UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

7. Dataflow analysis

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

Copyright note

Credits

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

Outline

1. Struttura della *Data Flow Analysis*
2. Esempio 1: *Reaching definitions*
3. Esempio 2: *Liveness analysis*
4. Generalizzazione
5. Esempio 3: *Available Expressions*

Cos’è la *Data Flow Analysis*?

- **Analisi locale (es., Local Value Numbering)**
  - Analizza l’effetto di ogni istruzione
  - Compone l’effetto delle istruzioni per derivare informazione dall’inizio del *basic block* ad ogni istruzione

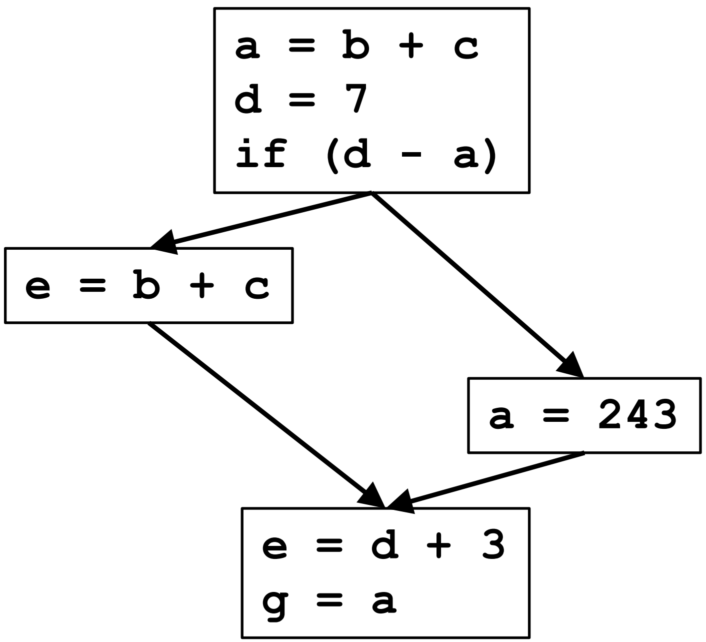
- **Analisi globale - Data flow analysis**
  - Analizza l’effetto di ogni *basic block*
  - Compone l’effetto dei *basic blocks* per derivare informazione ai confini (inizio, fine) dei *basic blocks*
  - Dai confine dei *basic blocks* si possono applicare tecniche locali per ragionare (e generare informazione) sulle istruzioni

Cos’è la *Data Flow Analysis*? (2)

- **Data flow analysis:**
  - Sensibile al flusso di controllo in una funzione
  - Analisi intraprocedurale (singola funzione, singolo CFG)

- **Esempi di ottimizzazione:**
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

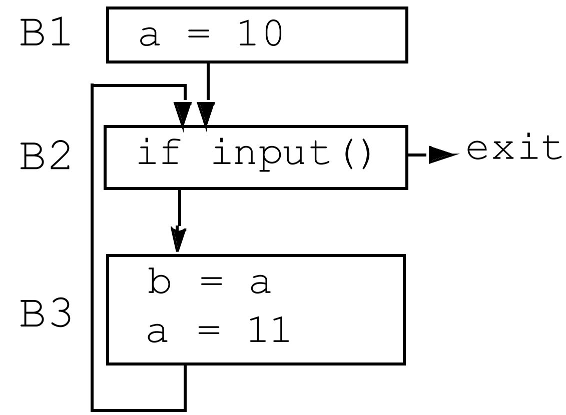
Cos’è la *Data Flow Analysis*? (3)



Per ogni variabile *x* consente di derivare informazione come:

- Valore di *x*?
- Quale “definizione” definisce *x*?
- La definizione è ancora valida (*live*)?

Rappresentazione del Programma Statica o Dinamica



- **Rappresentazione statica**: Un programma finito, un pezzo di codice
- **Rappresentazione dinamica**: Può avere infiniti percorsi di esecuzione
- Data flow analysis:
  - Per ogni punto del programma:
    combina informazioni relative a tutte le possibili istanze dello stesso punto.
- Esempio di problema DFA:
  - Quale definizione definisce il valore usato nello *statement* “b = a”?

Effetti di un *basic block*

- Effetti di un'istruzione (*statement*): **a = b + c**
    - **Usa** (**Use**s) delle variabili (**b**, **c**)
    - **Uccide** (**Kill**s) una precedente definizione (**a**)
    - **Definisce** (**Def**ines) una variabile (**a**)
- Componendo gli effetti delle singole istruzioni si definiscono gli effetti di un *basic block*
    - Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
    - Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
    - Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

Effetti di un *basic block*

```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Usi localmente esposti:

Definizioni uccise:

Definizioni localmente disponibili:

- Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
- Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
- Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

Effetti di un *basic block*

```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Usi localmente esposti:

Definizioni uccise:

Definizioni localmente disponibili:

- Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
- Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
- Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

Effetti di un *basic block*

```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Usi localmente esposti: r1, r2

Definizioni uccise:

Definizioni localmente disponibili:

- Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
- Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
- Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

Effetti di un *basic block*

```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Usi localmente esposti:

Definizioni uccise:

Definizioni localmente disponibili:

- Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
- Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
- Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

Effetti di un *basic block*

```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Usi localmente esposti:

Definizioni uccise: r2

Definizioni localmente disponibili:

- Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
- Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
- Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

Effetti di un *basic block*

```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Usi localmente esposti:

Definizioni uccise:

Definizioni localmente disponibili: ?

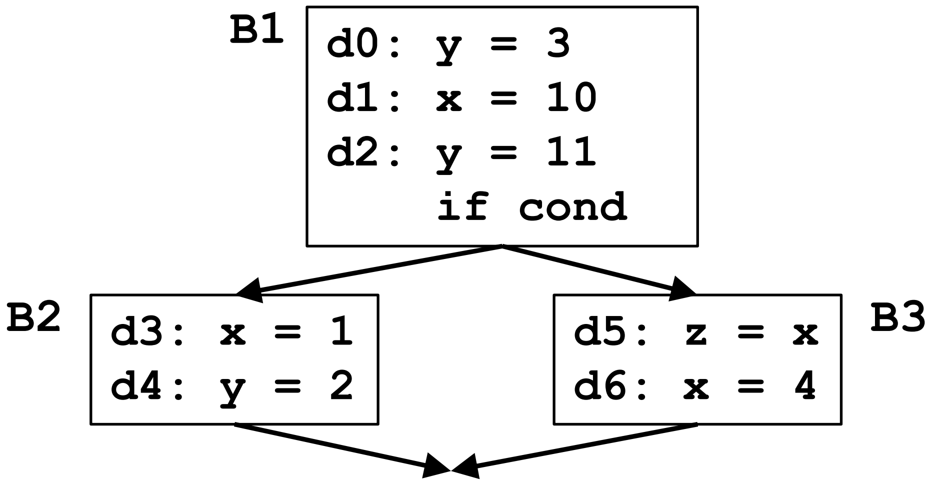
- Un **uso localmente esposto** (**locally exposed use**) in un BB è un uso di una variabile che non è preceduto nel BB da una definizione della stessa variabile
- Ogni definizione di una variabile nel BB uccide (**kills**) tutte le definizioni della stessa variabile che raggiungono il BB.
- Una **definizione localmente disponibile** (**locally available definition**) è l'ultima definizione di una variabile nel BB.

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

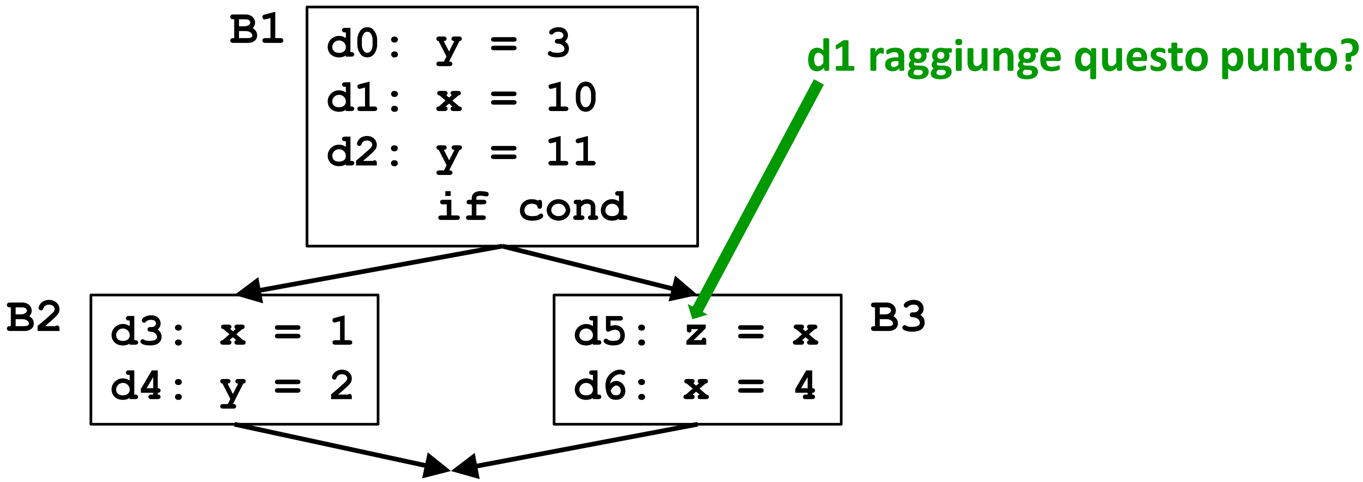
Reaching definitions

Reaching Definitions



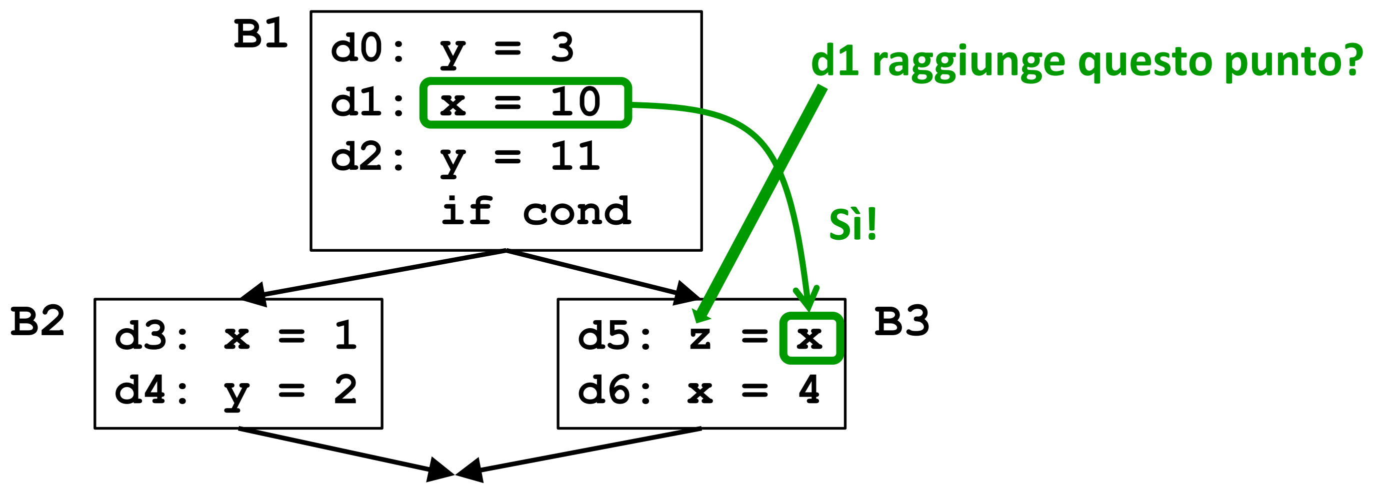
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions



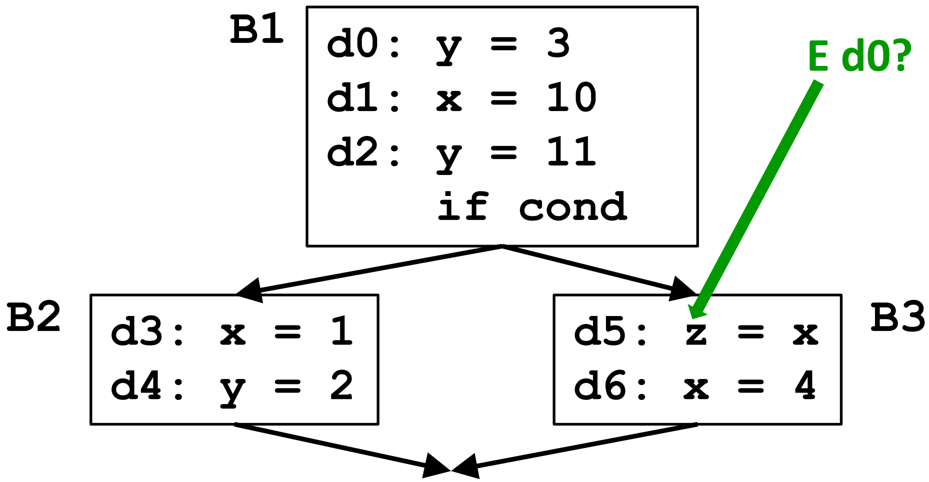
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions



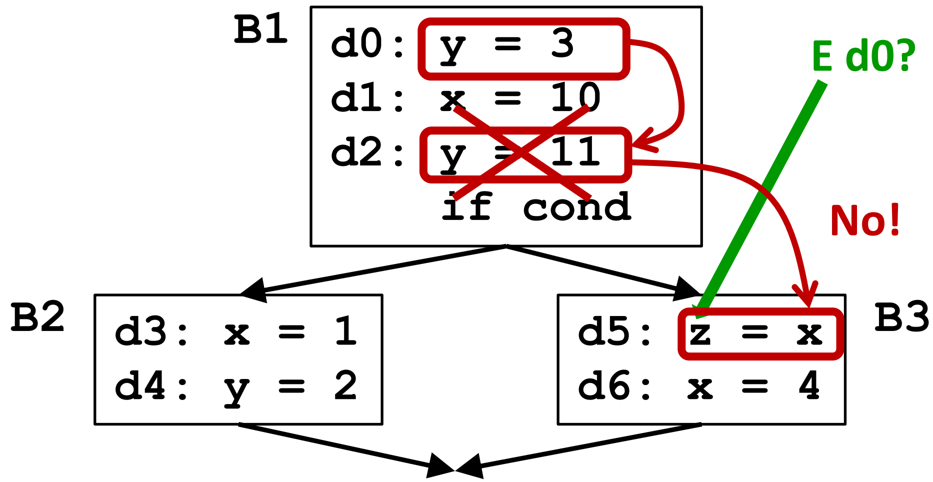
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions



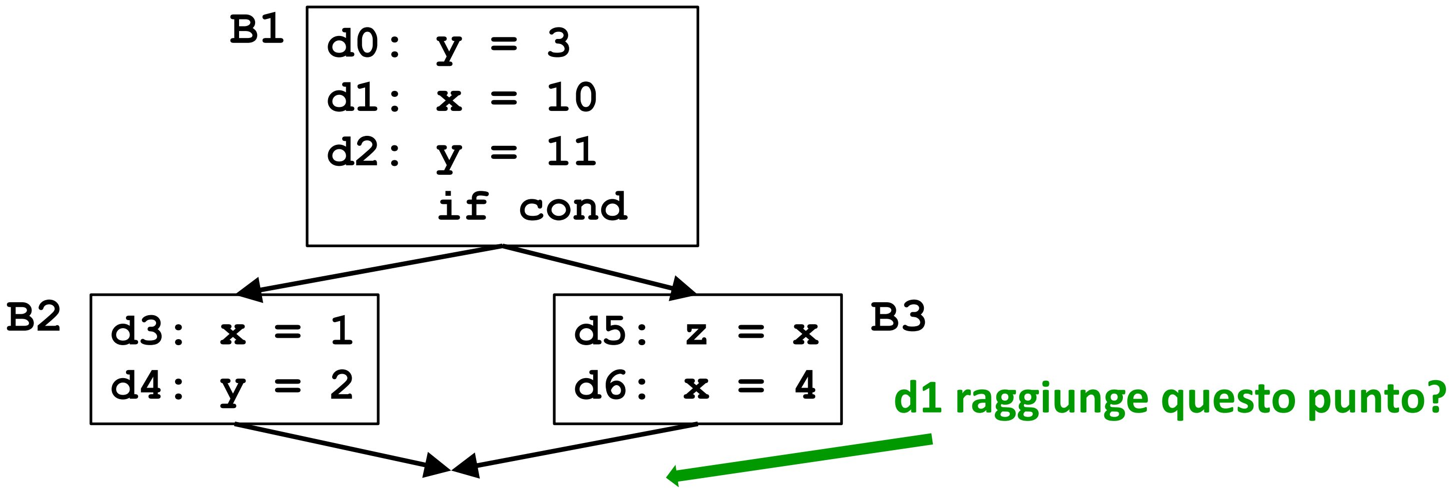
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions



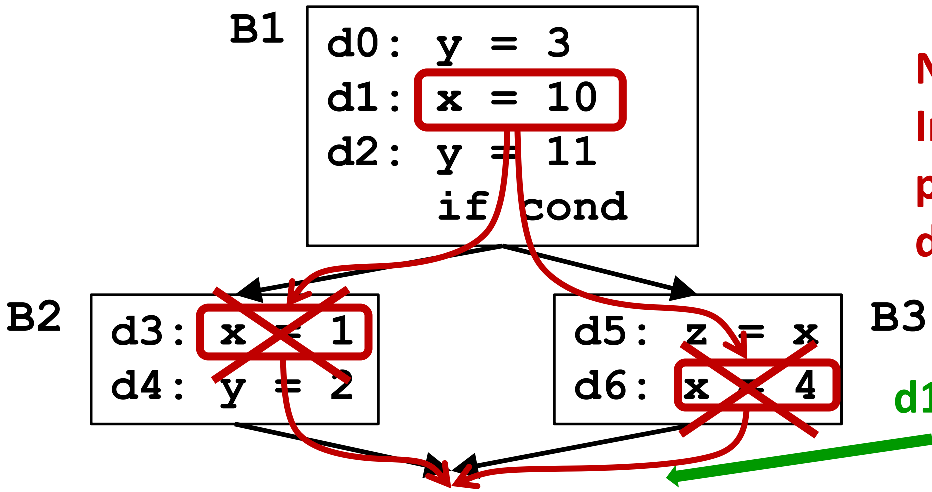
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions



- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions

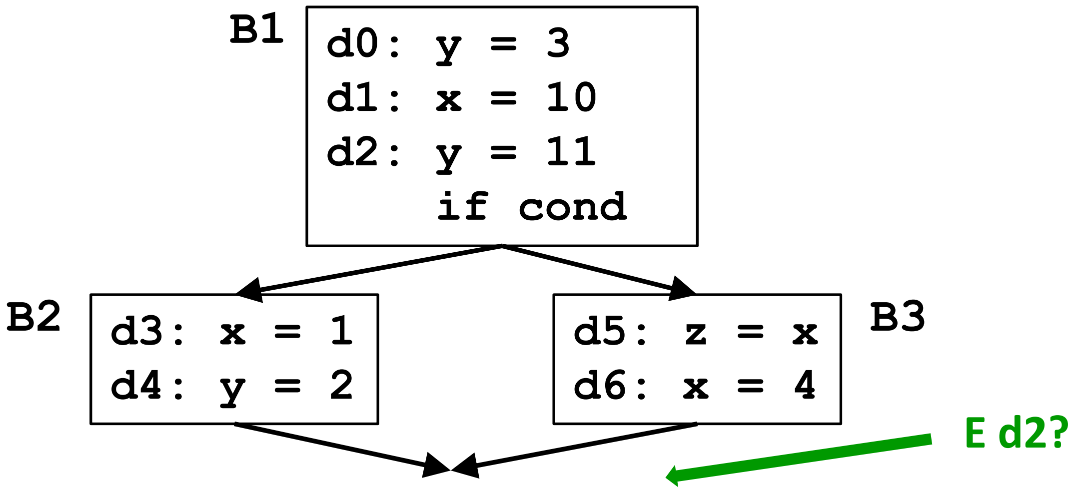


No!
Indipendentemente dal percorso preso la definizione è uccisa.

d1 raggiunge questo punto?

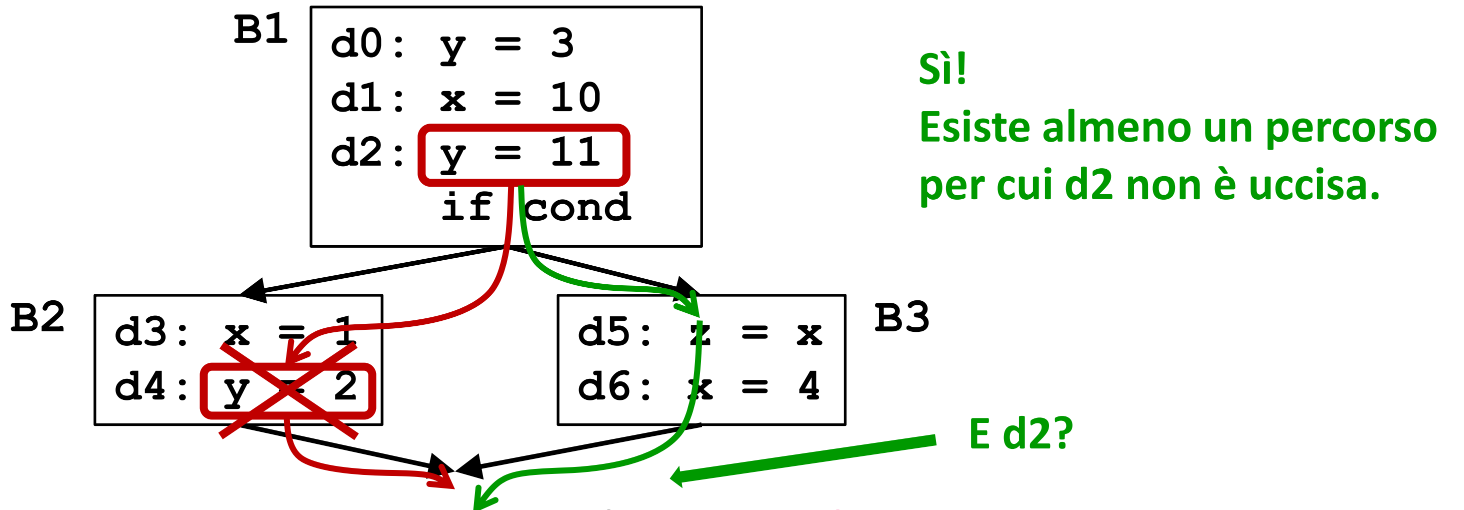
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions



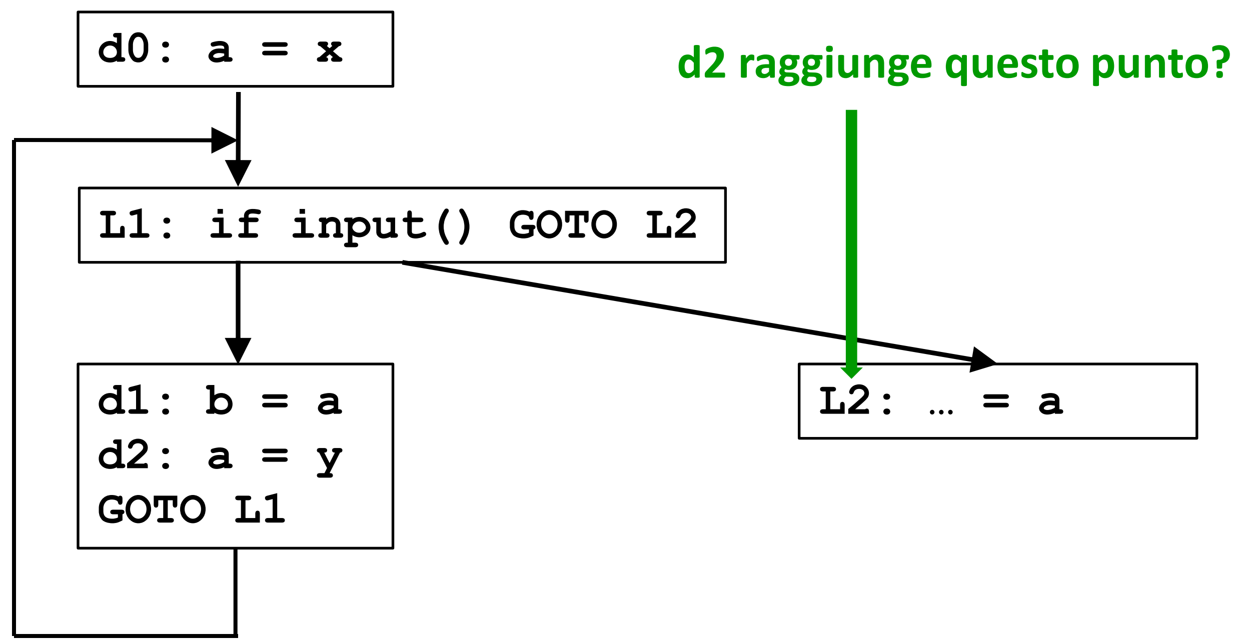
- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions

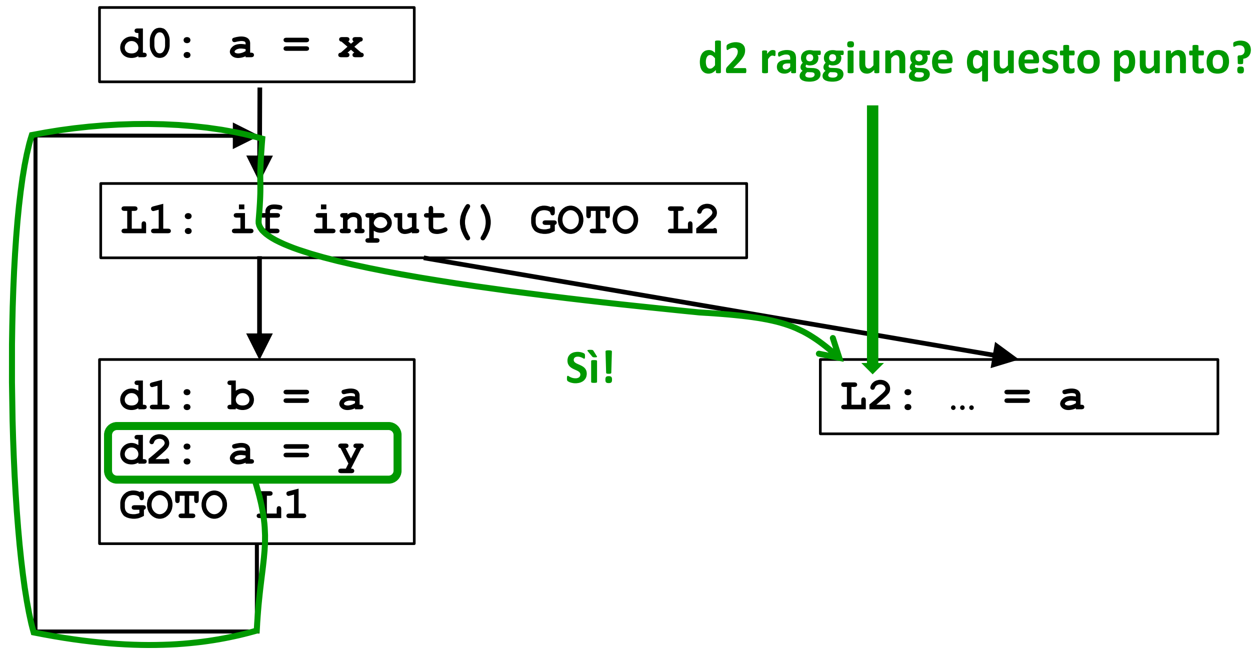


- Ogni istruzione di assegnamento è una **definizione**
- Una **definizione** *d* **raggiunge (reaches)** un punto *p* se **esiste** un percorso da *d* a *p* tale per cui *d* non è uccisa (killed) (sovrascritta) lungo quel percorso.
- Definizione del problema
  - **Per ogni punto nel programma determinare se ogni definizione nel programma raggiunge quel punto**
  - Un *bit vector* per ogni punto del programma (istruzione)
  - La lunghezza del vettore è pari al numero di definizioni

Reaching Definitions

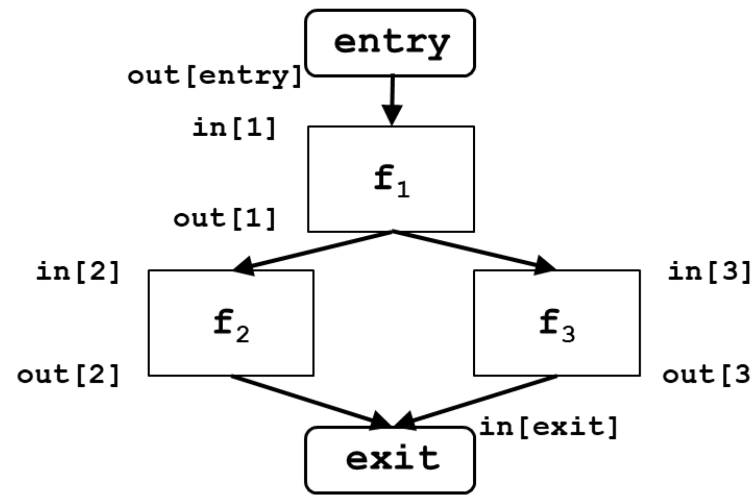


Reaching Definitions

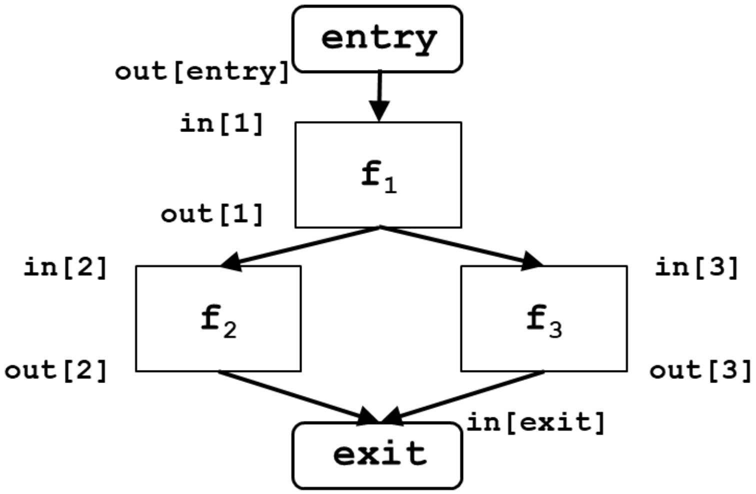


Schema della Data Flow Analysis

- Consideriamo un *flow graph*
- Aggiungiamo un entry BB e un exit BB
    - Single-entry, single-exit
    - Sempre possibile

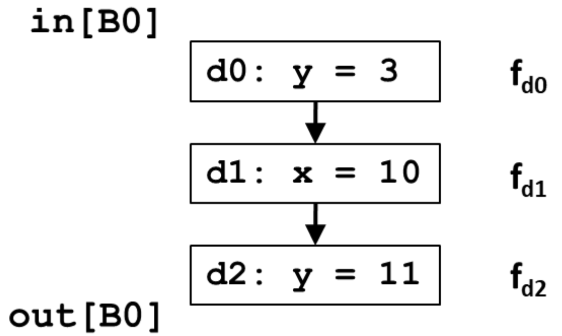


Schema della Data Flow Analysis



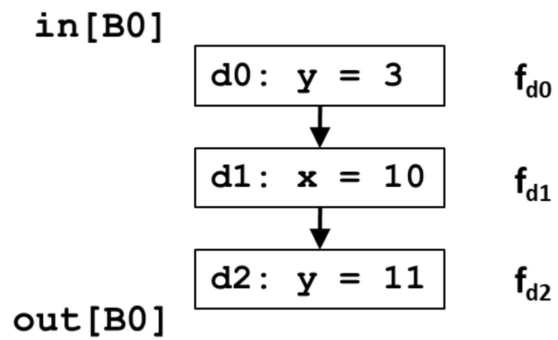
- Consideriamo un *flow graph*
- Definiamo un insieme di equazioni tra *in[b]* e *out[b]* per tutti i *basic blocks b*
  - Qual è l'effetto del codice nei *basic blocks*?
    - La funzione di trasferimento f_b correla *in[b]* e *out[b]* per un dato *b*
  - Qual è l'effetto del flusso di controllo?
    - correla *out[b1], in[b2]* se *b1* e *b2* sono adiacenti
- Risolviamo le equazioni

Effetti di uno *Statement*



- f_s : La funzione di trasferimento di uno *statement*
  - Astrae l'esecuzione rispetto al problema di interesse
- Per uno *statement* s (d: x = y + z)
  out[s] = f_s(in[s]) = Gen[s] U (in[s]-Kill[s])
  - **Gen[s]**: definizioni generate: Gen[s] = {d}
  - Definizioni **Propagate**: in[s] - Kill[s],
    dove **Kill[s]** = altre definizioni di x nel resto del programma

Effetti di uno *Statement*



$f_B = f_{d2} \cdot f_{d1} \cdot f_{d0}$

- Funzione di trasferimento di uno *statement* s:
  - out[s] = f_s(in[s]) = Gen[s] U (in[s]-Kill[s])
- Funzione di trasferimento di un basic block *B*:
  - Composizione di funzioni di traserimento degli *statements* in *B*
- out[B] = f_B(in[B]) = $f_{d2}f_{d1}f_{d0}$(in[B])

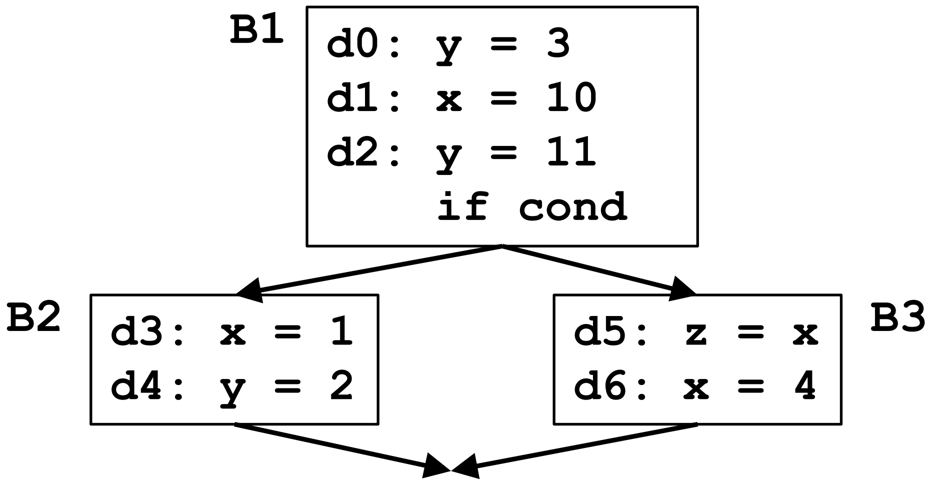
  = Gen[d_2] U (Gen[d_1] U (Gen[d_0] U (in[B]-Kill[d_0]))-Kill[d_1])) -Kill[d_2]

  = Gen[d_2] U (Gen[d_1] U (Gen[d_0] - Kill[d_1]) - Kill[d_2]) U
  in[B] - (Kill[d_0] U Kill[d_1] U Kill[d_2])

  = Gen[B] U (in[B] - Kill[B])

  - Gen[B]: definizioni localmente disponibili (alla fine del bb)
  - Kill[B]: insieme delle definizioni (in tutto il programma) uccise da *B*

Esempio

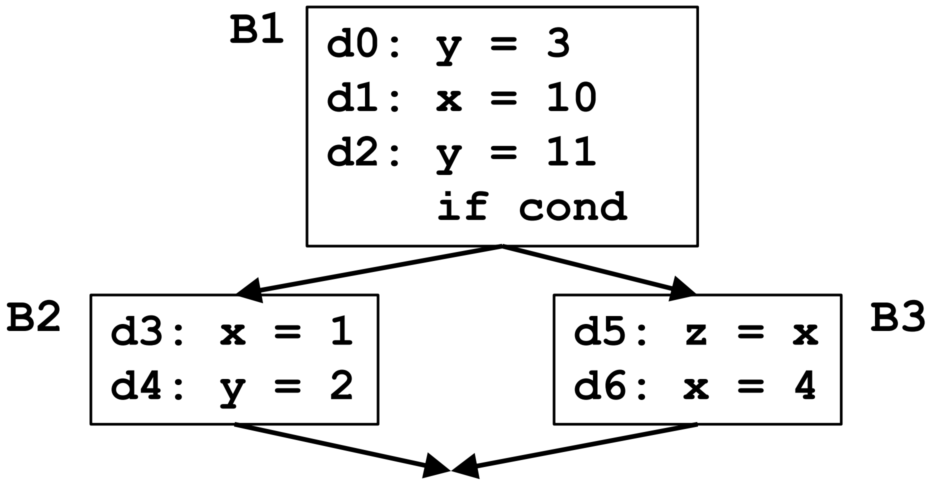


- una **funzione di trasferimento** f_b di un *basic block b*:

  $$OUT[b] = f_b(IN[b])$$

  incoming reaching definitions -> outgoing reaching definitions
- Un *basic block b*
  - **genera** definizioni: **Gen[b]**
    - L'insieme delle definizioni localmente disponibili in *b*
  - **Uccide (kills)** definizioni: **in[b] - Kill[b]**, dove **Kill[b]** = definizioni (nel resto del programma) uccise dalle definizioni in *b*
- **out[b] = Gen[b] U (in(b)-Kill[b])**

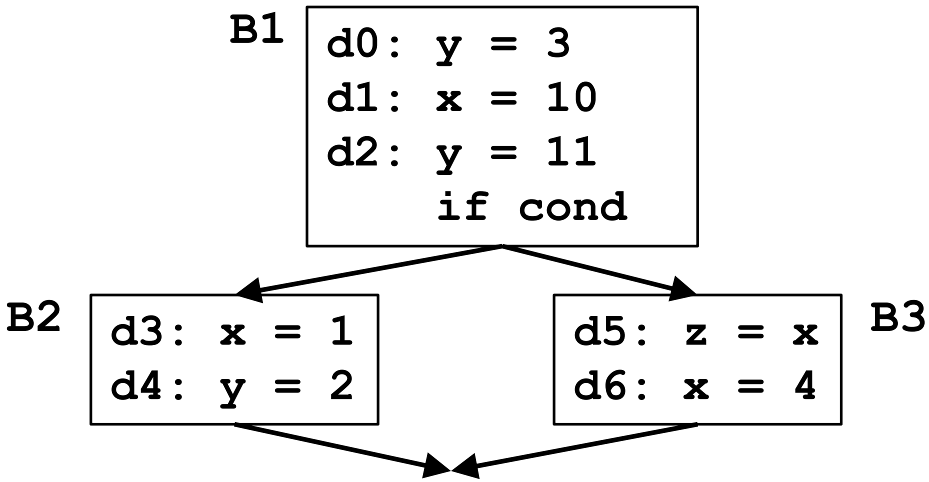
Esempio



- Proviamo a calcolare **Gen[b_i]** e **Kill[b_i]**

	Gen	Kill
B_1		
B_2		
B_3		

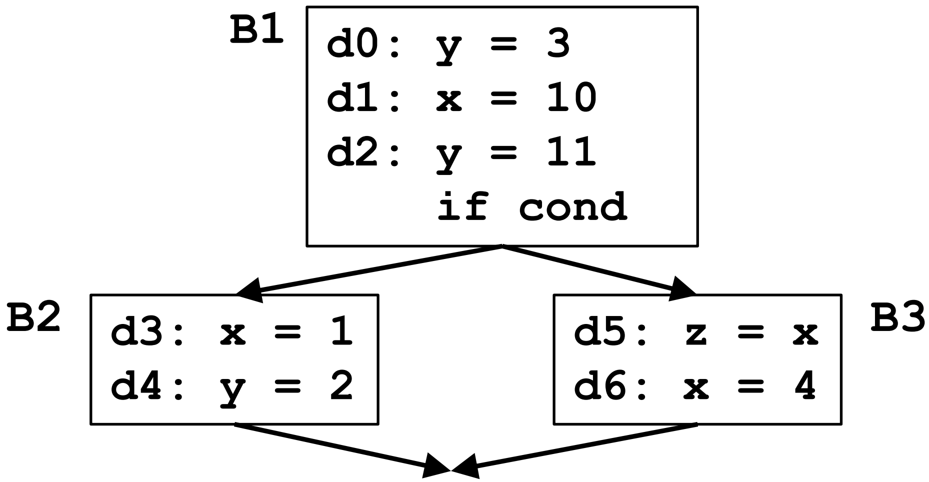
Esempio



- Proviamo a calcolare **Gen[b_i]** e **Kill[b_i]**

	Gen	Kill
B_1	1, 2	
B_2		
B_3		

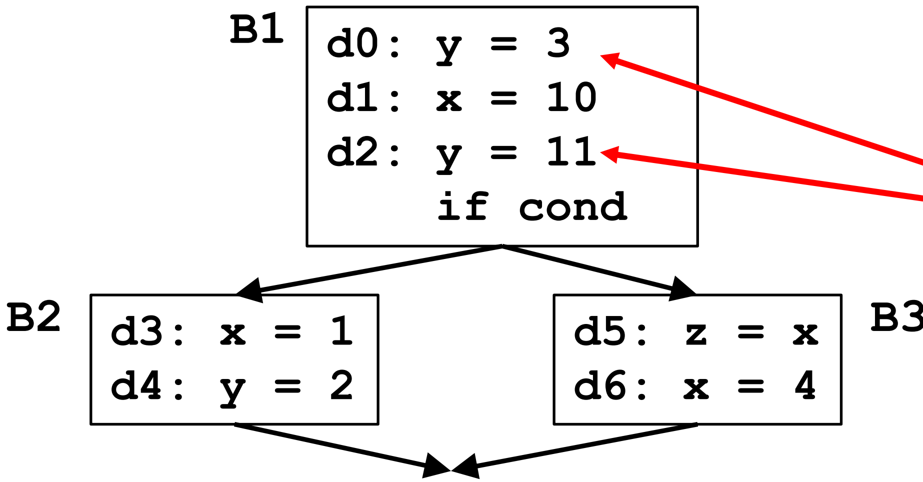
Esempio



- Proviamo a calcolare **Gen[*b_i*]** e **Kill[*b_i*]**

	Gen	Kill
B_1	1, 2	0, 2, 3, 4, 6
B_2		
B_3		

Esempio

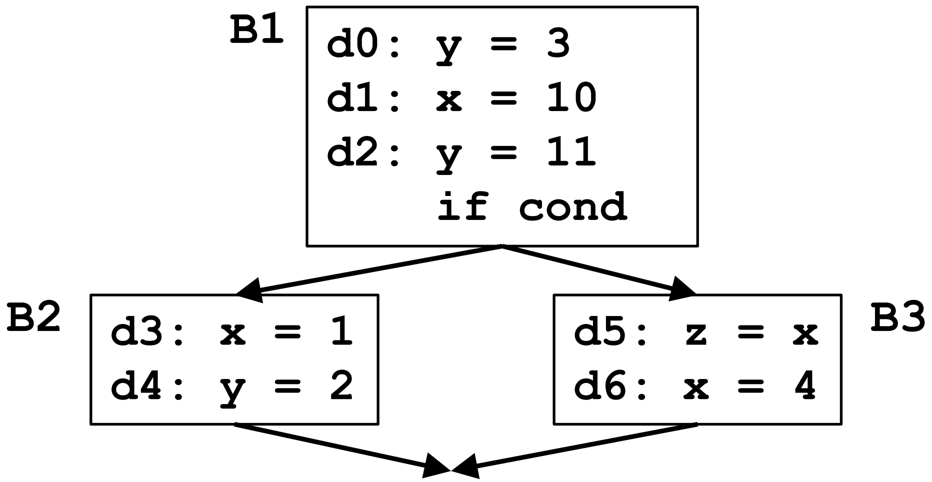


d0 e d2 si uccidono a vicenda (approccio *bit-vector*, non ha nozione di ordine di esecuzione)

- Proviamo a calcolare **Gen[b_i]** e **Kill[b_i]**

	Gen	Kill
B_1	1, 2	0, 2, 3, 4, 6
B_2		
B_3		

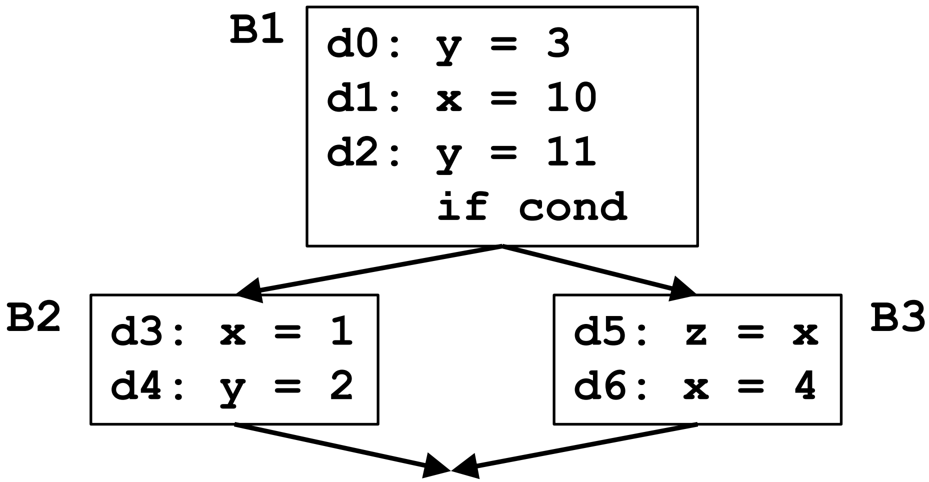
Esempio



- Proviamo a calcolare **Gen[*b_i*]** e **Kill[*b_i*]**

	Gen	Kill
B_1	1, 2	0, 2, 3, 4, 6
B_2	3, 4	
B_3		

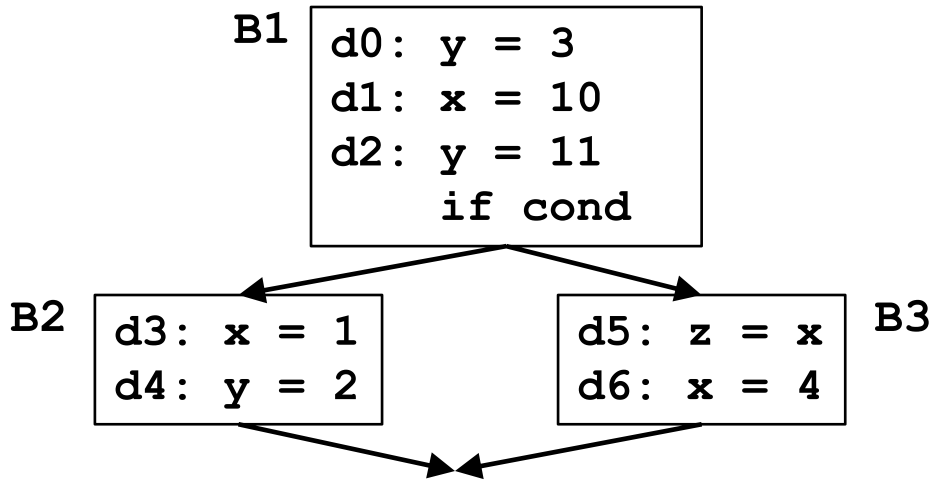
Esempio



- Proviamo a calcolare **Gen[b_i]** e **Kill[b_i]**

	Gen	Kill
B_1	1, 2	0, 2, 3, 4, 6
B_2	3, 4	0, 1, 2, 6
B_3		

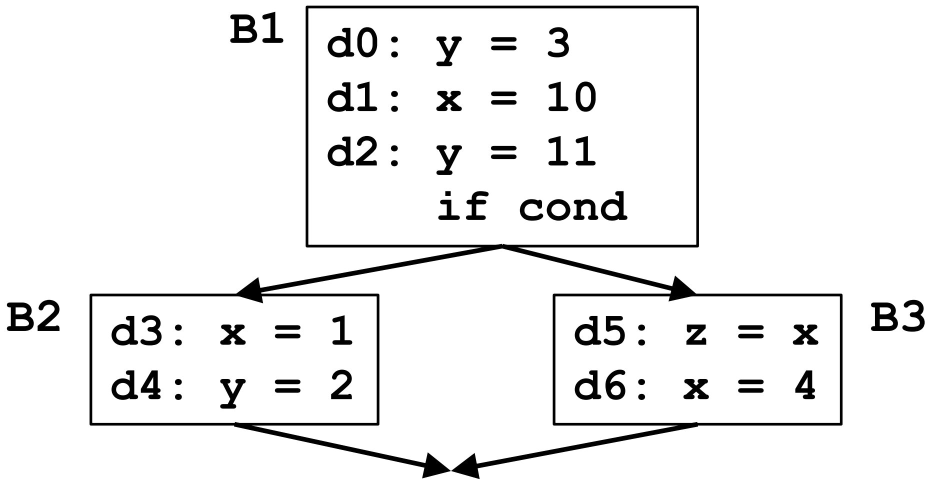
Esempio



- Proviamo a calcolare **Gen[*b*$_i$]** e **Kill[*b*$_i$]**

	Gen	Kill
B$_1$	1, 2	0, 2, 3, 4, 6
B$_2$	3, 4	0, 1, 2, 6
B$_3$	5, 6	

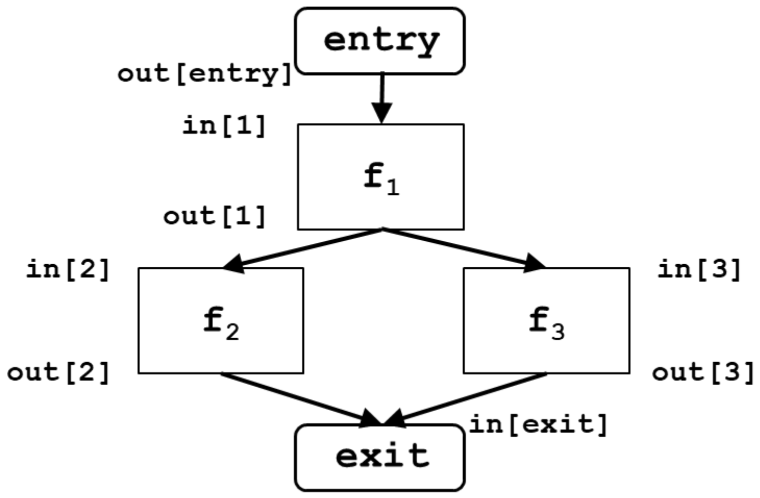
Esempio



- Proviamo a calcolare **Gen[b_i]** e **Kill[b_i]**

	Gen	Kill
B_1	1, 2	0, 2, 3, 4, 6
B_2	3, 4	0, 1, 2, 6
B_3	5, 6	1, 3

Effetti degli archi (aciclici)

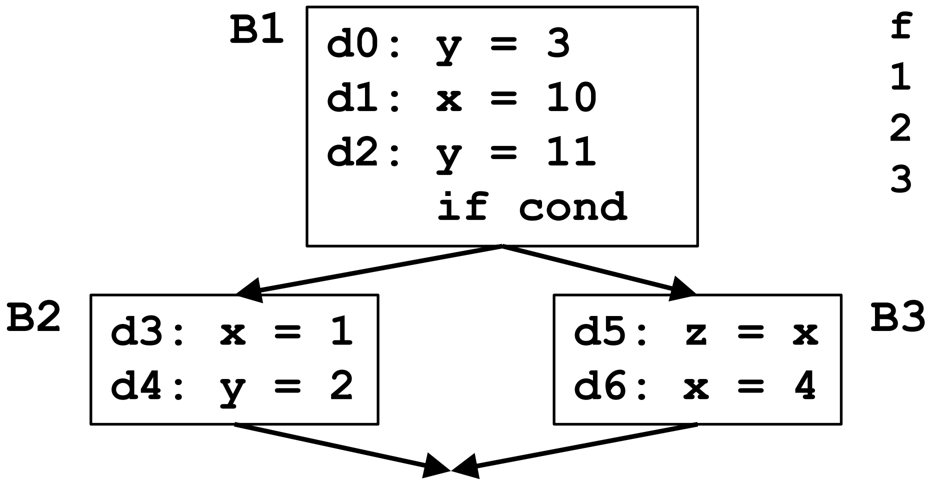


- $out[b] = f_b(in[b])$
- Nodo di unione (**join**): un nodo con **predecessori** multipli
- Operatore di unione (**meet**) :

  $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$, dove

  $p_1, \ldots, p_n$ sono tutti **predecessori** di *b*

Esempio



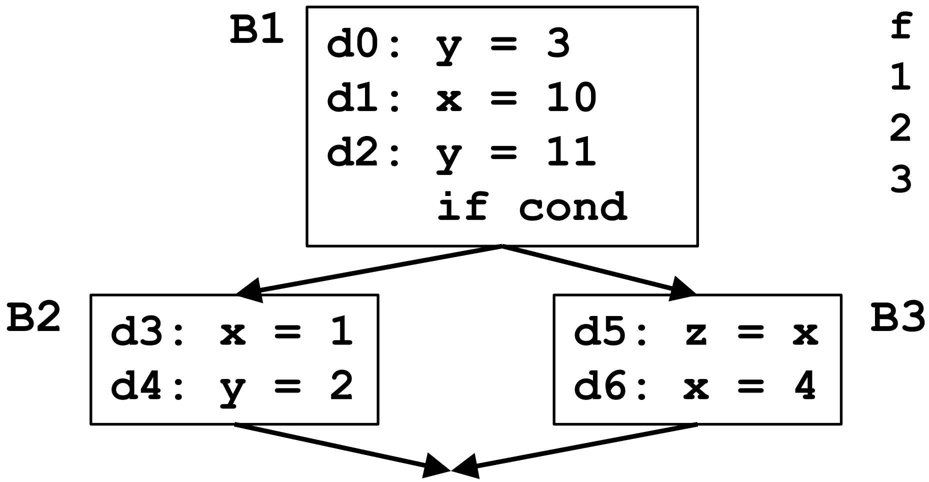
f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

- $out[b] = f_b(in[b])$
- Nodo di unione (**join**): un nodo con **predecessori** multipli
- Operatore di unione (**meet**) :

  $in[b] = out[p_1] \cup out[p_2] \cup \ldots \cup out[p_n]$, dove

  $p_1, \ldots, p_n$ sono tutti **predecessori** di *b*

Esempio



f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

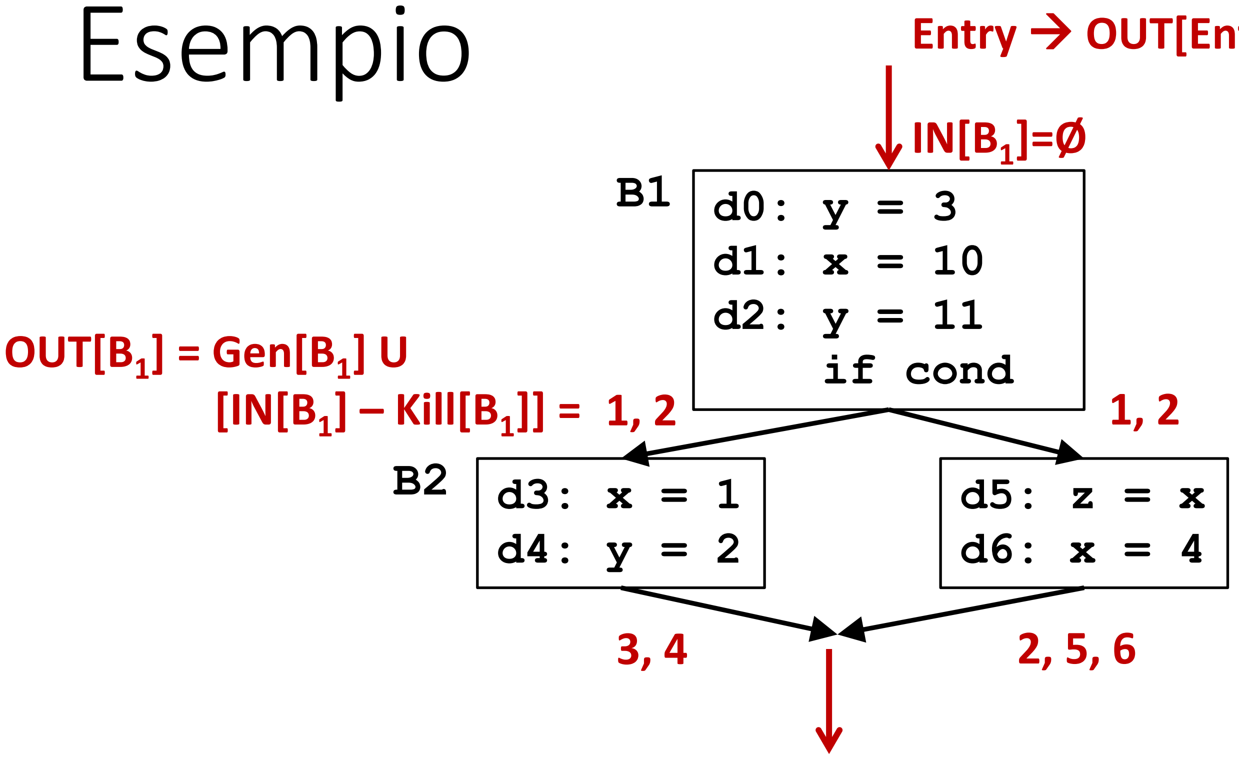
Cosa c'è in ingresso e in uscita ad ogni nodo?

- out[b] = f_b(in[b])
- Nodo di unione (**join**): un nodo con **predecessori** multipli
- Operatore di unione (**meet**) :

  in[b] = out[p_1] U out[p_2] U ... U out[p_n], dove

  p_1, ..., p_n sono tutti **predecessori** di *b*

Esempio



f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

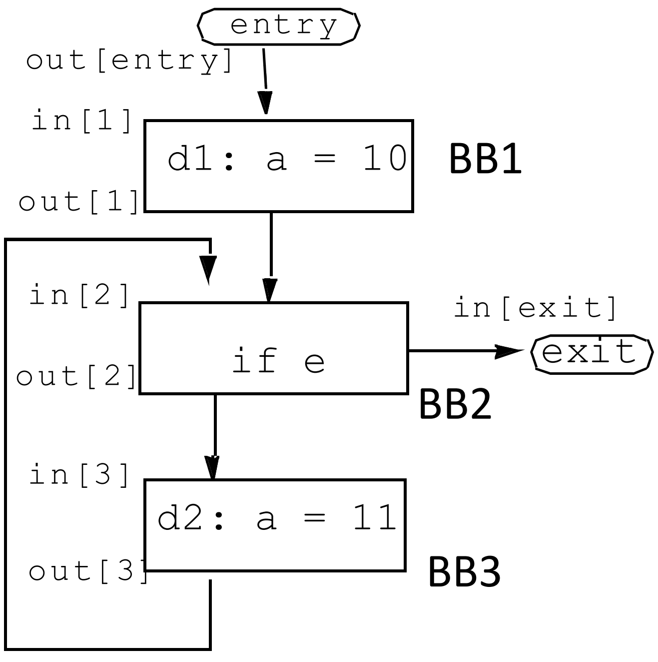
Exit → IN[EXIT] = OUT[B2] U OUT[B3] = 2, 3, 4, 5, 6

- out[b] = f_b(in[b])
- Nodo di unione (**join**): un nodo con **predecessori** multipli
- Operatore di unione (**meet**) :

  in[b] = out[p_1] U out[p_2] U ... U out[p_n], dove

  p_1, ..., p_n sono tutti **predecessori** di *b*

Grafi ciclici



- Le equazioni valgono ancora
  - out[b] = f_b(in[b])
  - in[b] = out[p_1] U out[p_2] U ... U out[p_n], p_1, ..., p_n pred.
- I backedges possono cambiare le equazioni **out[b]**
  - Iteriamo fino a convergenza

Reaching Definitions: Algoritmo iterativo

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
   out[Entry] = ∅

// Initialization for iterative algorithm
   for each basic block B other than Entry
      out[B] = ∅

// iterate
   while (changes to any out[] occur) {
      for each basic block B other than Entry {
         in[B] = ∪ (out[p]), for all predecessors p of B
         out[B] = fB(in[B])    // out[B]=gen[B]∪(in[B]-kill[B])
      }
```

Reaching Definitions: Algoritmo *Worklist*

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
    out[Entry] = ∅          // can set out[Entry] to special def
                            // if reaching then undefined use
    For all nodes i
        out[i] = ∅          // can optimize by out[i]=gen[i]
    ChangedNodes = N

// iterate
    While ChangedNodes ≠ ∅ {
        Remove i from ChangedNodes
        in[i] = U (out[p]), for all predecessors p of i
        oldout = out[i]
        out[i] = f_i(in[i])    // out[i]=gen[i]U(in[i]-kill[i])
        if (oldout ≠ out[i]) {
            for all successors s of i
                add s to ChangedNodes
        }
    }
```

Una **worklist** contiene le variabili che devono ancora essere processate. Quando la worklist è vuota abbiamo finito.

Esempio

- Applicare l’algoritmo fino a convergenza per il grafo d’esempio

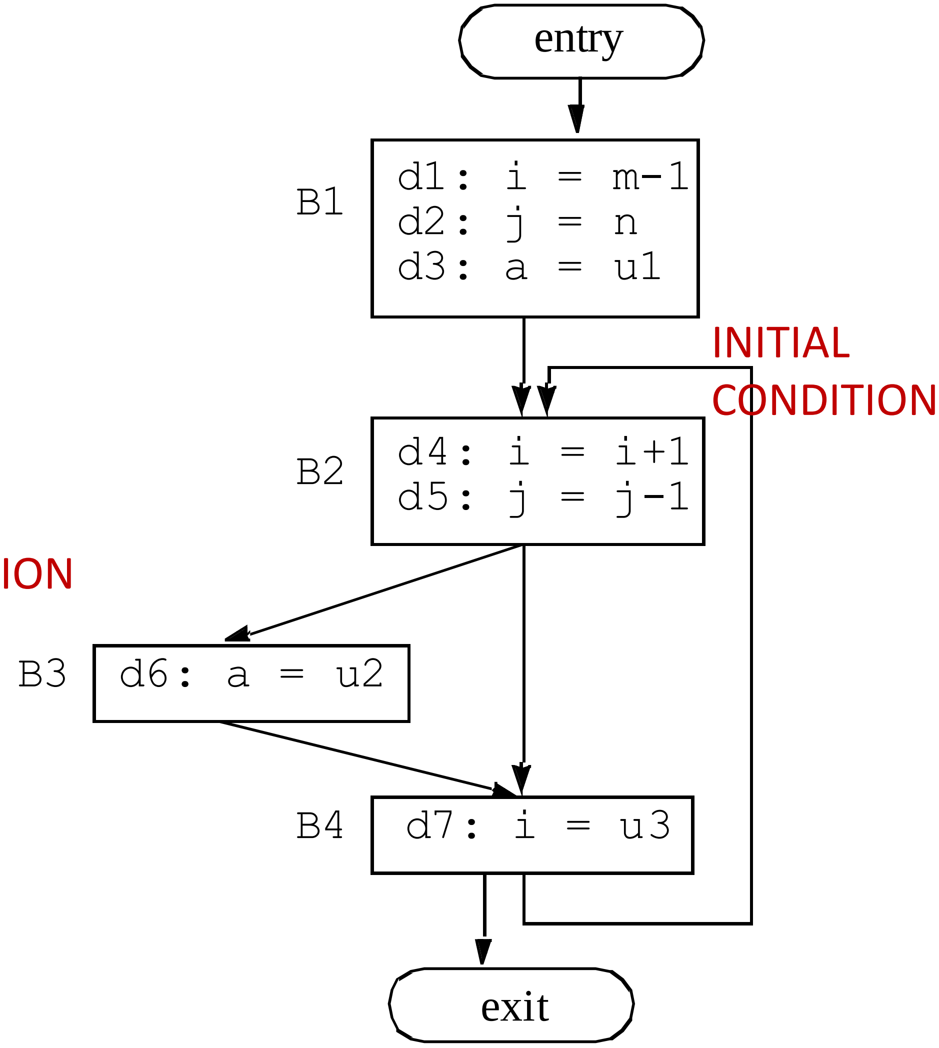
d1 d2 d3 d4 d5 d6 d7

IN[B1] = <000 00 0 0>

BOUNDARY CONDITION

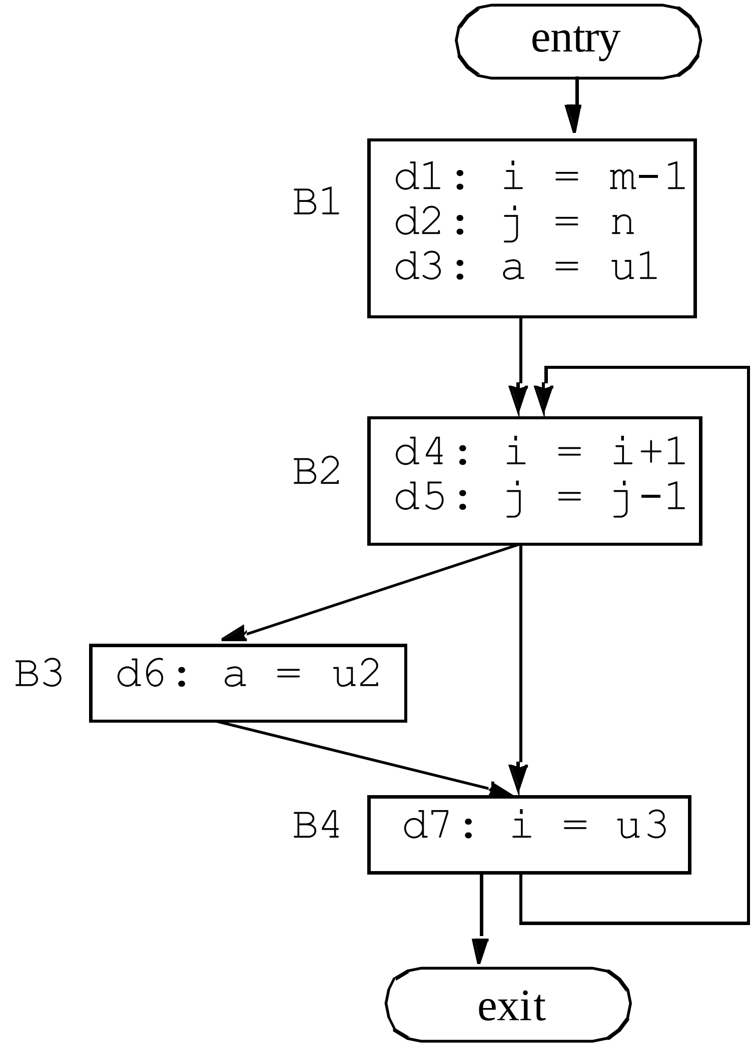
OUT[B1] = <… .. . . >

…



Esempio

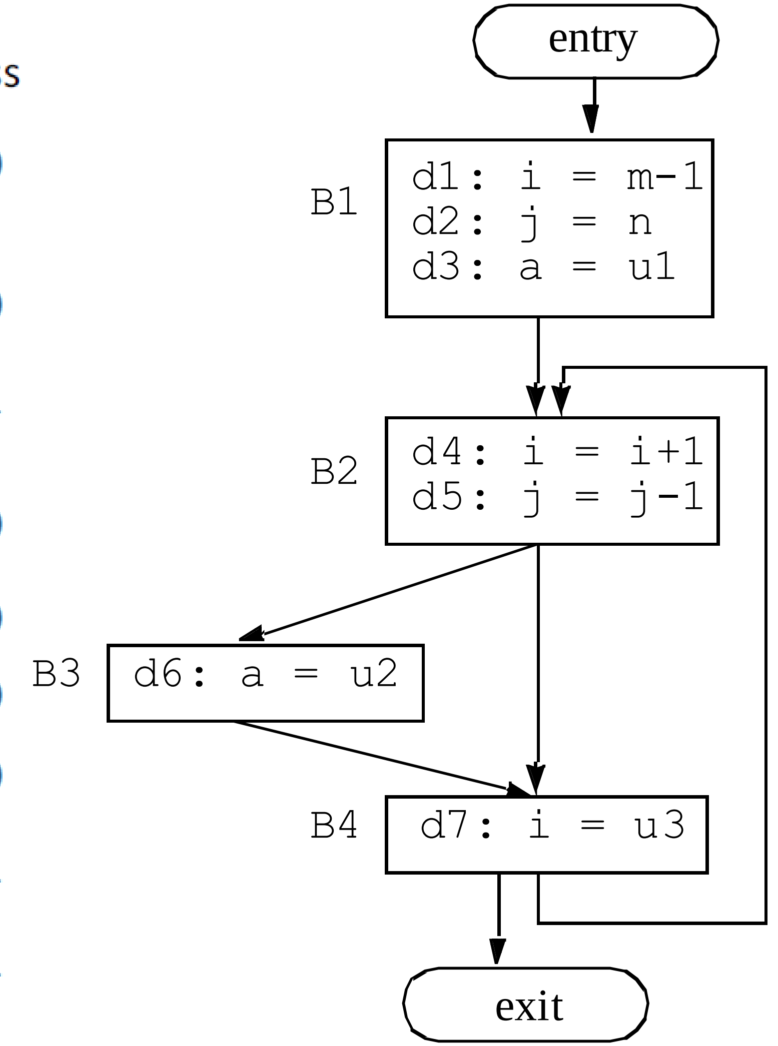
	First Pass
IN[B1]	000 00 0 0
OUT[B1]	111 00 0 0
IN[B2]	111 00 0 0
OUT[B2]	001 11 0 0
IN[B3]	001 11 0 0
OUT[B3]	000 11 1 0
IN[B4]	001 11 1 0
OUT[B4]	001 01 1 1
IN[exit]	001 01 1 1



Esempio

Dopo la seconda iterazione **out[B2]** non cambia più

	First Pass	Second Pass
IN[B1]	000 00 0 0	000 00 0 0
OUT[B1]	111 00 0 0	111 00 0 0
IN[B2]	111 00 0 0	111 01 1 1
OUT[B2]	001 11 0 0	001 11 1 0
IN[B3]	001 11 0 0	001 11 1 0
OUT[B3]	000 11 1 0	000 11 1 0
IN[B4]	001 11 1 0	001 11 1 0
OUT[B4]	001 01 1 1	001 01 1 1
IN[exit]	001 01 1 1	001 01 1 1



UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Liveness Analysis

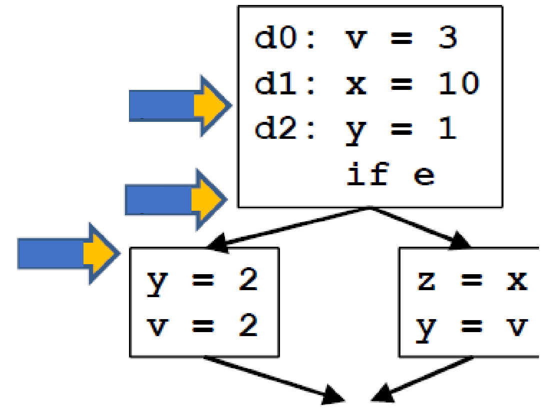
Live Variable Analysis

- **Definizione**
  - Una variabile **v** è viva (**live**) in un punto *p* del programma se
    - Il valore di **v** è usato lungo qualche percorso del flow graph a partire da *p*.
  - Altrimenti, la variabile è morta (**dead**).
- **Motivazione**
    - es., register allocation

```
for i = 0 to n
    … i …
…
for i = 0 to n
    … i …
```

Posso riusare lo stesso registro se *i* non è viva qui

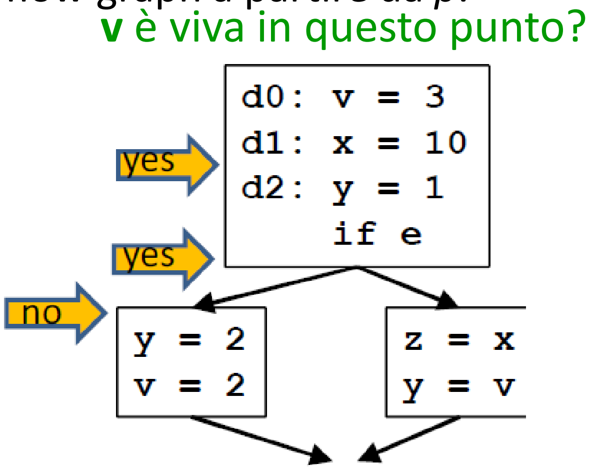
v è viva in questo punto?



- **Definizione del problema**
  - Per ogni *basic block*
    - Determinare se ogni variabile è *viva* in ogni *basic block*
  - Dimensione del bit vector: un bit per ogni variabile

Live Variable Analysis

- **Definizione**
  - Una variabile **v** è viva (**live**) in un punto *p* del programma se
    - Il valore di **v** è usato lungo qualche percorso del flow graph a partire da *p*.
  - Altrimenti, la variabile è morta (**dead**).



- **Motivazione**
    - es., register allocation

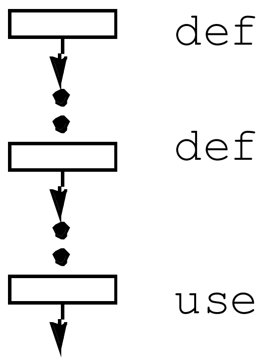
```
for i = 0 to n
    … i …
…
for i = 0 to n
    … i …
```

- **Definizione del problema**
  - Per ogni *basic block*
    - Determinare se ogni variabile è *viva* in ogni *basic block*
  - Dimensione del bit vector: un bit per ogni variabile

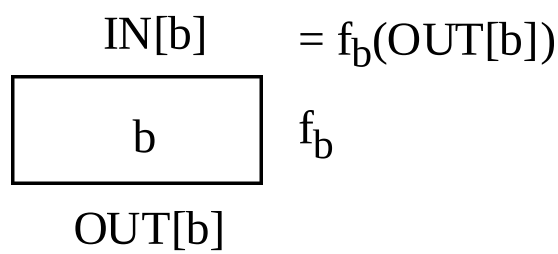
Funzione di trasferimento

- **Intuizione: Tracciamo gli usi all'indietro fino alle definizioni**

an execution path



control flow



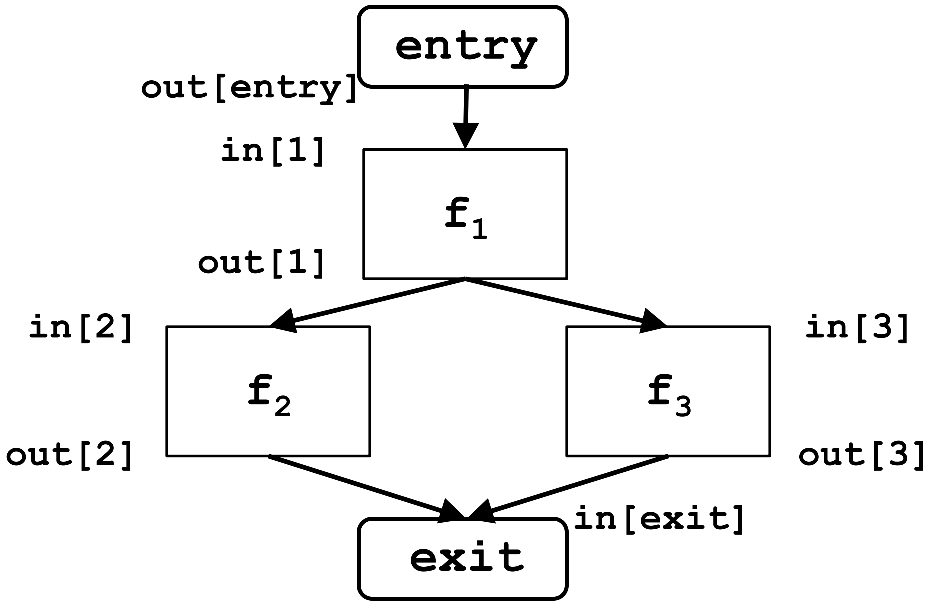
example

```
d3: a = 1
d4: b = 1

d5: c = a
d6: a = 4
```

- **Un *basic block b* può**
  - generare variabili vive: **Use[b]**
    - L'insieme degli usi localmente esposti in *b*
  - propagare variabili vive in ingresso: **OUT**[b] - **Def[b]**,
    - dove Def[b] = insieme delle variabili definite nel bb
- **Funzione di trasferimento** per il blocco *b*:

in[b] = Use[b] U (out(b)-Def[b])

Flow Graph

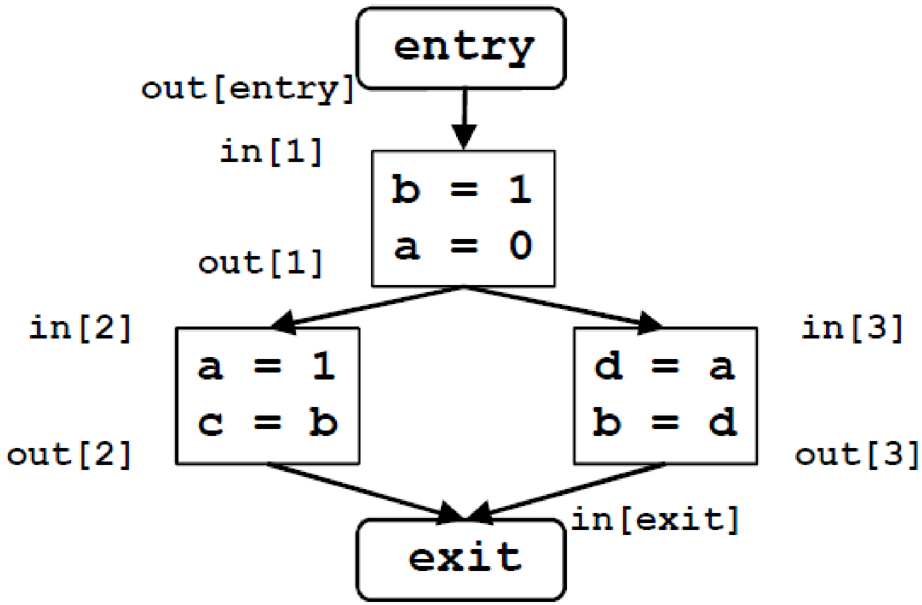


- $in[b] = f_b(out[b])$
- Join node: un nodo con successori multipli
- **meet** operator:

  out[b] = in[s_1] U in[s_2] U ... U in[s_n], dove

  s_1, ..., s_n sono tutti successor di *b*

Flow Graph



f	Use	Def
1		
2		
3		

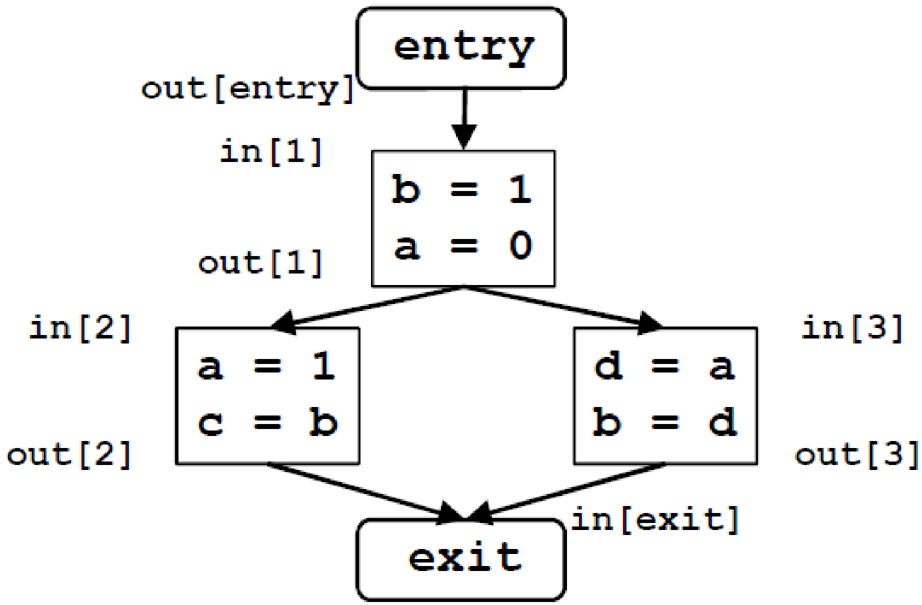
Troviamo usi e definizioni per tutti i nodi

- in[b] = f_b(out[b])
- Join node: un nodo con successori multipli
- **meet** operator:

  out[b] = in[s_1] U in[s_2] U ... U in[s_n], dove

  s_1, ..., s_n sono tutti successor di *b*

Flow Graph



f	Use	Def
1	{}	{a,b}
2	{b}	{a,c}
3	{a}	{b,d}

- in[b] = f_b(out[b])
- Join node: un nodo con successori multipli
- **meet** operator:

  out[b] = in[s_1] U in[s_2] U ... U in[s_n], dove

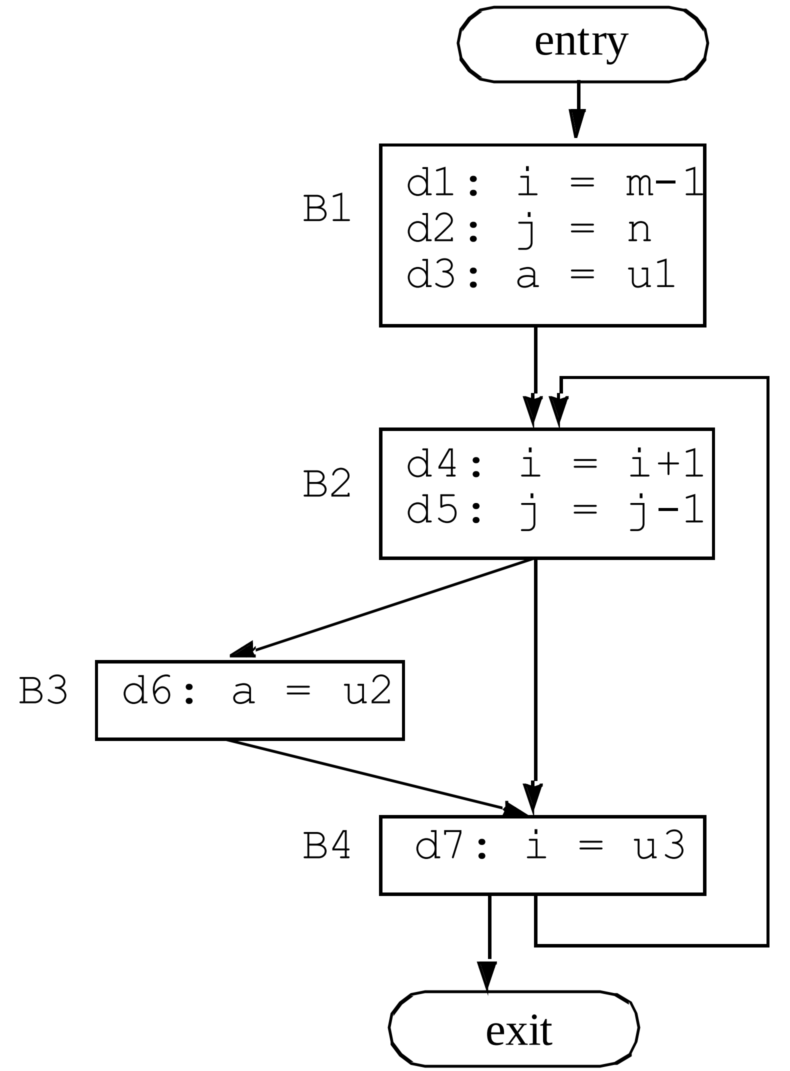
  s_1, ..., s_n sono tutti successor di *b*

Liveness: Algoritmo iterativo

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
   in[Exit] = ∅

// Initialization for iterative algorithm
   For each basic block B other than Exit
      in[B] = ∅

// iterate
   While (Changes to any in[] occur) {
      For each basic block B other than Exit {
         out[B] = ∪ (in[s]), for all successors s of B
         in[B] = f_B(out[B])    // in[B]=Use[B]∪(out[B]-Def[B])
      }
```

Esempio

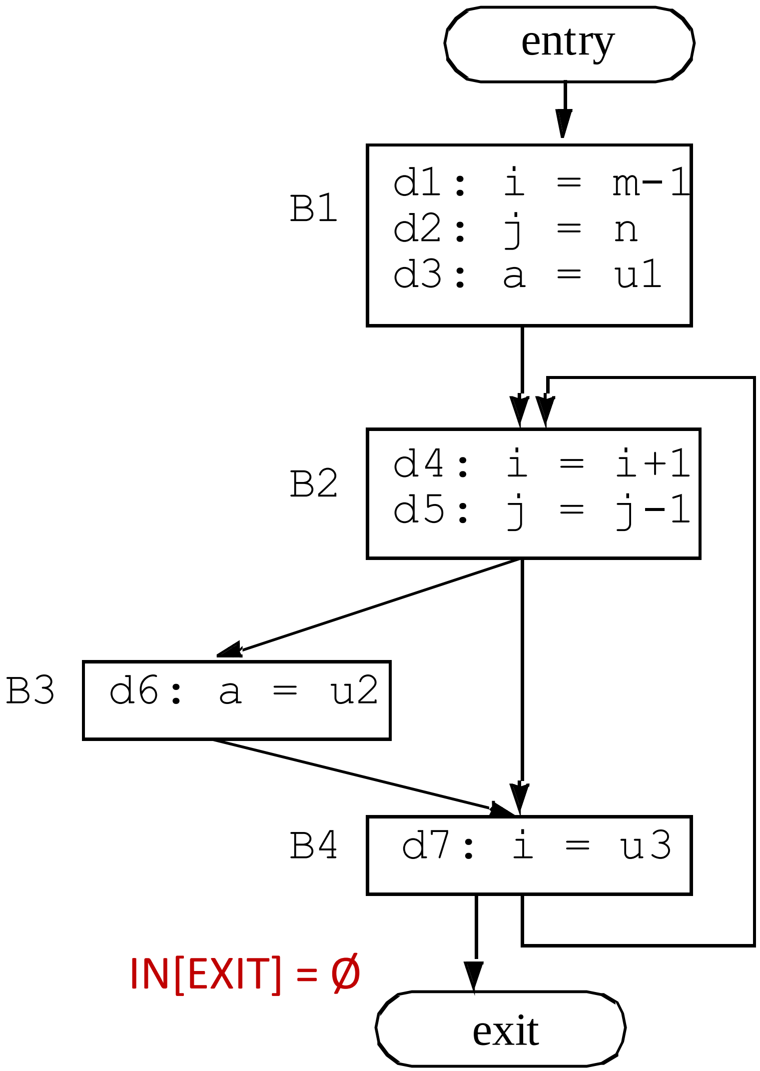
- Applicare l’algoritmo fino a convergenza per il grafo d’esempio



Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio

- Boundary condition
  IN[EXIT] = Ø



Esempio

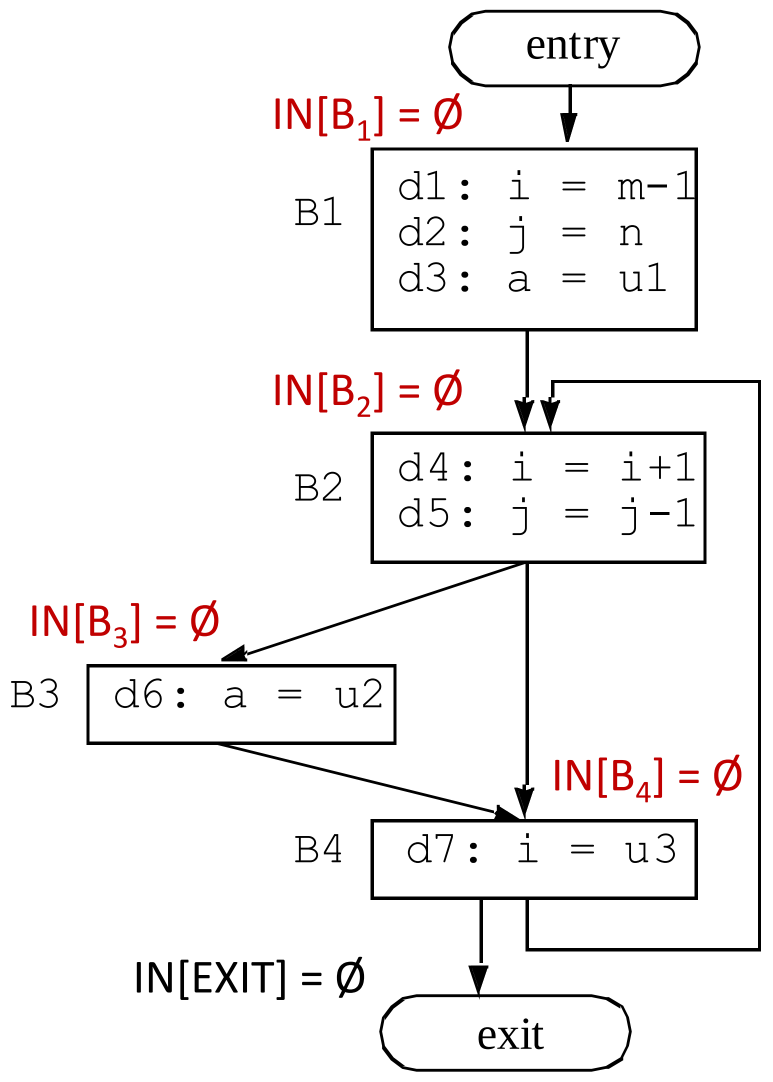
- Applicare l'algoritmo fino a convergenza per il grafo d'esempio

- Boundary condition

  IN[EXIT] = Ø

- Initial condition

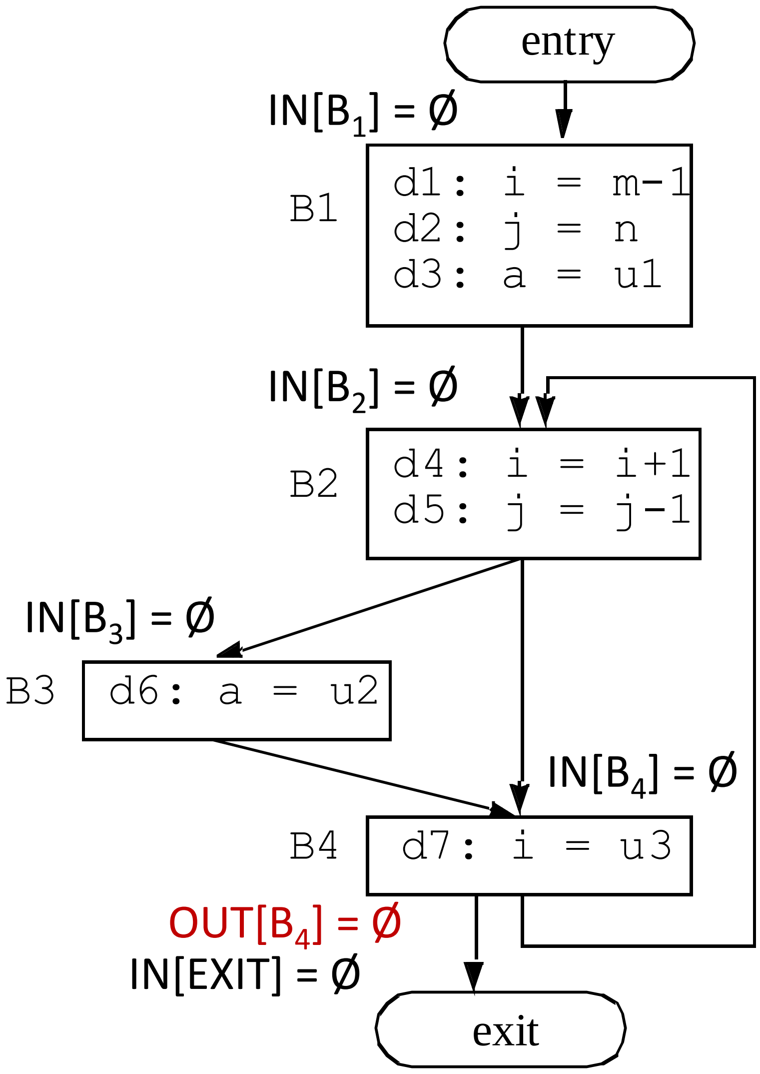
  $IN[B_i] = \emptyset, \forall i \neq ENTRY$



Esempio

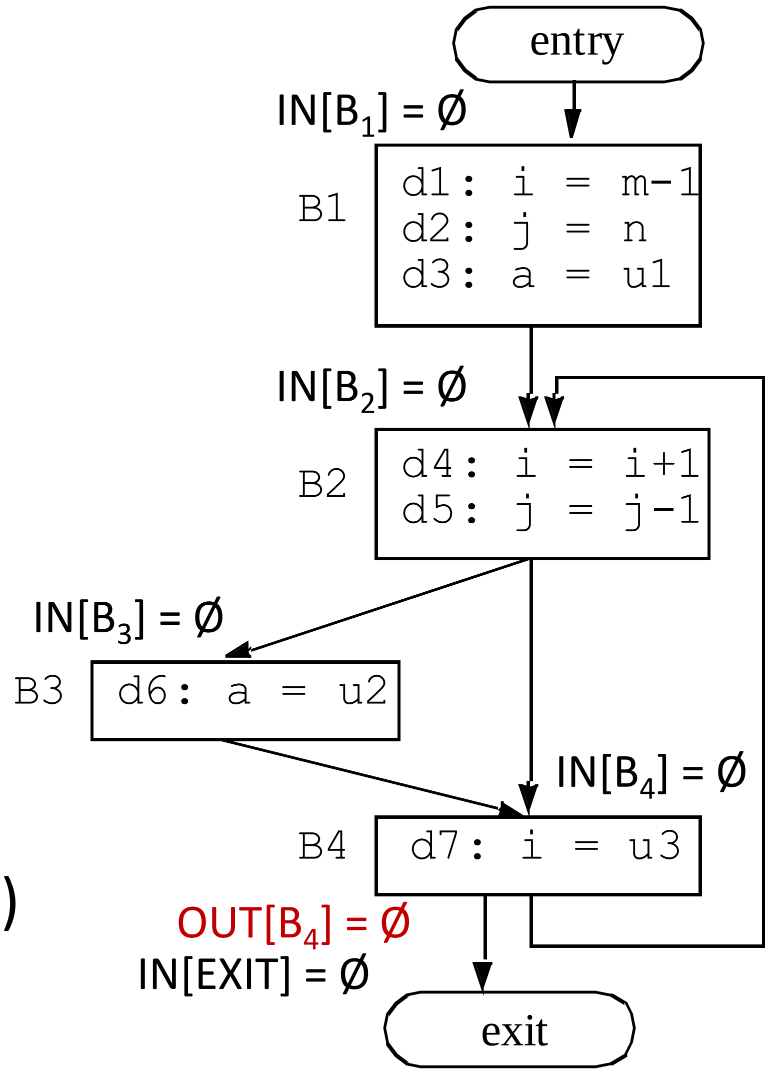
- Applicare l'algoritmo fino a convergenza per il grafo d'esempio

- Fallthrough



Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio



- Funzione di trasferimento

  in[B]=Use[B] U (out[B]-Def[B])

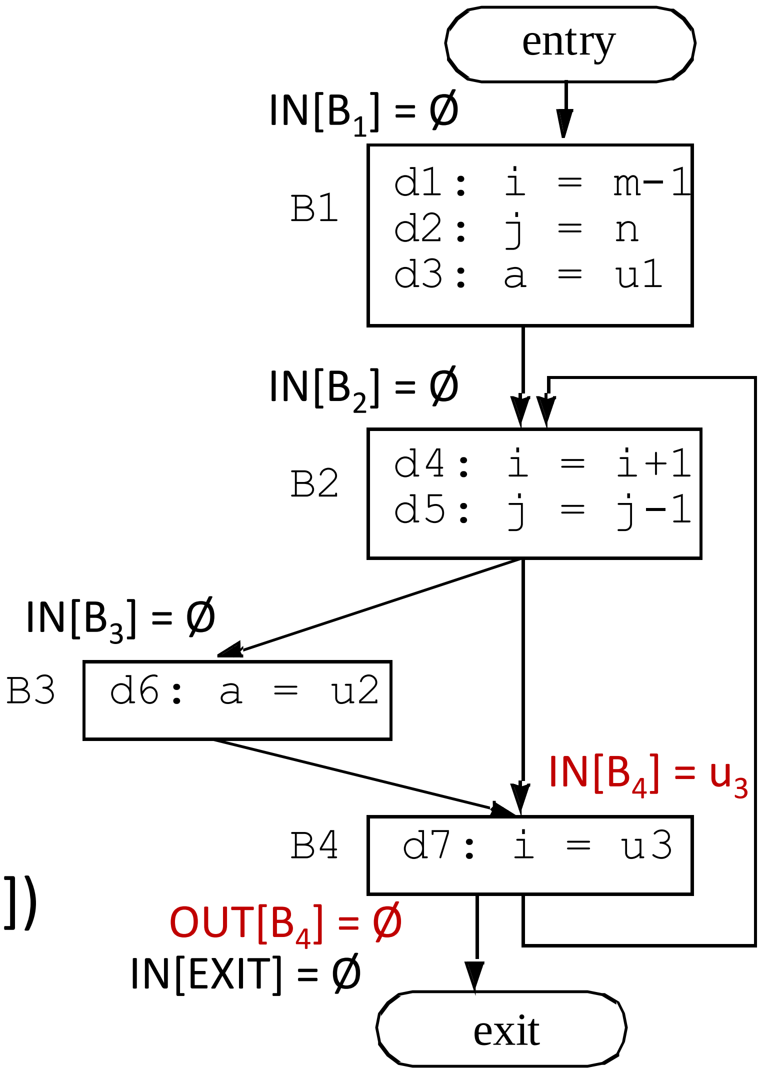
Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio

- Funzione di trasferimento

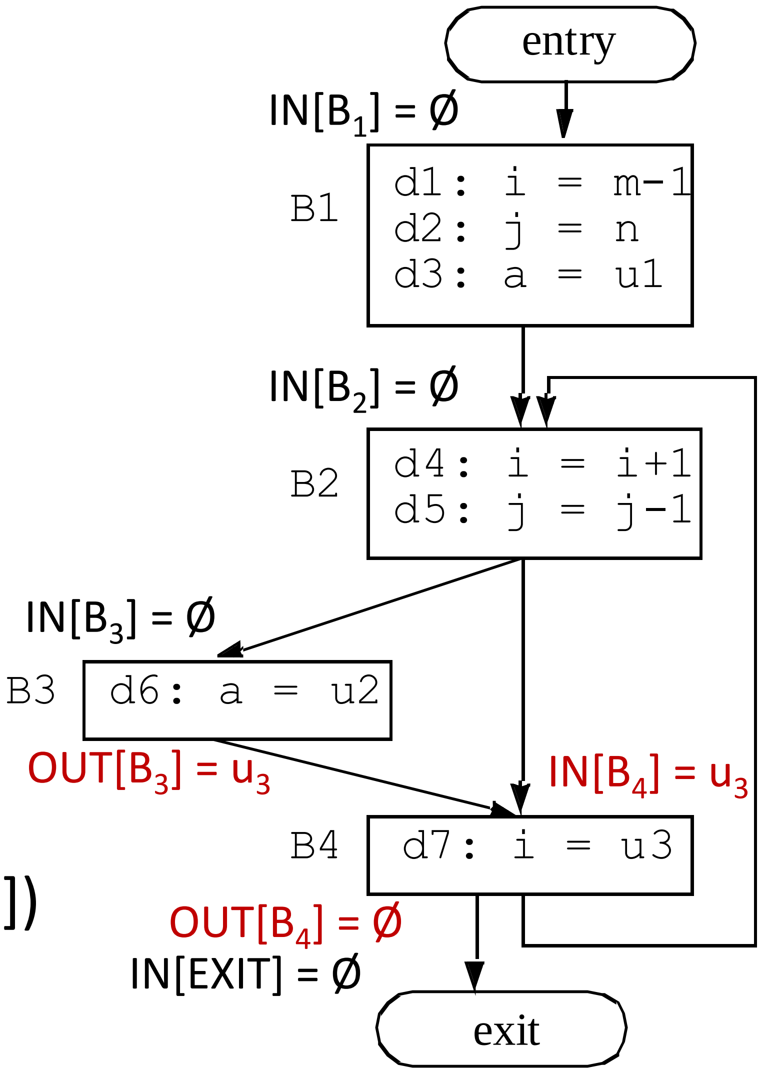
  in[B] = Use[B] U (out[B]-Def[B])

  $in[B_4] = \{u_3\} \cup (\{\emptyset\} - \{i\}) = u_3$



Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio



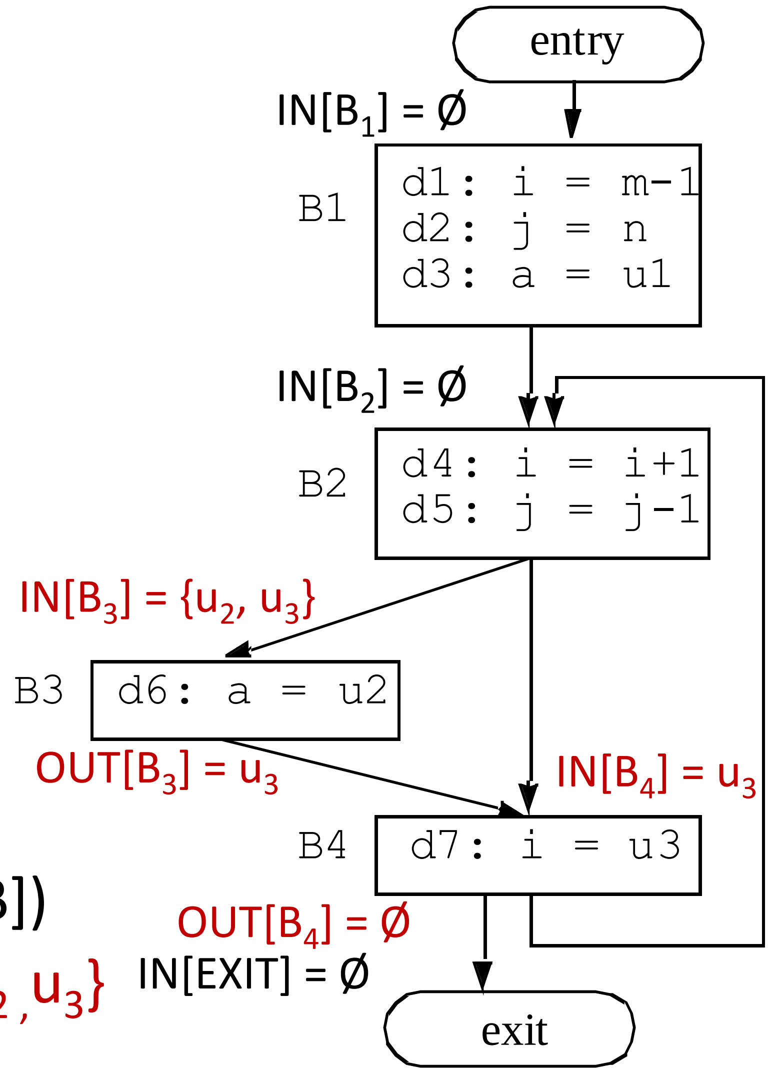
- Funzione di trasferimento

  in[B] = Use[B] U (out[B]-Def[B])

  $in[B_4] = \{u_3\} \cup (\{\emptyset\} - \{i\}) = u_3$

Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio



- Funzione di trasferimento

  in[B] = Use[B] U (out[B]-Def[B])

  $in[B_3] = \{u_2\} \cup (\{u_3\} - \{a\}) = \{u_2, u_3\}$

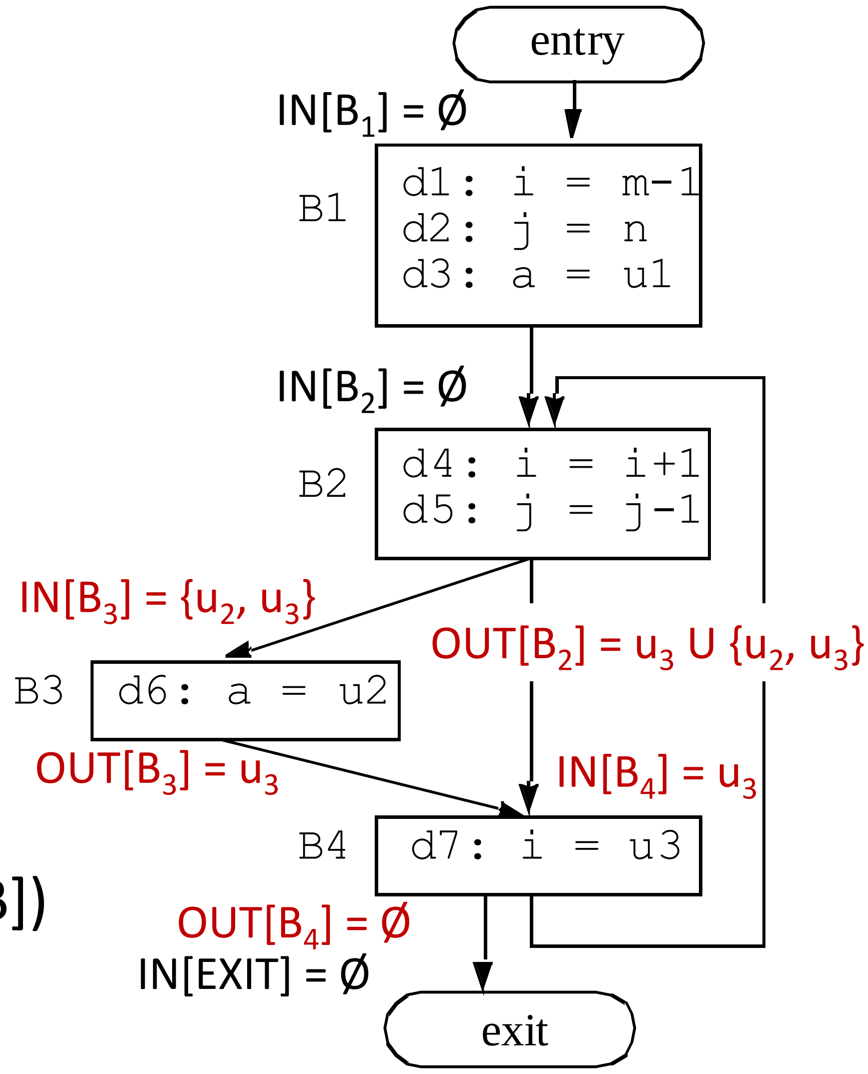
Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio
- Unione degli input

  out[B] = U (in[*succ*])

- Funzione di trasferimento

  in[B] = Use[B] U (out[B]-Def[B])



Esempio

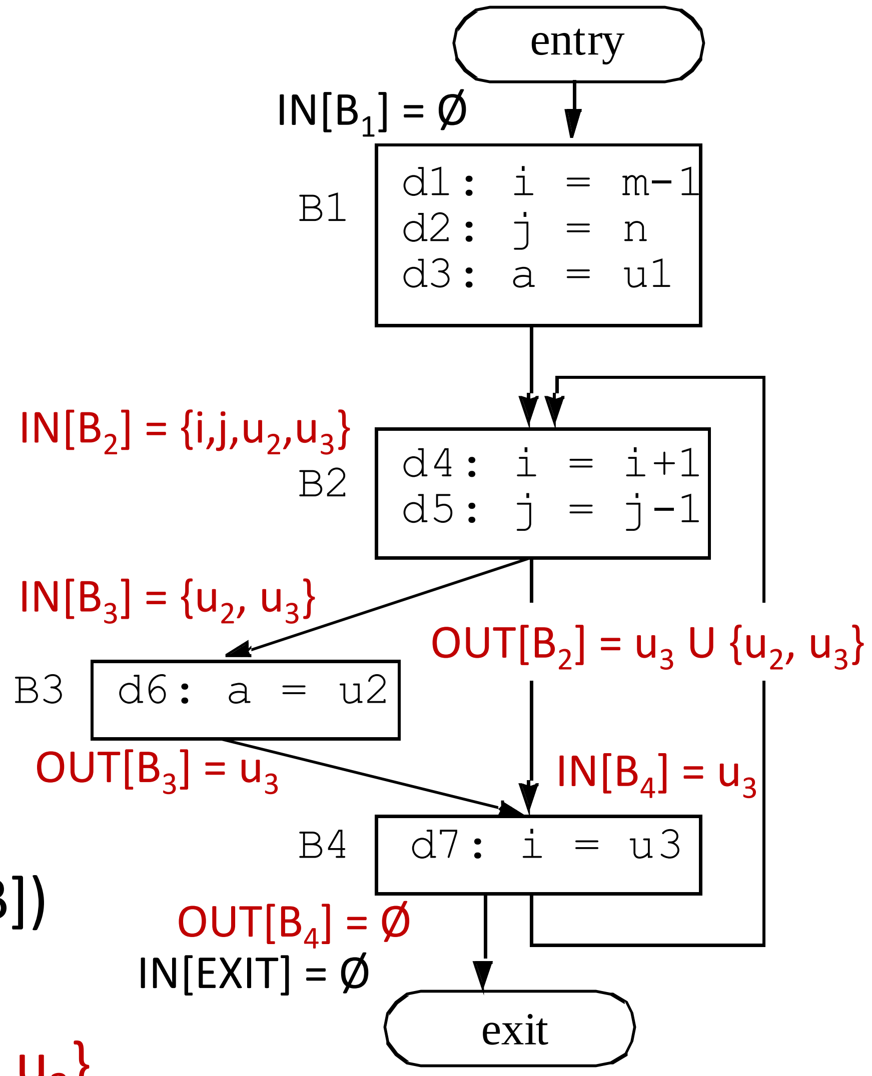
- Applicare l'algoritmo fino a convergenza per il grafo d'esempio
- Unione degli input

  out[B] = U (in[*succ*])

- Funzione di trasferimento

  in[B] = Use[B] U (out[B]-Def[B])

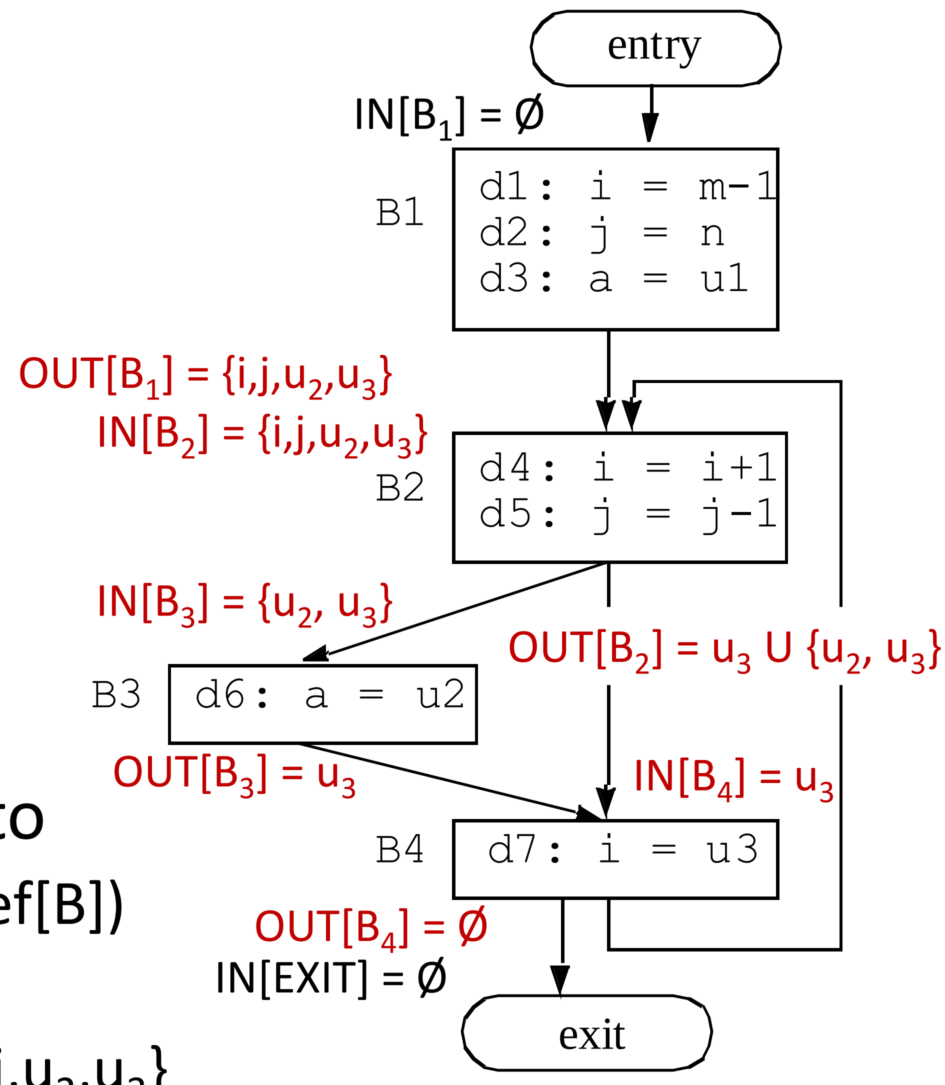
  $in[B_2] = \{i,j\} \cup$
  $(\{u_2,u_3\} - \{i,j\}) = \{i,j,u_2,u_3\}$



Esempio

- Applicare l'algoritmo fino a convergenza per il grafo d'esempio
- Unione degli input

  out[B] = U (in[*succ*])



- Funzione di trasferimento

  in[B] = Use[B] U (out[B]-Def[B])

  $in[B_2] = \{i,j\} \cup$

  $(\{u_2,u_3\} - \{i,j\}) = \{i,j,u_2,u_3\}$

Esempio

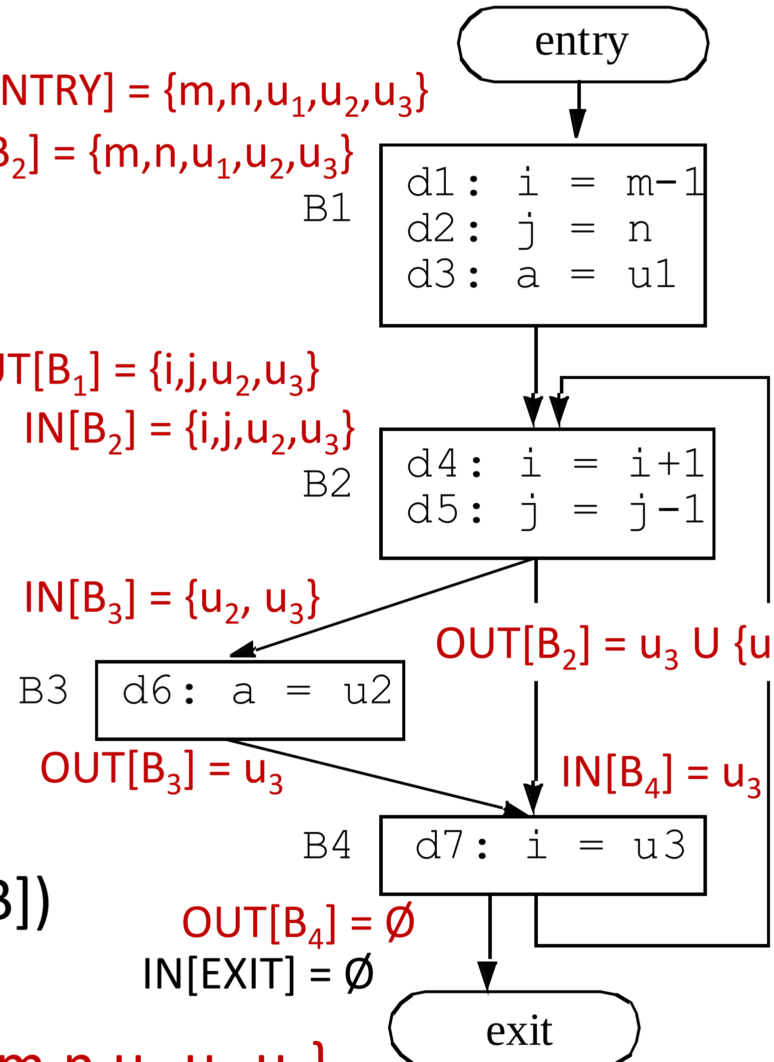
- Applicare l'algoritmo fino a convergenza per il grafo d'esempio
- Unione degli input

  out[B] = U (in[*succ*])

- Funzione di trasferimento

  in[B] = Use[B] U (out[B]-Def[B])

  $in[B_1] = \{m,n,u_1\} \cup (\{i,j,u_2,u_3\} - \{i,j,a\}) = \{m,n,u_1,u_2,u_3\}$



$OUT[ENTRY] = \{m,n,u_1,u_2,u_3\}$

$IN[B_2] = \{m,n,u_1,u_2,u_3\}$

$OUT[B_1] = \{i,j,u_2,u_3\}$

$IN[B_2] = \{i,j,u_2,u_3\}$

$IN[B_3] = \{u_2, u_3\}$

$OUT[B_2] = u_3 \cup \{u_2, u_3\}$

$OUT[B_3] = u_3$

$IN[B_4] = u_3$

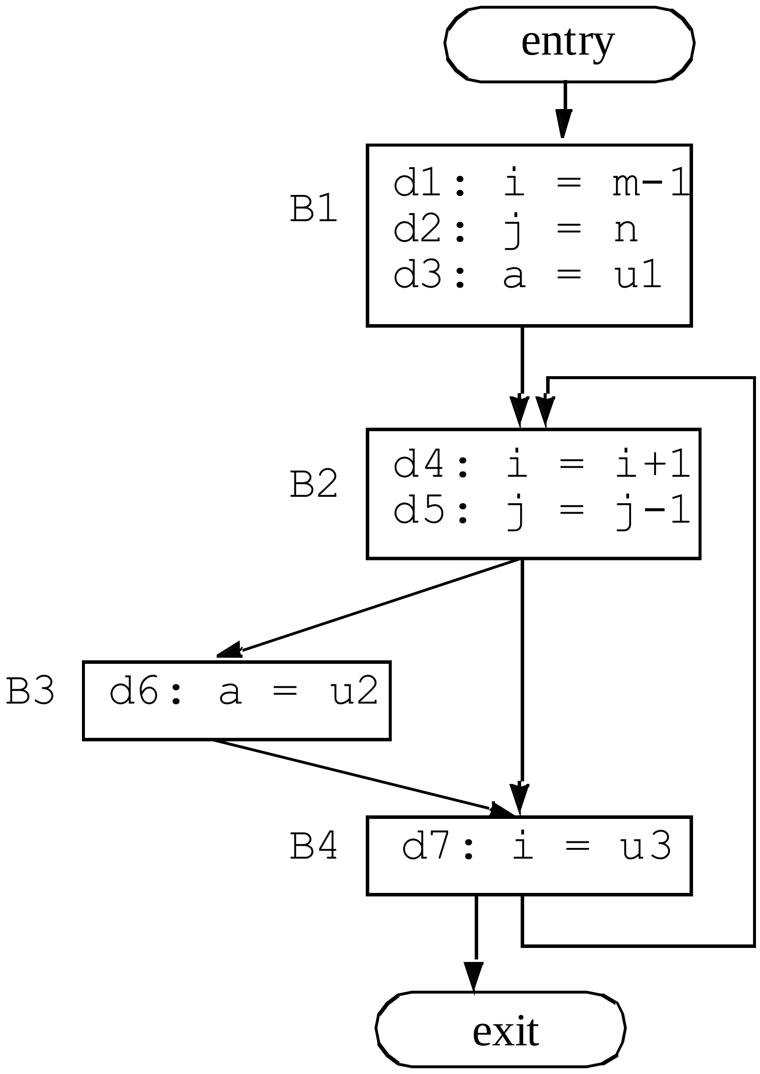
$OUT[B_4] = \emptyset$

$IN[EXIT] = \emptyset$

Esempio

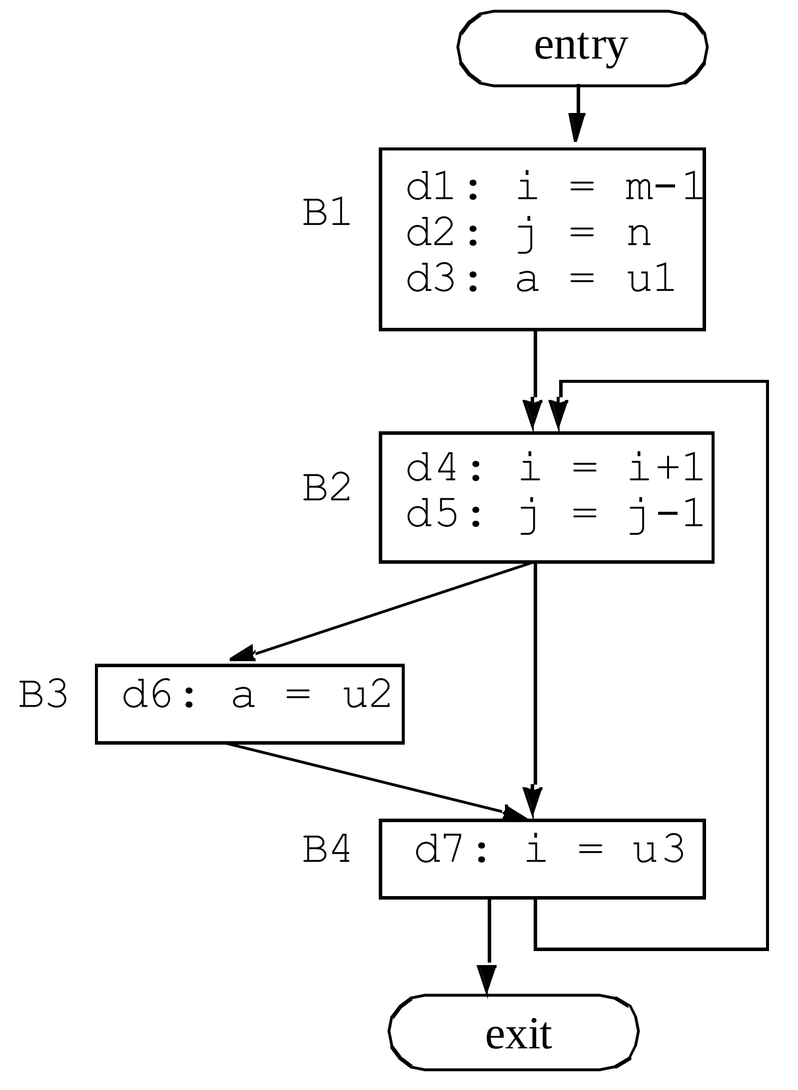
	First Pass
OUT[entry]	{m,n,u1,u2,u3}
IN[B1]	{m,n,u1,u2,u3}
OUT[B1]	{i,j,u2,u3}
IN[B2]	{i,j,u2,u3}
OUT[B2]	{u2,u3}
IN[B3]	{u2,u3}
OUT[B3]	{u3}
IN[B4]	{u3}
OUT[B4]	{}

PRIMA ITERAZIONE COMPLETA



Esempio

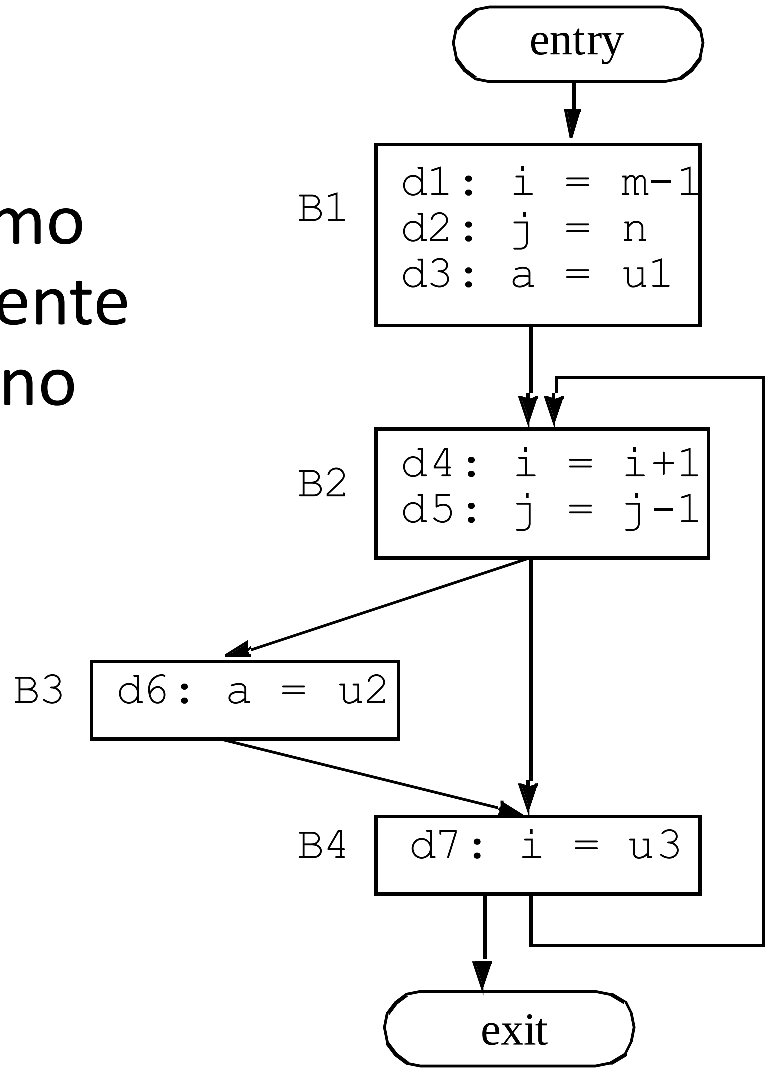
	First Pass	Second Pass
OUT[entry]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
IN[B1]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
OUT[B1]	{i,j,u2,u3}	{i,j,u2,u3}
IN[B2]	{i,j,u2,u3}	{i,j,u2,u3}
OUT[B2]	{u2,u3}	{j,u2,u3}
IN[B3]	{u2,u3}	{j,u2,u3}
OUT[B3]	{u3}	{j,u2,u3}
IN[B4]	{u3}	{j,u2,u3}
OUT[B4]	{}	{i,j,u2,u3}



SECONDA ITERAZIONE COMPLETA

Esempio

- La convergenza dell'algoritmo è garantita indipendentemente dall'ordine col quale vengono processati i blocchi
- Ripetere l'esercizio usando un ordine crescente di processing dei blocchi



Framework

	Reaching Definitions	**Live Variables**
Domain	Sets of definitions	Sets of variables
Direction	forward: out[b] = f_b(in[b]) in[b] = $\wedge$ out[pred(b)]	backward: in[b] = f_b(out[b]) out[b] = $\wedge$ in[succ(b)]
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$	$f_b(x) = Use_b \cup (x - Def_b)$
Meet Operation ($\wedge$)	$\cup$	$\cup$
Boundary Condition	out[entry] = $\emptyset$	in[exit] = $\emptyset$
Initial interior points	out[b] = $\emptyset$	in[b] = $\emptyset$

Ci sono molti altri problemi che la *dataflow analysis* può risolvere (es., *available expressions, dominators, CP, …*)

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Available Expressions

Available Expressions

- Utili in ottimizzazioni come la Global Common Subexpression Elimination

```
if (…) {
   x = m + n;
} else {
   y = m + n;
}
z = m + n;
```

m+n è già stato calcolato, quindi è ridondante

Available Expressions

- Utili in ottimizzazioni come la Global Common Subexpression Elimination

```
if (…) {
   x = m + n;
} else {
   …;
}
z = m + n;
```

Ma cosa succeed se *m+n* NON viene calcolato nel ramo **else**?

Available Expressions

- Ci serve una maniera rigorosa di ragionare sulla ridondanza
  - → `Available Expressions`

- Nel problema di Dataflow delle `Available Expressions` ci interessano le **espressioni**
  - **Dominio**: Insieme delle espressioni
  - Solo espressioni binarie del tipo $x \oplus y$

Available Expressions

- **Terminologia**
  - Una espressione $x \oplus y$ è **available** in un punto p del programma se ogni percorso che parte dal blocco ENTRY e arriva a p valuta l'espressione $x \oplus y$

  - Un blocco **genera** l'espressione $x \oplus y$ se valuta $x \oplus y$ e non ridefinisce in seguito x o y

  - Un blocco **uccide** l'espressione $x \oplus y$ se assegna (o potrebbe assegnare) un valore a x o y e non ricalcola successivamente $x \oplus y$

Available Expressions

- **Esempio**

```
x = y + 1; // generates 'y + 1'
y = m + n; // generates 'm + n', kills 'y + 1'
```

→ Transfer Function: $f_B \coloneqq gen_B \cup (x - kill_B)$

Available Expressions

- Qual è la **direzione** dell’analisi?
- Nell’analisi delle **available expressions** eliminiamo un’espressione perché è stata calcolata **in passato**
- Nell’analisi delle **live variables** eliminiamo una variabile perché non verrà usata **in futuro**

Available Expressions

- Qual è la **direzione** dell'analisi?
- Nell'analisi delle **available expressions** eliminiamo un'espressione perché è stata calcolata **in passato**
- Nell'analisi delle **live variables** eliminiamo una variabile perché non verrà usata **in futuro**
- **Direzione → In avanti (Forward)**

Available Expressions

- Come definiamo le equazioni per **IN[B]** e **OUT[B]**?
- **Equazioni OUT:** $OUT[B] = f_B(IN[B])$
- **Equazioni IN:** $IN[B] = \wedge_{p \in pred(B)} (OUT[p])$

meet operator

- Quale dovrebbe essere l'operatore di **meet**?

Available Expressions

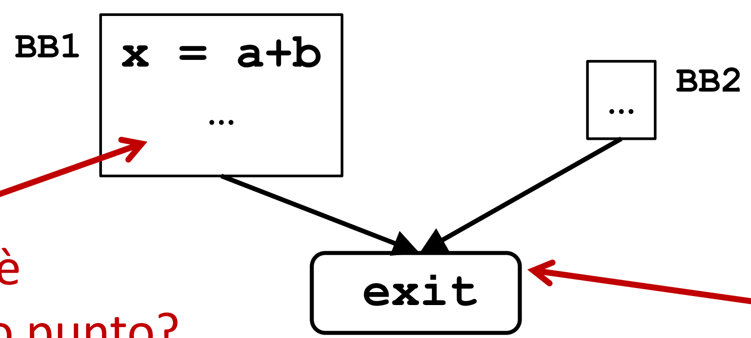
- Ricordiamo la definizione del problema

*Una espressione $x \oplus y$ è **available** in un punto p del programma se **ogni** percorso che parte dal blocco ENTRY e arriva a p valuta l'espressione $x \oplus y$*

Available Expressions

- Ricordiamo la definizione del problema

*Una espressione x $\oplus$ y è **available** in un punto p del programma se **ogni** percorso che parte dal blocco ENTRY e arriva a p valuta l'espressione x $\oplus$ y*



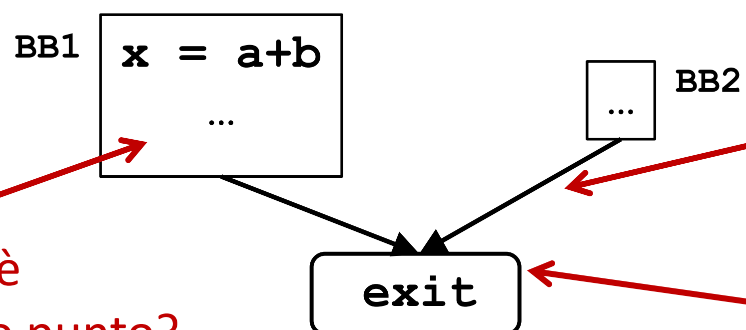
L'espressione **a+b** è **available** in questo punto?

E in questo punto?

Available Expressions

- Ricordiamo la definizione del problema

*Una espressione $x \oplus y$ è **available** in un punto p del programma se **ogni** percorso che parte dal blocco ENTRY e arriva a p valuta l'espressione $x \oplus y$*



No, perché non è **available** lungo questo percorso

L'espressione **a+b** è **available** in questo punto?

E in questo punto?

Available Expressions

- Come definiamo le equazioni per **IN[B]** e **OUT[B]**?
- **Equazioni OUT:** $OUT[B] = f_B(IN[B])$
- **Equazioni IN:** $IN[B] = \wedge_{p \in pred(B)} (OUT[p])$

meet operator

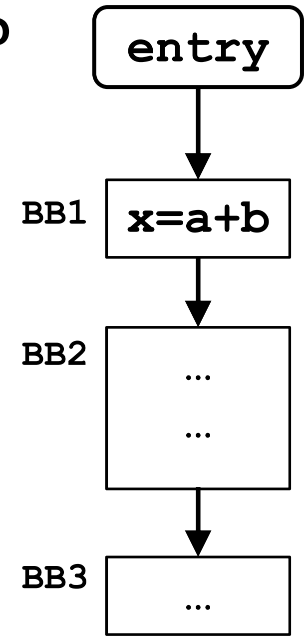
- Quale dovrebbe essere l'operatore di **meet**?
  - L'operatore di intersezione $\cap$

Available Expressions

- Quali sono le **condizioni al contorno**?
- Quali sono le **condizioni iniziali**?

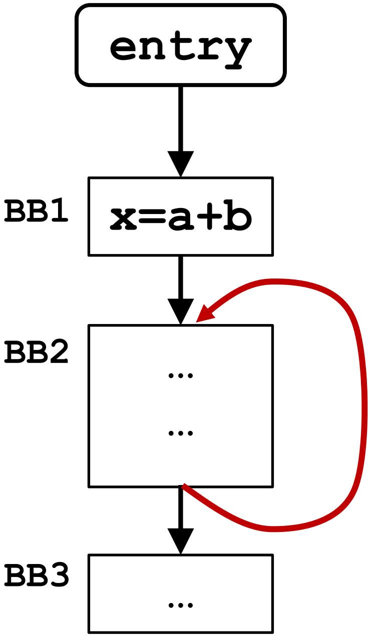
Available Expressions

- Quali sono le **condizioni al contorno**?
  - **OUT[ENTRY] = Ø**
- Quali sono le **condizioni iniziali**?



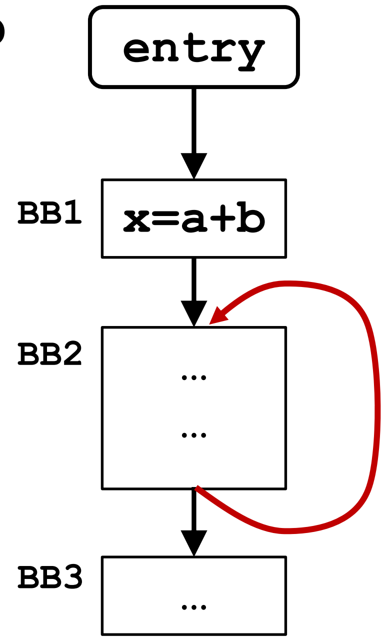
Available Expressions

- Quali sono le **condizioni al contorno**?
  - **OUT[ENTRY] = Ø**
- Quali sono le **condizioni iniziali**?
  - **OUT[B_i] = Ø ?**



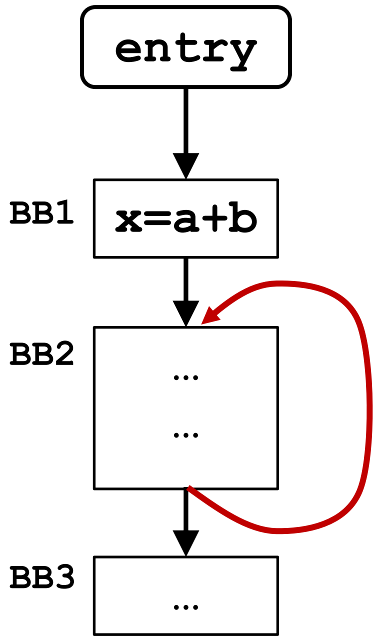
Available Expressions

- Quali sono le **condizioni al contorno**?
  - **OUT[ENTRY] = Ø**
- Quali sono le **condizioni iniziali**?
  - ~~**OUT[B_i] = Ø ?**~~
  - **Il nostro meet operator è ∩**
  - **OUT[B_i] =** $\mathcal{U}$ (universal set)



Available Expressions

- Quali sono le **condizioni al contorno**?
  - **OUT[ENTRY] = Ø**
- Quali sono le **condizioni iniziali**?
  - ~~**OUT[B_i] = Ø ?**~~
  - **Il nostro meet operator è ∩**
  - **OUT[B_i] =** $\mathcal{U}$ (universal set)



Available Expressions

- Dataflow Analysis

	Available Expressions
Domain	Sets of Expressions
Direction	Forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet Operation ($\wedge$)	$\cap$
Boundary Condition	$out[entry] = \varnothing$
Initial interior points	$out[b] = U$

Esercizio

- Risolvere il problema di DFA delle *Available Expressions* per il CFG in figura