



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# 11. Single Static Assignment (SSA)

## Compilatori – Middle end [I215-014]

*Corso di Laurea in INFORMATICA*  
(D.M.270/04) [16-215]  
Anno accademico 2024/2025

**Prof. Andrea Marongiu**  
[andrea.marongiu@unimore.it](mailto:andrea.marongiu@unimore.it)

# Copyright note

*È vietata la copia e la riproduzione dei contenuti e immagini in qualsiasi forma.*

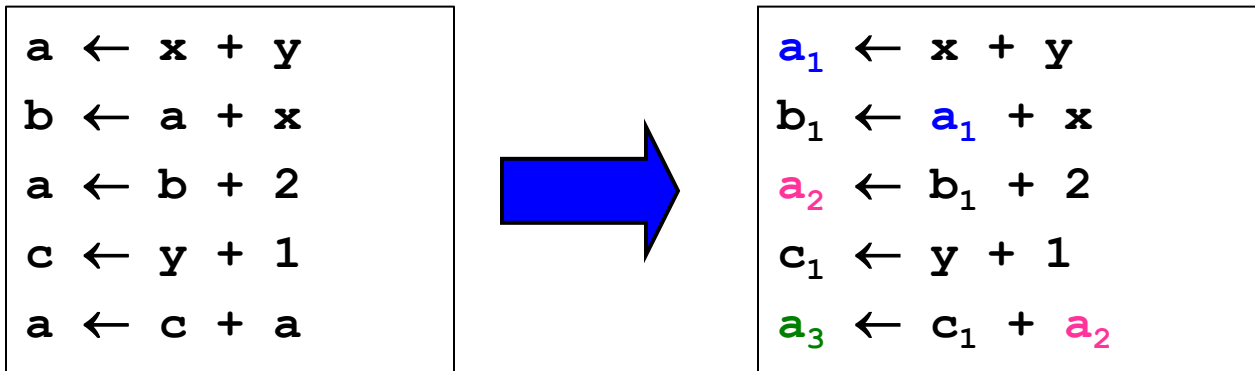
*È inoltre vietata la redistribuzione e la pubblicazione dei contenuti e immagini non autorizzata espressamente dall'autore o dall'Università di Modena e Reggio Emilia.*

# Credits

- Cooper, Torczon, “Engineering a Compiler”, Elsevier
- Sampson, Cornell University, “Advanced Compilers”
- Gibbons, Carnegie Mellon University, “Optimizing Compilers”
- Pekhimenko, University of Toronto, “Compiler Optimization”

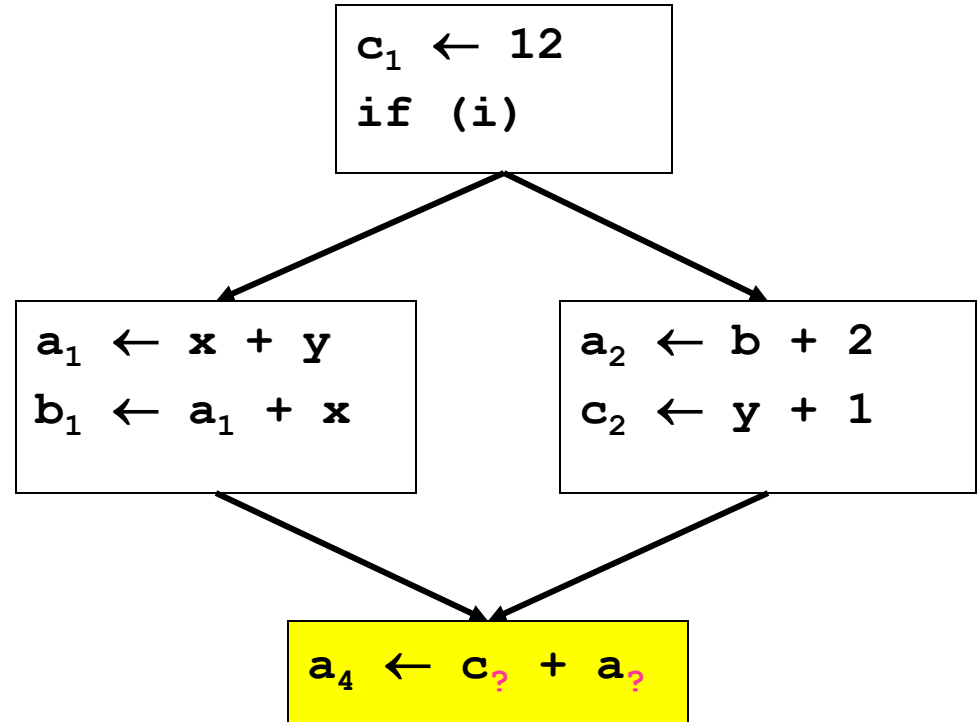
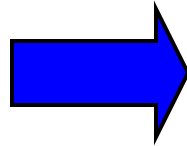
# Static Single Assignment (SSA)

- La forma Static Single Assignment (SSA) è una IR dove ad ogni variabile viene assegnato un valore solo una volta
- Facile da fare dentro un Basic Block (ricordate il *Value Numbering?*):
  - Visita ogni istruzione nell'ordine del programma:
    - LHS: assegna ad una nuova versione della variabile
    - RHS: usa la versione più recente di quella variabile



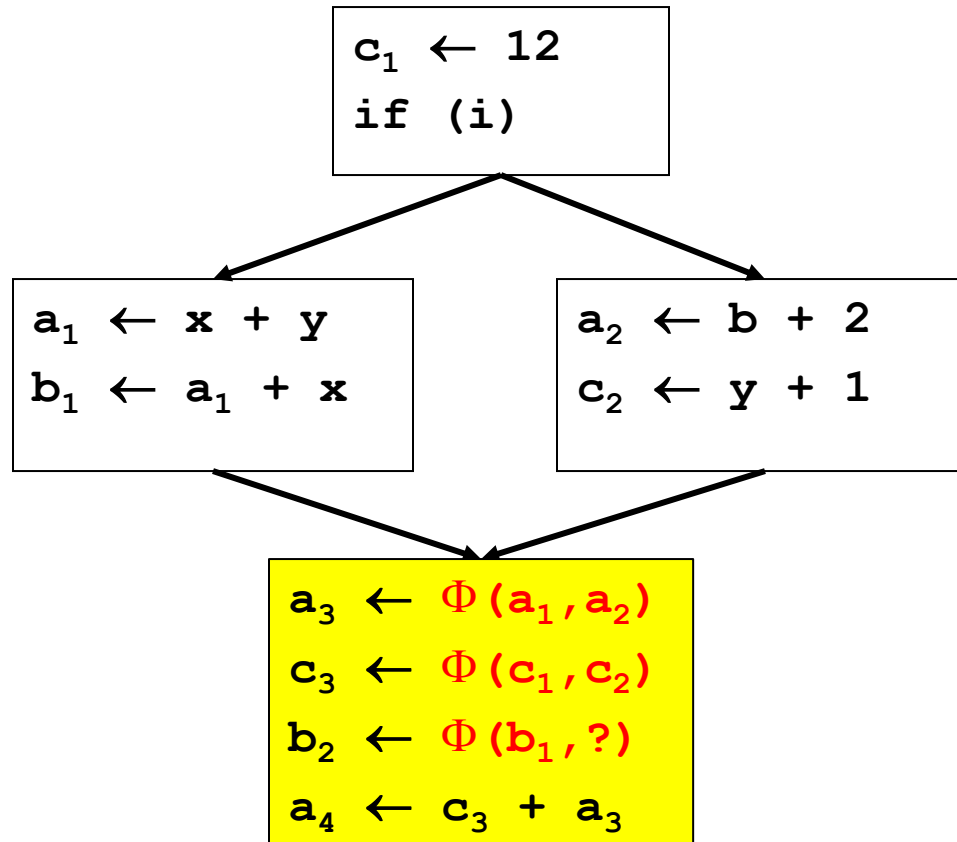
# Cosa succede nei punti di *join* di un CFG?

```
c ← 12
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
a ← c + a
```



→ Usiamo una **notazione fittizia**: una  $\Phi$  function

# La funzione $\Phi$



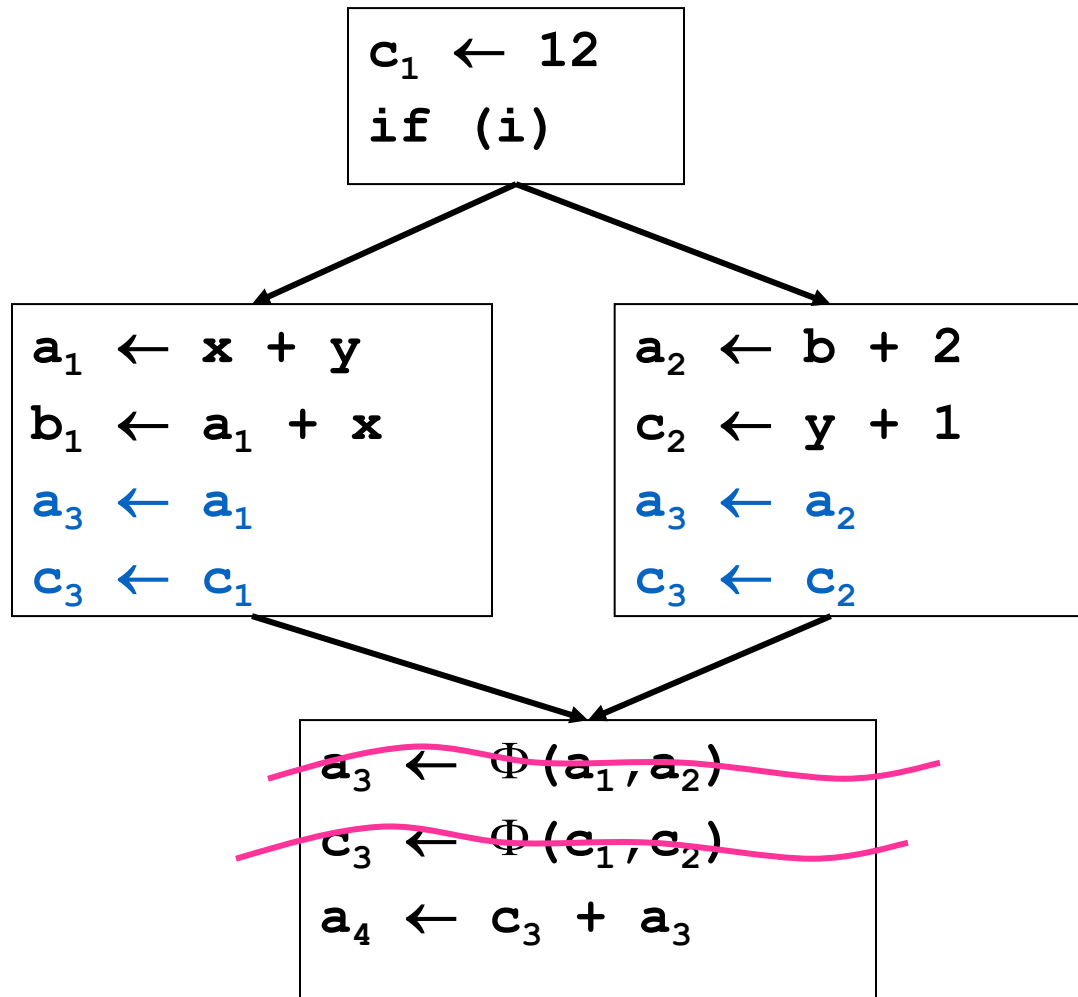
# La funzione $\Phi$

- $\Phi$  fonde multiple definizioni derivanti da multipli percorsi in una singola definizione.
- Per un basic block con  $p$  predecessori ci sono  $p$  argomenti nella funzione  $\Phi$ .

$$x_{\text{new}} \leftarrow \Phi(x_1, x_1, x_1, \dots, x_p)$$

- Come scegliamo quale  $x_i$  usare?
  - In realtà tipicamente non ci interessa!
  - Se è rilevante, usiamo le definizioni derivanti dall'arco di interesse

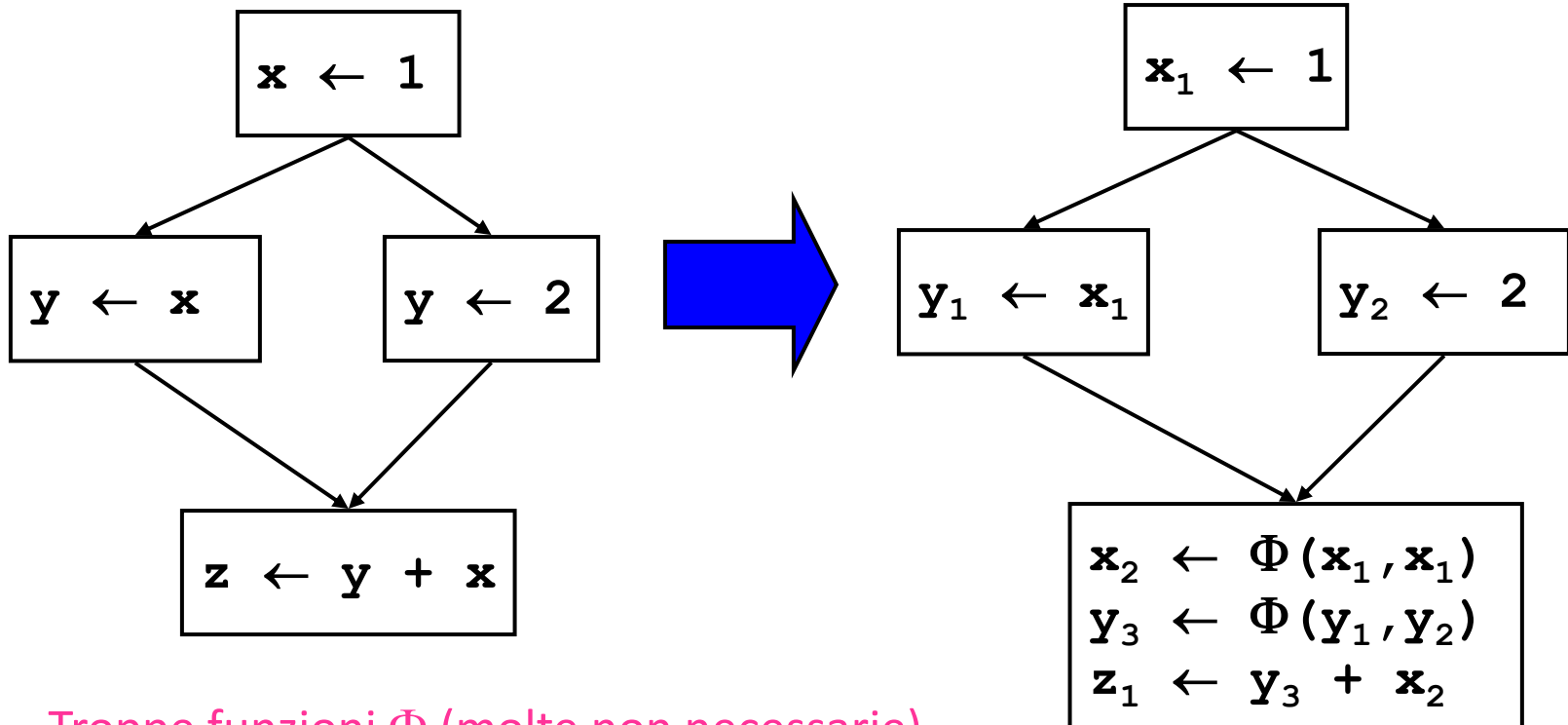
# Come si implementa $\Phi$ ?





# SSA Triviale

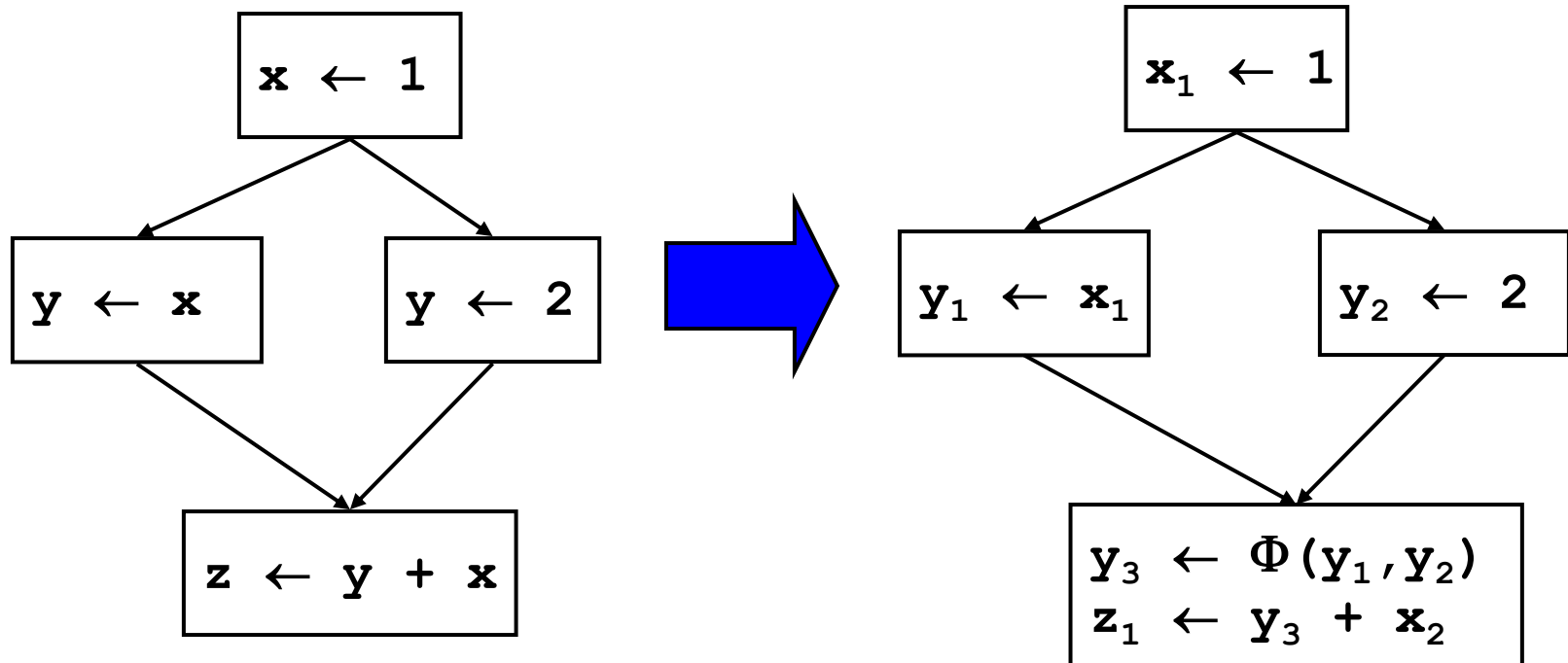
- Ogni assegnamento genera una nuova versione della variabile.
- Aggiungiamo una funzione  $\Phi$  ad ogni punto di join per tutte le variabili live.



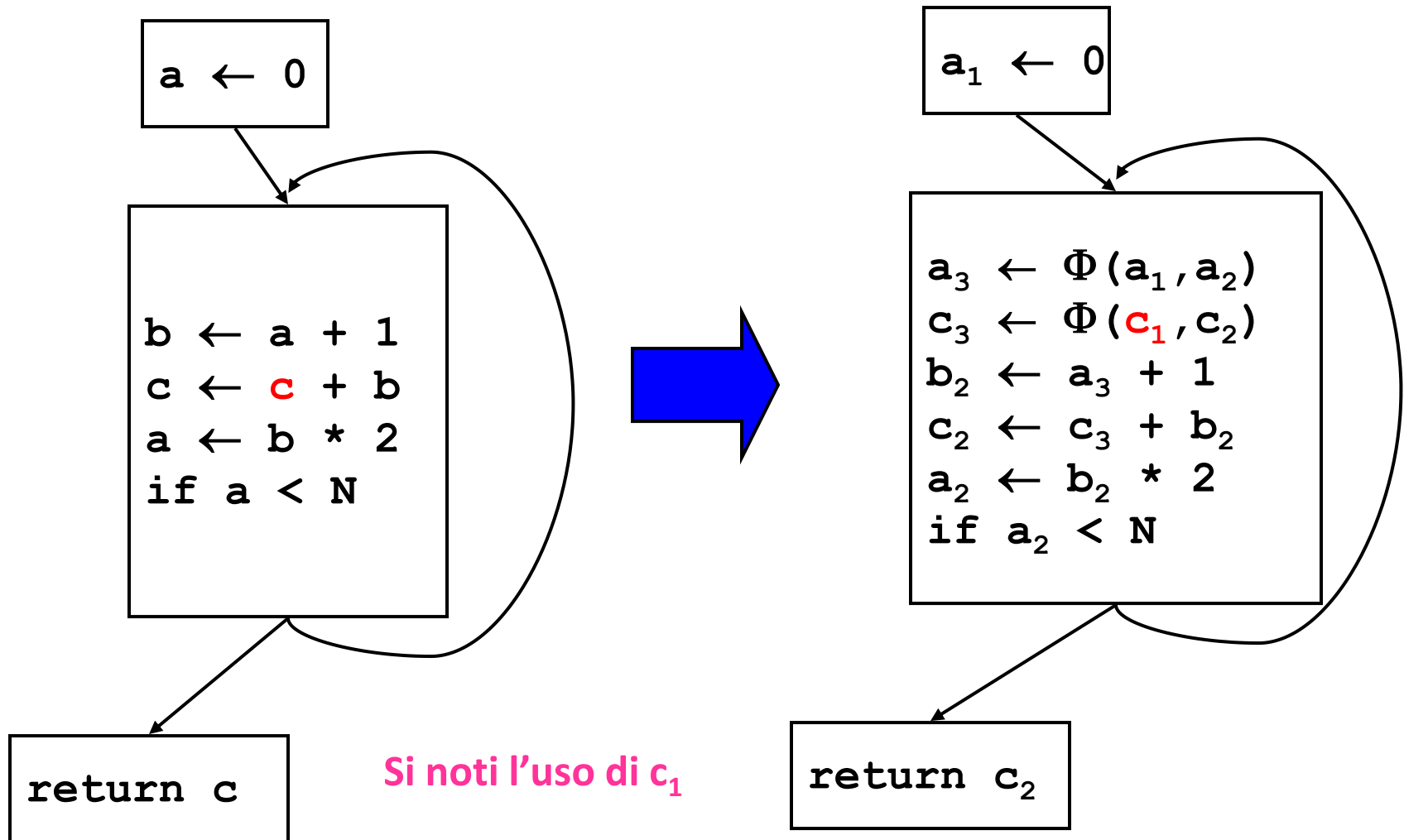
Troppe funzioni  $\Phi$  (molte non necessarie).

# SSA Minimale

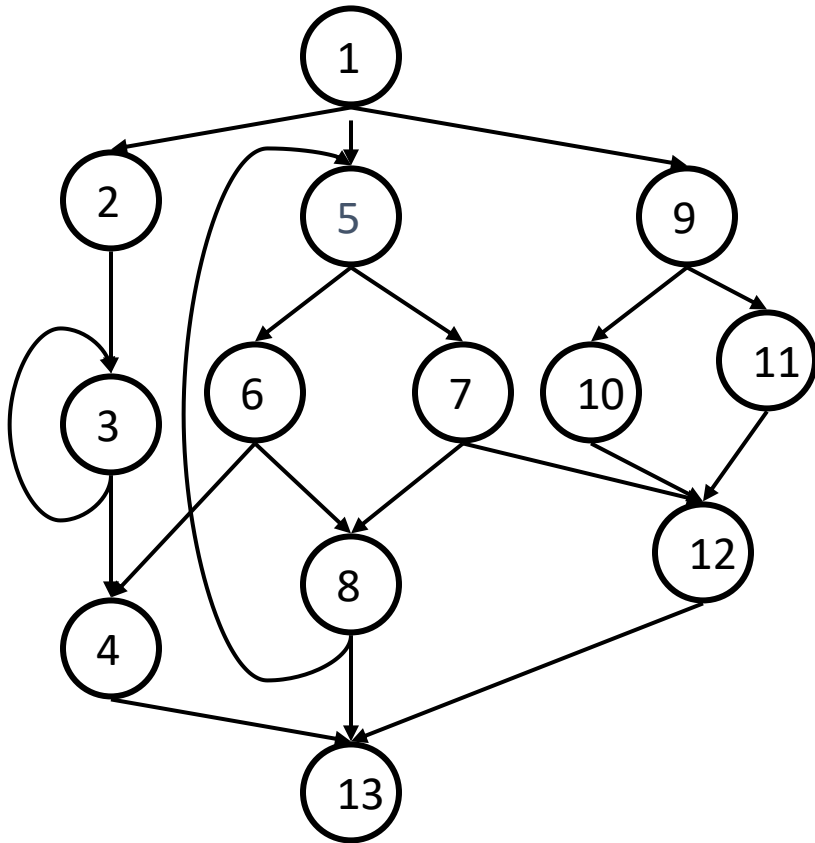
- Ogni assegnamento genera una nuova versione della variabile.
- Aggiungiamo una funzione  $\Phi$  ad ogni punto di join per tutte le variabili live **con definizioni multiple**.



# Un altro esempio



# Dove inseriamo le funzioni $\Phi$ ?

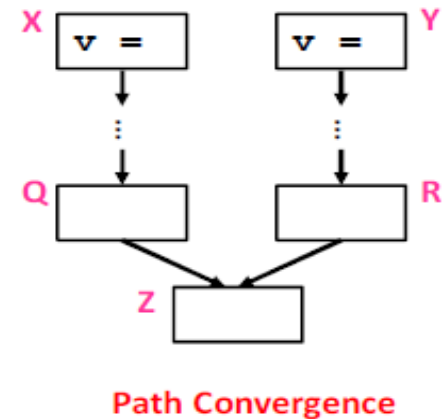


CFG

- Se c'è una definizione della variabile **a** nel blocco **5**, quali nodi hanno bisogno di una funzione  $\Phi()$ ?

# Dove inseriamo le funzioni $\Phi$ ?

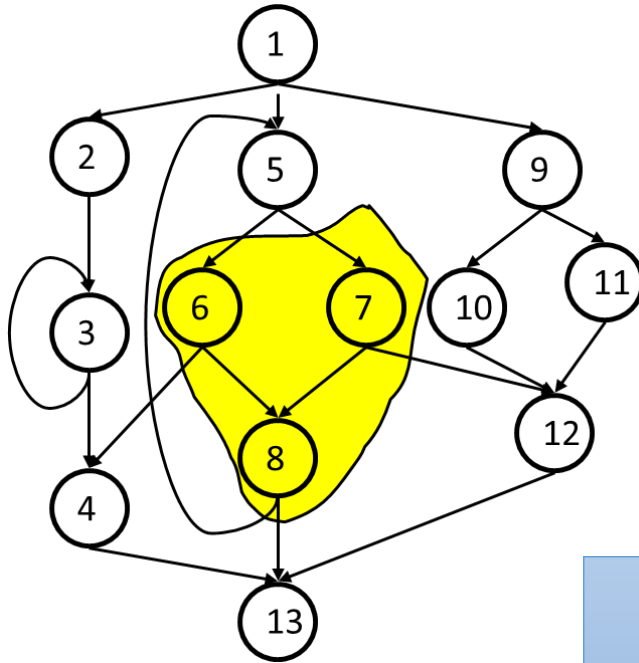
- Inseriamo una funzione  $\Phi$  per una variabile  $A$  nel blocco  $Z$  iff:
  - $A$  è stata definita più di una volta
    - (es.,  $A$  definita in  $X$  e  $Y$  e  $X \neq Y$ )
  - Esistono due percorsi da  $x$  a  $z$  ( $P_{xz}$ ) e da  $y$  a  $z$  ( $P_{yz}$ ) tali che:
    - $P_{xz} \cap P_{yz} = \{ z \}$   
( $Z$  è l'unico blocco comune tra i percorsi)
    - $z \notin P_{xq}$  or  $z \notin P_{yr}$  where  $P_{xz} = P_{xq} \rightarrow z$  and  $P_{yz} = P_{yr} \rightarrow z$   
(almeno un percorso raggiunge  $Z$  per la prima volta)
- Il blocco ENTRY contiene una definizione implicita di tutte le variabili
- Nota:  $v = \Phi(\dots)$  è una definizione di  $v$



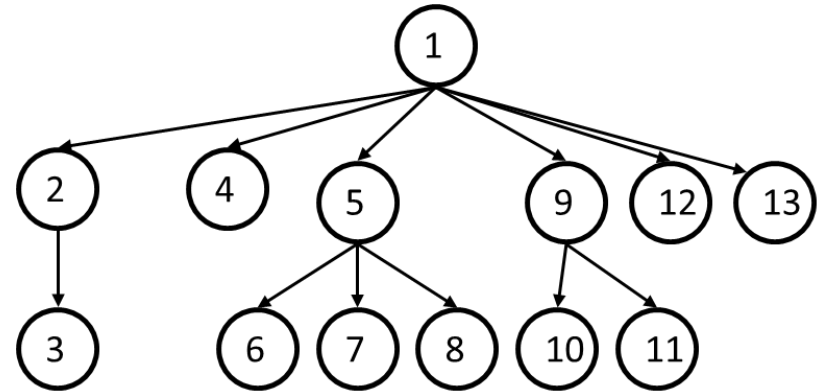
# Proprietà di dominanza della forma SSA

- Nella forma SSA le definizioni dominano gli usi.
  - Se  $x_i$  è usato in  $x \leftarrow \Phi(\dots, x_i, \dots)$ , allora  $BB(x_i)$  domina il predecessore  $i$ -esimo di  $BB(PHI)$
  - Se  $x$  è usato in  $y \leftarrow \dots x \dots$ , allora  $BB(x)$  domina  $BB(y)$
- Si può usare questa proprietà per derivare un algoritmo efficiente per convertire la IR in forma SSA

# Dominanza



CFG

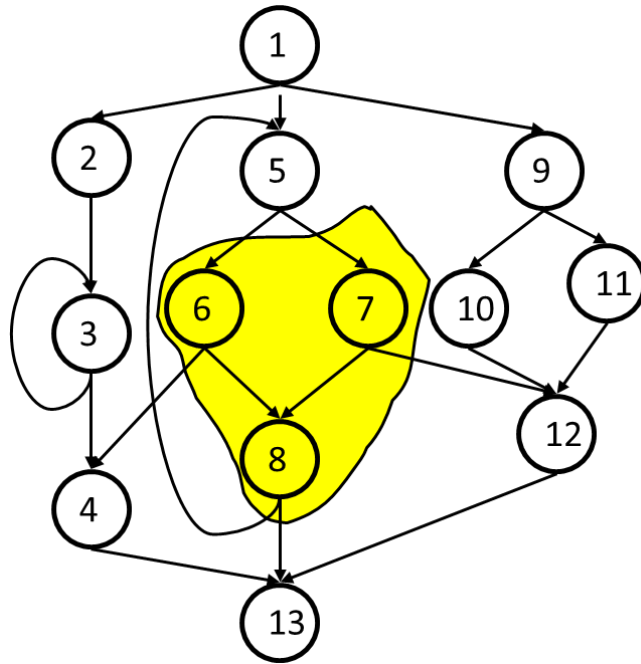


D-Tree

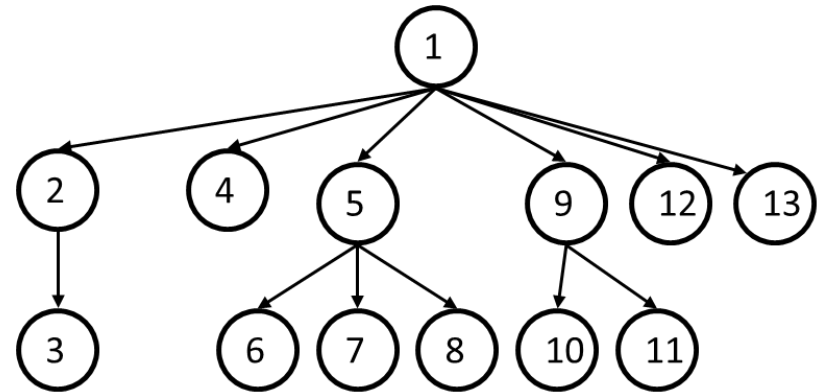
Se c'è una definizione di **a** nel blocco **5**, quali nodi hanno bisogno di una funzione  $\Phi()$ ?

**x** domina strettamente **w** (**x** **sdom** **w**) **iff** **x** **dom** **w**  $\wedge$  **x**  $\neq$  **w**

# Dominance Frontier



CFG



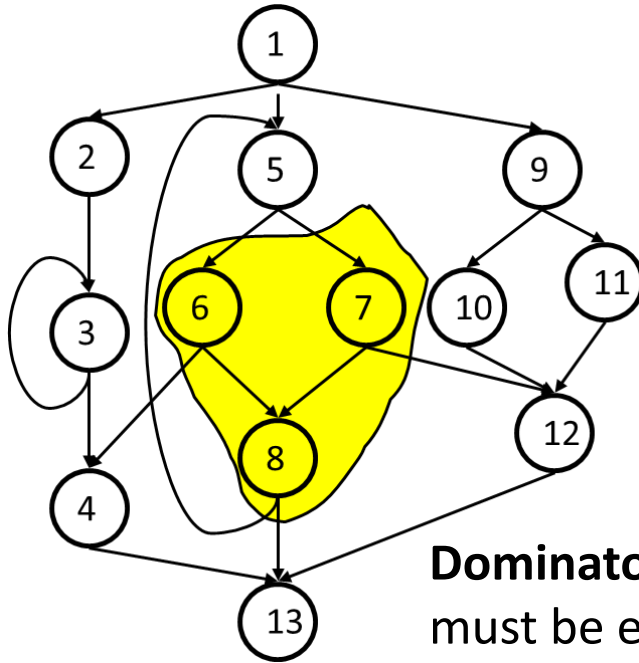
D-Tree

La **Dominance Frontier** di un nodo  $x =$   
 $\{ w \mid x \text{ dom pred}(w) \wedge \neg(x \text{ sdom } w) \}$

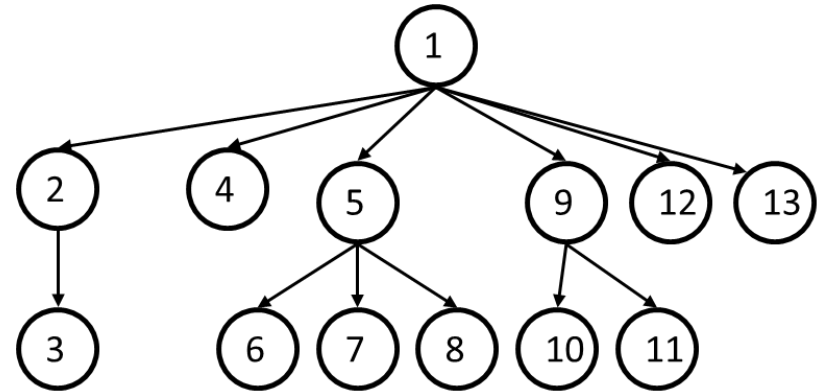
$x$  domina strettamente  $w$  ( $x \text{ sdom } w$ ) iff  $x \text{ dom } w \wedge x \neq w$



# Dominance Frontier



CFG



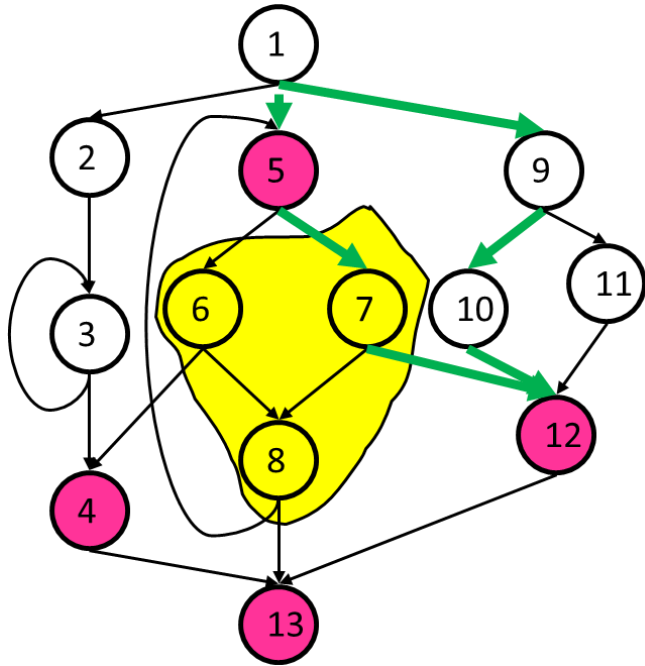
D-Tree

**Dominators** and **postdominators** tell us which basic block must be executed prior to, of after, a block  $x$ .

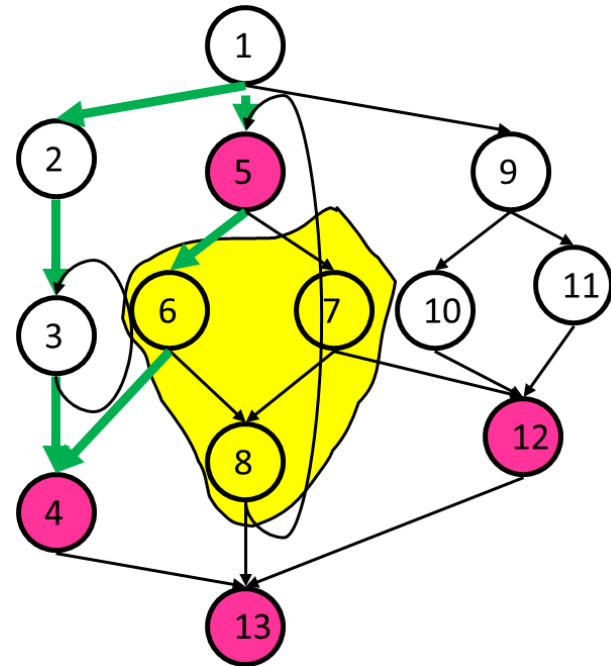
It is interesting to consider blocks “just before” or “just after” blocks we’re dominated by, or blocks we dominate.

The **Dominance Frontier** of a basic block  $x$ ,  $DF(x)$ , is the set of all blocks that are immediate successors to blocks dominated by  $x$ , but which aren’t themselves strictly dominated by  $x$ .

# Dominance Frontier



CFG



Se c'è una definizione di **a** nel blocco **5** i nodi nella  $DF(5)$  necessitano di una funzione  $\Phi()$  per **a**

# Utilizzo della Dominance Frontier per calcolare la forma SSA

- Posizionare tutte le funzioni  $\Phi()$
- Rinominare tutte le variabili

# Utilizzo della Dominance Frontier per posizionare le funzioni $\Phi()$

- Identificare tutti i siti dove vengono definite le variabili


- Quindi, per ogni variabile

- Per ogni sito di definizione

- Per ogni nodo in DF (del sito di definizione)

- Se non c'è una  $\Phi()$  nel nodo, mettiamone una
      - Se questo nodo non definiva la variabile in precedenza, aggiungiamo questo nodo alla lista dei siti di definizione

Non siamo ancora in  
forma SSA (la stiamo  
costruendo)



- Questo algoritmo calcola iterativamente la Dominance Frontier, inserendo il numero minimo necessario di funzioni  $\Phi()$

# Utilizzo della Dominance Frontier per posizionare le funzioni $\Phi()$

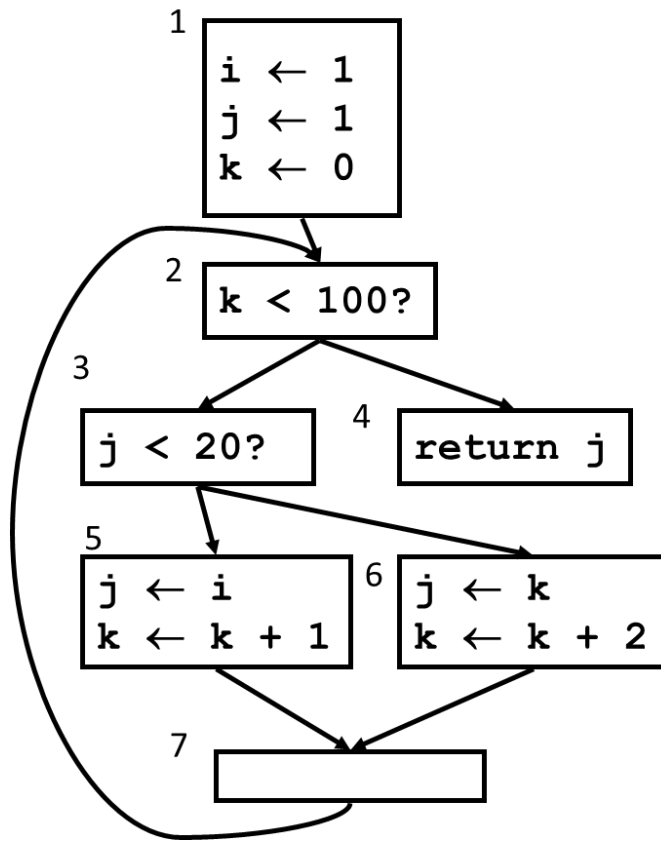
```
foreach node n {
    foreach variable v defined in n {
        orig[n]  $\cup$ = {v}
        defsites[v]  $\cup$ = {n}
    }
}

foreach variable v {
    W = defsites[v]
    while W not empty {
        n = remove node from W
        foreach y in DF[n]
            if y  $\notin$  PHI[v] {
                insert " $v \leftarrow \Phi(v,v,...)$ " at top of y
                PHI[v] = PHI[v]  $\cup$  {y}
                if v  $\notin$  orig[y]: W = W  $\cup$  {y}
            }
    }
}
```

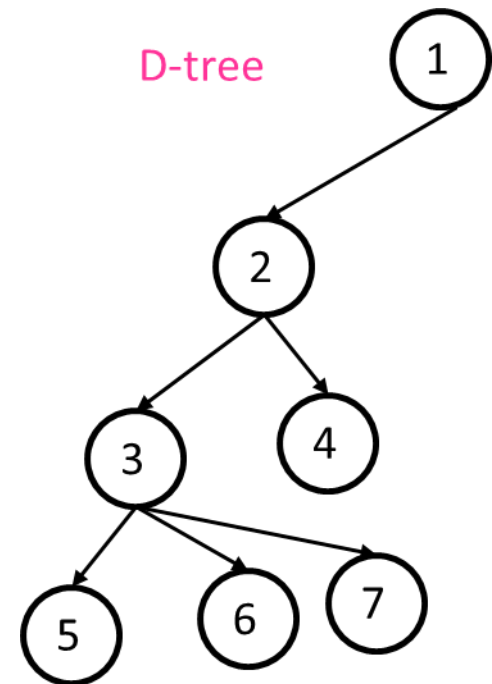
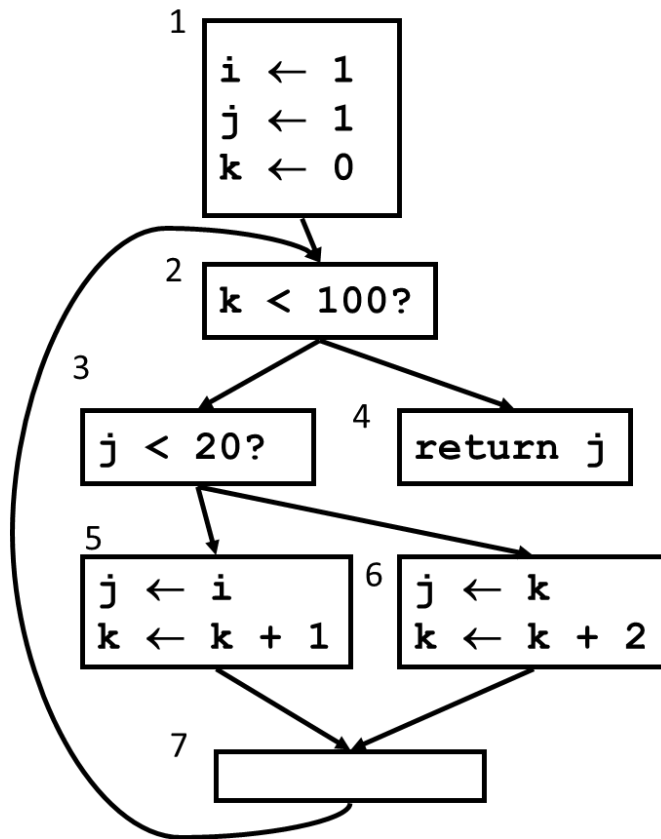
# Rinominare le variabili

- Algoritmo:
  - Scorrere il D-tree, rinominando le variabili come le si incontra
  - Rimpiazzare gli usi con la più recente definizione rinominata
- Per codice con control flow lineare questo è semplice
- Cosa succede in presenza di branch e join?
  - Utilizzare la definizione più vicina tale per cui la definizione si trova prima dell'uso nel D-tree
- Un'implementazione semplice a stack:
  - Per ogni var: rename (v)
  - rename(v): rimpiazza gli usi col top dello stack
    - se si incontra una definizione: push sullo stack
    - invocare rename(v) su tutti i figli nel D-tree
    - per ogni definizione nel blocco pop dallo stack

# Esempio: Calcolo del Dominance Tree

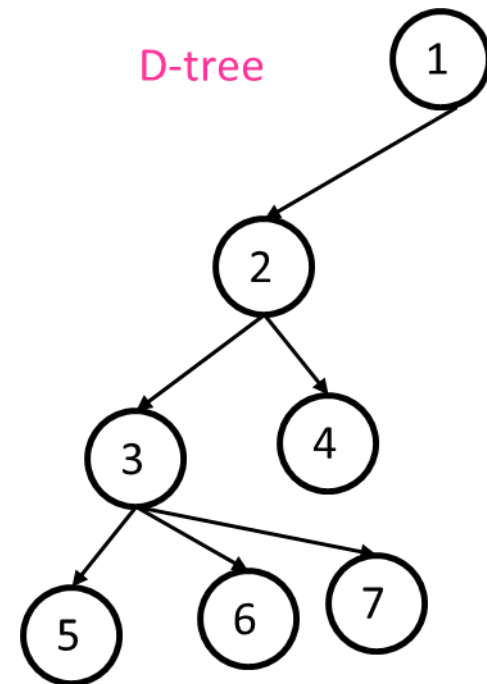
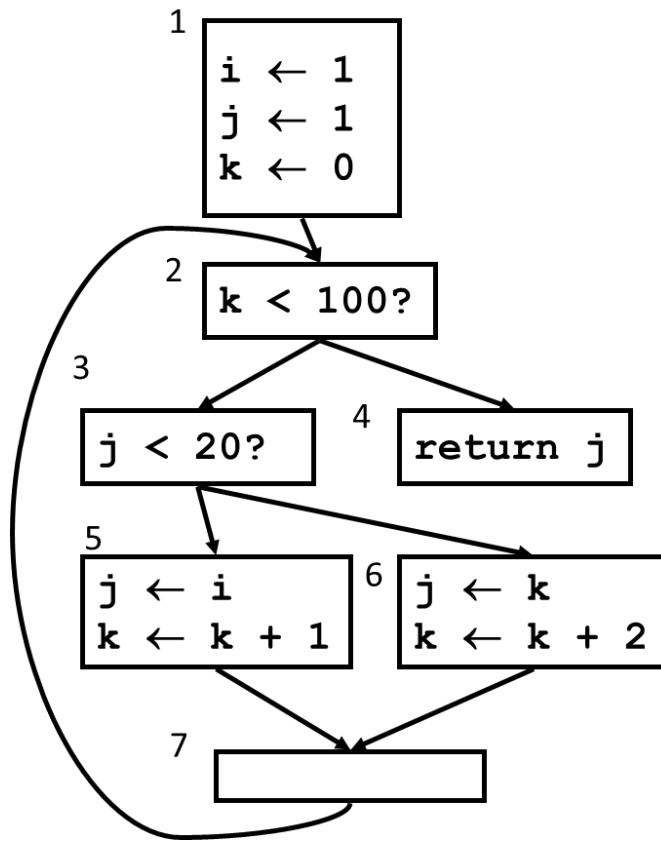


# Esempio: Calcolo del Dominance Tree

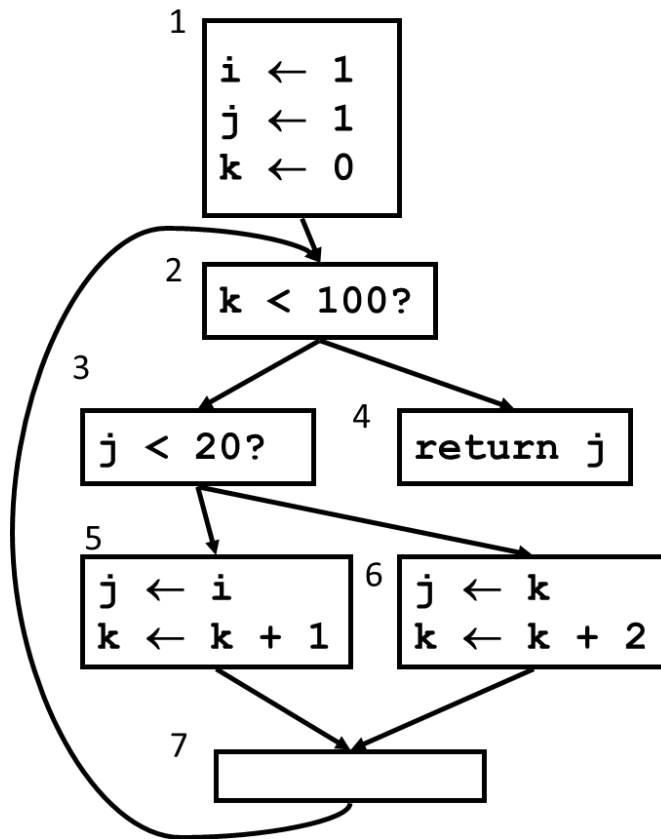




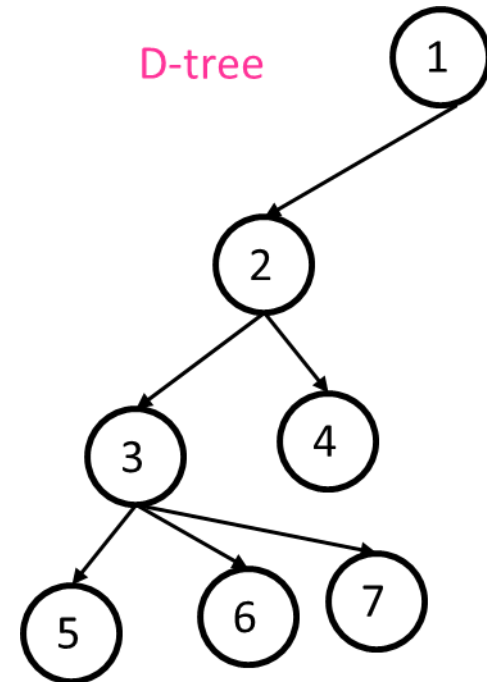
# Esempio: Calcolo delle Dominance Frontiers



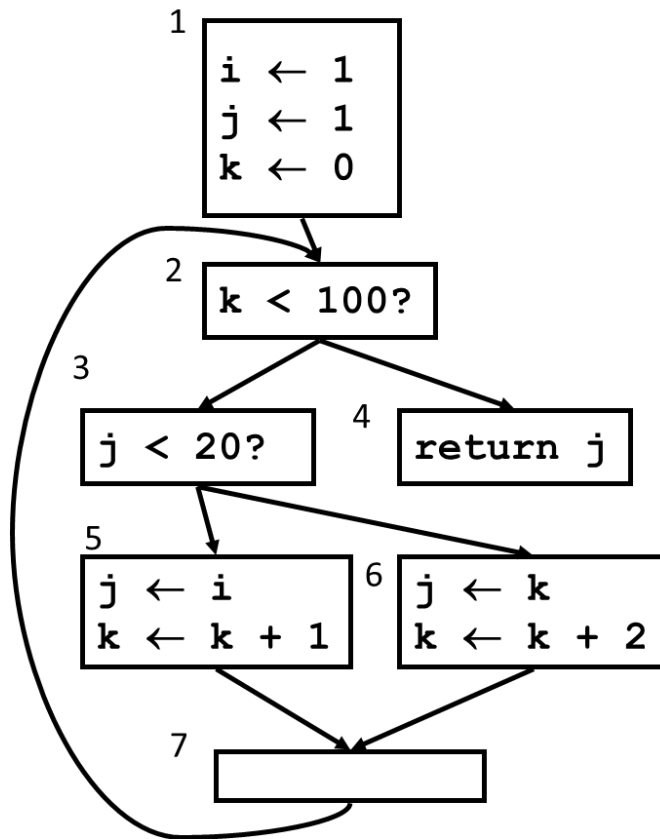
# Esempio: Calcolo delle Dominance Frontiers



DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}



# Esempio: $\Phi()$



DFs		orig[n]			
1	{}	1	{ i,j,k}		
2	{2}	2	{}		
3	{2}	3	{}	defsites[v]	
4	{}	4	{}	i	{1}
5	{7}	5	{j,k}	j	{1,5,6}
6	{7}	6	{j,k}	k	{1,5,6}
7	{2}	7	{}		

```

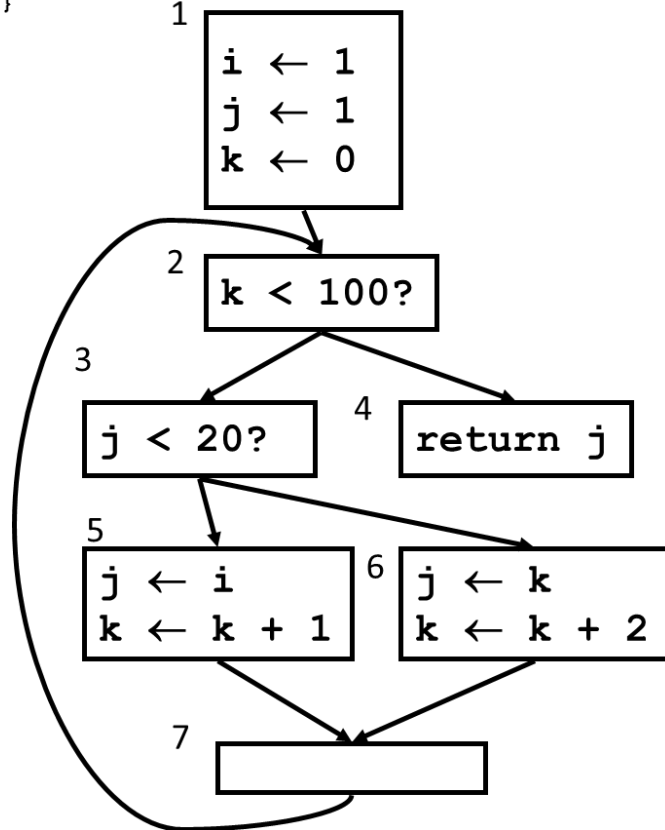
foreach node n {
    foreach variable v defined in n {
        orig[n] ∪= {v}
        defsites[v] ∪= {n}
    }
}
  
```

# Esempio: $\Phi()$

```

foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n]
      if y  $\notin$  PHI[v] {
        insert " $v \leftarrow \Phi(v,v,...)$ " at top of y
        PHI[v] = PHI[v]  $\cup$  {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
  }
}

```



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]

1	{i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

var i: W={1}

var j: W={1,5,6}

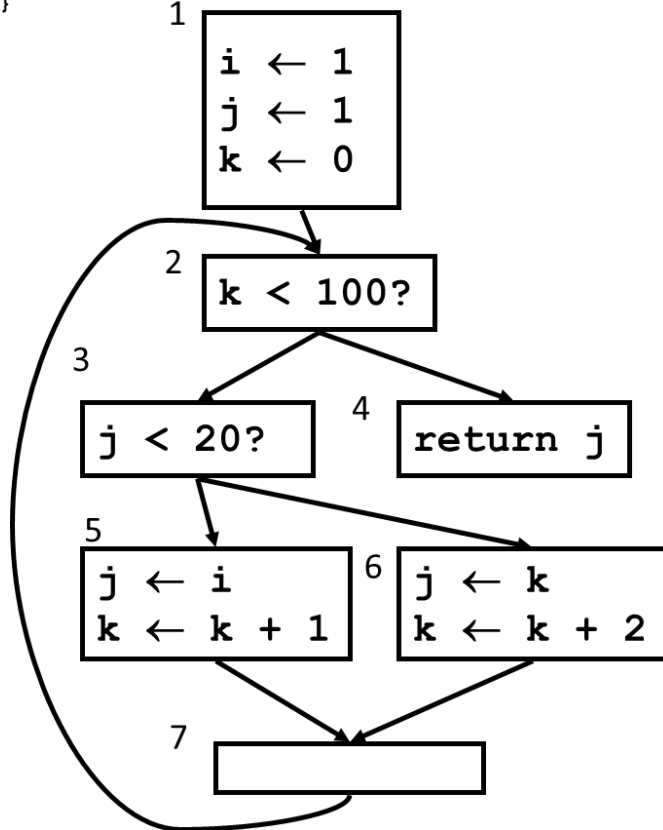
DF{1}      DF{5}

# Esempio: $\Phi()$

```

foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n]
      if y  $\notin$  PHI[v] {
        insert " $v \leftarrow \Phi(v,v,...)$ " at top of y
        PHI[v] = PHI[v]  $\cup$  {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
  }
}

```



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]

1	{i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

var j: W={1,5,6}

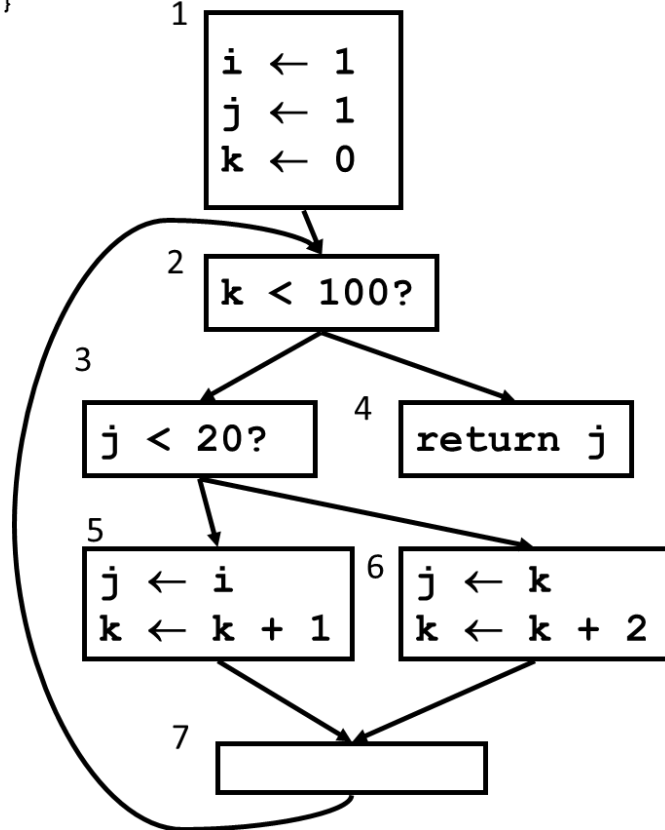
DF{1}      DF{5}

# Esempio: $\Phi()$

```

foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n]
      if y  $\notin$  PHI[v] {
        insert " $v \leftarrow \Phi(v,v,...)$ " at top of y
        PHI[v] = PHI[v]  $\cup$  {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
  }
}

```



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]

1	{i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

i	{1}
j	{1,5,6,7}
k	{1,5,6}

DFs

var j: W={1,5,6,7}

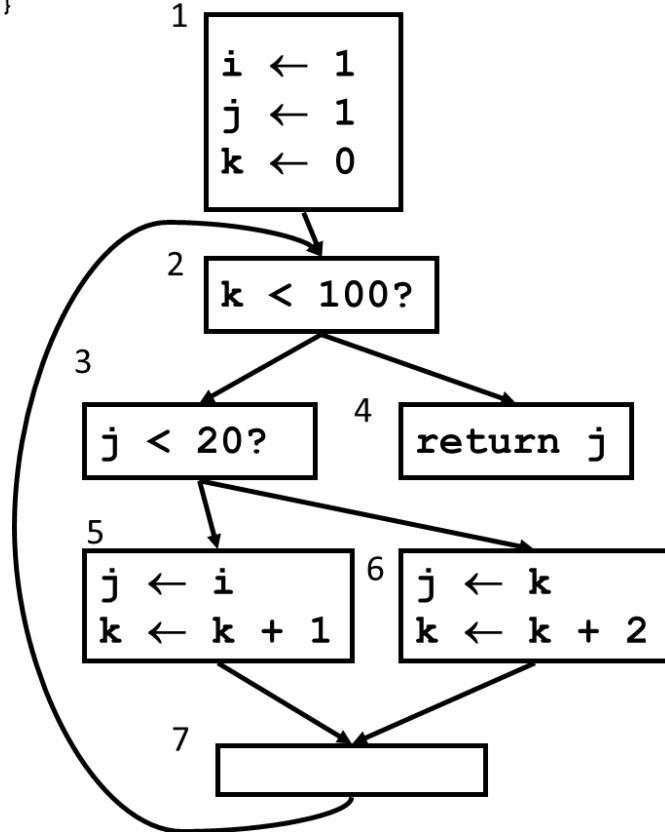
DF{1}    DF{5}    DF{7}

# Esempio: $\Phi()$

```

foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n]
      if y  $\notin$  PHI[v] {
        insert " $v \leftarrow \Phi(v,v,...)$ " at top of y
        PHI[v] = PHI[v]  $\cup$  {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
  }
}

```



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]

1	{i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

var j: W={1,5,6,7}

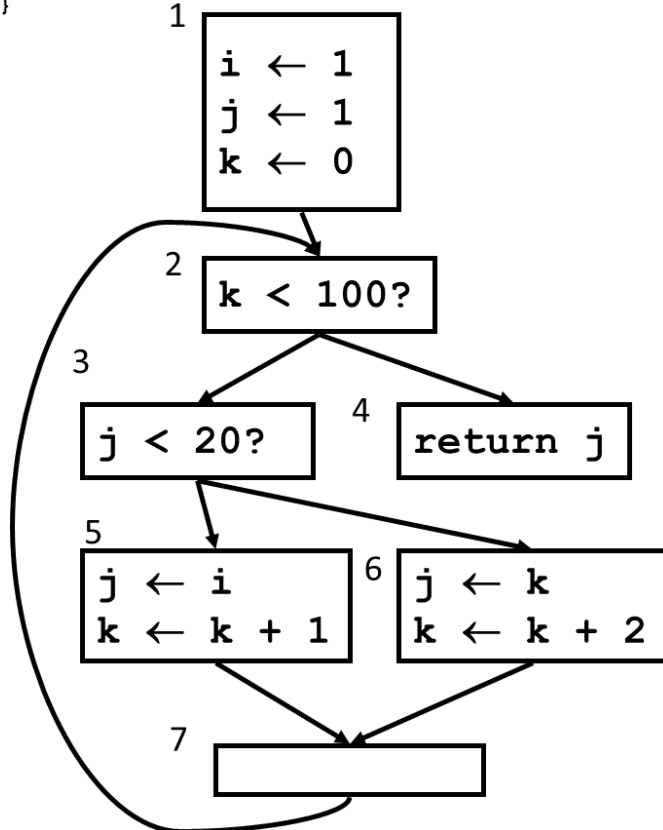
DF{1}    DF{5}    DF{7}    **DF{6}**

# Esempio: $\Phi()$

```

foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n]
      if y  $\notin$  PHI[v] {
        insert " $v \leftarrow \Phi(v,v,...)$ " at top of y
        PHI[v] = PHI[v]  $\cup$  {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
  }
}

```



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]

1	{i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

Def sites[v]

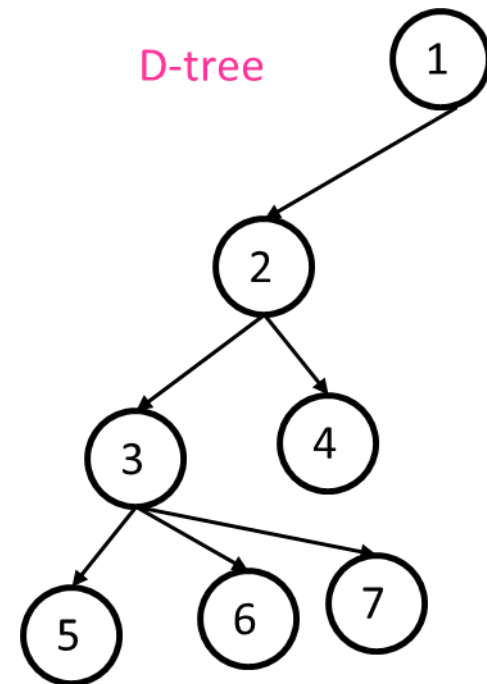
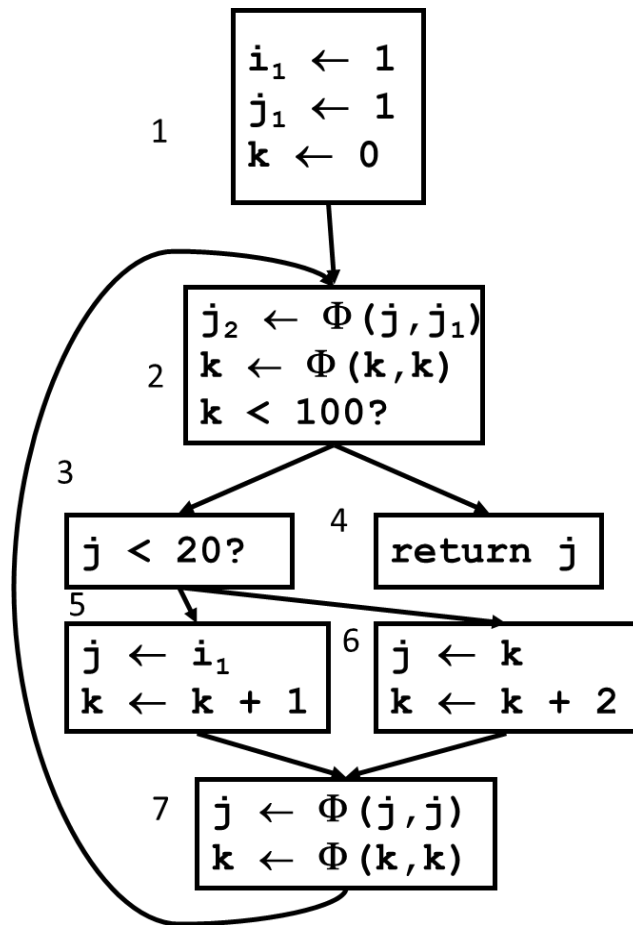
i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

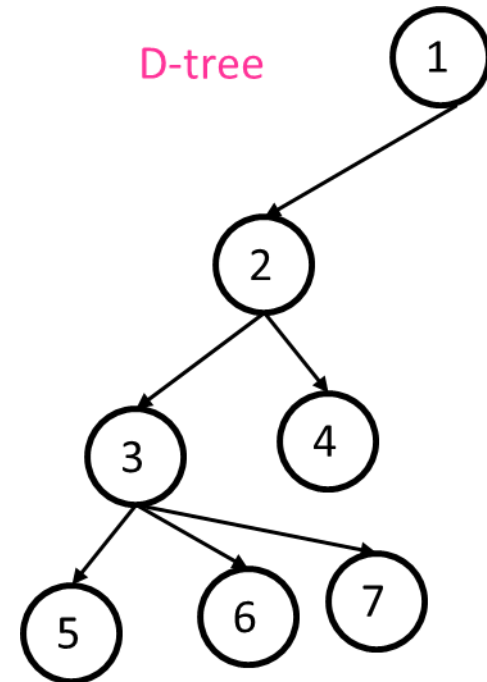
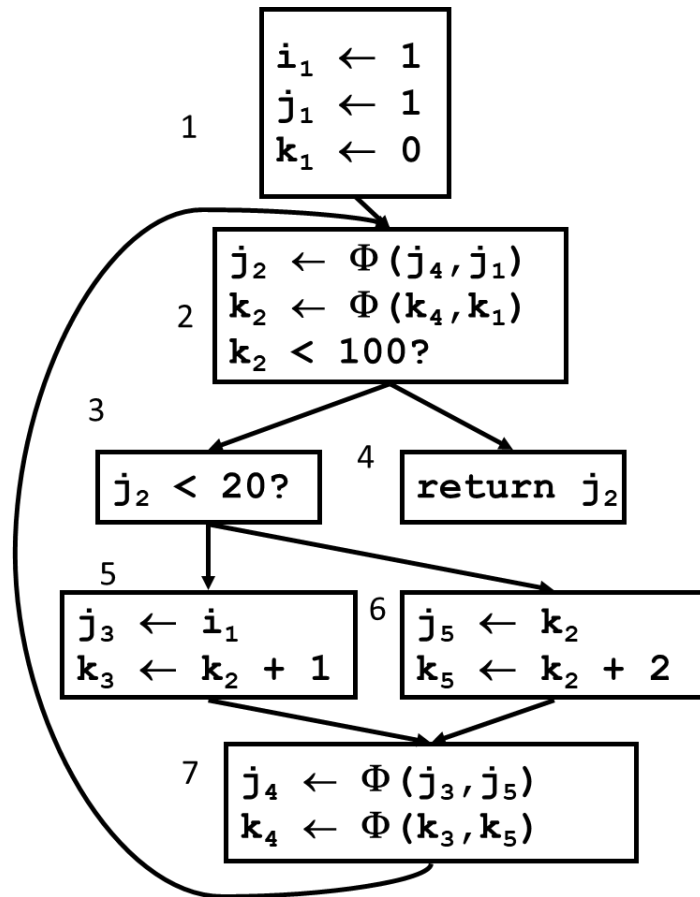
var k: W={1,5,6}



# Esempio: Rinominare le variabili



# Esempio: Rinominare le variabili



# Proprietà della forma SSA

- Solo un assegnamento per variabile
- Le definizioni dominano gli usi
- Un cambio di filosofia sull'analisi e ottimizzazione
- Se rimuoviamo la possibilità di riassegnare valori alle variabili alcuni concetti si fondono:
  - Definizioni = Variabili
  - Istruzioni = Valori
  - Operandi = Archi di un DFG



Da qui la rappresentazione di LLVM



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche

# Esempi di ottimizzazione con SSA

# Constant Propagation con SSA

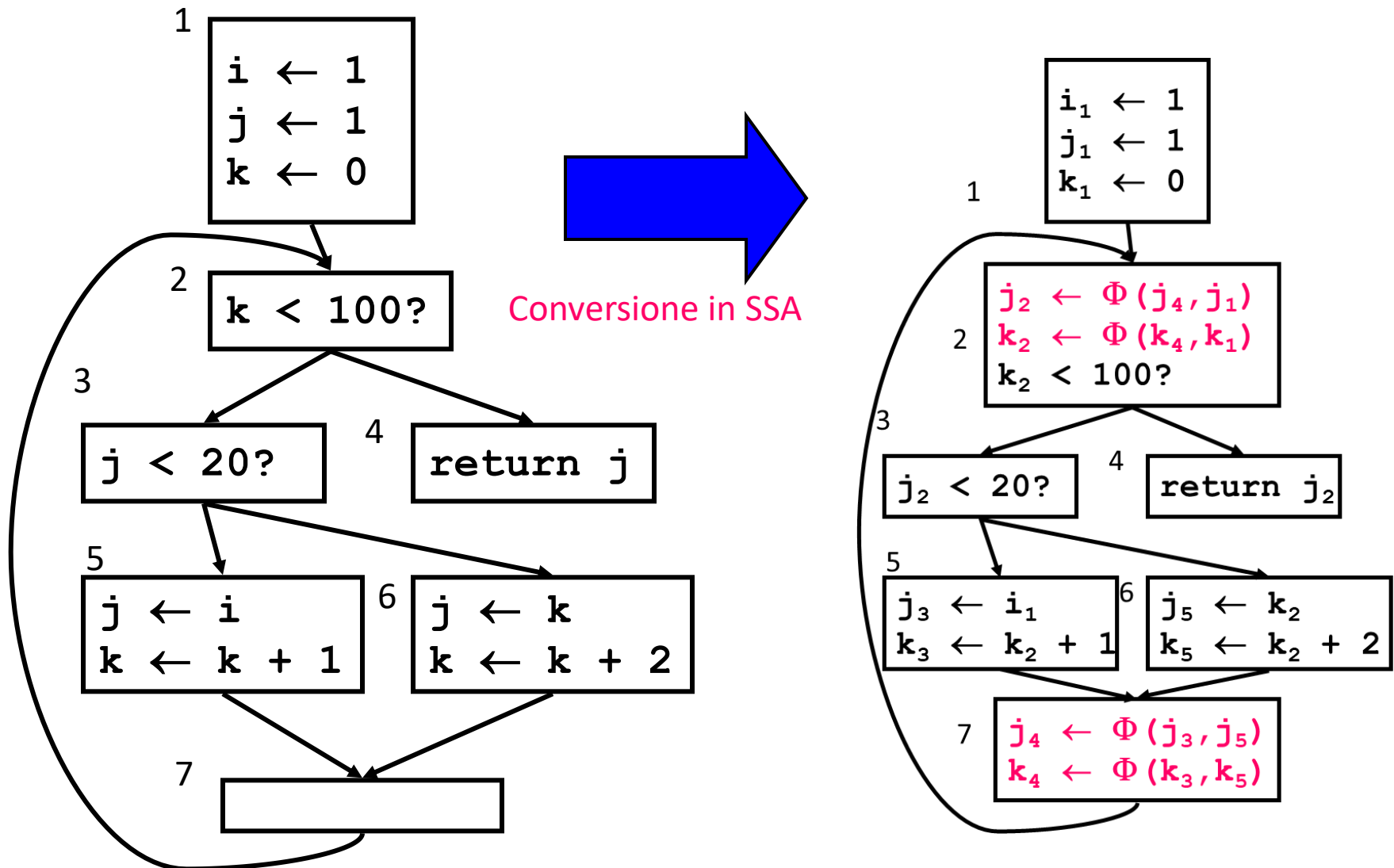
- Se " $v \leftarrow c$ ", sostituisci tutti gli usi di  $v$  con  $c$
- Se " $v \leftarrow \Phi(c,c,c)$ " (ogni input ha la stessa costante), sostituisci tutti gli usi di  $v$  con  $c$

```
W ← lista di tutte le defs
while !W.isEmpty {
    Stmt S ← W.removeOne
    if ((S has form " $v \leftarrow c$ ") ||
        (S has form " $v \leftarrow \Phi(c, \dots, c)$ ")) then {
        delete S
        foreach stmt U that uses v {
            replace v with c in U
            W.add(U)
        }
    }
}
```

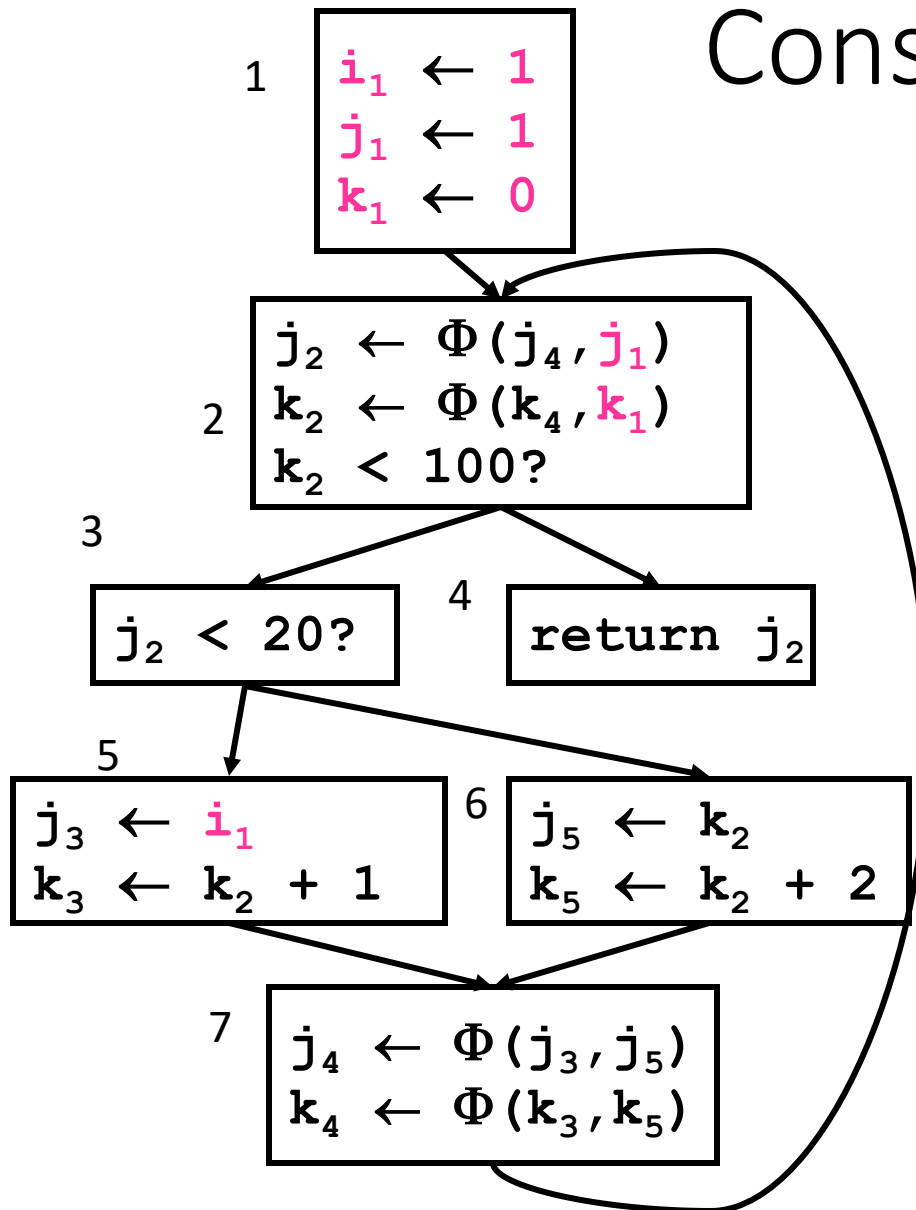
# Altre ottimizzazioni con SSA

- Copy propagation
  - rimuovi “ $x \leftarrow \Phi(y,y,y)$ ” e sostituisci  $y$  a tutte le  $x$
  - rimuovi “ $x \leftarrow y$ ” e sostituisci  $y$  a tutte le  $x$
- Constant Folding
  - (Also, constant conditions too!)

# Constant Propagation

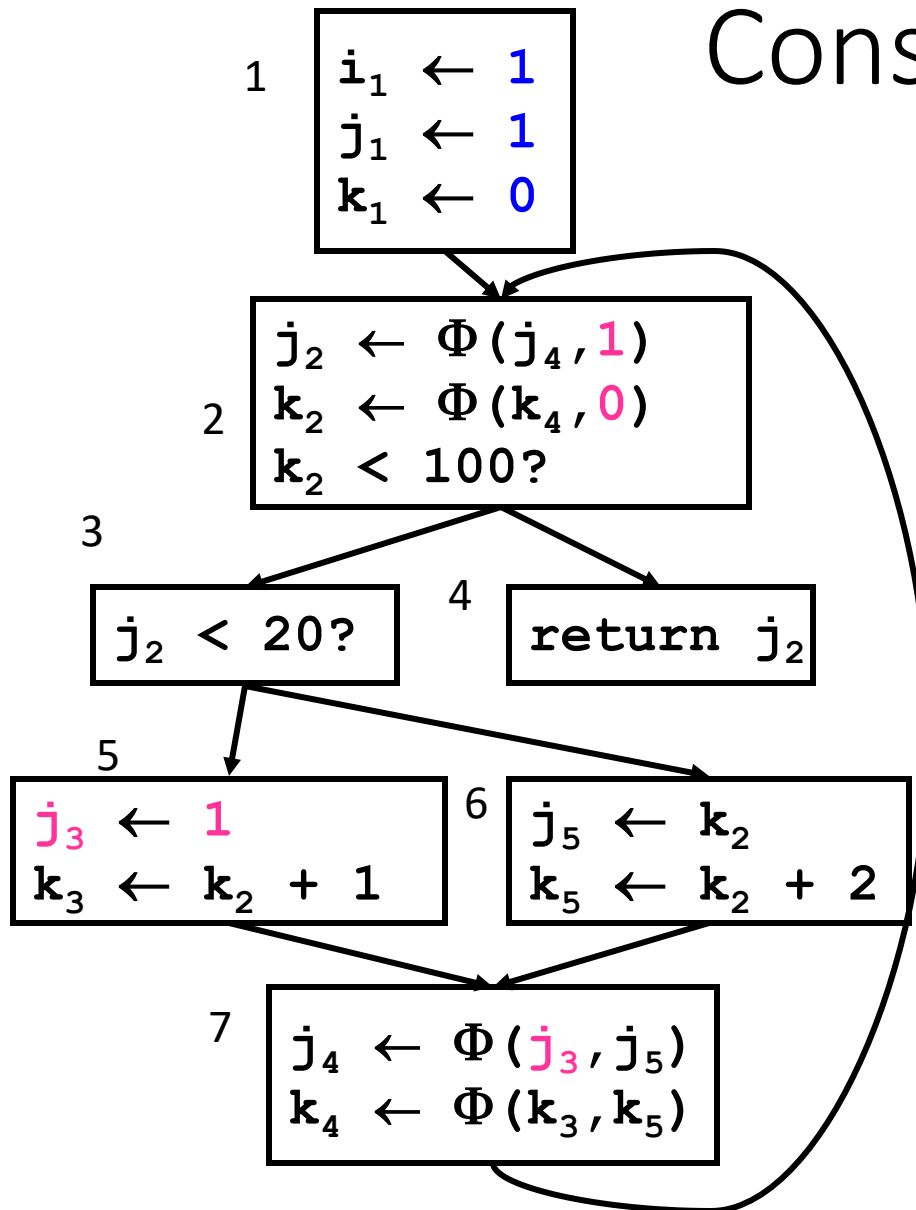


# Constant Propagation

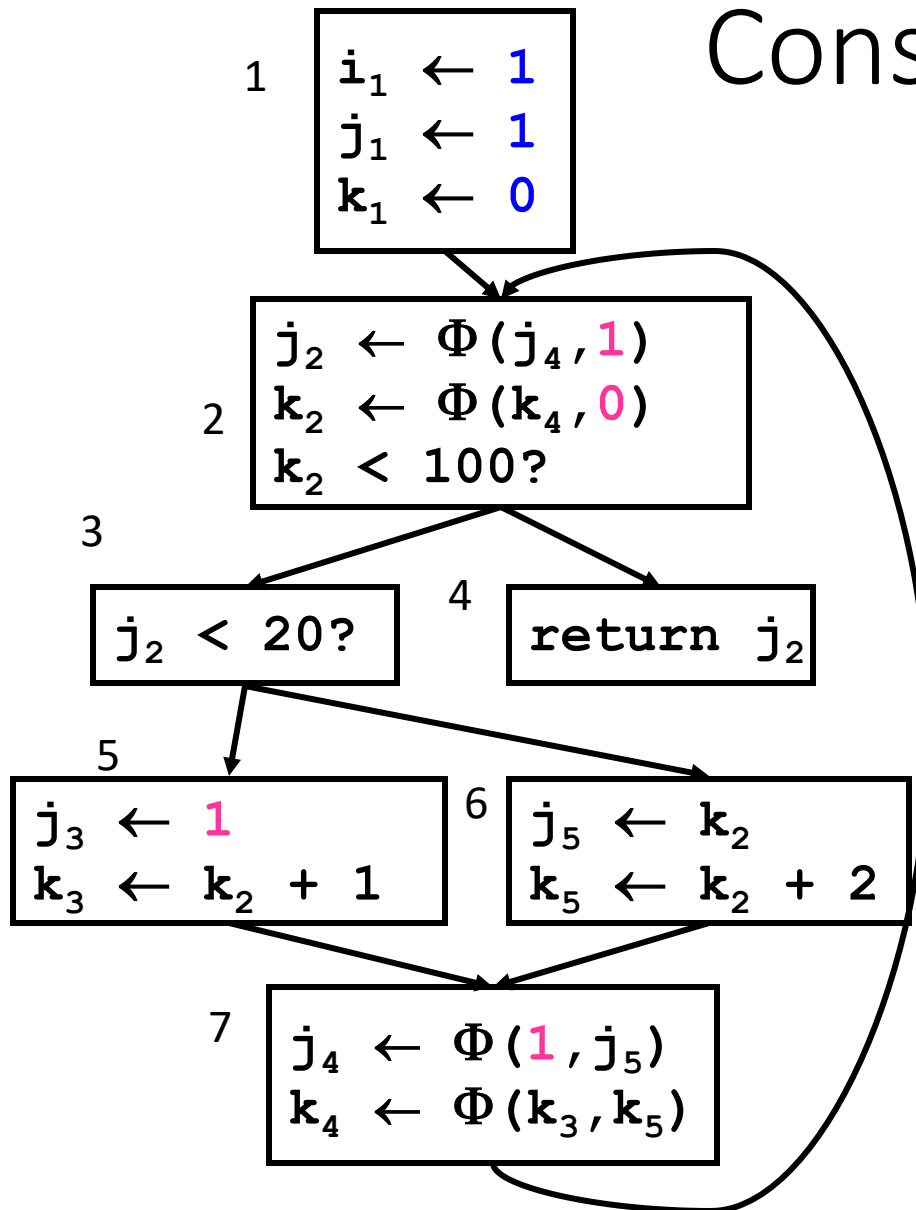




# Constant Propagation

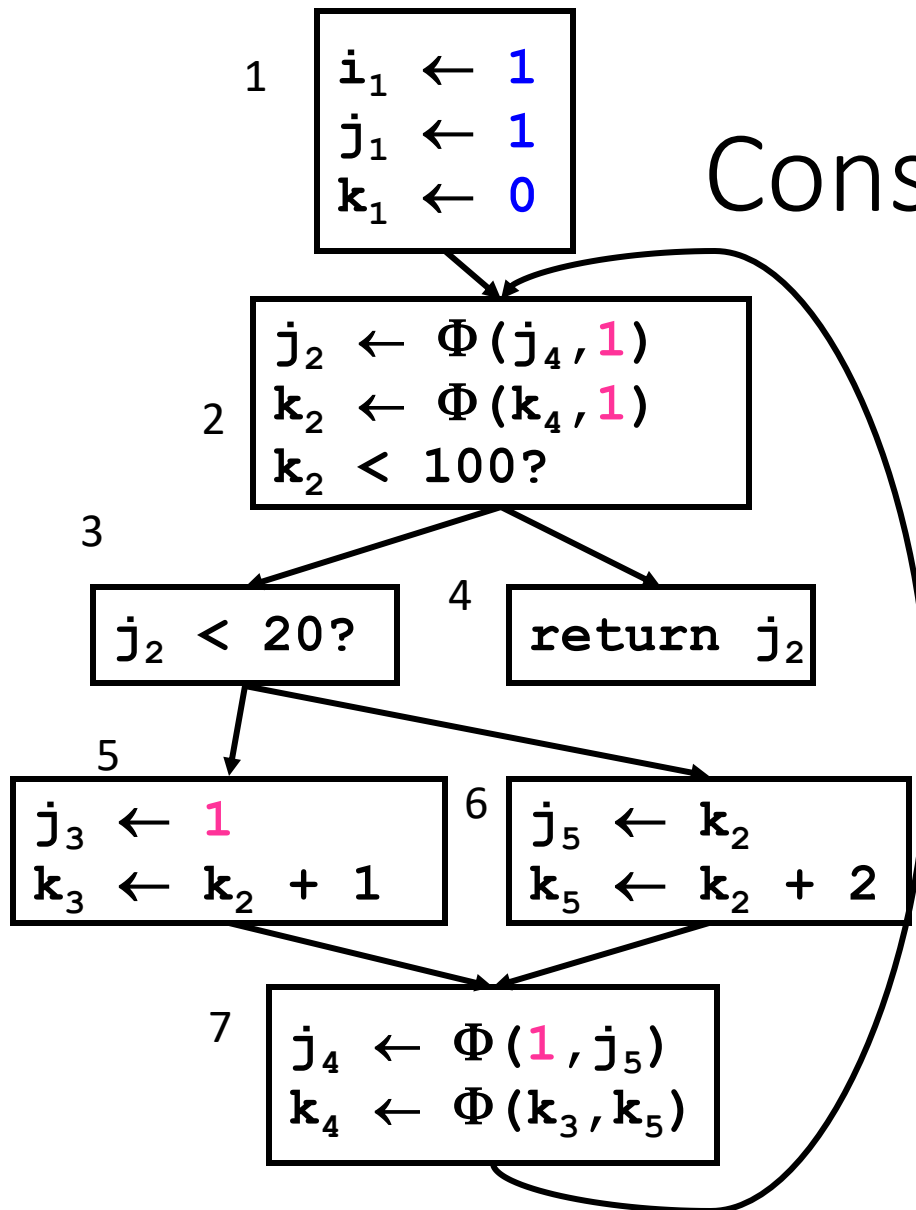


# Constant Propagation



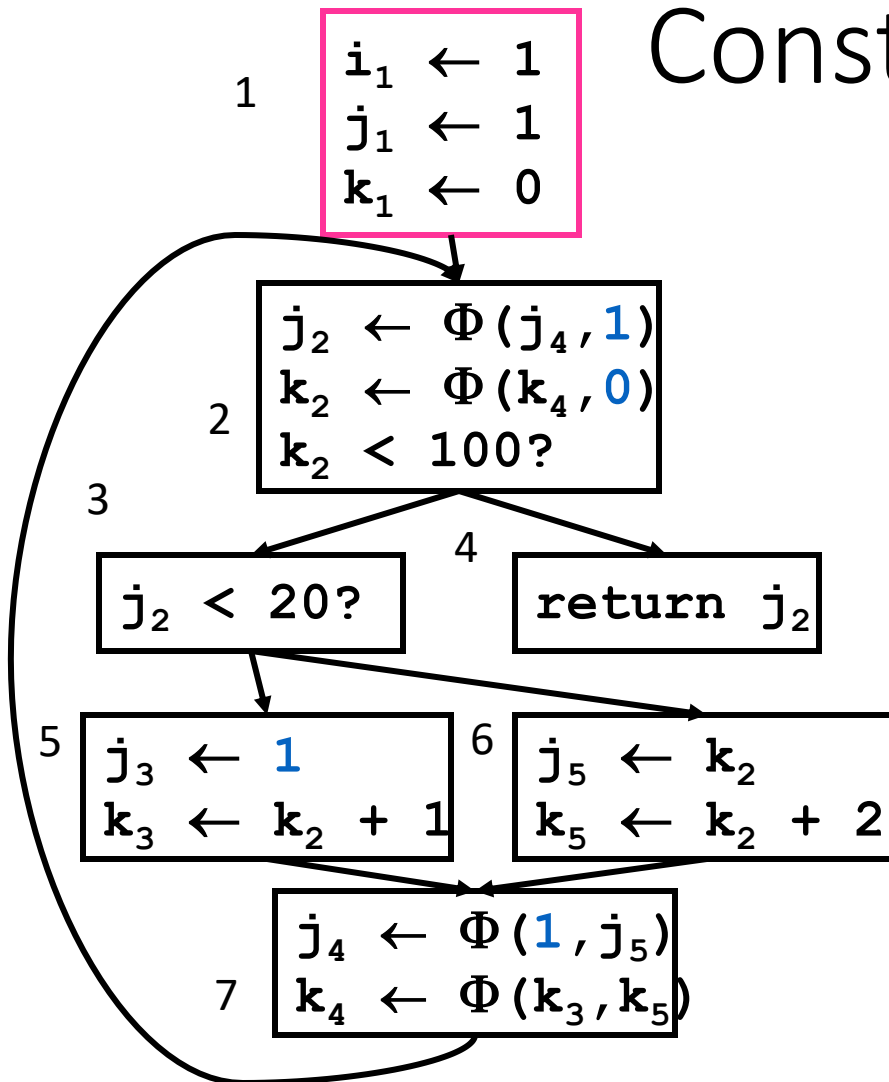
Non molto esaltante (per ora...)

# Conditional Constant Propagation

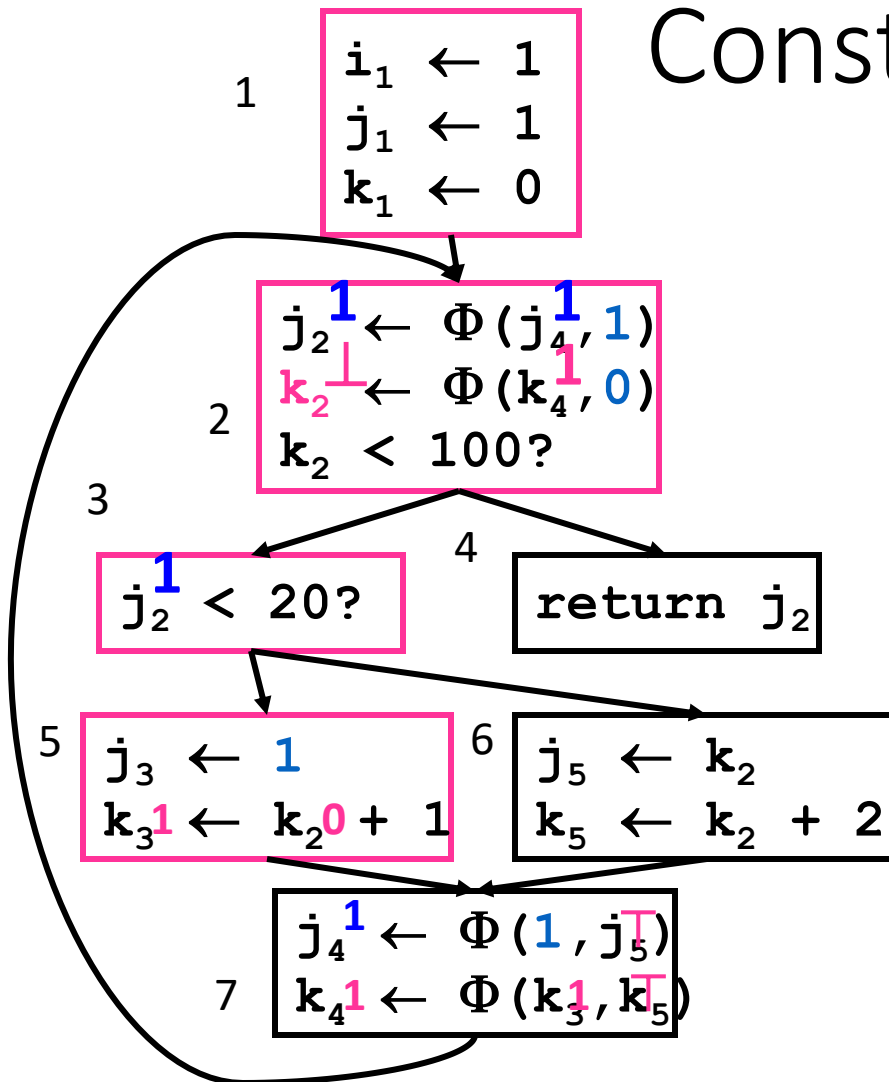


- Il blocco 6 esegue mai?
- La semplice CP non ce lo sa dire
- Ma la Conditional CP sì:
  - Assume che i blocchi non eseguano finché non si dimostra il contrario
  - Assume che i valori siano costanti finché non si dimostra il contrario

# Conditional Constant Propagation



# Conditional Constant Propagation



# Conditional Constant Propagation

