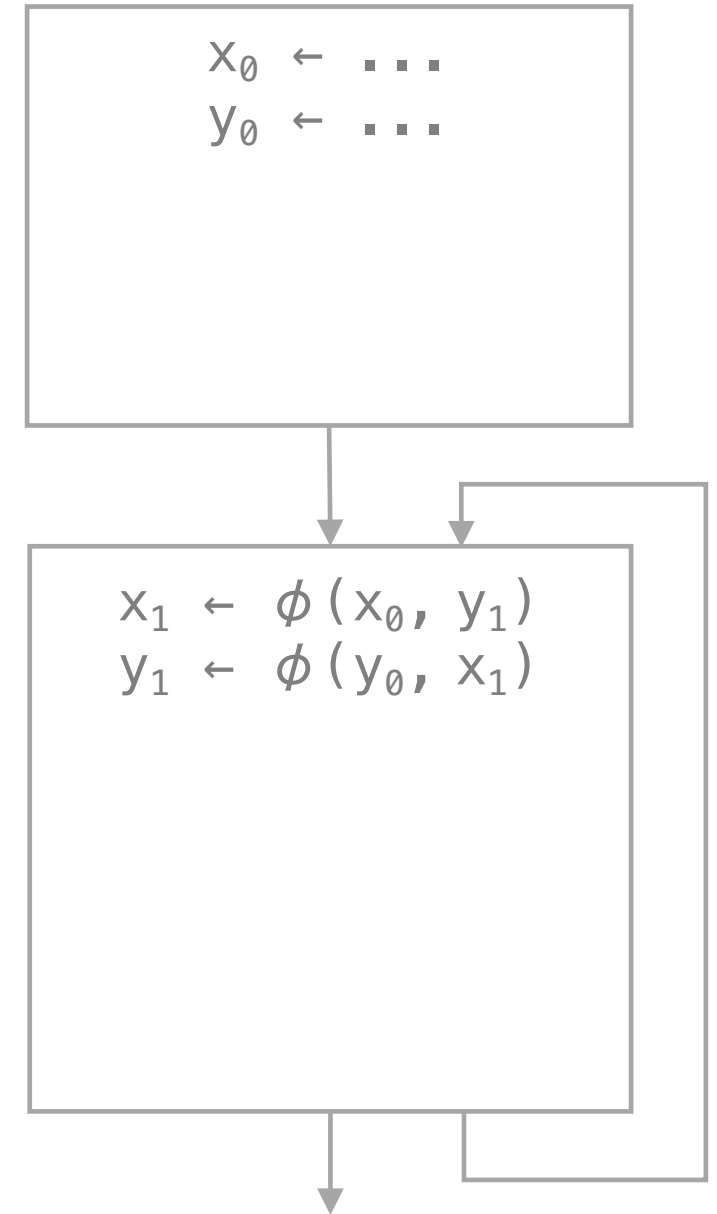
A Unified Approach to Out-of-SSA Translation

- ・ 素朴な SSA 逆変換とその問題点を2つ見てきた
 - ・ The Lost-Copy Problem
 - ・ The Swap Problem
- ・ 前に見た解決法は美しくないなので、統一された SSA 逆変換のアルゴリズムを説明する

Phase One

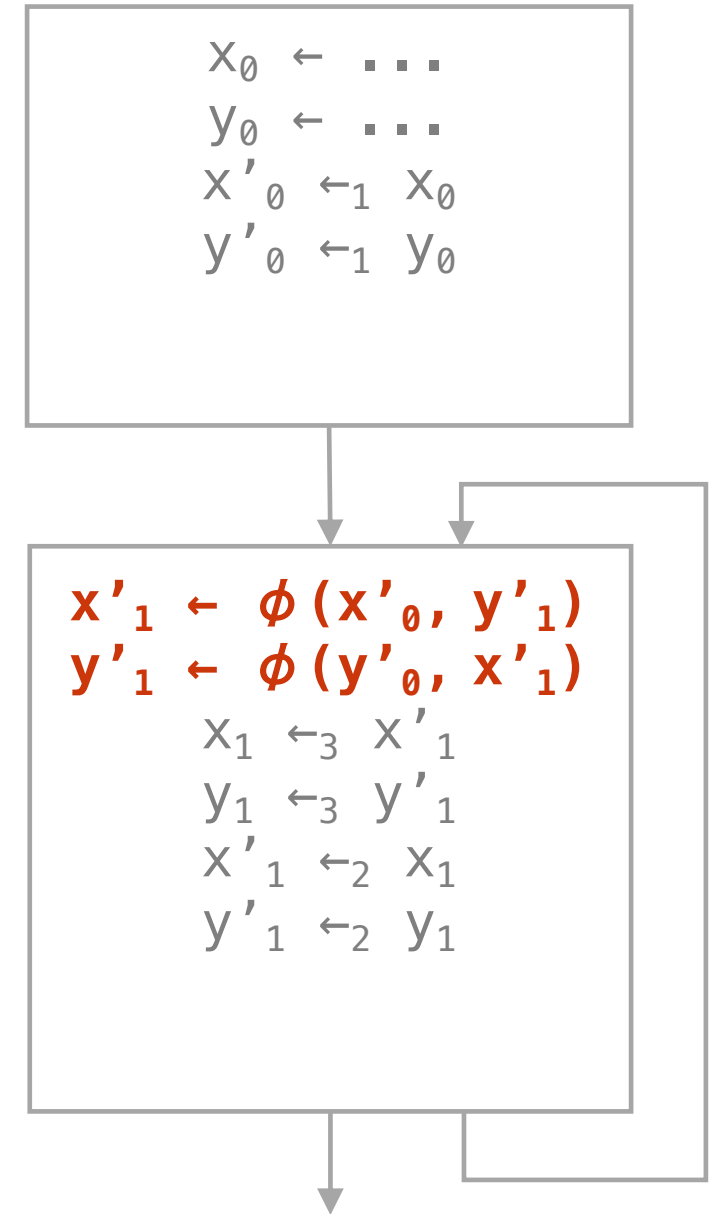
Pruned SSA Form, Copies Folded

- ・ この例も x と y の値をスワップするプログラム



Phase One

- ϕ 関数の名前空間を分離する



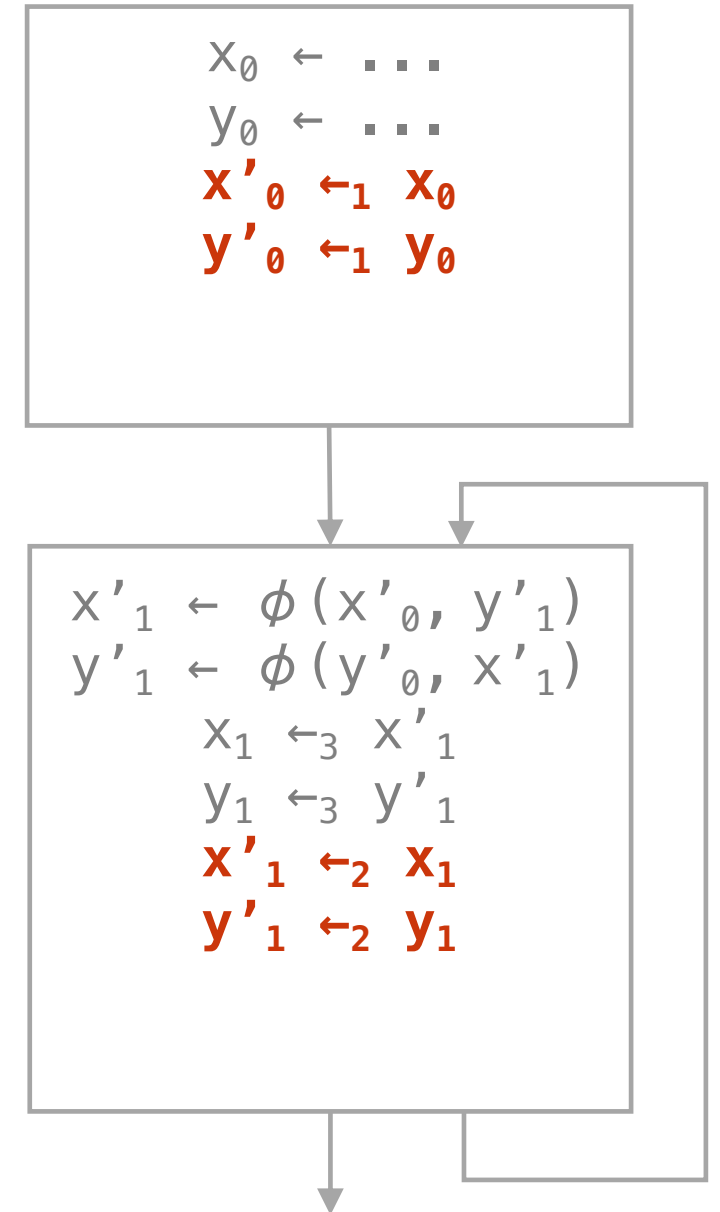
Phase One

- ϕ 関数の名前空間を分離する
 - 引数をコードの名前空間と対応させる
 - ϕ 関数の前の BB の最後にコピー操作を追加

並列コピーグループ: \leftarrow_i

同じグループの操作は並列に実行される

→ 命令順を入れ替えても結果は同じ



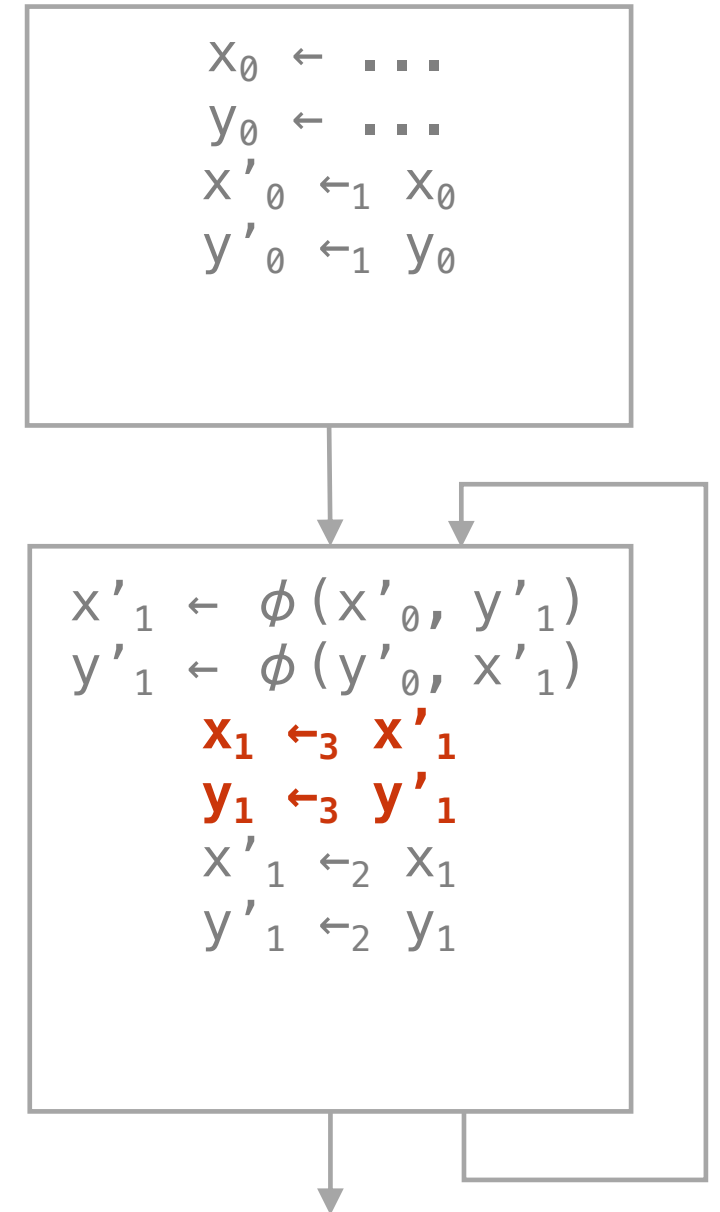
Phase One

- ϕ 関数の名前空間を分離する
 - 引数をコードの名前空間と対応させる
 - ϕ 関数の前の BB の最後にコピー操作を追加
- ϕ 関数の定義をコードの名前空間と対応させる

並列コピーグループ: \leftarrow_i

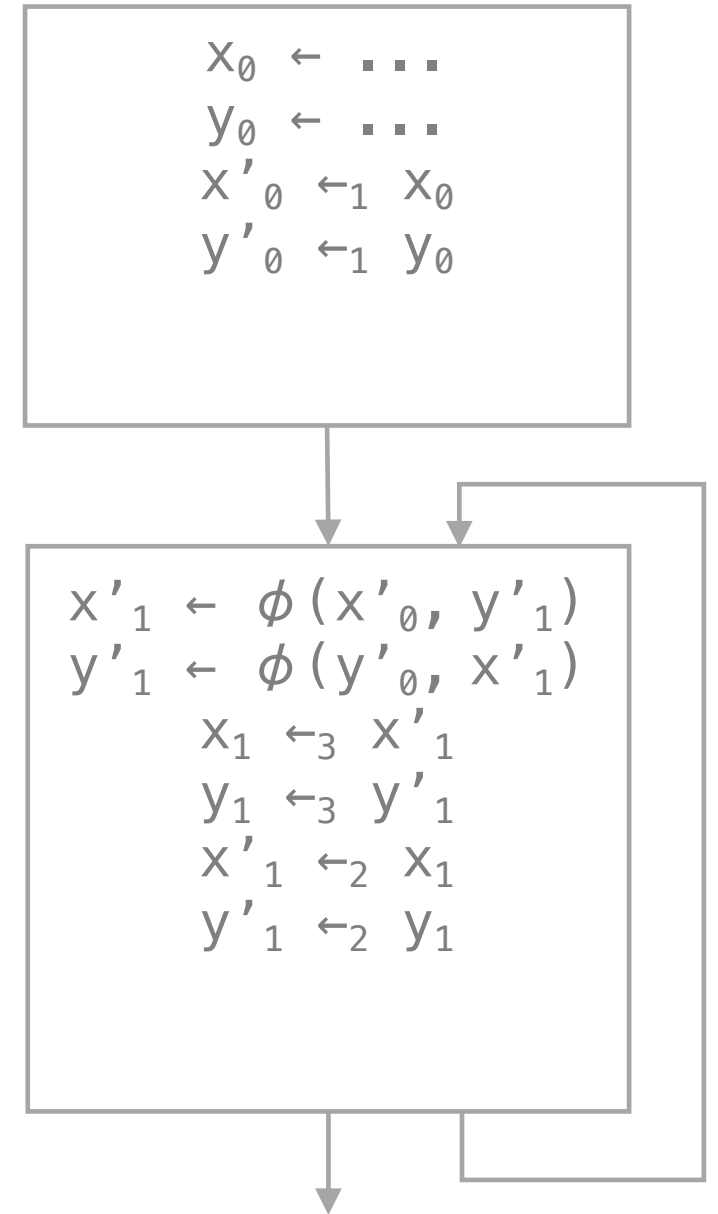
同じグループの操作は並列に実行される

→ 命令順を入れ替えても結果は同じ



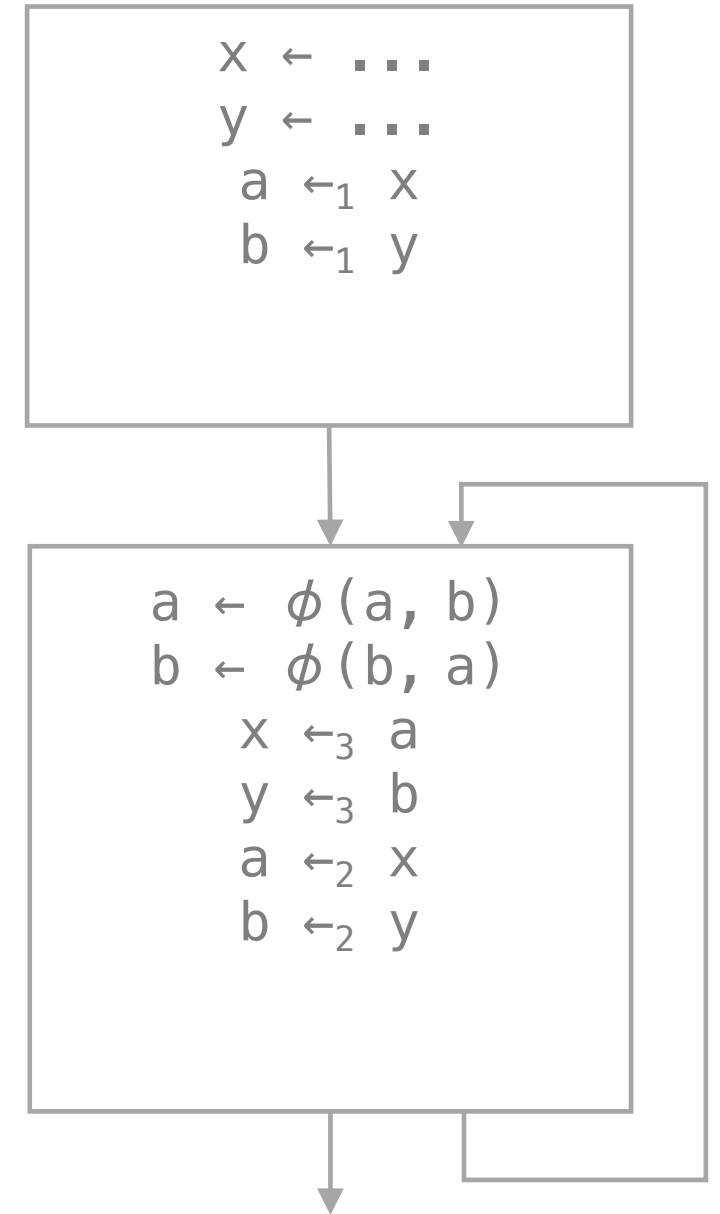
Phase One

- ϕ 関数の名前空間を分離する
 - ϕ 関数で使われる変数を外部と分離
 - ϕ 関数の外部 (コピー操作) で並列実行の影響を表せる



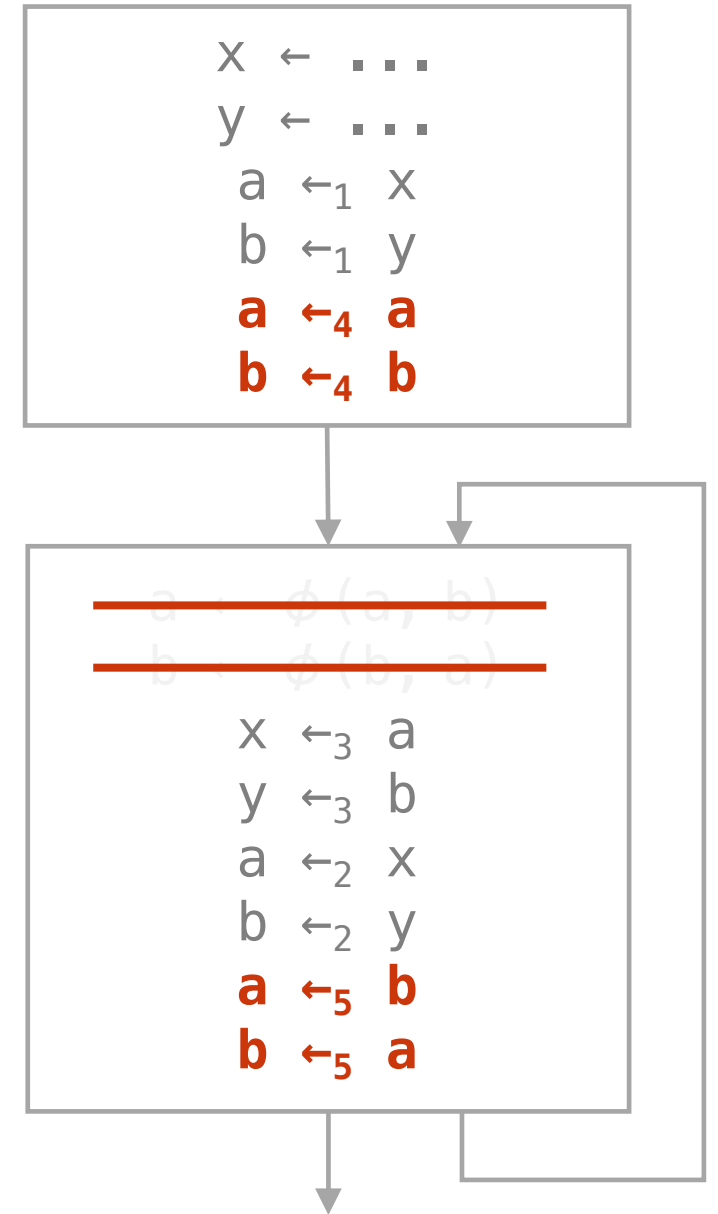
Phase One

- ϕ 関数の名前空間を分離する
 - ϕ 関数で使われる変数を外部と分離
 - ϕ 関数の外部 (コピー操作) で並列実行の影響を表せる
- (説明の都合上?)
元のコードと意味が同じになるため、
変数を改名し、変数の添え字を外す



Phase Two

- 素朴な手法のように、 ϕ 関数を削除し、コピー操作を追加
 - ϕ 関数の並列セマンティクスを表すため、並列コピー操作を挿入



Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
→ 並列コピーグループのデータ依存グラフを作成

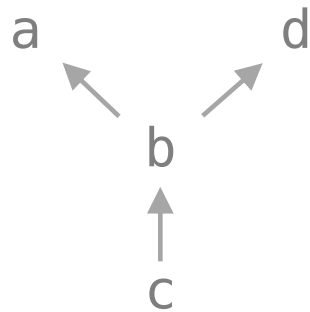
Data-dependence graph (データ依存グラフ)

定義から使用に値の流れを表したグラフ

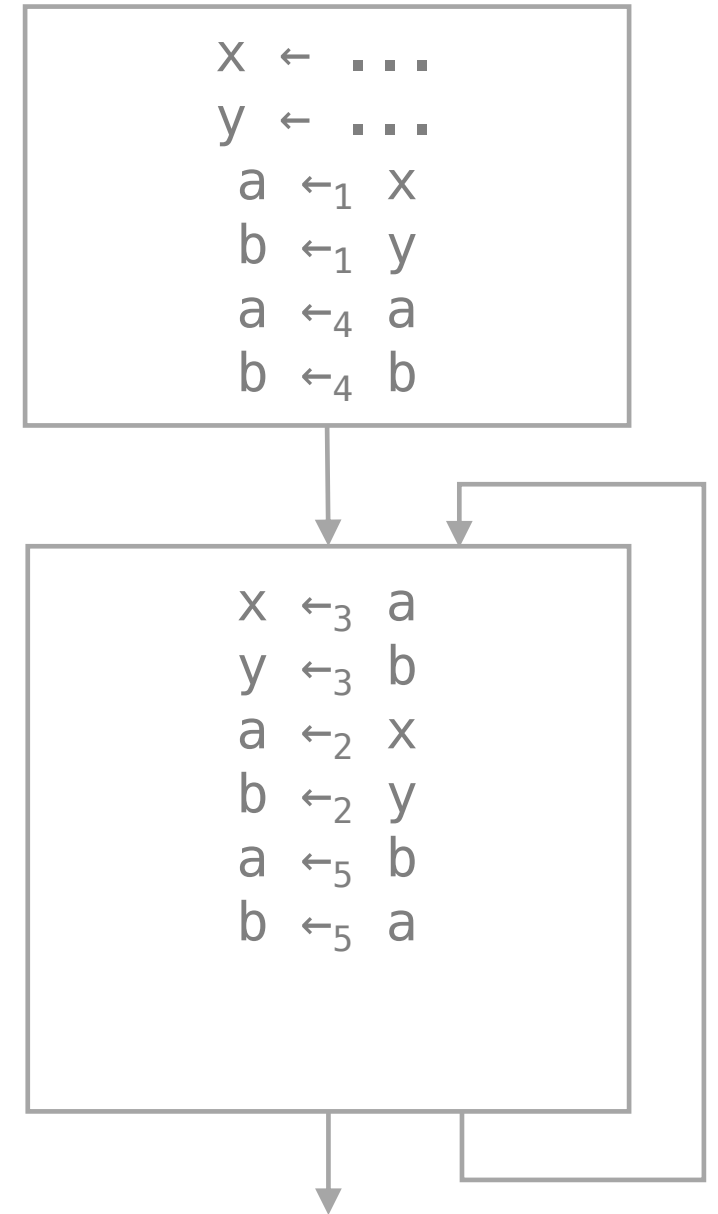
```

a ←1 b
b ←1 c
d ←1 b

```



データ依存グラフ

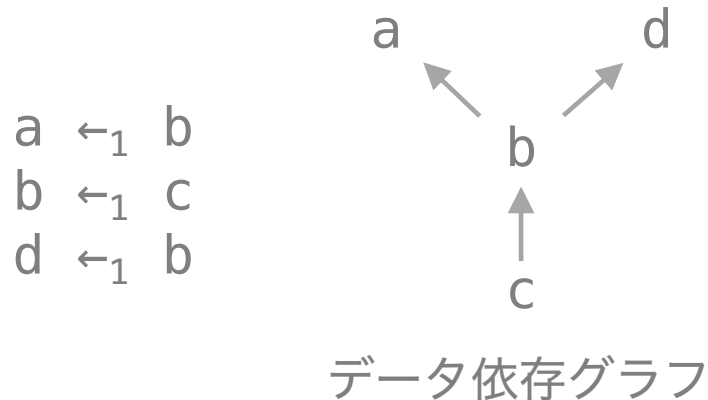


Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
- 並列コピーグループのデータ依存グラフを作成

依存グラフにサイクルがないとき

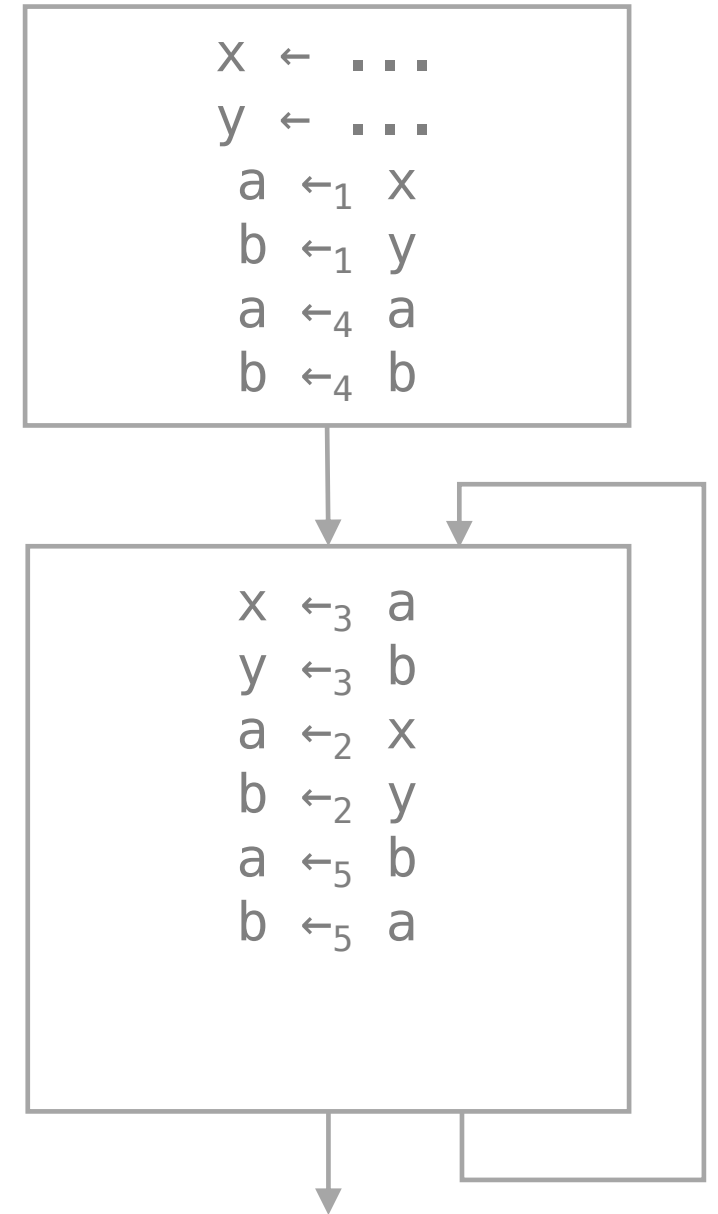
グラフが表す順序で逐次コピー操作に置き換える



Q.

$$\begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & c \\ d & \leftarrow & b \end{array}$$

の順序の制約は？

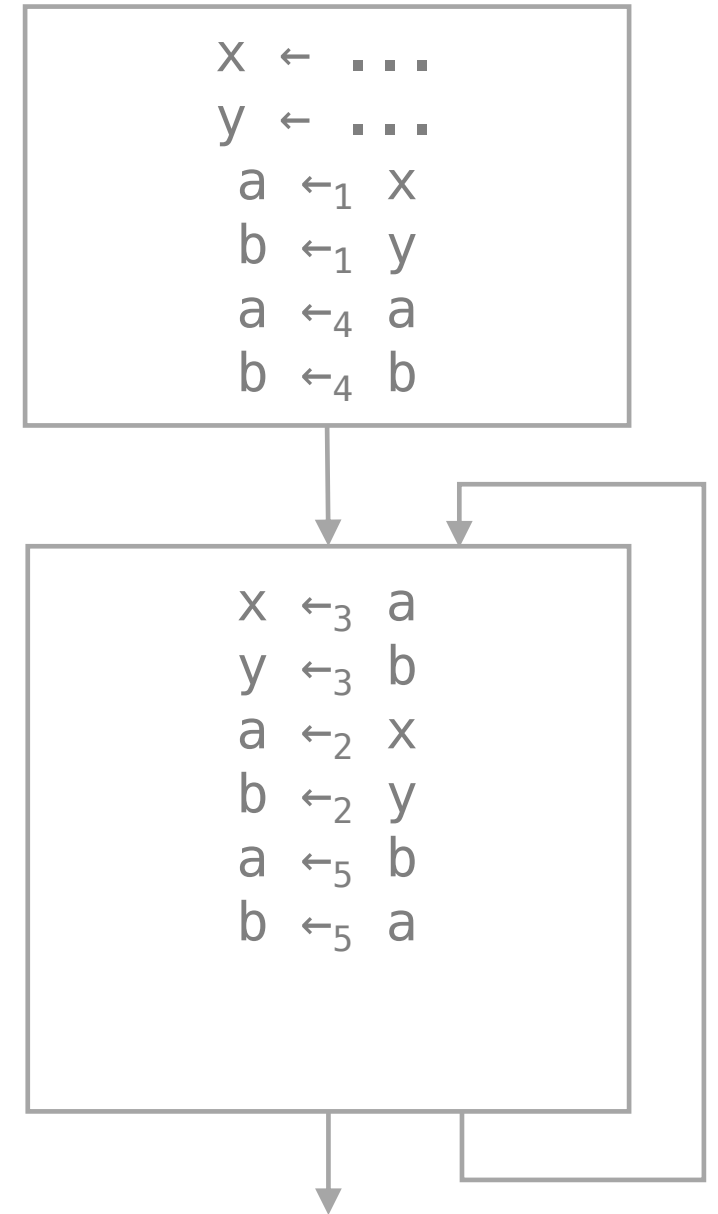
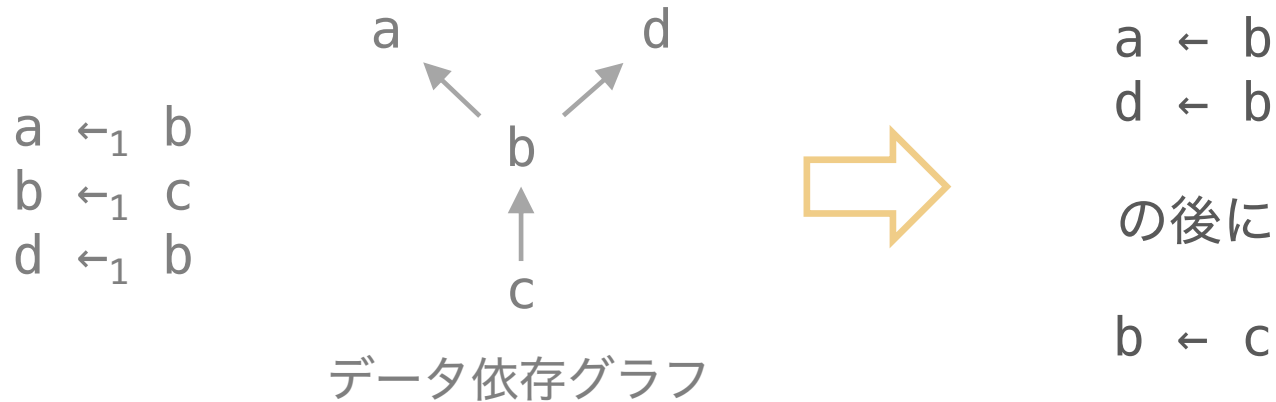


Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
- 並列コピーグループのデータ依存グラフを作成

依存グラフにサイクルがないとき

グラフが表す順序で逐次コピー操作に置き換える

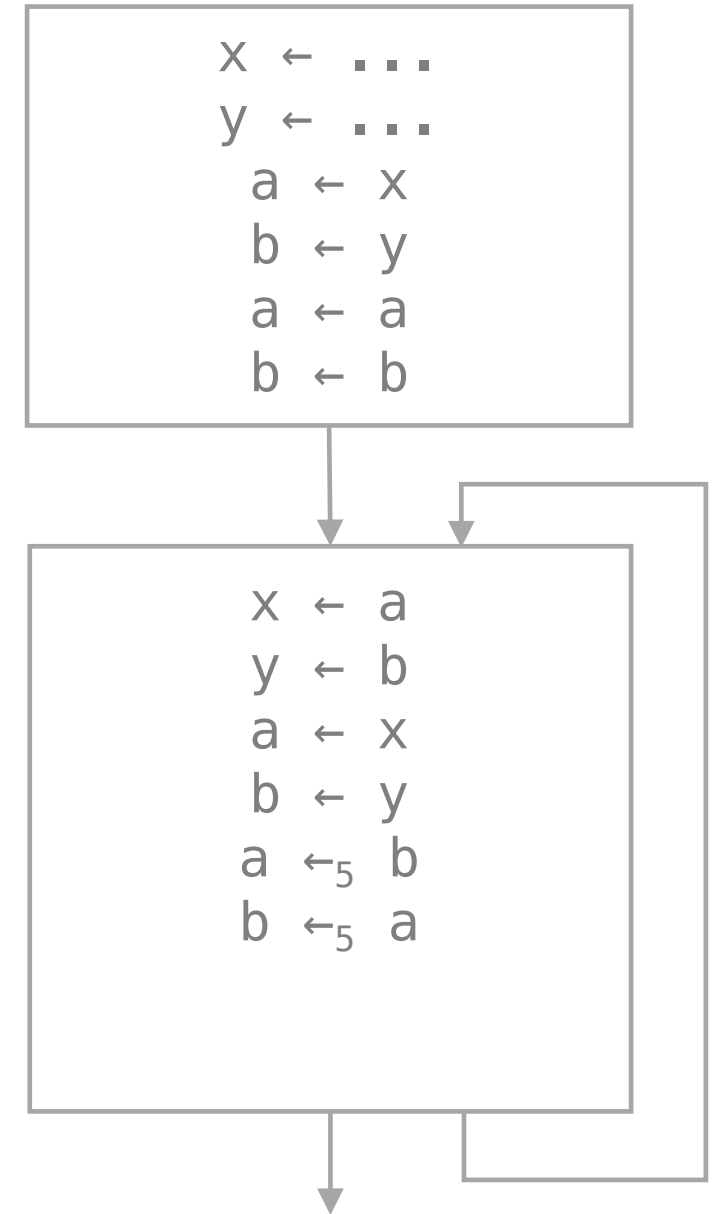
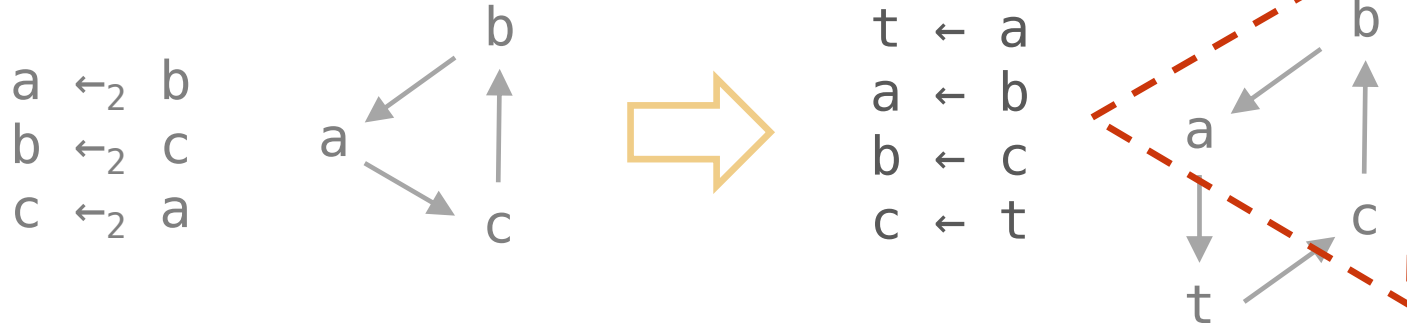


Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
- 並列コピーグループのデータ依存グラフを作成

依存グラフにサイクルがあるとき

サイクルをなくすようにコピー操作に書き換える

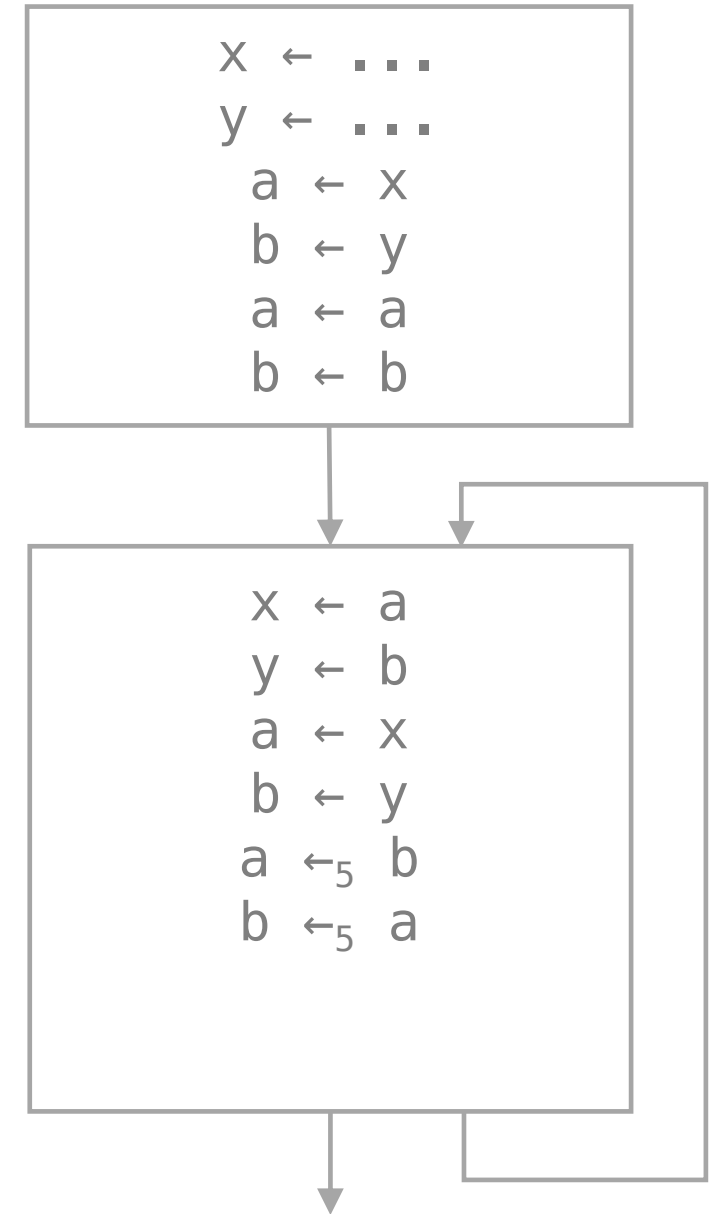
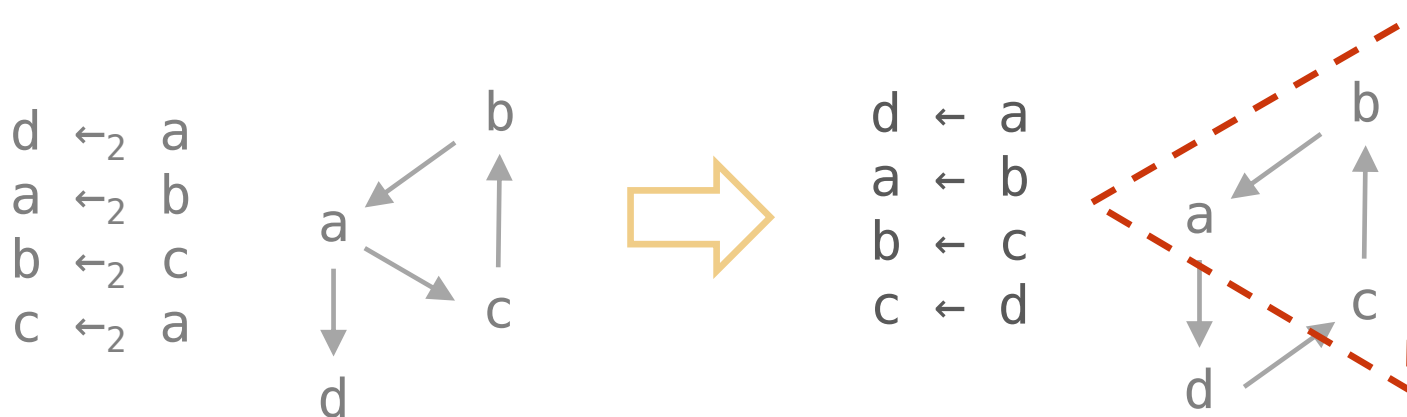


Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
- 並列コピーグループのデータ依存グラフを作成

依存グラフにサイクルがあるとき

サイクルがあっても新しい変数を追加しなくて良い場合も

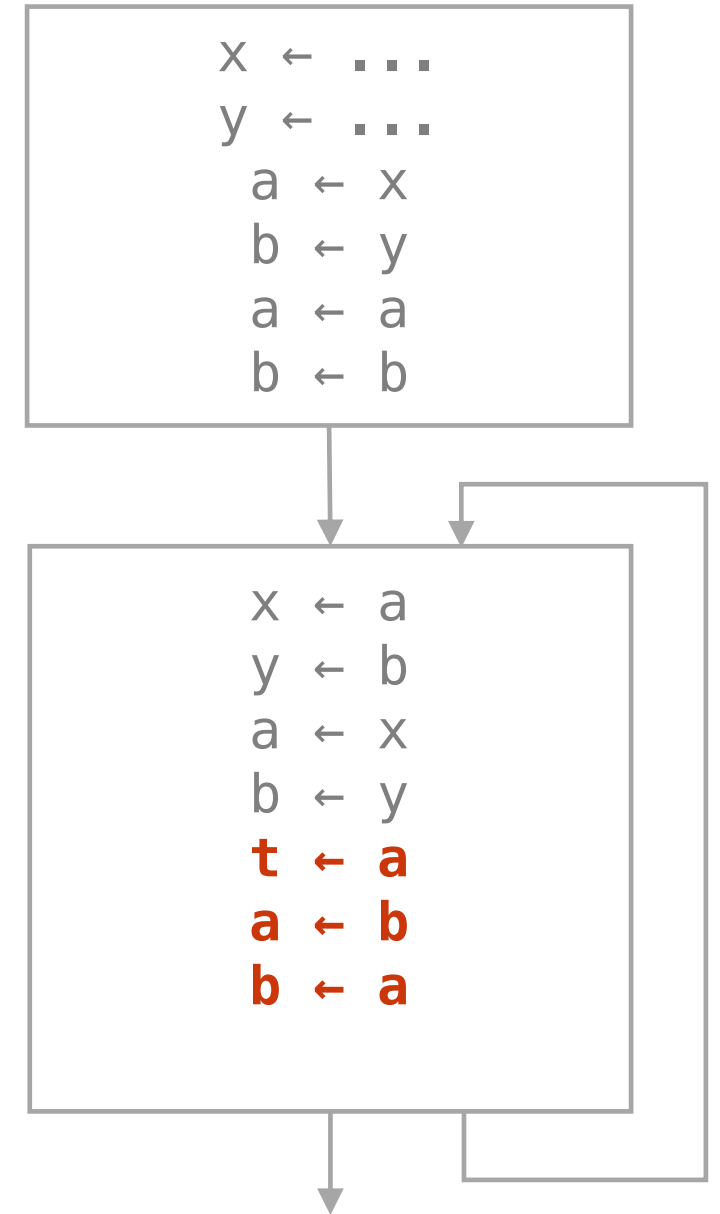
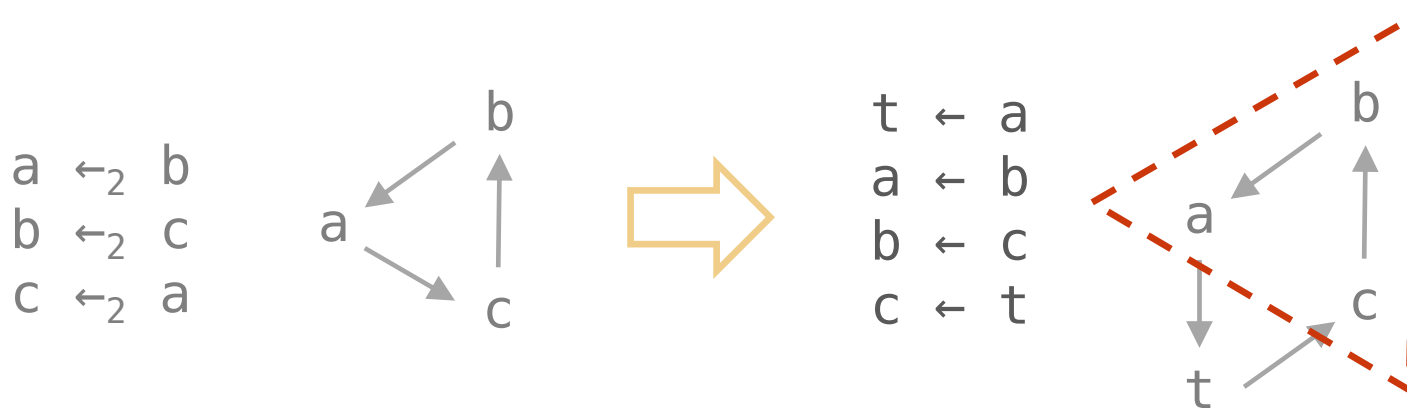


Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
- 並列コピーグループのデータ依存グラフを作成

依存グラフにサイクルがあるとき

サイクルをなくすようにコピー操作に書き換える

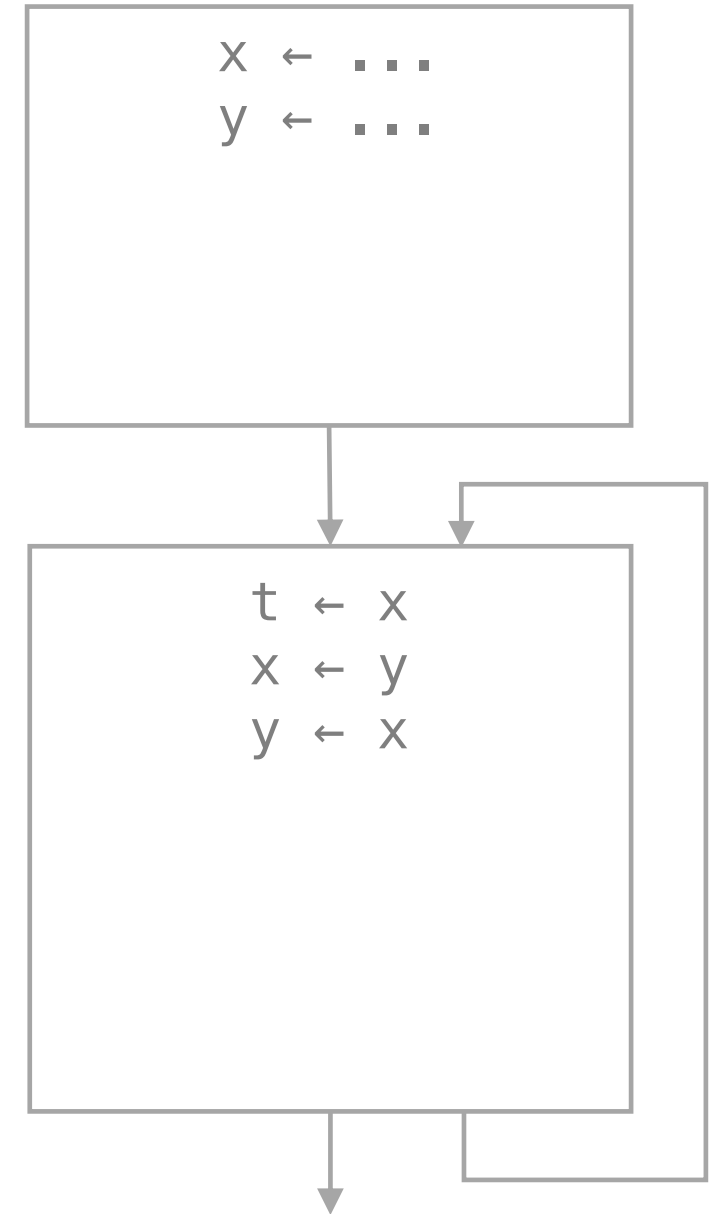


Phase Three

- ・ 並列コピーグループを意味が等しくなるように
逐次コピー操作に変更
 - 並列コピーグループのデータ依存グラフを作成

Copy folding 後

- ・ 元々のコードの意味を保持しつつ
SSA ではない形式に戻すことができた





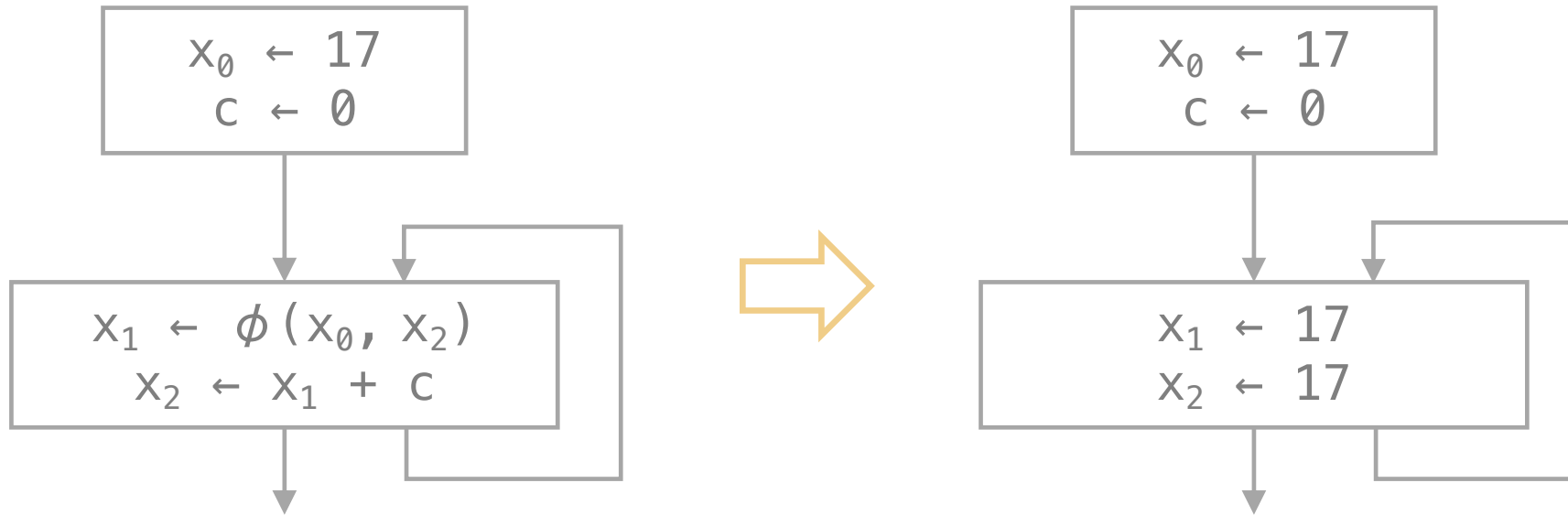
いわき vs 山口
at いわき

9.3.6 Using SSA Form

- SSA 形式を使うことで解析や最適化の質を上げることができる

Sparse Simple Constant Propagation (SSCP)

定数伝播問題



Meet-Semilattice

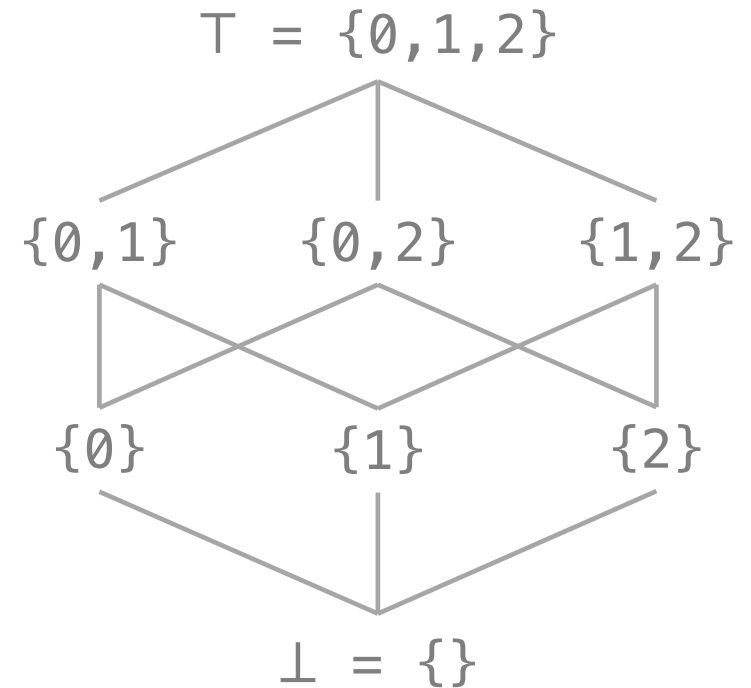
順序集合 (L, \geq) が下半束 (meet-semilattice) :

集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

meet operator \wedge

$\forall a, b, c \in L$

1. 冪等性 : $a \wedge a = a$
2. 可換性 : $a \wedge b = b \wedge a$
3. 結合性 : $a \wedge (b \wedge c) = (a \wedge b) \wedge c$



Meet-Semilattice

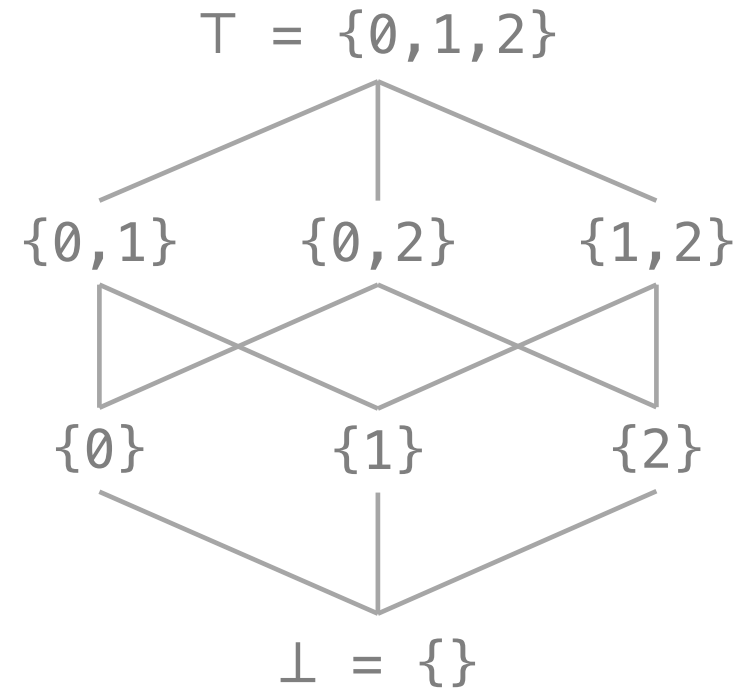
順序集合 (L, \geq) が下半束 (meet-semilattice) :

集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

meet operator \wedge

$$a \geq b \iff a \wedge b = b$$

$$a > b \iff a \geq b \text{ and } a \neq b$$



Meet-Semilattice

順序集合 (L, \geq) が下半束 (meet-semilattice) :

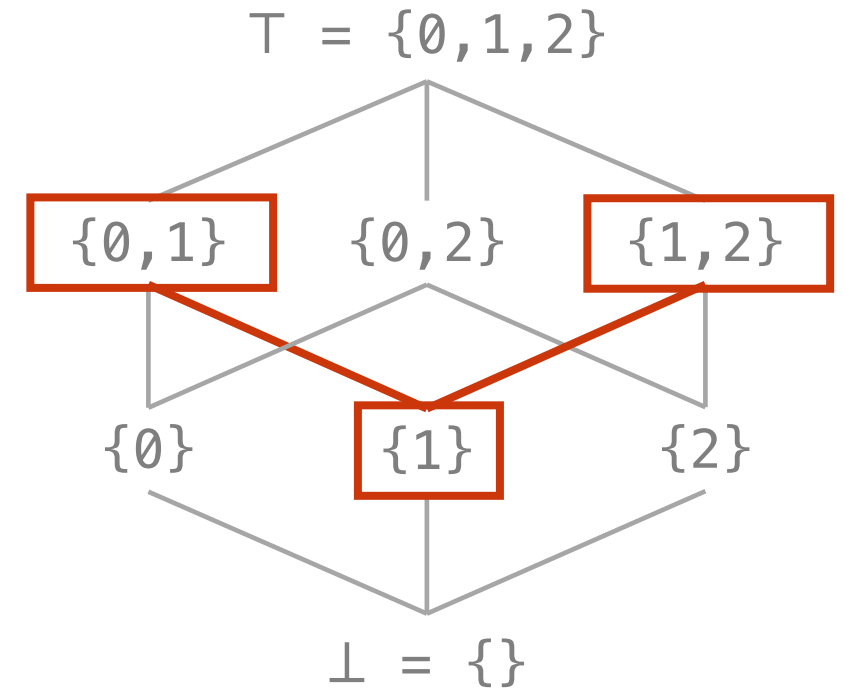
集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

L : $\{0, 1, 2\}$ の冪集合

二項関係: \subseteq

meet operator: \cap

$$\{0, 1\} \cap \{1, 2\} = \{1\}$$



Meet-Semilattice

順序集合 (L, \geq) が下半束 (meet-semilattice) :

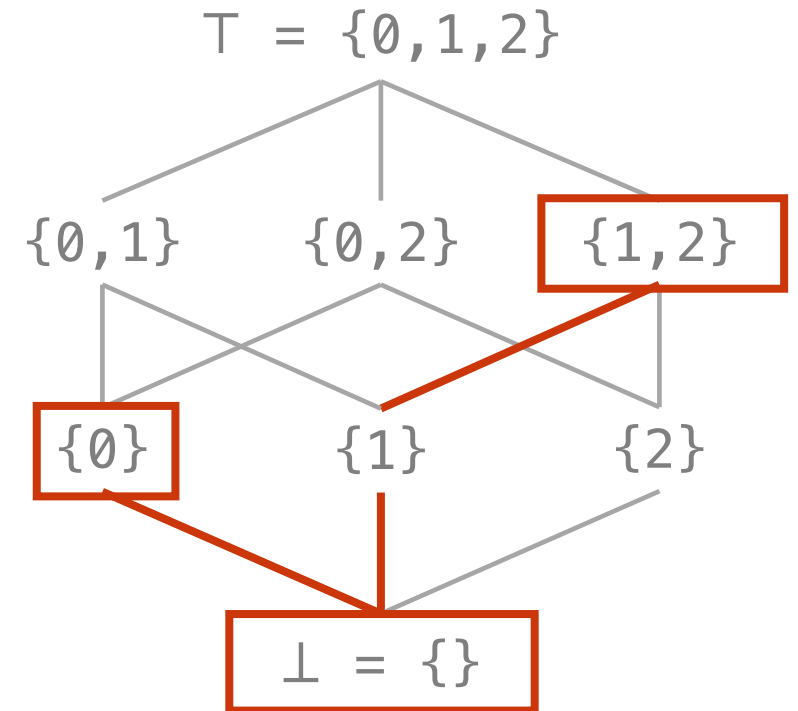
集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

L : $\{0, 1, 2\}$ の冪集合

二項関係: \subseteq

meet operator: \cap

$$\{0\} \cap \{1, 2\} = \{\}$$



Meet-Semilattice

順序集合 (L, \geq) が下半束 (meet-semilattice) :

集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

bottom : \perp

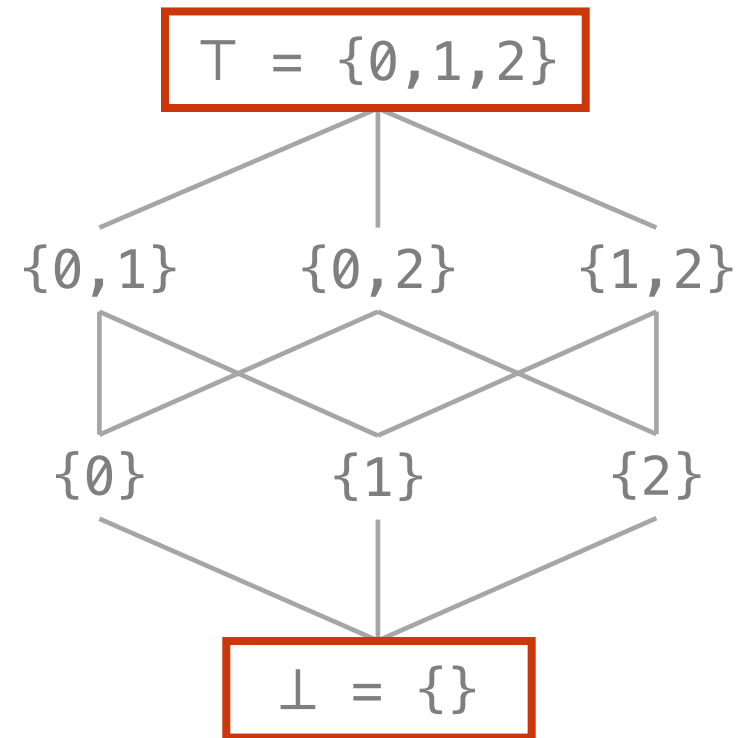
→ すべての要素の下限

$$\forall a \in L, a \wedge \perp = \perp, \text{ and } \forall a \in L, a \geq \perp$$

top : \top

→ すべての要素の上限 (なくてもよい)

$$\forall a \in L, a \wedge \top = a, \text{ and } \forall a \in L, \top \geq a$$



Meet-Semilattice

順序集合 (L, \geq) が下半束 (meet-semilattice) :

集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

L: 自然数の集合

二項関係: 通常の数的大小関係

Q. meet operator は?

$$a \geq b \iff a \wedge b = b$$

⋮
3
2
1
0

Meet-Semilattice

順序集合 (L, \geq) が下半束 (meet-semilattice) :

集合の任意の2元 x, y に対して、
それらの下限 $x \wedge y$ が存在する

L : 自然数の集合

二項関係: 通常の数的大小関係

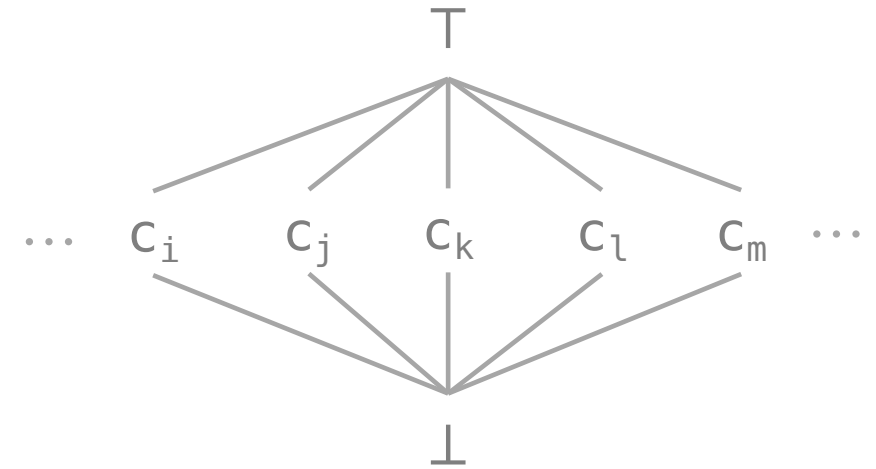
meet operator : \min

$$a \geq b \iff a \wedge b = b$$

\vdots
 $|$
 3
 $|$
 $|$
 2
 $|$
 $|$
 1
 $|$
 0

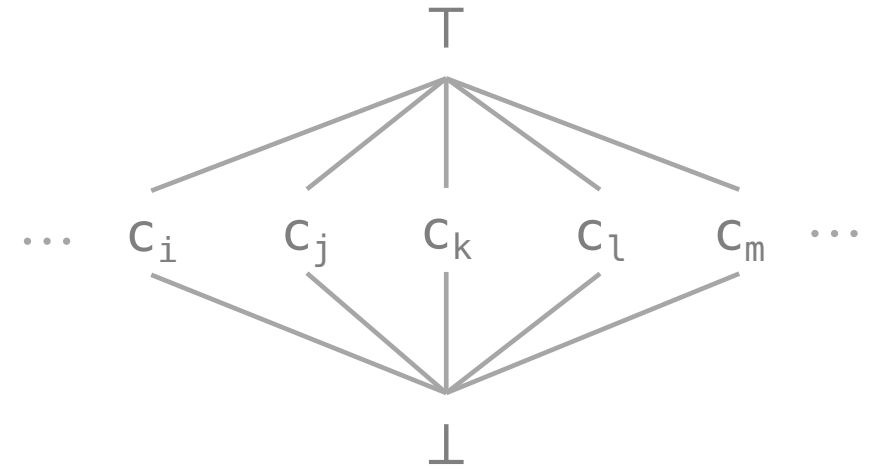
SSCP

- ・ 定数伝播のアルゴリズム
- ・ 各 SSA 名について右図のような半束を構築
- ・ \perp , \top , 定数値の無限集合
 - \top : 定数かどうか分からない
→ その値に関する情報を今後発見するかも
 - \perp : 定数ではない
- ・ 2つのステップ
 1. 初期化フェーズ
 2. 伝播フェーズ



SSCP : 初期化フェーズ

- $\text{Value}(n)$: SSA名 n の半束上の値
- $\text{Value}(n)$ を以下のように初期化
 1. n が ϕ 関数で定義された変数なら \perp
 2. n が定数かどうか分からないなら \perp
 3. n が定数 c_i なら c_i
 4. n が定数でないなら T
 - n が外部の入力によって与えられる場合など



T : 定数かどうか分からない

\perp : 定数ではない

SSCP : 伝播フェーズ

- ・ 定数・定数ではない SSA 名の使用箇所
定義された SSA 名に情報を伝播

→ `worklist` の初期値は定数・定数ではない
SSA 名すべての集合

$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) != ⊥:
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : 伝播フェーズ

- ・ 定数・定数ではない SSA 名の使用箇所
で定義された SSA 名に情報を伝播
- ・ m は n の使用箇所
で定義された変数

$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) != ⊥:
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : 伝播フェーズ

- ・ 定数・定数ではない SSA 名の使用箇所
で定義された SSA 名に情報を伝播
- ・ m は n の使用箇所
で定義された変数
- ・ $\text{Value}(m)$ を計算する

Q. $\text{value}(n) = \perp$ のとき

$$m \leftarrow n + 10$$
$$\text{value}(m) = ?$$
$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) !=  $\perp$ :
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```


SSCP : 伝播フェーズ

- ・ 定数・定数ではない SSA 名の使用箇所
で定義された SSA 名に情報を伝播
- ・ m は n の使用箇所
で定義された変数
- ・ $\text{Value}(m)$ を計算する

Q. $\text{value}(n) = \perp$ のとき

$$m \leftarrow n + 10$$
$$\text{value}(m) = \perp$$
$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) !=  $\perp$ :
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : 伝播フェーズ

- ・ 定数・定数ではない SSA 名の使用箇所
で定義された SSA 名に情報を伝播
- ・ m は n の使用箇所
で定義された変数
- ・ $\text{Value}(m)$ を計算する

Q. $\text{value}(n) = \perp$ のとき

$$m \leftarrow n * 0$$
$$\text{value}(m) = ?$$
$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) !=  $\perp$ :
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : 伝播フェーズ

- ・ 定数・定数ではない SSA 名の使用箇所
で定義された SSA 名に情報を伝播
- ・ m は n の使用箇所
で定義された変数
- ・ $\text{Value}(m)$ を計算する

Q. $\text{value}(n) = \perp$ のとき

$$m \leftarrow n * 0$$

$$\text{value}(m) = 0$$

$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) !=  $\perp$ :
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : 伝播フェーズ

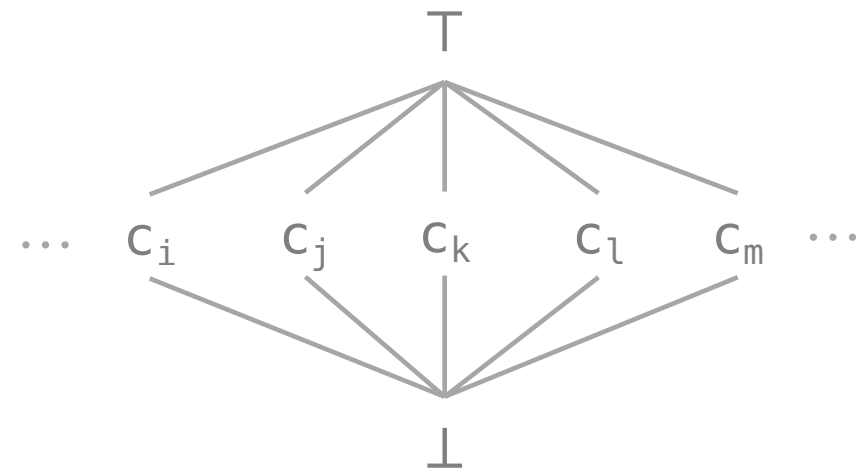
- ・ 定数・定数ではない SSA 名の使用箇所
で定義された SSA 名に情報を伝播
- ・ m は n の使用箇所
で定義された変数
- ・ $\text{Value}(m)$ を計算する
- ・ $\text{Value}(m)$ が変化していれば
`worklist` に追加
→ 不動点に到達するまで

$$m \leftarrow n + 10$$

```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) != ⊥:
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : 複雑性

- SSCP は各 SSA 名の Value が不動点に到達するまで計算する
 - 各 SSA 名の Value は最大2回変化
 - Value が変化すると worklist に追加される
 - 各 SSA 名の使用箇所で評価
- 評価回数は最悪でもコード内の使用回数の2倍



```
1 while len(worklist) > 0:
2     n = worklist.remove()
3     for op in use_operations(n):
4         m = define_SSA_name(op)
5         if Value(m) != ⊥:
6             t = Value(m)
7             Value(m) = result(op)
8             if Value(m) != t:
9                 worklist.add(m)
```

SSCP : SSA を使う利点

- ・ SSCP で SSA を使用する利点はシンプルさと効率性
- ・ 古典的なデータフロー（9.2節のような）解析を考えると各基本ブロックで (variable, value) の組を考えることになる
 - どの程度で収束するのか分かりにくい
 - ブロック内で Value が変化しない変数も再評価する場合も

