Static single assignment

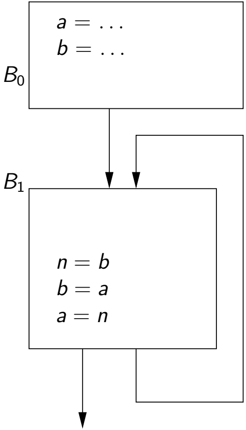
SSA with dominance property

- Unique definition for each variable.
- Each definition dominates its uses.

Static single assignment

SSA with dominance property

- Unique definition for each variable.
- Each definition dominates its uses.



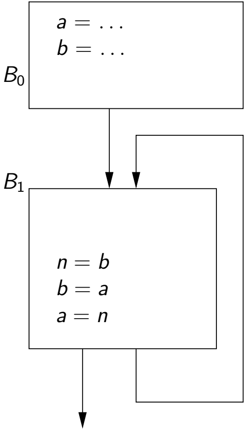
Static single assignment

SSA with dominance property

- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at the (iterated) dominance frontier.



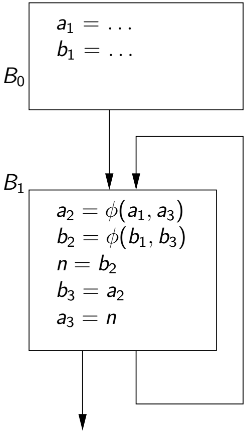
Static single assignment

SSA with dominance property

- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at the (iterated) dominance frontier.



Static single assignment

SSA with dominance property

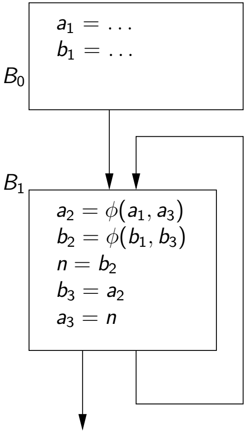
- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at the (iterated) dominance frontier.

Interests of SSA

- Link uses/definitions explicit.
- Code optimizations: efficient, easy-to-implement, fast.
- More accurate program analysis.



Static single assignment

SSA with dominance property

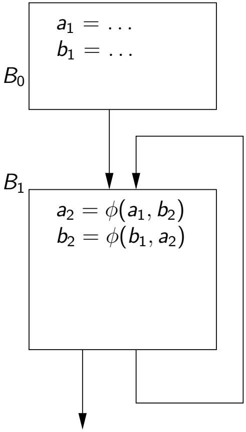
- Unique definition for each variable.
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at the (iterated) dominance frontier.

Interests of SSA

- Link uses/definitions explicit.
- Code optimizations: efficient, easy-to-implement, fast.
- More accurate program analysis.



Dominance frontier (elementary algorithm)

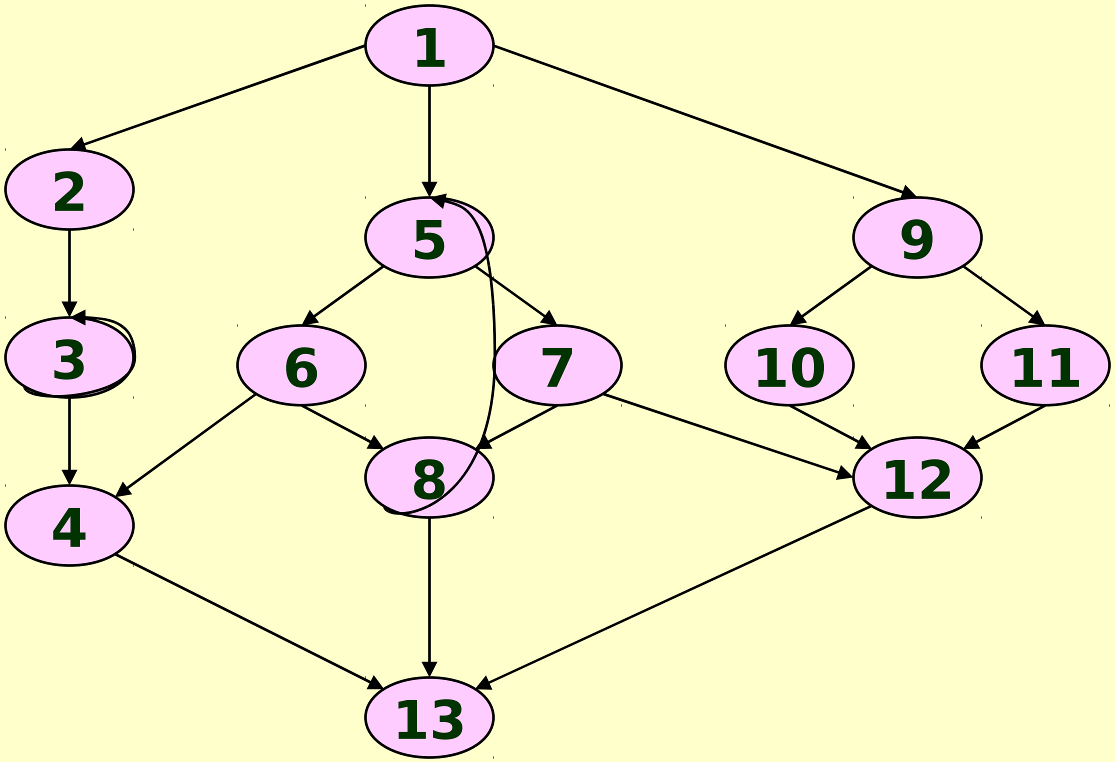
Dominance can be computed by fixed-point iteration:

$$D(r) = \{r\} \text{ and } D(n) = \{n\} \cup \left(\bigcap_{p \in \text{pred}[n]} D[p] \right)$$

Many other more efficient algorithms are possible. Then:

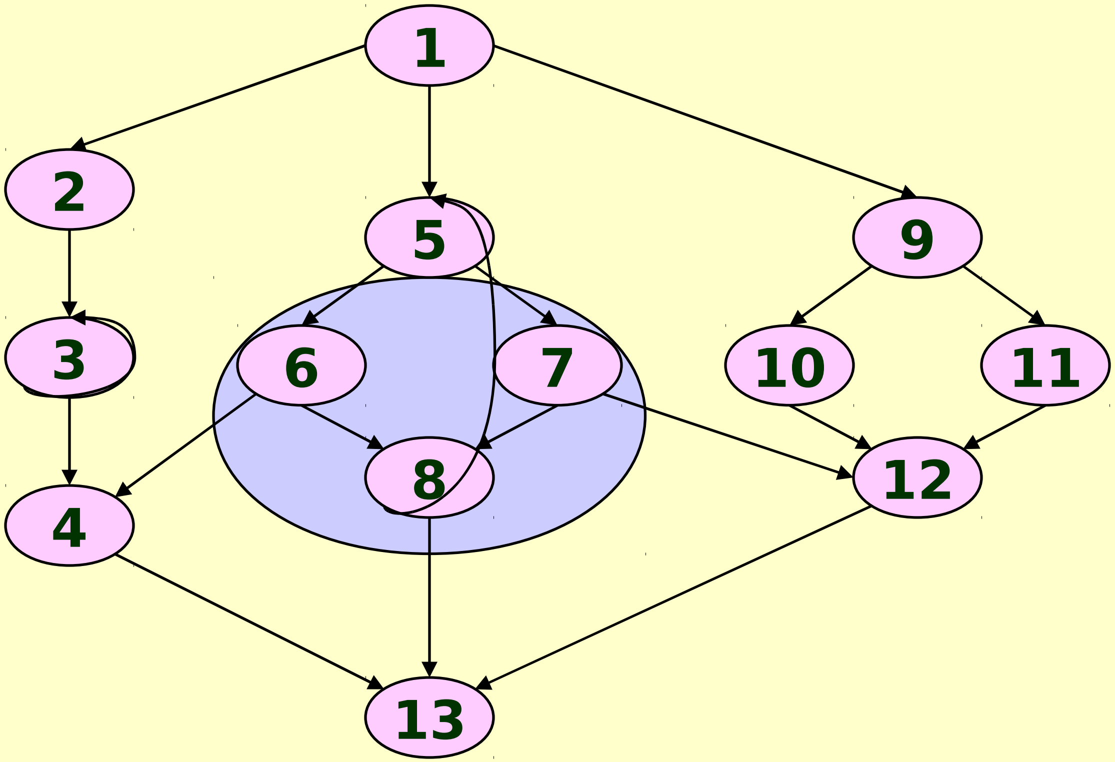
```
procedure computeDF(n)
  S := {}
  for each node y in succ[n] do
    if (idom(y) ≠ n) then S := S ∪ {y} /* successor of n not strictly dominated by n */
  endfor
  for each child c of n in the dominator tree do
    computeDF(c)
    for each element w of DF[c] do
      if (n does not dominate w) then S := S ∪ {w}
    endfor
  endfor
  DF[n] := S
```

Example



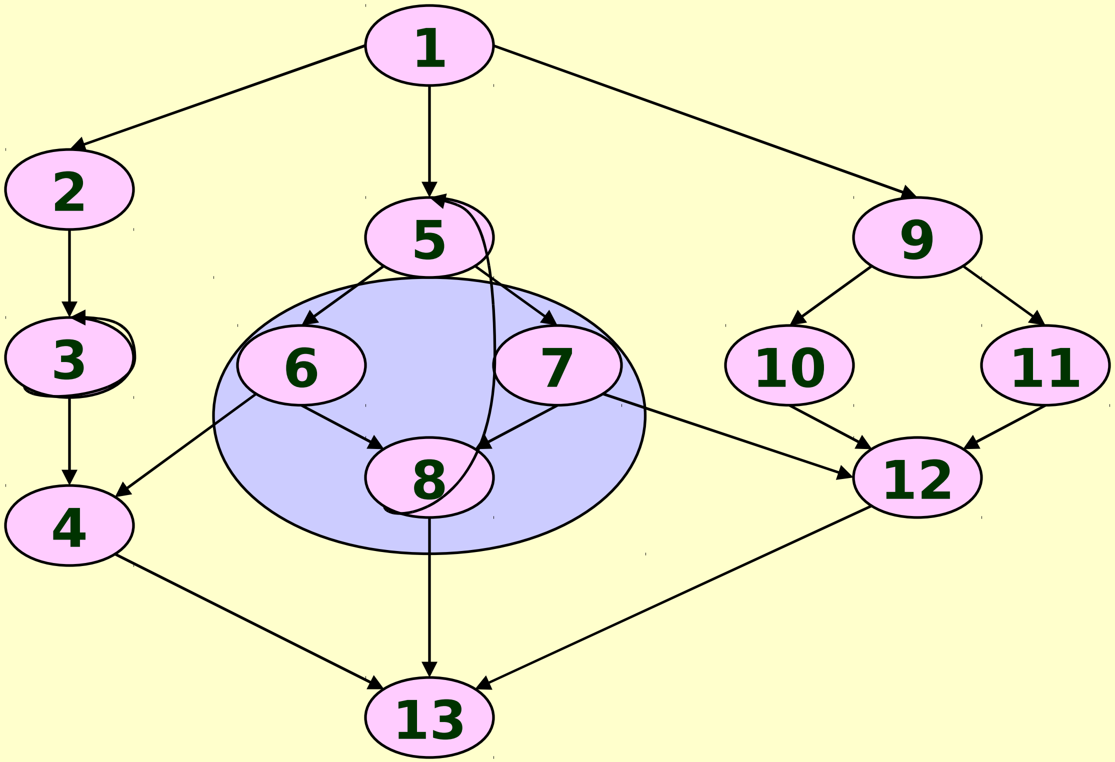
What is the dominance frontier of node 5?

Example



First we must find all nodes that node 5 strictly dominates.

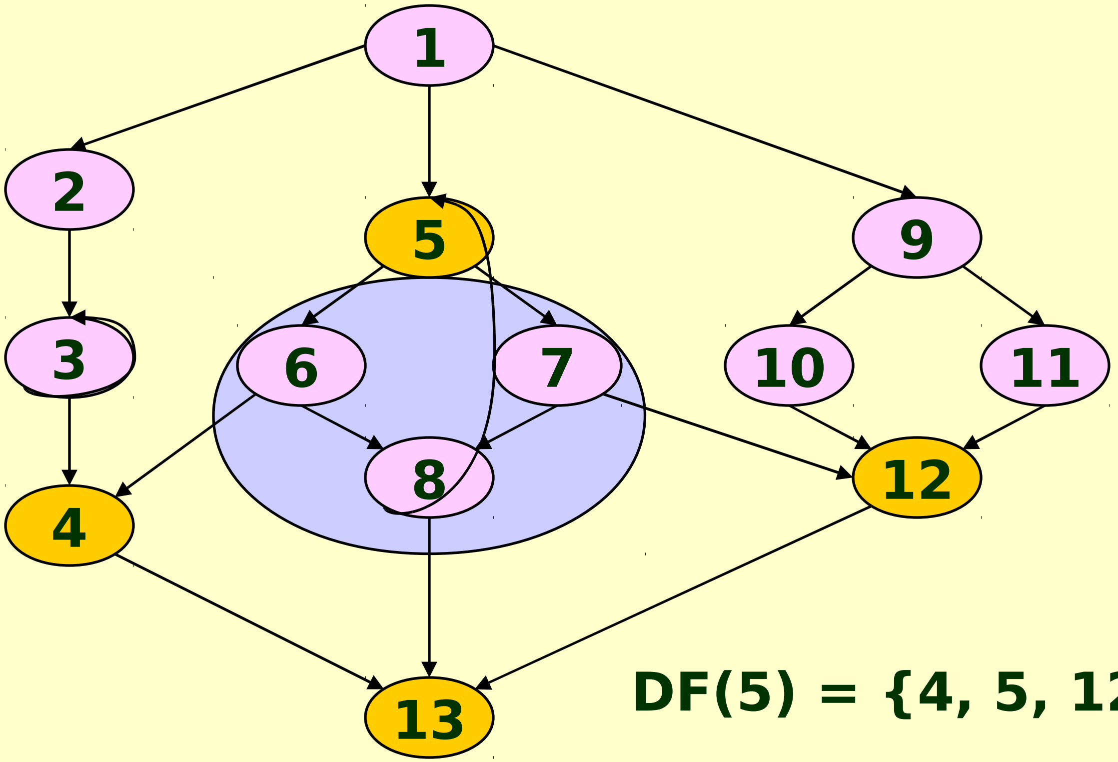
Example



A node **w** is in the dominance frontier of node 5 if 5 dominates a predecessor of **w**, but 5 does not strictly dominates **w** itself. What is the dominance frontier of 5?

nizatio

Example



DF(5) = {4, 5, 12, 13}

A node **w** is in the dominance frontier of node 5 if 5 dominates a predecessor of **w**, but 5 does not strictly dominates **w** itself. What is the dominance frontier of 5?

Placement of ϕ-functions

```
procedure Place-φ-functions(G, DF, 𝒟) /* 𝒟[n] is the set of variables defined in n */
  for each node n in G do
    for each variable a in 𝒟[n] do
      defsites[a] := defsites[a] ∪ {n}
    endfor
  endfor
  for each variable a do
    W := defsites[a]
    while (W not empty) do
      remove some node n from W
      for each Y in DF[n] do
        if (Y ∉ 𝒟_φ[n]) then
          insert statement a = φ(a, ..., a) at the top of Y
          𝒟_φ[n] := 𝒟_φ[n] ∪ {Y}
          if (Y ∉ 𝒟[n]) then W := W ∪ {Y}
        endif
      endfor
    endwhile
  endfor
```

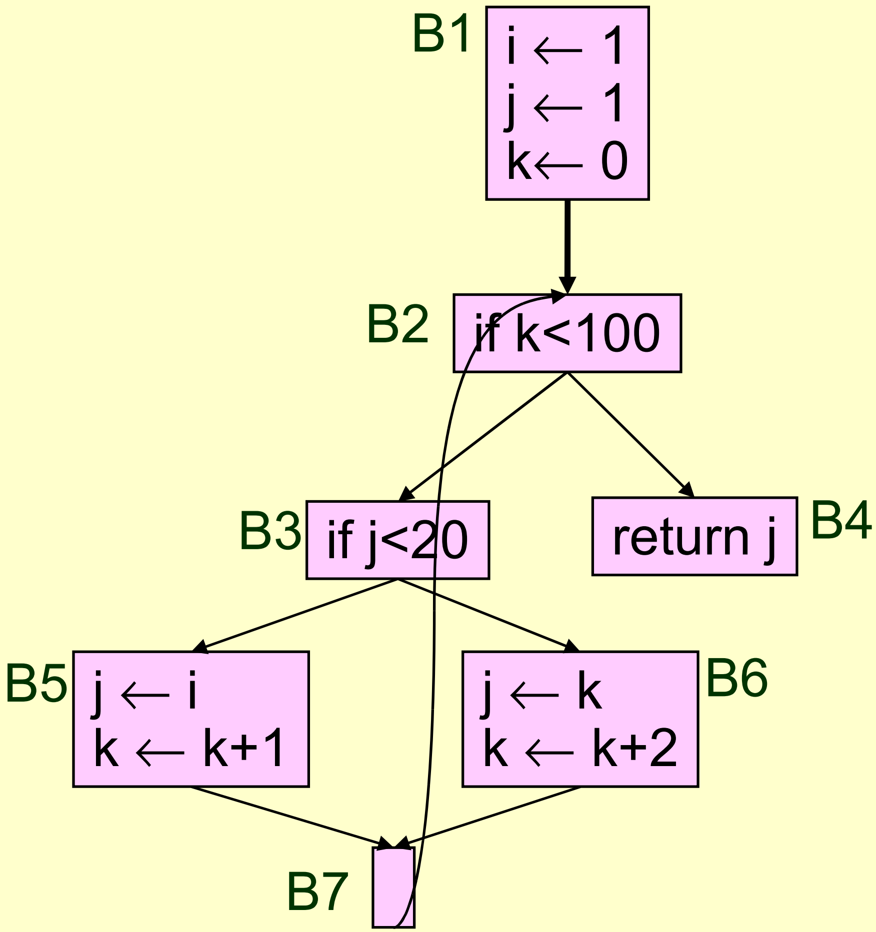
Renaming variables

```
procedure Rename(n)
  for each statement S in block n do
    if (S is not a φ-function) then
      for each use of some variable x in S do
        i := top(Stack[x]); replace the use of x with x_i in S
      endfor
    endif
    for each definition of some variable a in S
      Count[a]++; i := Count[a]; push i onto Stack[a]; replace definition with a_i
    endfor
  endfor
  for each successor Y of block n and each φ-function in Y do
    i := top(Stack[a]) where a is the argument coming from n; replace it with a_i
  endfor
  for each child (in the dominance tree) X of n do Rename(X)
  for each definition of some variable a (in the original code) do pop Stack[a]

procedure RenameAll(G)
  for each variable a do Count[a] := 0; Stack[a] := {}; push 0 onto Stack[a]
  Rename(r) /* root of the dominance tree */
```

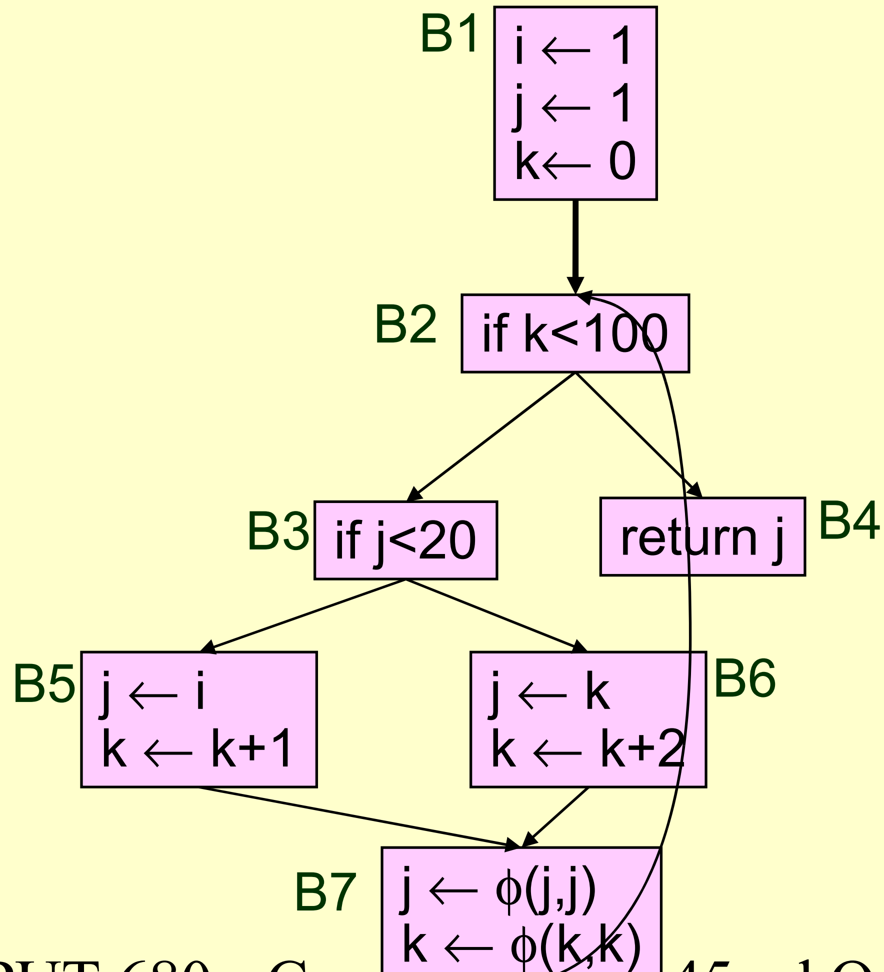
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```



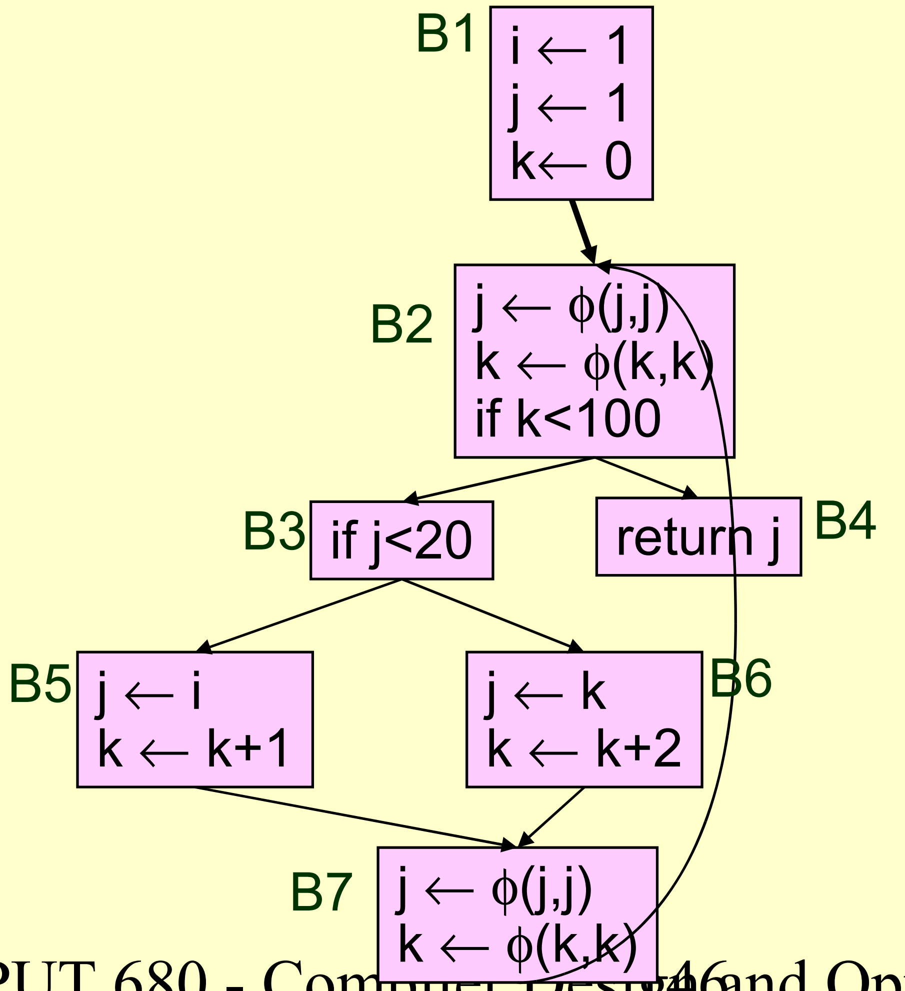
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```



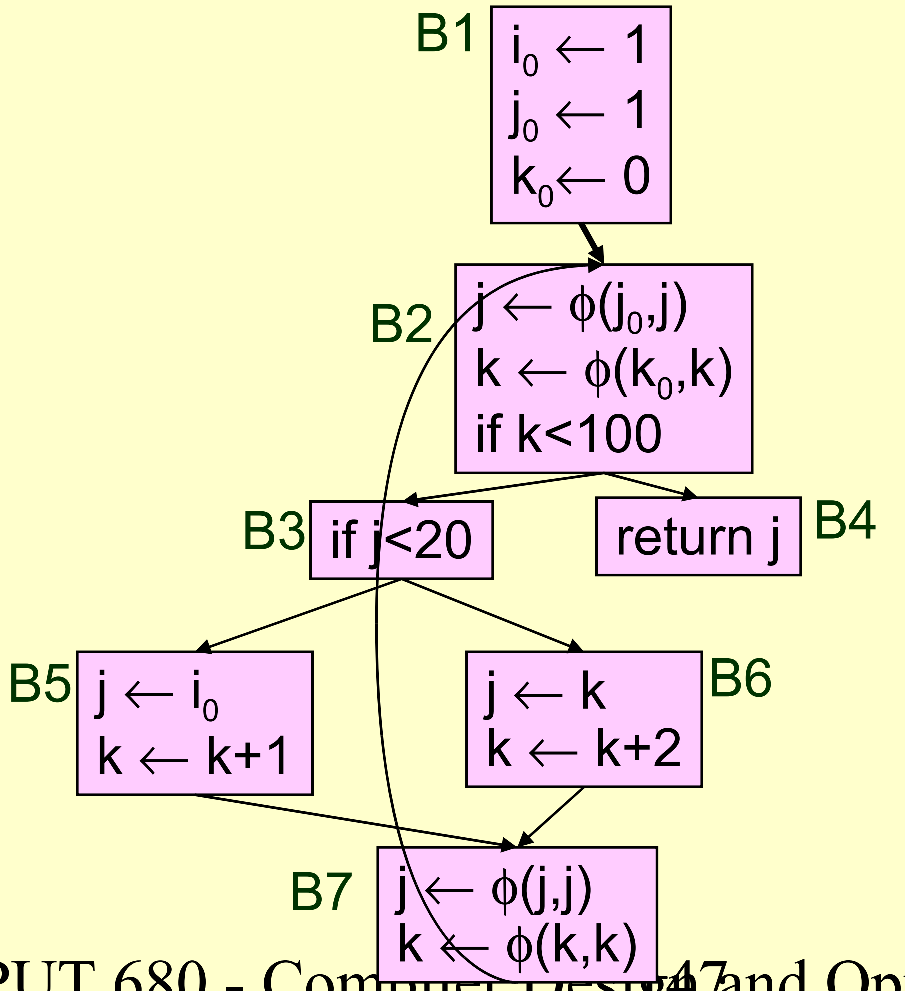
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```



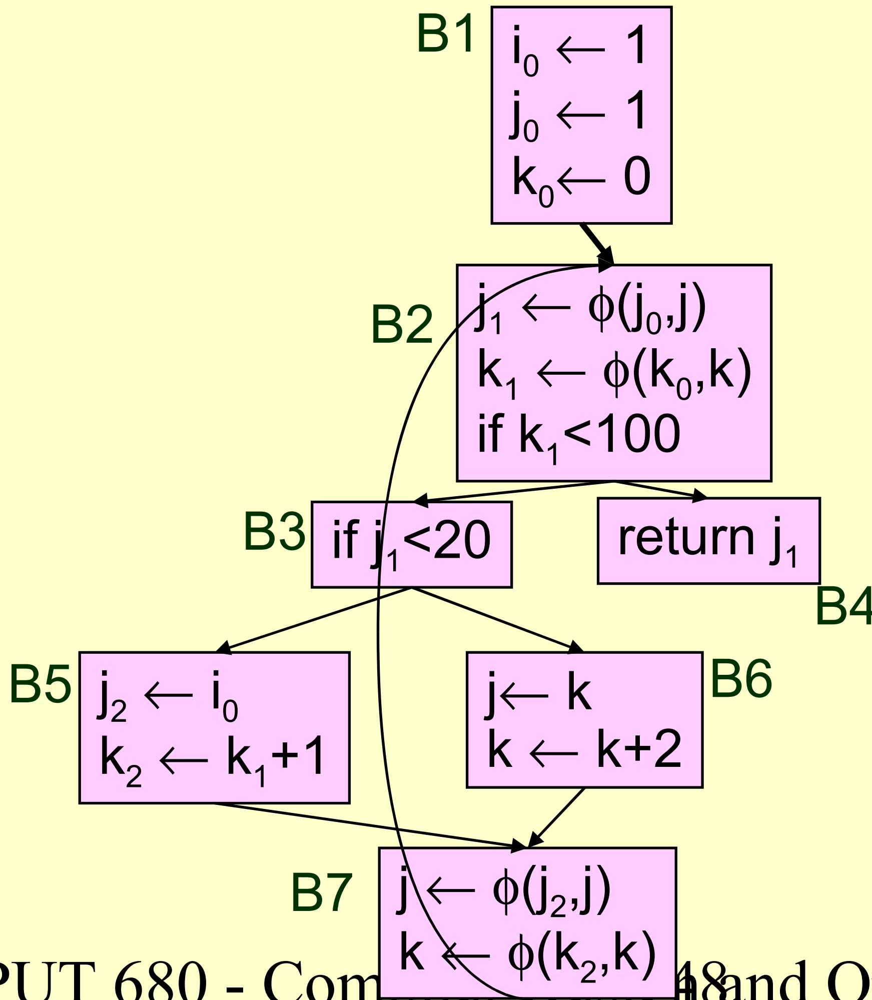
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```



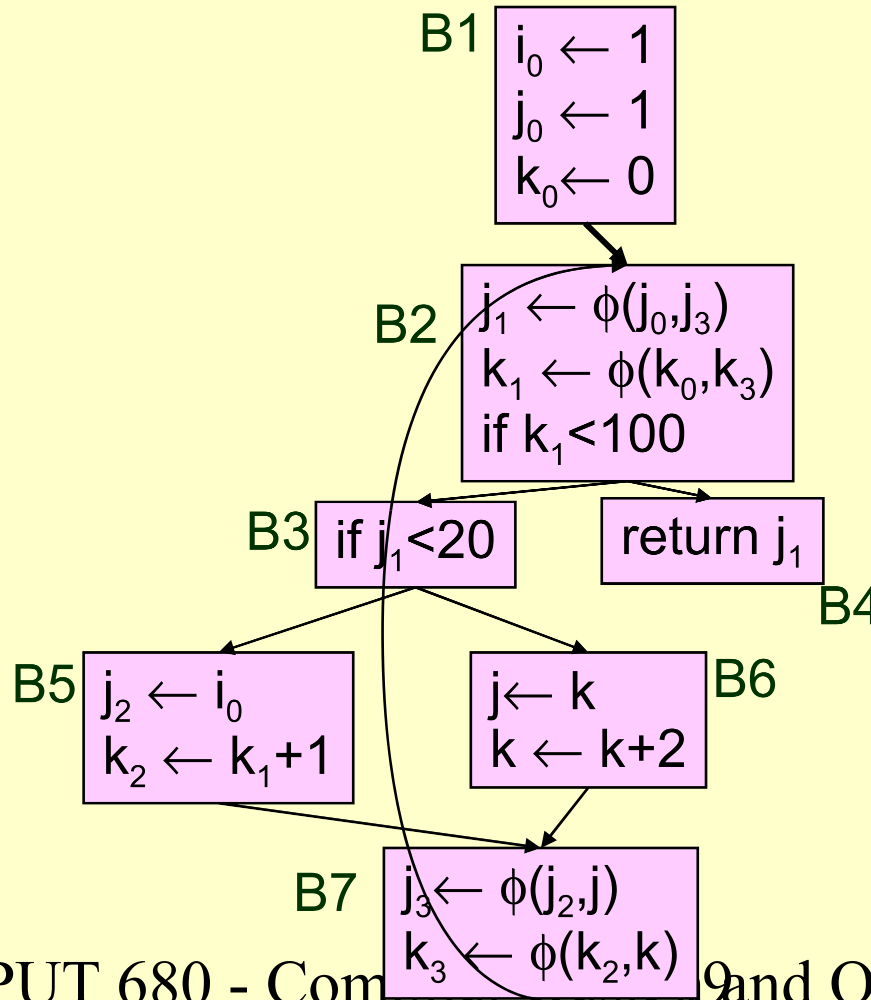
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```



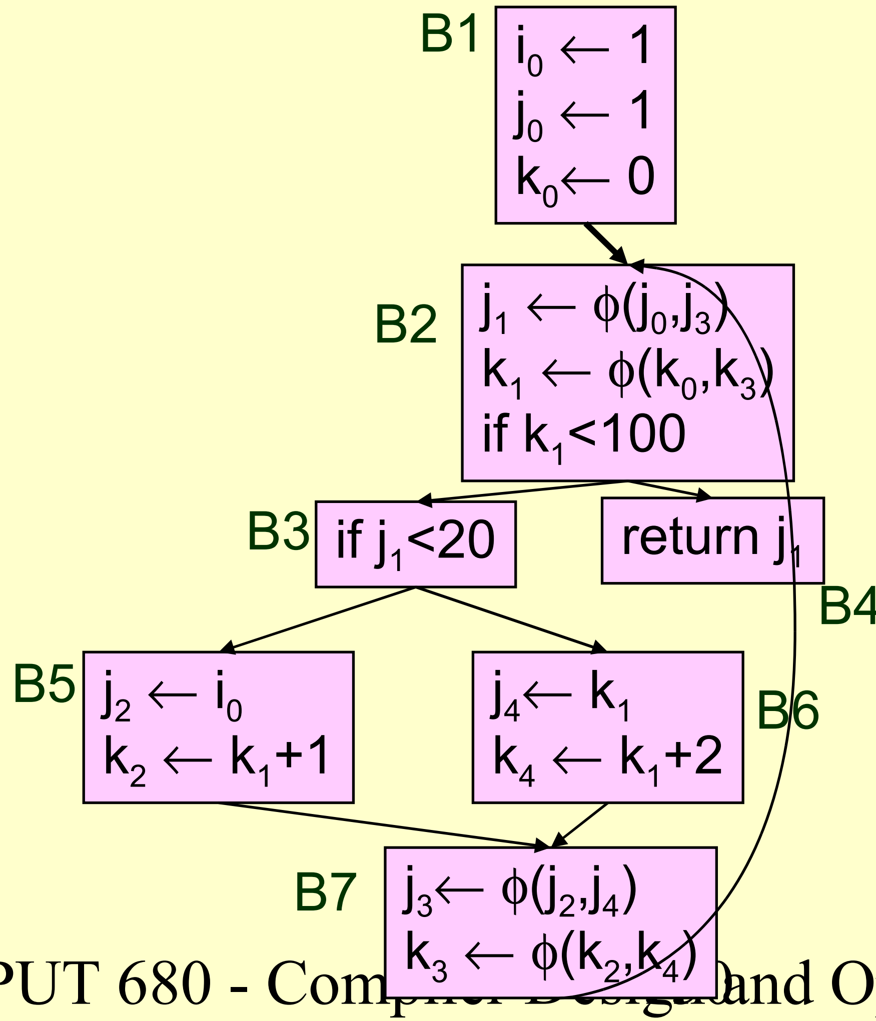
SSA: A Complete Example.

```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```

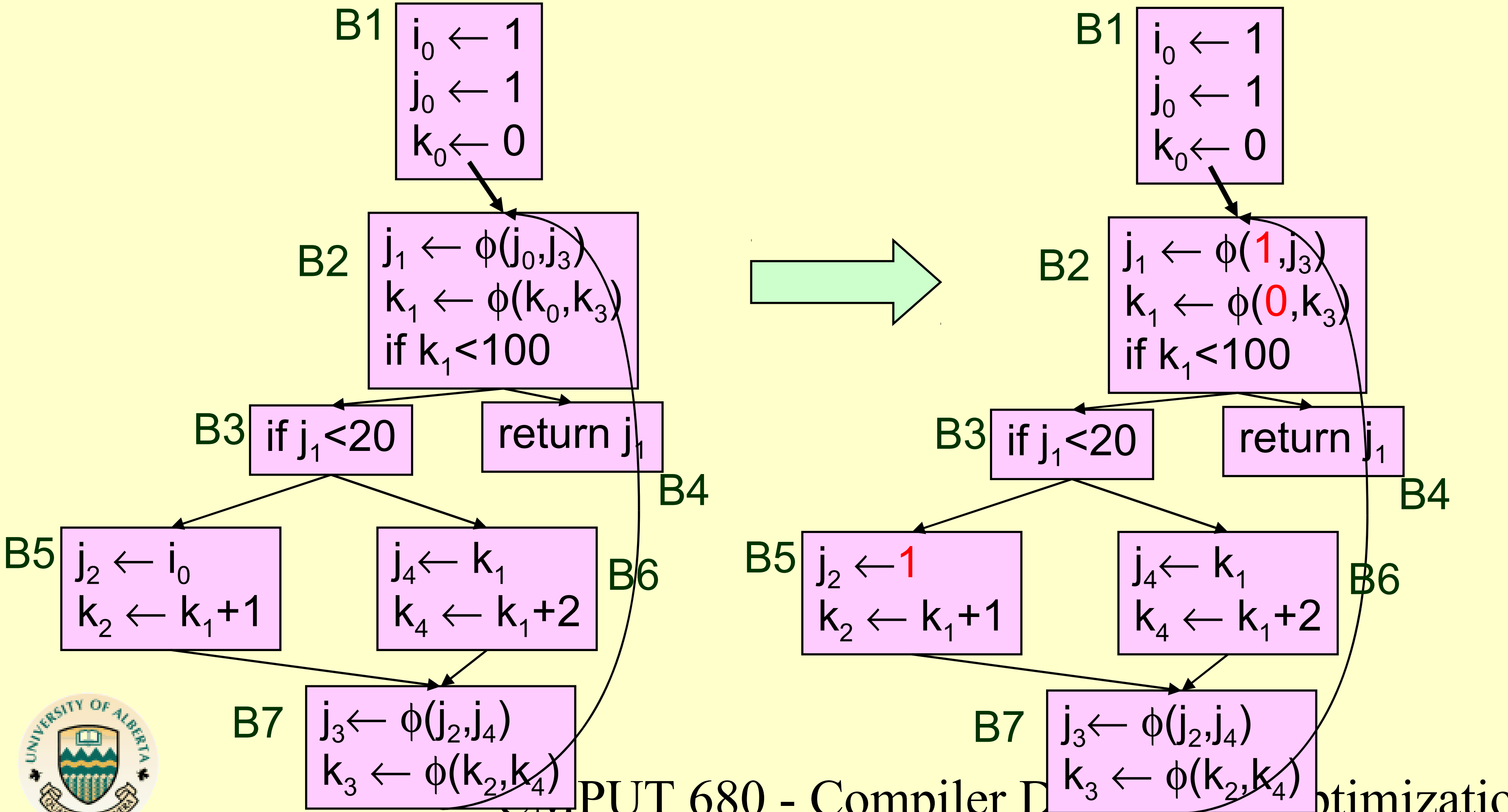


SSA: A Complete Example.

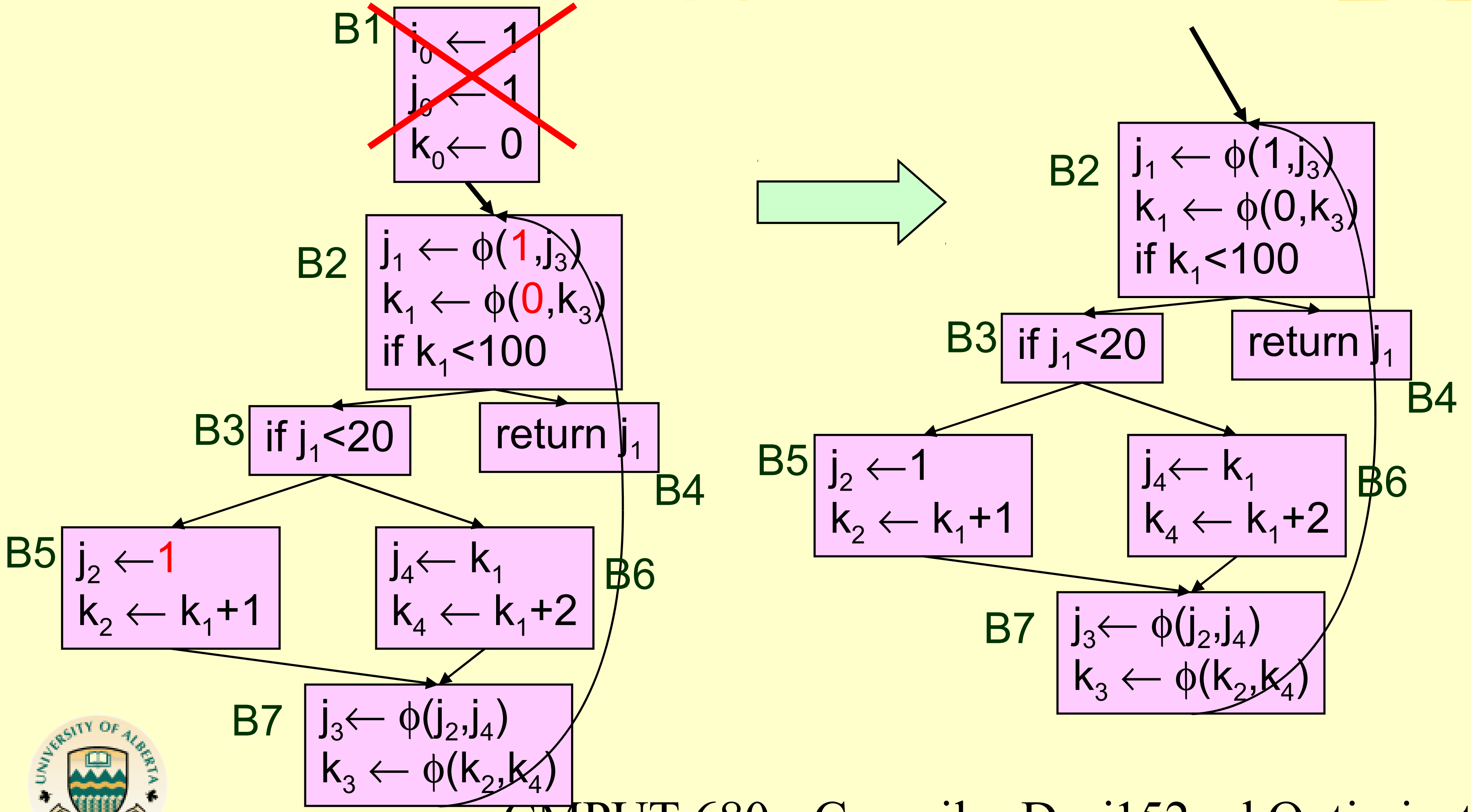
```
i=1;
j=1;
k=0;
while(k<100) {
    if(j<20) {
        j=i;
        k=k+1;
    }
    else {
        j=k;
        k=k+2;
    }
}
return j;
```



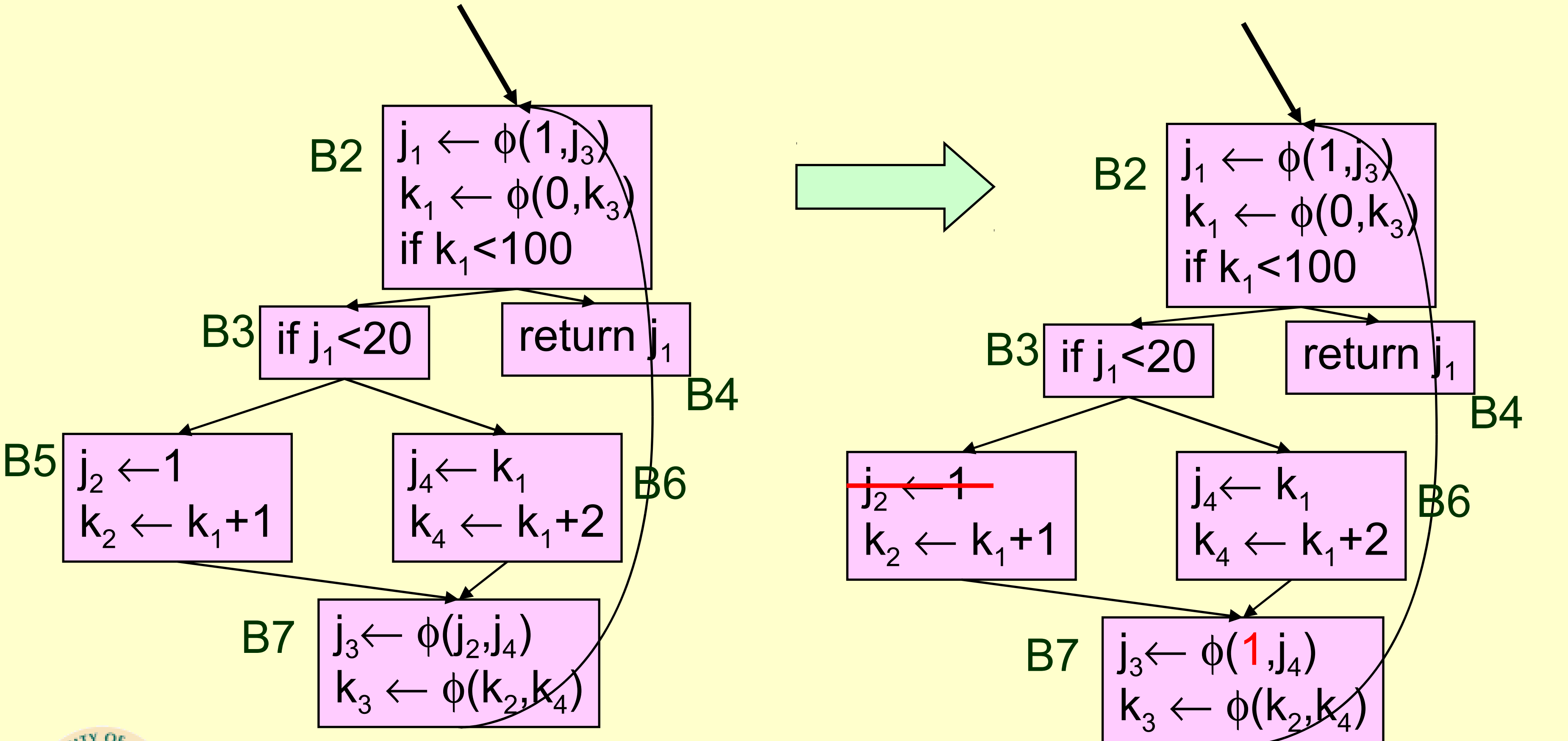
Example: Constant Propagation



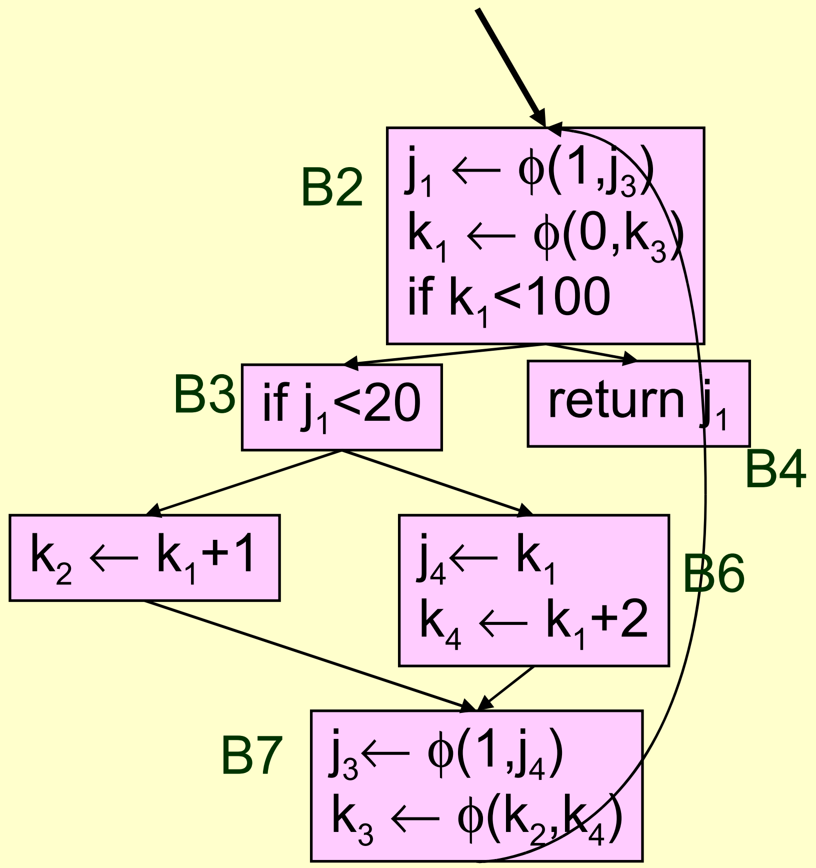
Example: Dead-code Elimination



Constant Propagation and Dead Code Elimination



Example:
Is this the end?

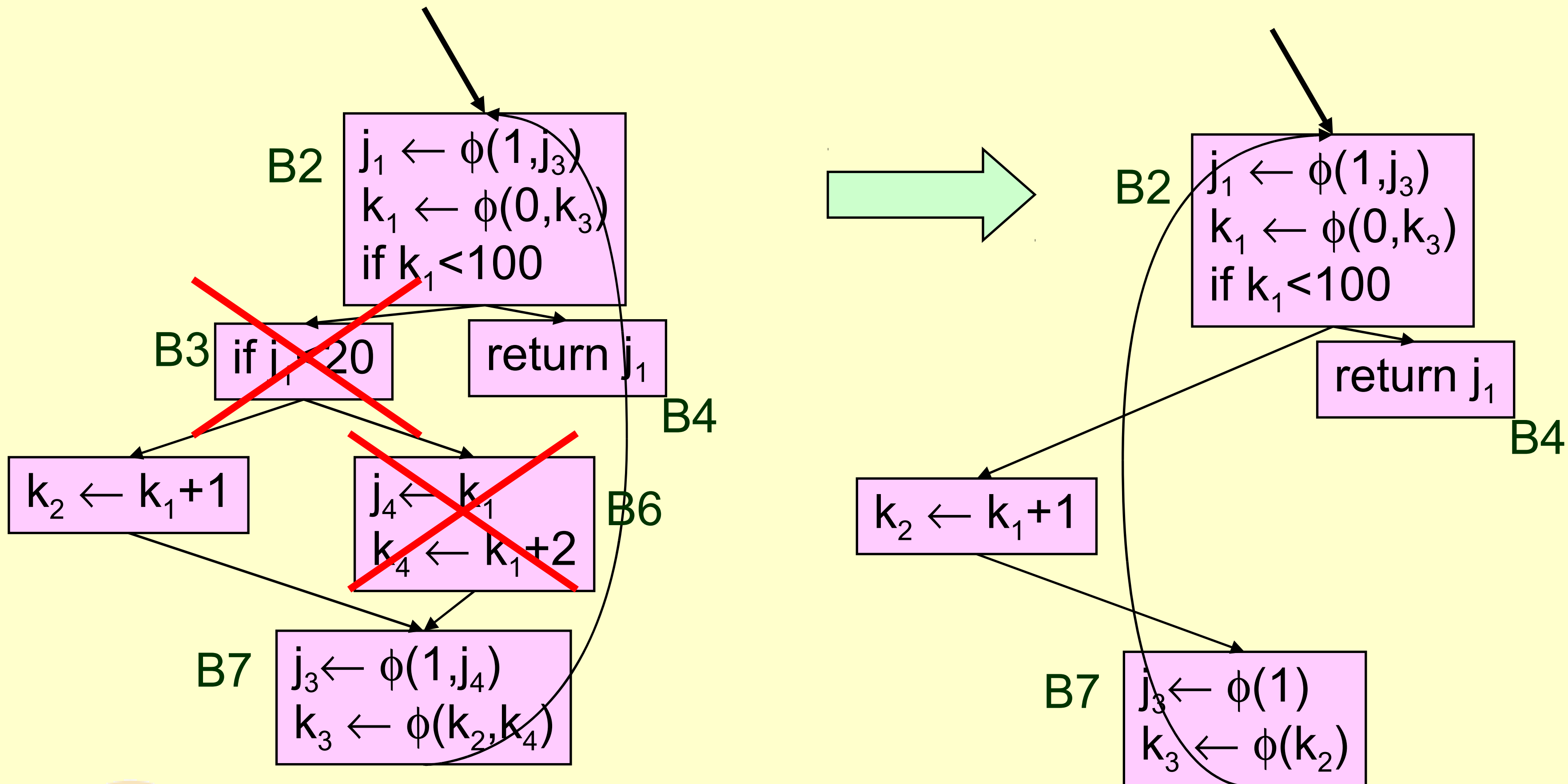


But block 6 is never executed! How can we find this out, and simplify the program?

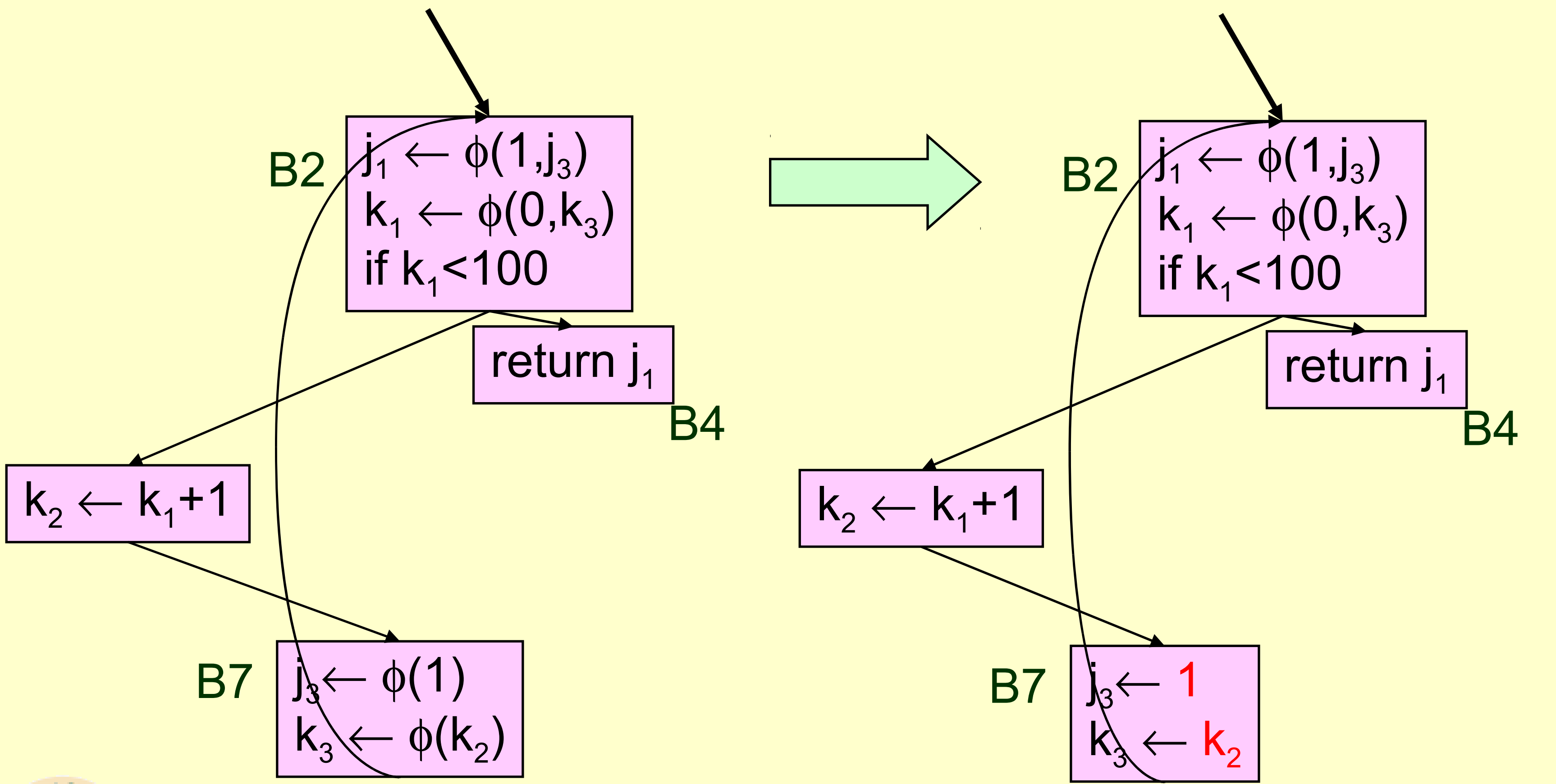
SSA conditional constant propagation finds the *least fixed point* for the program and allows further elimination of dead code.

See algorithm in Tiger book.

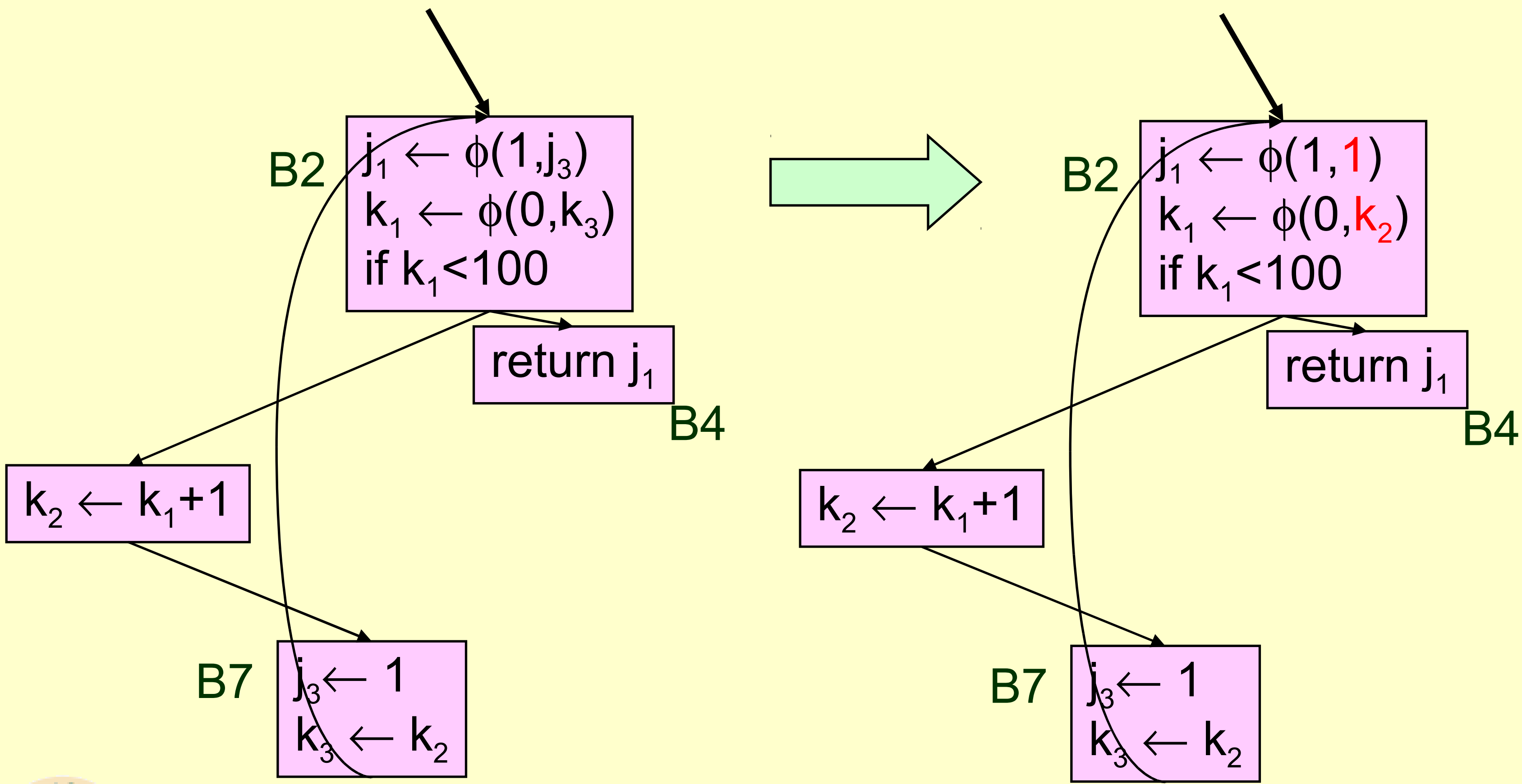
Example: Dead code elimination



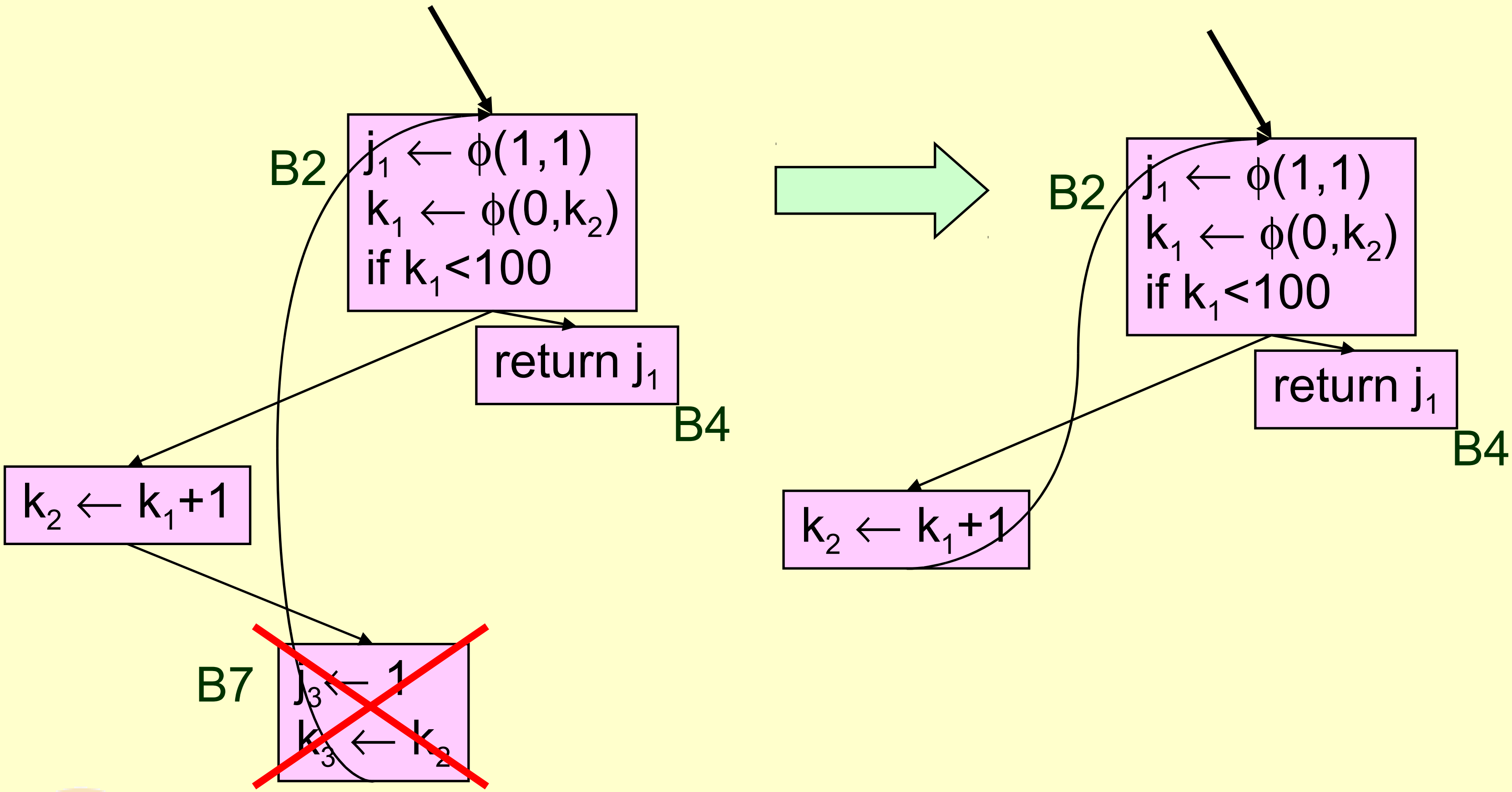
Example: Single Argument ϕ-Function Elimination



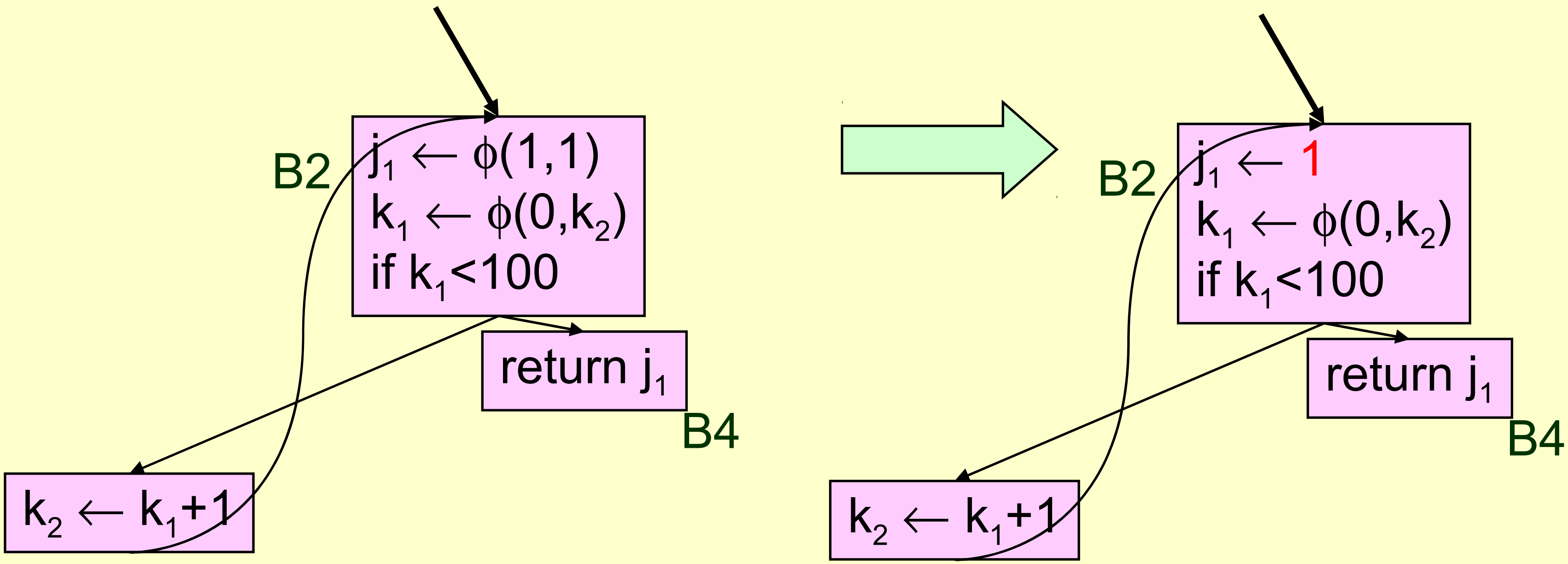
Example: Constant and Copy Propagation



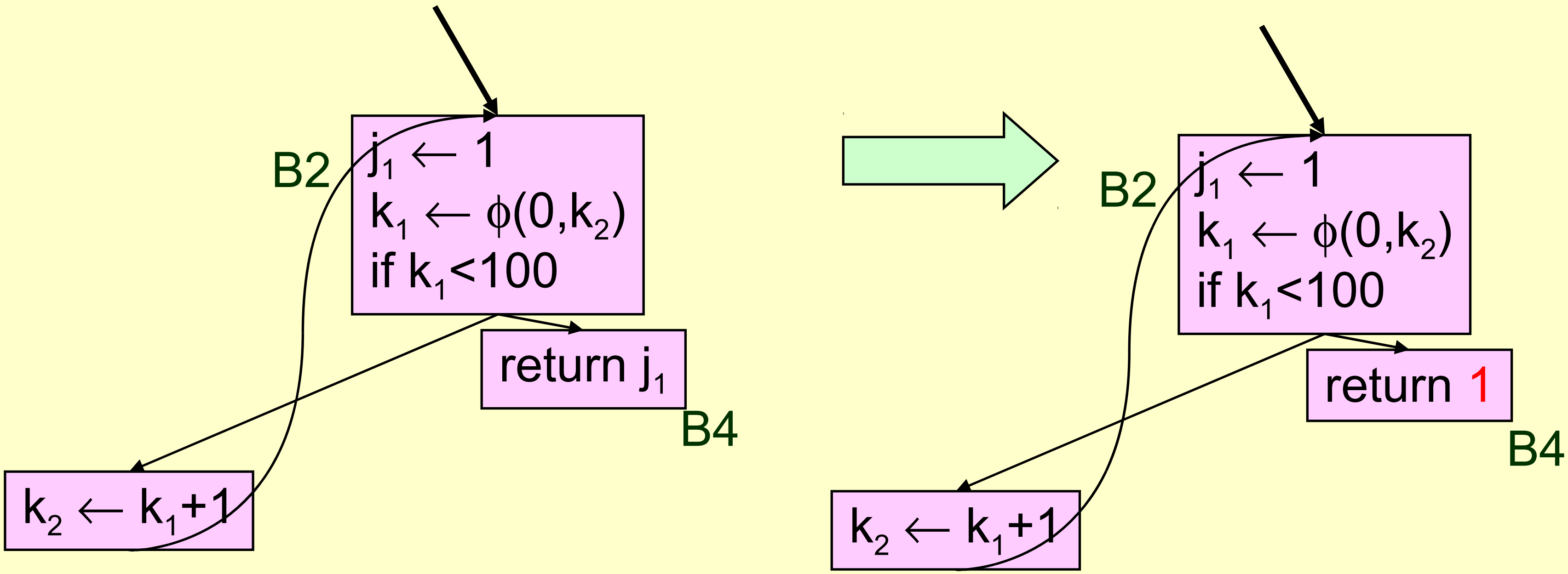
Example: Dead Code Elimination



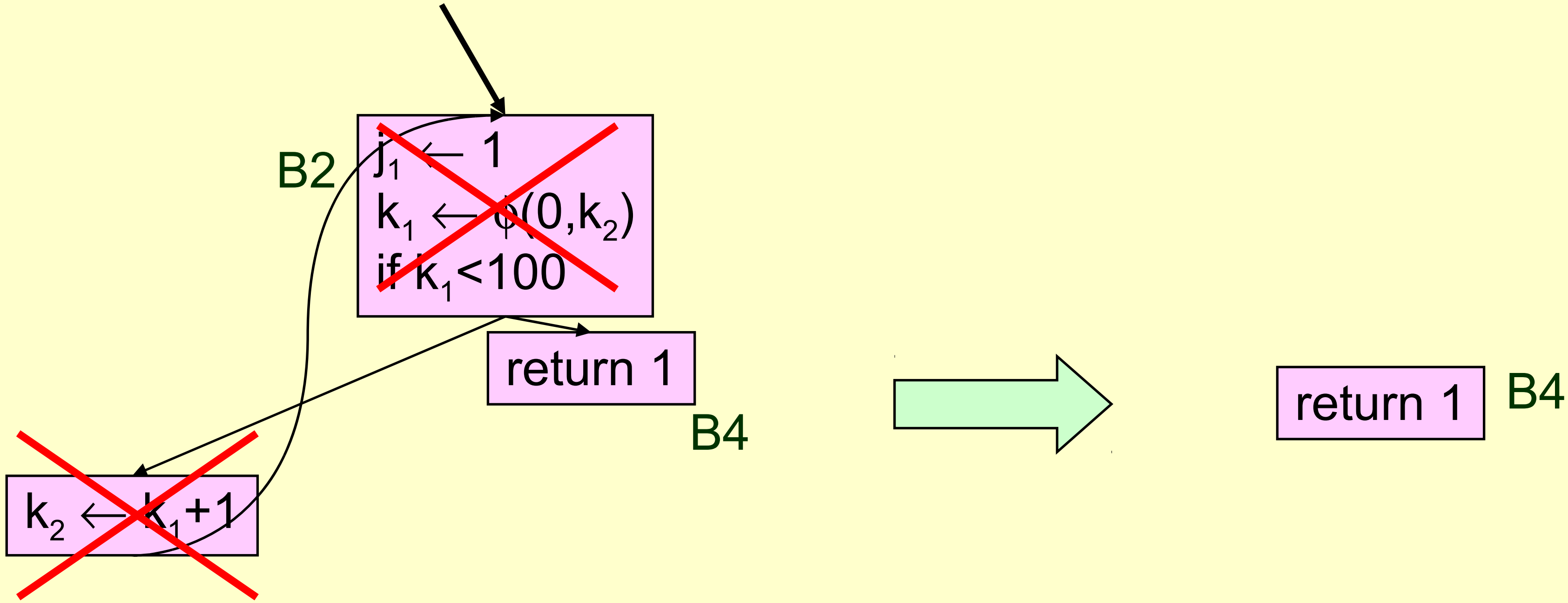
Example: ϕ-Function Simplification



Example: Constant Propagation



Example: Dead Code Elimination



More readings

References

- Cytron, Ferrante, Rosen, Wegman, Zadek. **Efficiently computing static single assignment form and the control dependence graph**, ACM Transactions on Programming Languages and Systems, 13(4):451–490, 1991.
- Ramalingam. **On loops, dominators, and dominance frontiers**. ACM Transactions on Programming Languages and Systems, 24(5):455–490, 2002.

Recent advances in SSA

- SSA-based compilers & JIT compilation.
- Register allocation, out-of-SSA conversion, liveness analysis.
- SSA extensions: SSI, gated SSA, psi-SSA, value state dependence graph, array SSA, safeTSA, etc.

Links between the different notions

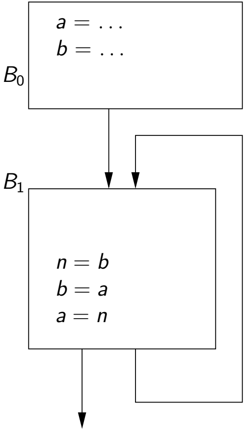
A few important results:

- If S contains the entry node, $J(S) = J^+(S) = DF^+(S)$.
- G is reducible
  - iff simplifiable by the rules T_1 and T_2.
  - iff each SCC has a unique entry node.
  - iff removing all (u, v) where v dominates u makes G acyclic.
  - ...
- Dominators and iterated dominance frontiers can be computed quickly from loop-nesting forest, especially if G is reducible.
- Conversely, DJ-graphs can be used to build loop forests.
- Advanced algorithms use Tarjan's union-find with almost-linear complexity (see Ramalingam, Sreedhar, Havlak, Steensgaard).

Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.

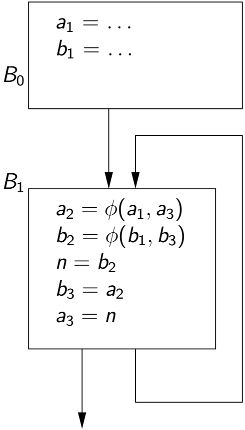
Swap problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.

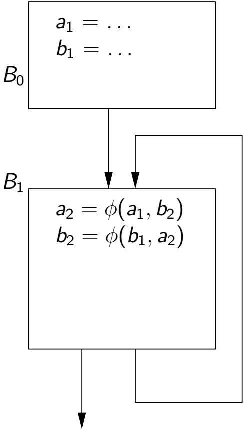
Swap problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.

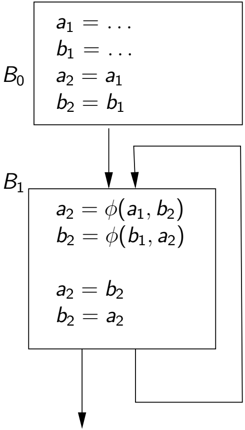
Swap problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks.

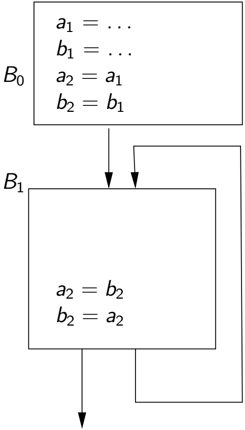
Swap problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
    - Bad understanding of parallel copies.

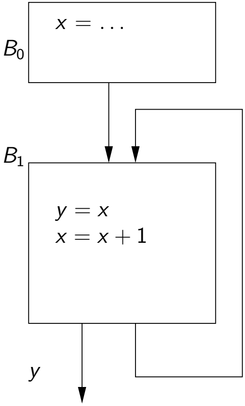
Swap problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

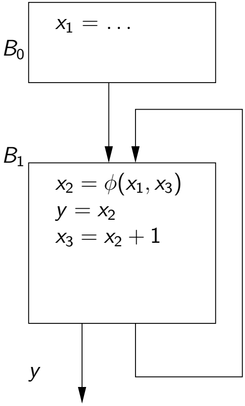
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

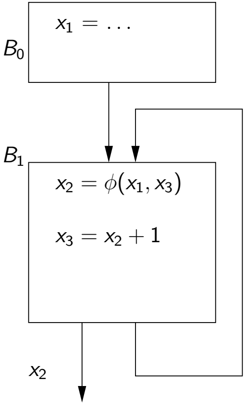
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

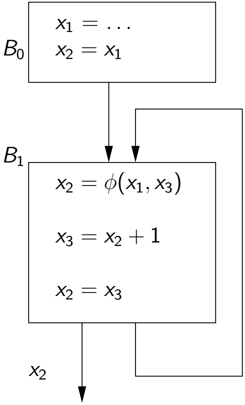
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

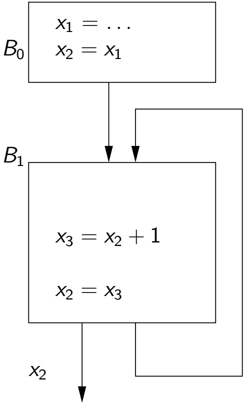
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

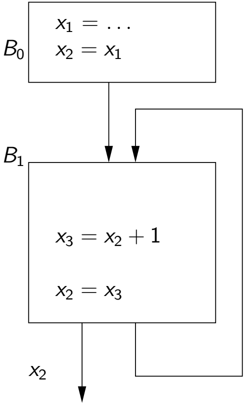
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.

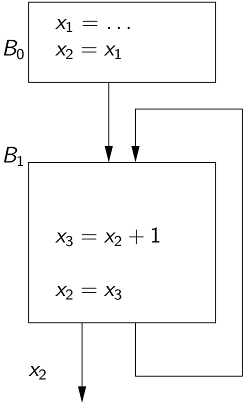
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - "virtualization" hard to implement.

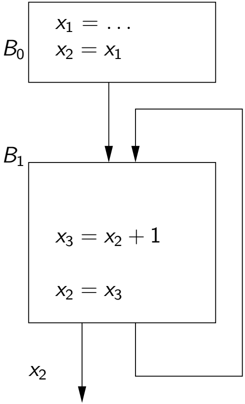
Lost copy problem



Early attempts and pitfalls

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - "virtualization" hard to implement.

Lost copy problem



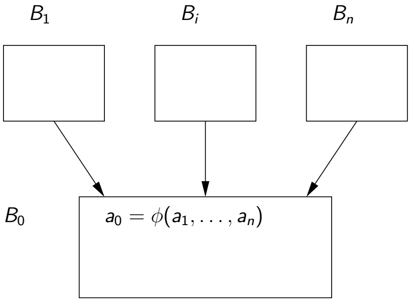
☛ Many SSA optimizations turned off in gcc and Jikes.

Going to CSSA (conventional SSA): Sreedhar et al.

Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, a_0, ..., a_n have the same name.

From SSA to CSSA



Going to CSSA (conventional SSA): Sreedhar et al.

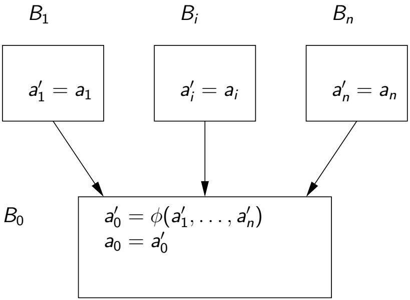
Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, a_0, ..., a_n have the same name.

Correctness

After introduction of variables a'_i and copies, the code is in CSSA.

From SSA to CSSA



Going to CSSA (conventional SSA): Sreedhar et al.

Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

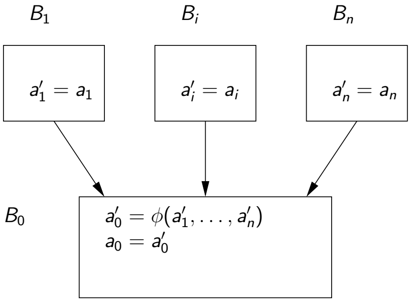
Correctness

After introduction of variables a'_i and copies, the code is in CSSA.

Code quality

Aggressive coalescing can remove useless copies. But better use accurate notion of interferences.

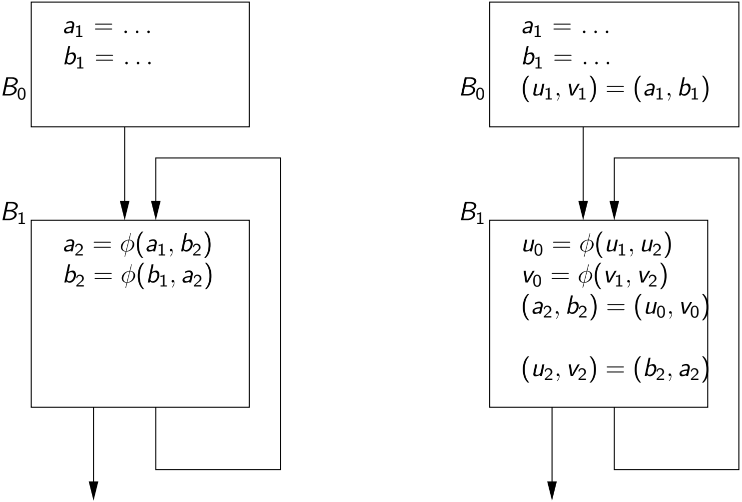
From SSA to CSSA



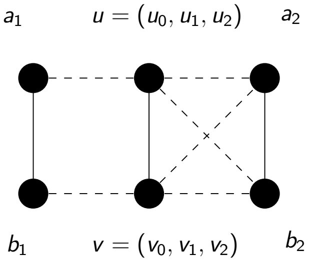
"Liveness of ϕ" defined by the a'_i.

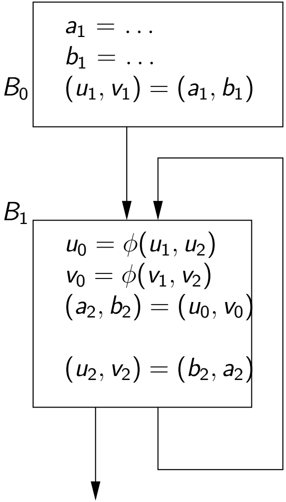
✝ Be careful with potential bugs due to conditional branches that use or define variables.

Coalesced example: the swap problem

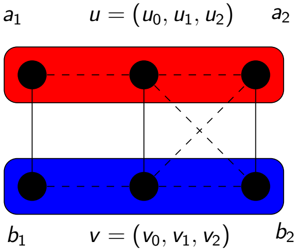


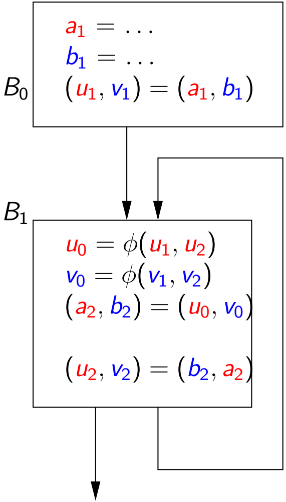
Coalesced example: the swap problem



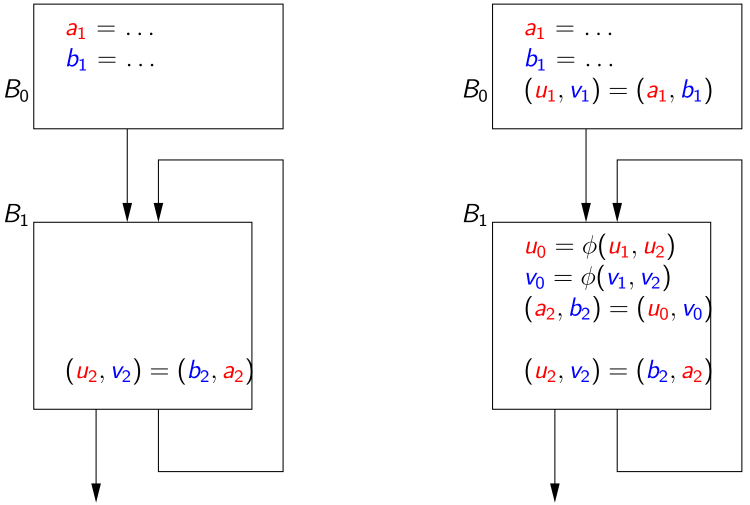


Coalesced example: the swap problem

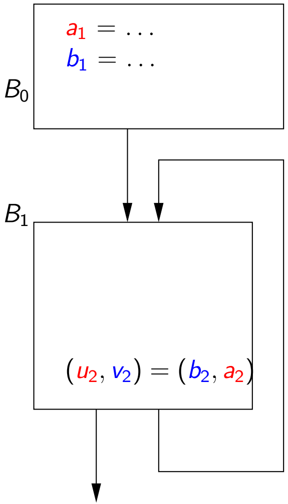


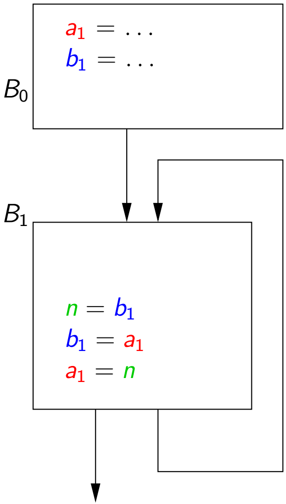


Coalesced example: the swap problem

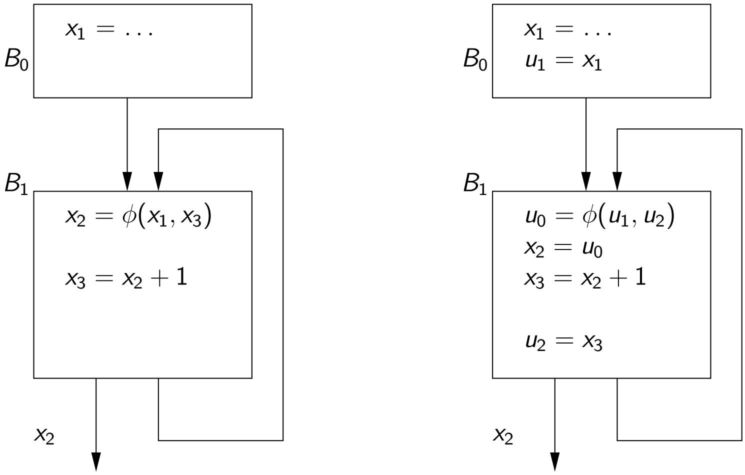


Coalesced example: the swap problem

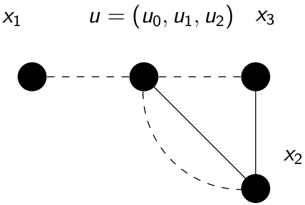


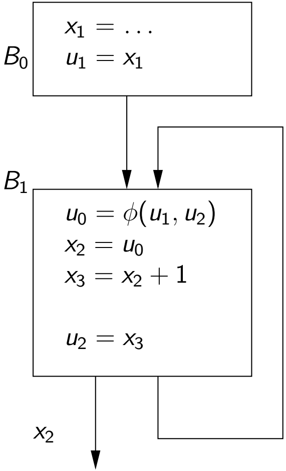


Coalesced example: the lost copy problem

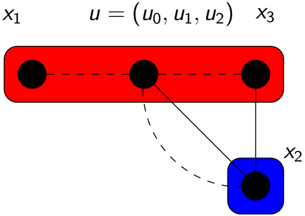


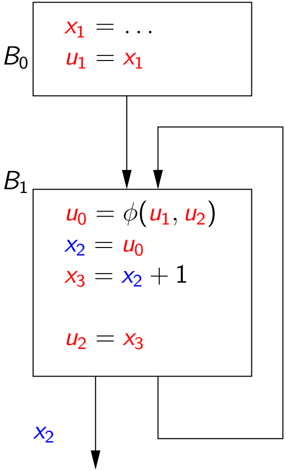
Coalesced example: the lost copy problem



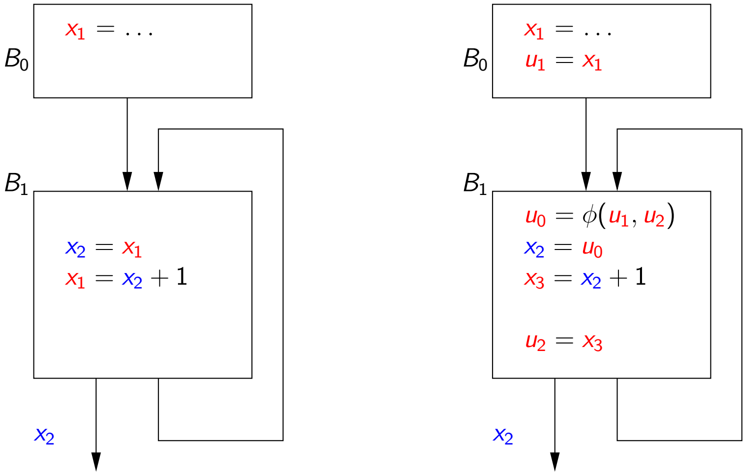


Coalesced example: the lost copy problem





Coalesced example: the lost copy problem

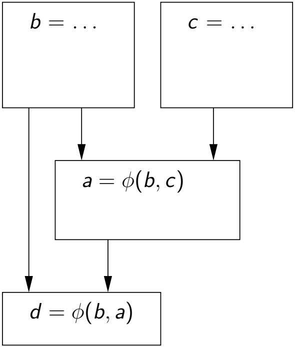


Outline

Exploiting SSA: value-based interferences

Definition (Chaitin interference)

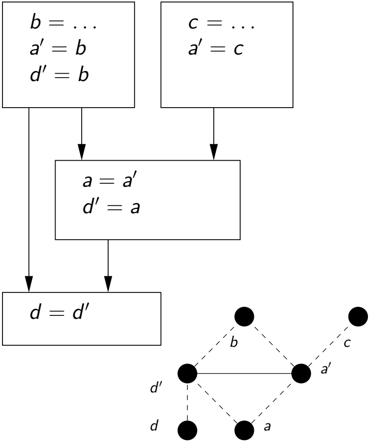
Two variables interfere if one is live at the definition of the other, which is not a copy of the first.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

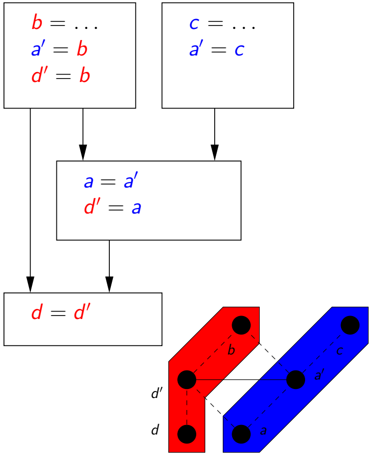
Two variables interfere if one is live at the definition of the other, which is not a copy of the first.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

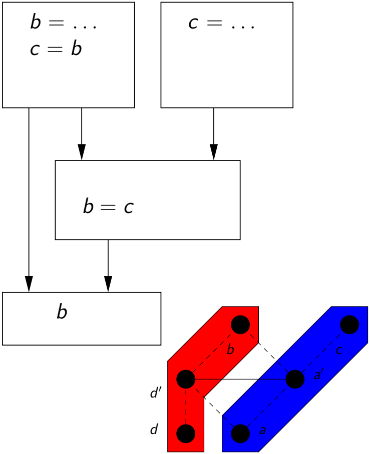


Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

☛ Need to update interference graph after coalescing.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

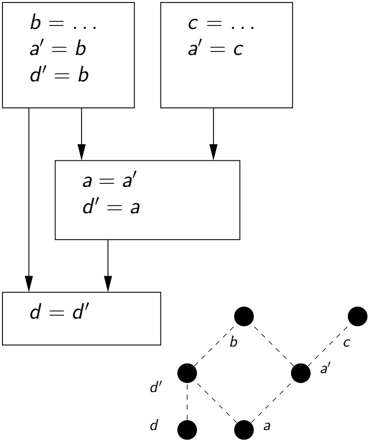
☛ Need to update interference graph after coalescing.

Unique value V of a SSA variable

For a copy `b` = `a`, $V(\texttt{b}) = V(\texttt{a})$ (traversal of dominance tree).

Value-based interference

`a` and `b` interfere if $V(\texttt{a}) \neq V(\texttt{b})$ and Live-range(`a`) $\cap$ Live-range(`b`) $\neq \emptyset$.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

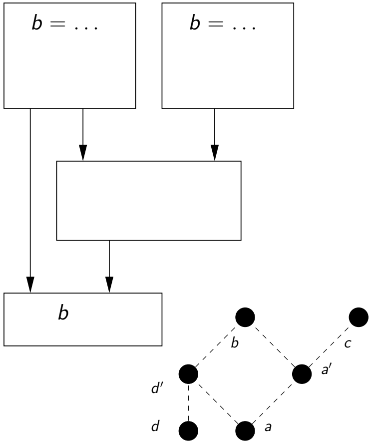
☛ Need to update interference graph after coalescing.

Unique value V of a SSA variable

For a copy $\mathtt{b} = \mathtt{a}$, $V(\mathtt{b}) = V(\mathtt{a})$ (traversal of dominance tree).

Value-based interference

$\mathtt{a}$ and $\mathtt{b}$ interfere if $V(\mathtt{a}) \neq V(\mathtt{b})$ and Live-range($\mathtt{a}$) $\cap$ Live-range($\mathtt{b}$) $\neq \emptyset$.



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

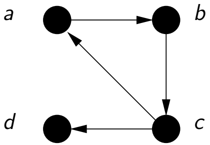
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

$$(a, b, c, d) = (c, a, b, c)$$



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

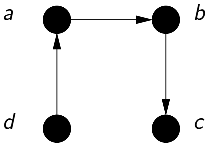
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \to a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

$d = c$
$(a, b, c) = (d, a, b)$



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

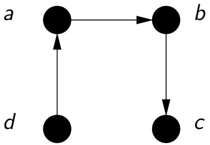
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \to a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

$$
\begin{aligned}
d &= c \\
c &= b \\
b &= a \\
a &= d
\end{aligned}
$$



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

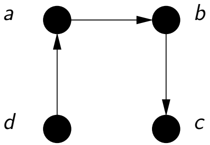
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.
- Simple circuit: one more copy.

$d = c$
$c = b$
$b = a$
$a = d$



Algorithm 1: Parallel copy sequentialization algorithm

Data: Set P of parallel copies $a \mapsto b$, $a \neq b$, one extra fresh variable n

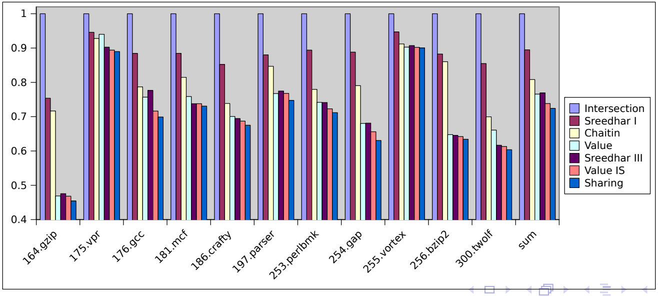
Output: List of copies in sequential order

```
ready ← [] ; to_do ← [] ; pred(n) ← ⊥ ;
forall the (a ↦ b) ∈ P do
    loc(b) ← ⊥ ; pred(a) ← ⊥ ;                                /* initialization */
forall the (a ↦ b) ∈ P do
    loc(a) ← a ; pred(b) ← a ; to_do.push(b) ;                /* copy into b to be done */
forall the (a ↦ b) ∈ P do
    if loc(b) = ⊥ then ready.push(b) ;                        /* b is not used and can be overwritten */
while to_do ≠ [] do
    while ready ≠ [] do
        b ← ready.pop() ; a ← pred(b) ;                       /* pick a free location */
        c ← loc(a) ; emit_copy(c ↦ b) ; loc(a) ← b ;          /* generate the copy */
        if a = c and pred(a) ≠ ⊥ then ready.push(a) ;         /* first time copied */
    b ← to_do.pop() ;                                         /* look for remaining copy */
    if b = loc(b) then
        emit_copy(b ↦ n) ; loc(b) ← n ; ready.push(b) ;       /* break circuit */
```

Qualitative experiments with SPEC CINT2000

Key points of the out-of-SSA translation

- Copy insertion (to go to CSSA and to handle register renaming constraints) followed by coalescing.
- Value-based interferences ☛ coalescing is improved and independent of virtualization (i.e., as in Sreedhar III).
- Parallel copies followed by sequentialization.

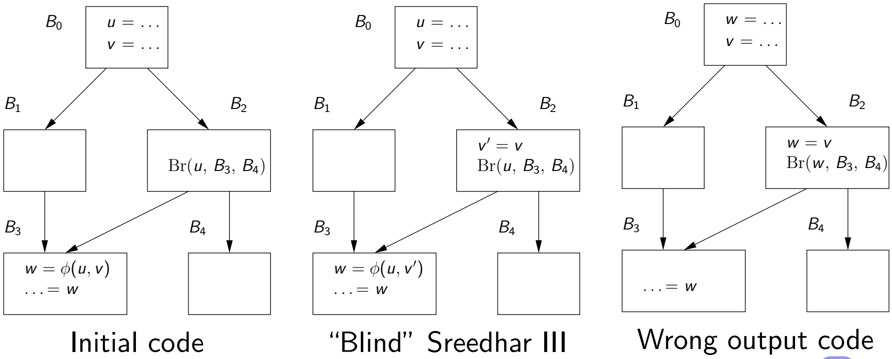


Bug tracking RVM-254 of Jikes RVM

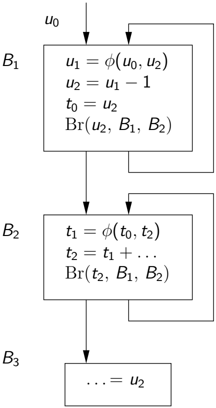
Problems with SSA form: lack of loop unrolling breaks VM

This problem is probably one of the most serious in the RVM currently. When loop unrolling is disabled and SSA enabled the created IR is corrupt. The error has in the past look like we were suffering from the "lost copy" problem, but implementing a naive solution to this didn't solve the problem. Their is sound logic behind the code so we need to identify a small test case where things are broken and then reason about what's wrong in leave SSA. This has been attempted once (with the code that removes an element from the live set) but the problem no longer appears to surface here. Currently these optimizations are disabled but by RVM 3.0 they should be re-enable and this bug cured.

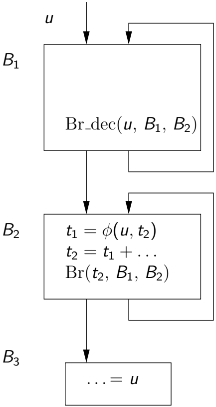
Potential bugs with conditional branches



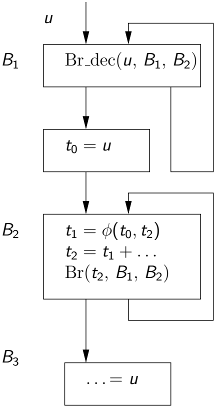
Unfeasible out-of-SSA translation example



Initial code



After optimization



Needs edge splitting