UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

9. Loops e UD-DU chains

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”

Cos'è un *Loop*?

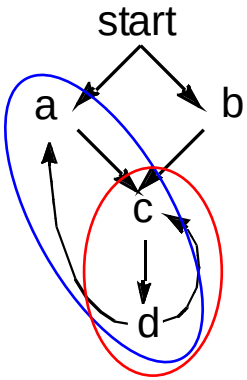
- Come già sappiamo, i programmi spendono la maggior parte del tempo nei loop
  - È conveniente quindi saperli rappresentare nella IR in maniera specifica

- Obiettivo:
  - Definire un loop in termini di teoria dei grafi (control flow graph)
  - Indipendentemente dalla sintassi
  - Una rappresentazione unica per tutti i tipi di *loop*: `for`, `while`, `goto`, …

Cos'è un *Loop*?

- Non tutti i cicli sono un "loop" da un punto di vista dell'ottimizzazione

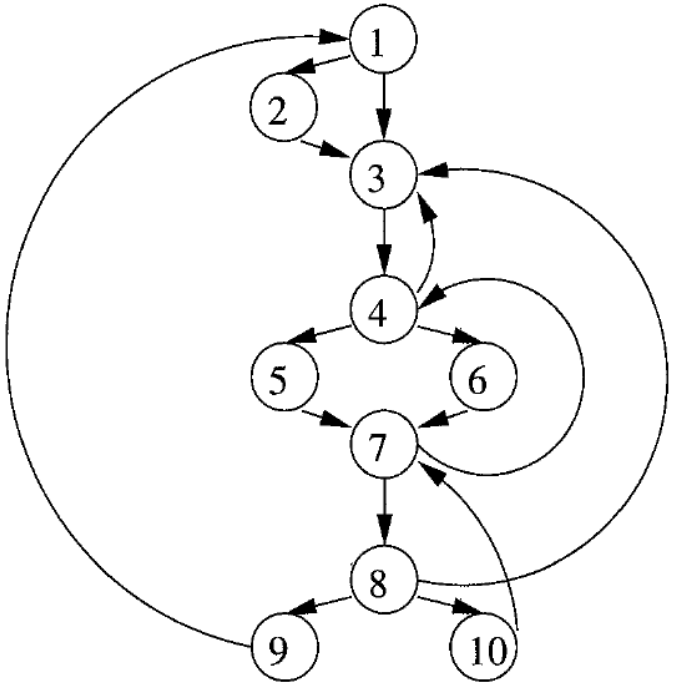
Questo è un loop?



Questo è un loop?

- Proprietà intuitive di un *loop*
  - Singolo *entry point*
  - Gli archi devono formare almeno un ciclo

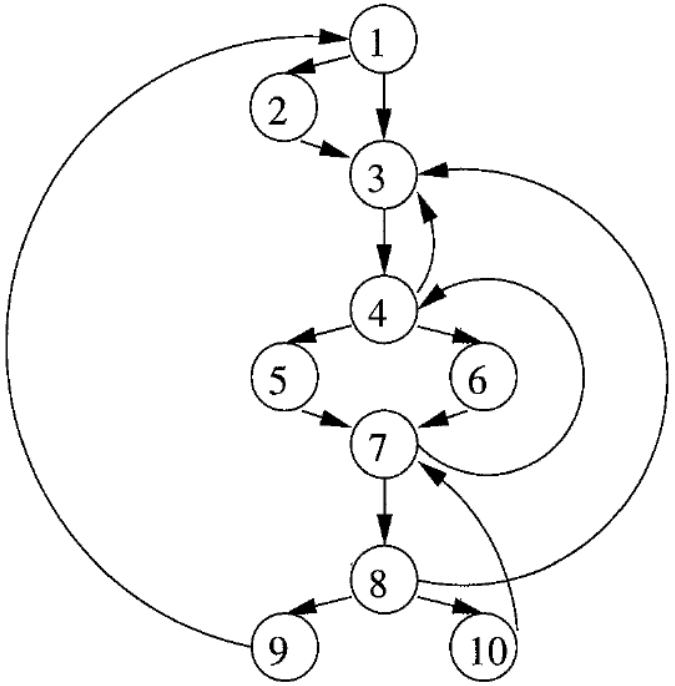
Definizioni Formali



- **Dominator**
  - Un nodo *d* domina un nodo *n* in un grafo (*d* ***dom*** *n*) se ogni percorso dall'ENTRY node a *n* passa per *d*

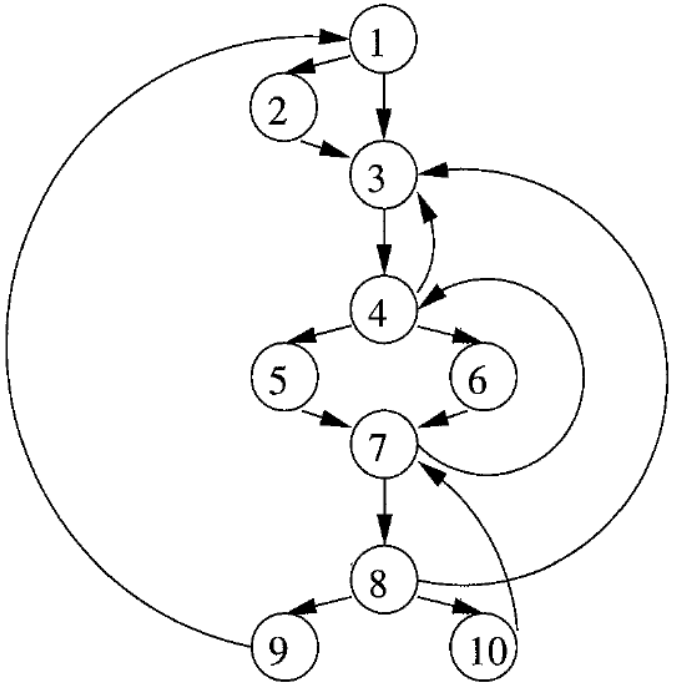
Definizioni Formali

- **Dominator tree**
  - I **dominators** possono essere rappresentati come un albero
    - *a* -> *b* nel dominator tree **iff** *a* domina **immediatamente** *b*
    - Il nodo *entry* è la radice, ogni nodo *d* domina solo i suoi discendenti nell'albero



Definizioni Formali

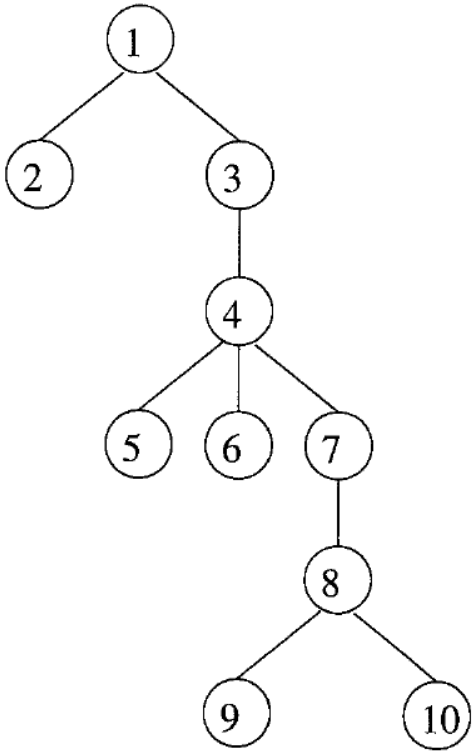
- **Dominator tree**
  - I **dominators** possono essere rappresentati come un albero
    - *a* -> *b* nel dominator tree **iff** *a* domina **immediatamente** *b*
    - Il nodo *entry* è la radice, ogni nodo *d* domina solo i suoi discendenti nell'albero
- **Immediate dominator**
  - L'ultimo **dominator** di *n* su qualsiasi percorso da *entry* a *n*

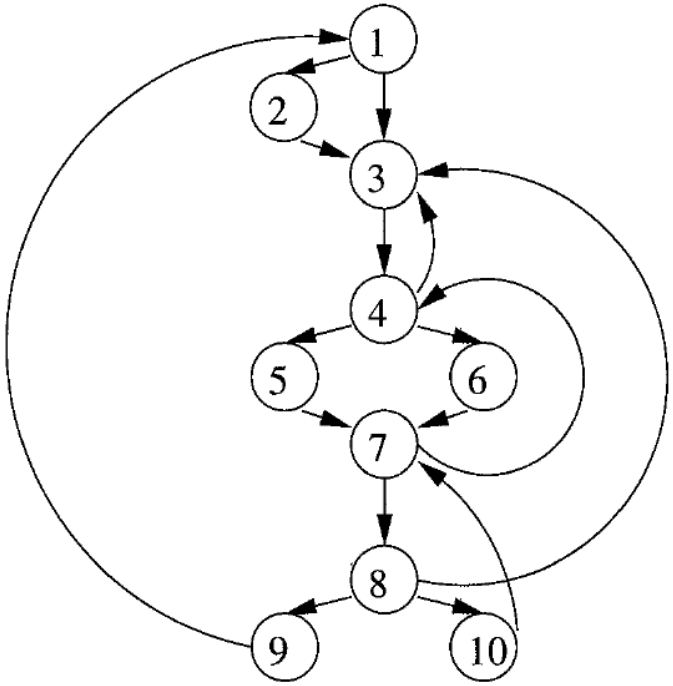


m domina immediatamente (strettamente) *n* (*m* sdom *n*) **iff** *m* dom *n* E $m \neq n$

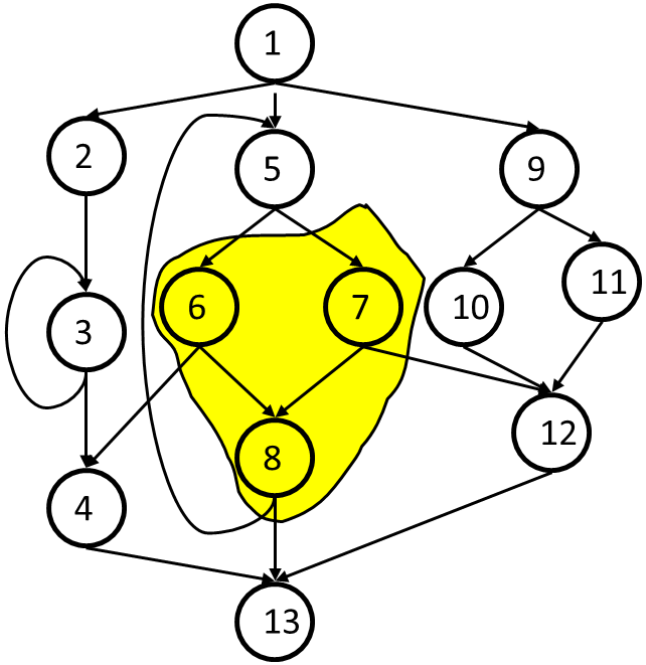
Definizioni Formali

- **Dominator tree**





Un altro esempio

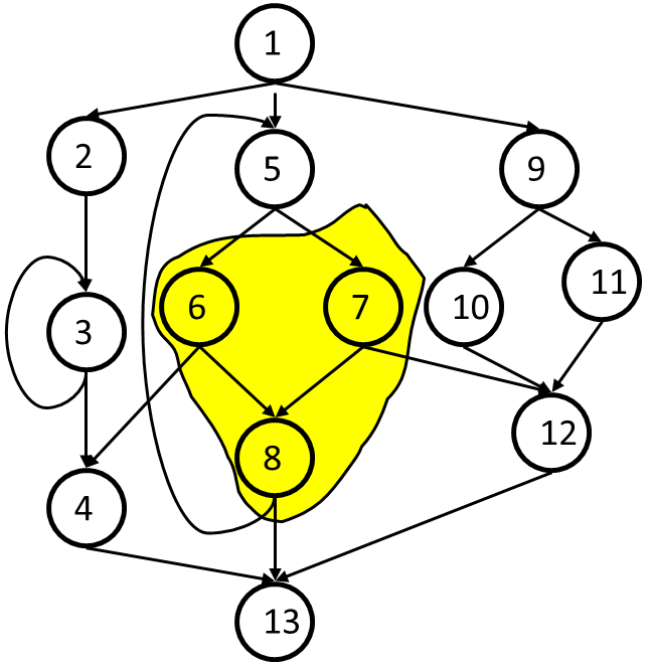


CFG

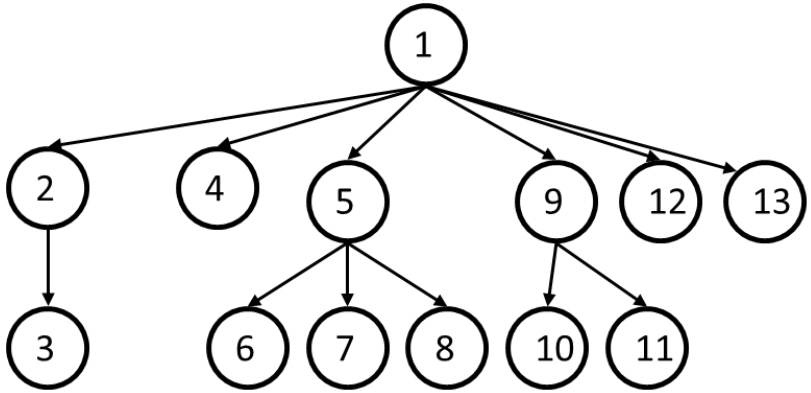
?

D-Tree

Un altro esempio



CFG



D-Tree

Loop Naturali

- I loop possono essere specificati in molti modi diversi nel sorgente (**for**, **while**, **goto**, …)
- Dal punto di vista dell'analisi importa solo che la rappresentazione abbia delle proprietà che facilitino l'ottimizzazione:
  - Singolo entry-point: *header*
    - *L'header* domina tutti i nodi nel loop
  - Un *back edge* è un arco la cui testa domina la propria coda (tail -> head)
    - un *back edge* deve far parte di almeno un loop

Identificare i Loop Naturali

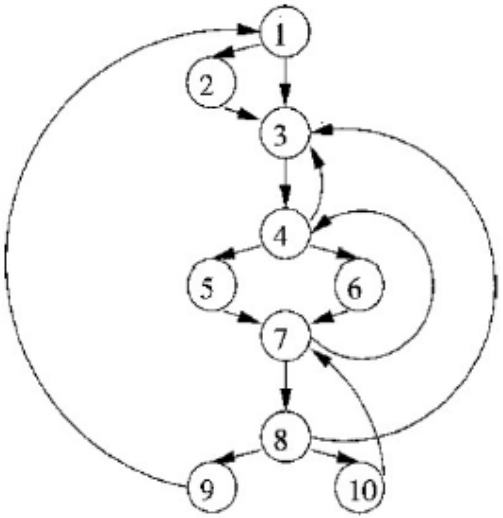
1. Trovare le relazioni di dominanza nel flow graph
2. Identificare i *back edges*
3. Trovare il loop naturale associato al *back edge*

1. Trovare i Dominatori

- Definizione
  - Un nodo *d* domina un node *n* in un grafo (*d* **dom** *n*) se ogni percorso dall'ENTRY node a *n* passa per *d*

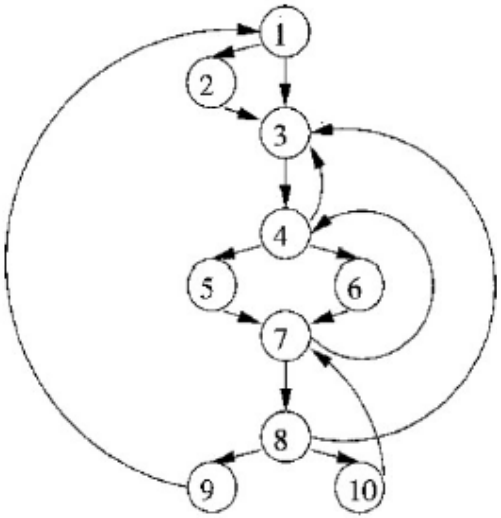
- Formulato come un problema DFA:
  - Direzione:
  - Dominio (Valori):
  - Meet operator:
  - Condizioni al contorno:
  - Condizioni iniziali:
  - Funzione di trasferimento:

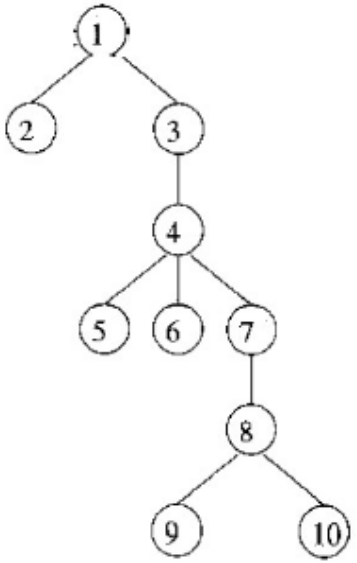
Esempio



- Trovare il Dominance Tree

Esempio





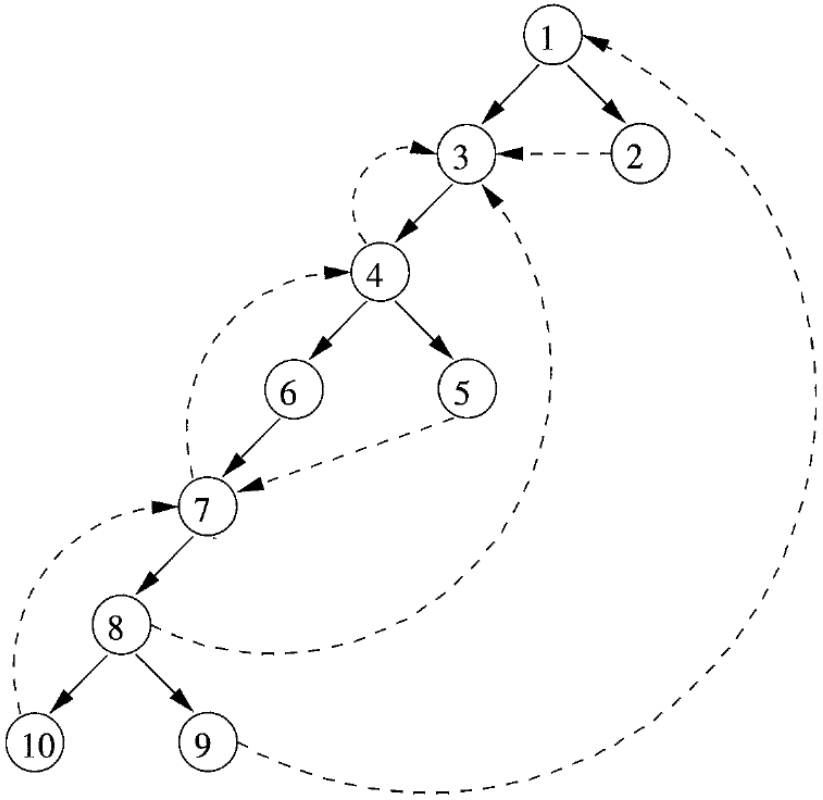
2. Trovare i *Back Edges*

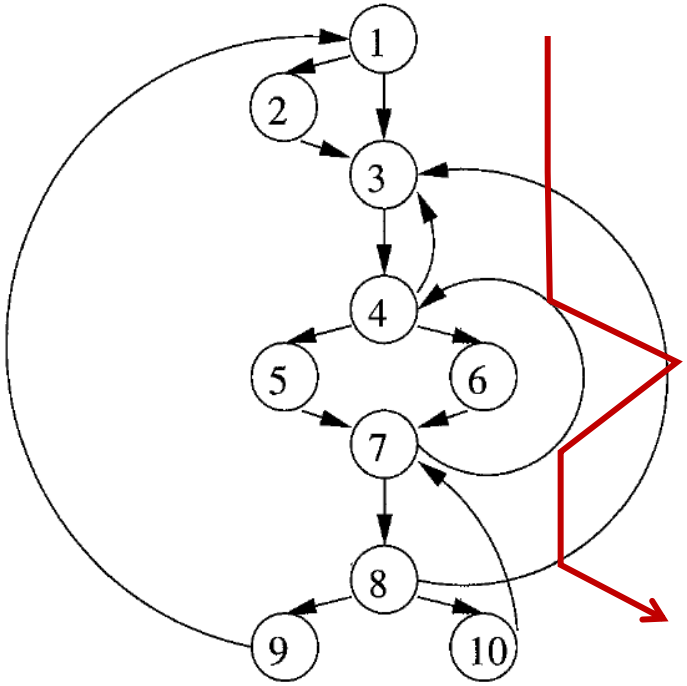
- Depth-first traversal
  - A depth-first traversal starts at the root and recursively visits the children of each node in any order, not necessarily left to right
  - It is called "depth-first" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "deep") as quickly as it can

The Dragon book, Sec. 2.3.4

2. Trovare i *Back Edges*

Una possibile visita *depth-first*





Il percorso della visita definisce un ***depth-first spanning tree*** (DFST)

- Archi solidi: struttura dell'albero
- Archi tratteggiati: altri archi del CFG

2. Trovare i *Back Edges*

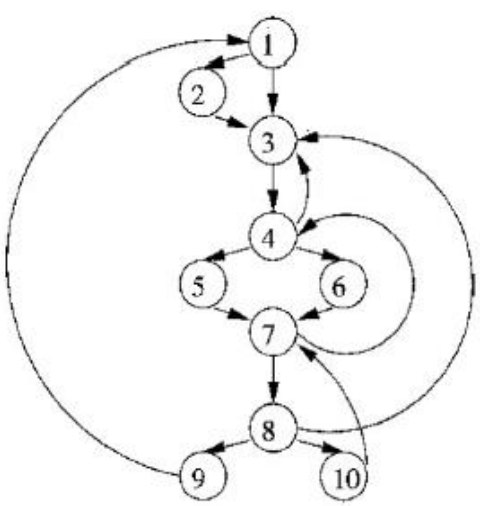
- Categorizzazione degli archi nel grafo:
  - ***Advancing*** (A) edges: dall'antenato al discendente (*proper*). Tutti gli archi solidi del DFST sono A.
  - ***Retreating*** (R) edges: dal discendente all'antenato (non necessariamente *proper* → da un nodo a sé stesso). Solo archi tratteggiati del DFST (4→3, 7→4, 10→7, 9→1)
  - ***Cross*** (C) edges: Esistono archi *m* → *n* tali per cui né *m* né *n* è un antenato dell'altro (si considerano solo gli archi solidi, es. 2→3, 5→7)
    - Se disegniamo il DFST in modo che i figli di un nodo siano aggiunti da sinistra a destra nell'ordine in cui sono visitati, allora i *cross edges* vanno sempre **da destra a sinistra**

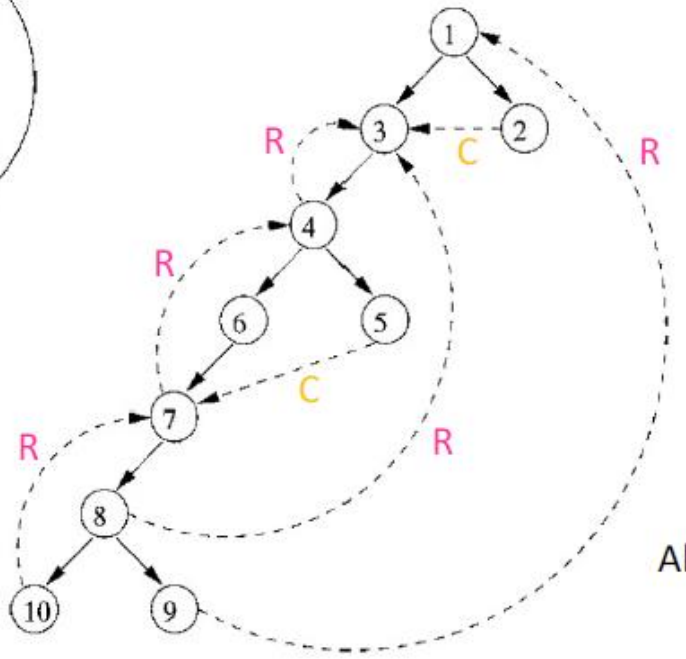
2. Trovare i *Back Edges*

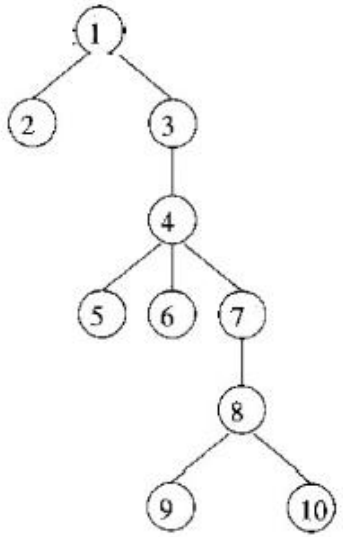
- Definizione
  - Back edge: tail (*t)* → head (*h)*, *h* domina *t*
- Algoritmo
  - Esegui una *depth first search*
  - Per ogni retreating edge *t* -> *h* controlla se *h* è nella lista dei dominatori di *t*

- La maggior parte dei programmi (tutto il codice strutturato e la maggior parte dei GOTO) hanno control flow graphs riducibili
  - retreating edges = back edges

2. Trovare i *Back Edges*



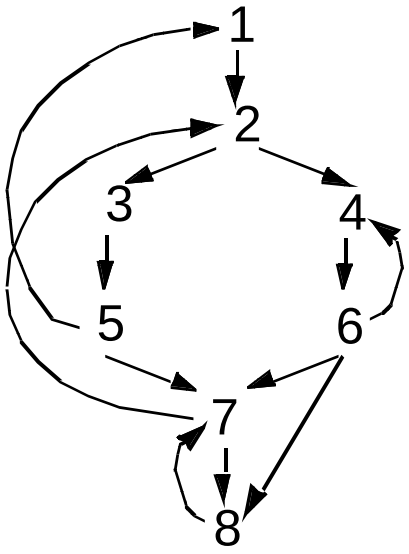




All the retreating edges are back edges

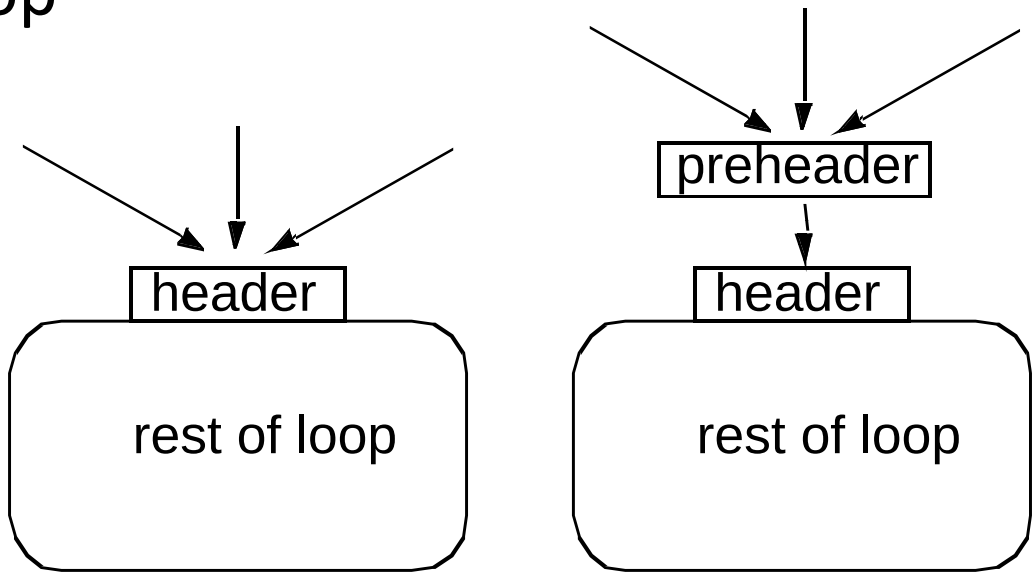
3. Trovare il Loop Naturale

- Il loop naturale di un *back edge* è il più piccolo insieme di nodi che include *head* e *tail* del *back edge* e non ha predecessor fuori da questo insieme (a parte di predecessor dell'*header*).
- Algoritmo
  - eliminare *h* dal CFG
  - Trovare i nodi che raggiungono *t* (questi nodi, più *h* formano il loop naturale *t* -> *h*)



Preheader

- Le ottimizzazioni sui loop spesso richiedono che del codice venga eseguito una volta, prima del loop
- A questo scopo si crea un blocco preheader per ogni loop



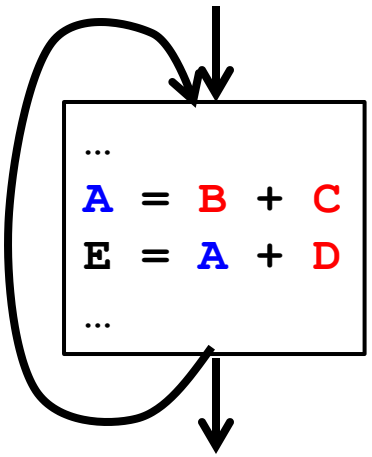
UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

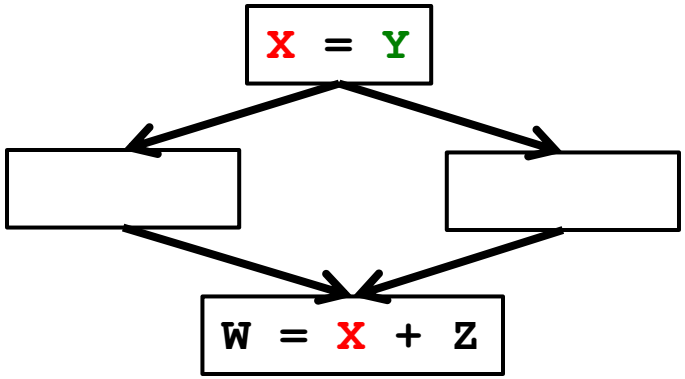
Use-Def e Def-Use Chains

Dove viene definita o usata una variabile?

- Esempio: Loop-Invariant Code Motion
  - B, C, e D sono definite solo fuori dal loop?
  - Ci sono altre definizioni di A dentro il loop?
  - Ci sono usi di A dentro il loop?



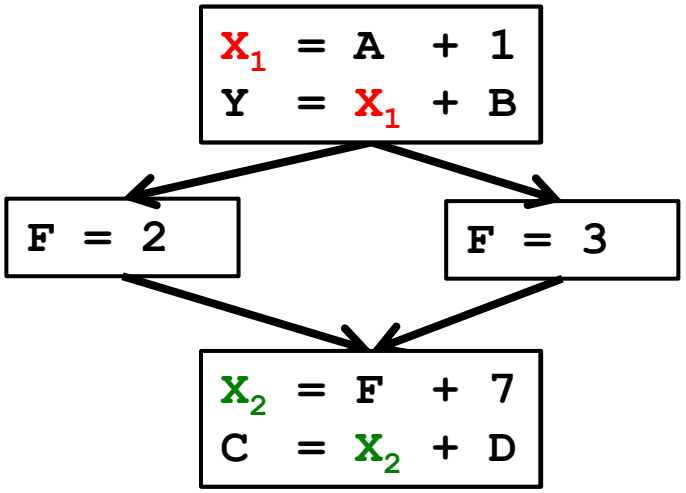
- Esempio: Copy Propagation
  - Per un dato uso di X:
    - Sono tutte le reaching definitions di X:
      - Copie della stessa variable: e.g., X = Y
    - Dove Y non è ridefinita da quella copia?
  - In questo caso, sostituisci gli usi di X con usi di Y



- Per questo genere di ottimizzazione è molto utile poter scorrere agevolmente le relazioni di definizioni e usi delle stesse variabili
  - Ciò abiliterebbe una forma di analisi "sparsa" (ignoriamo i casi “don’t care”)

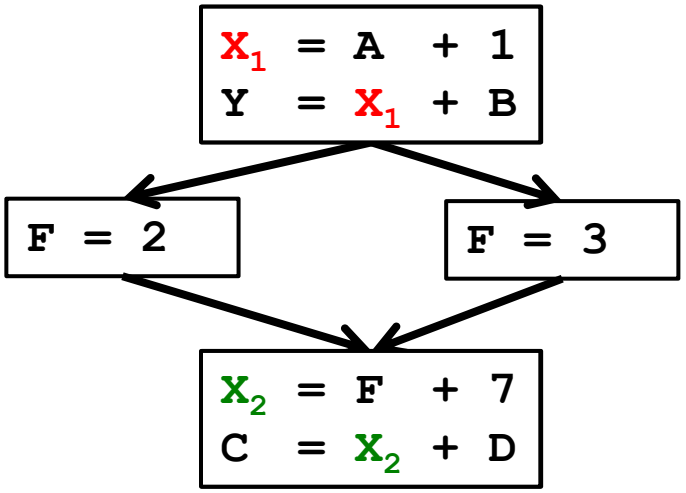
Occorrenze della stessa variabile potrebbero essere scorrelate

- I valori contenuti in celle di memoria riusate potrebbero essere indipendenti
  - Nel qual caso il compilatore potrebbe ottimizzarli come valori separati
- Si potrebbe rinominare le variabili per evidenziare le diverse versioni



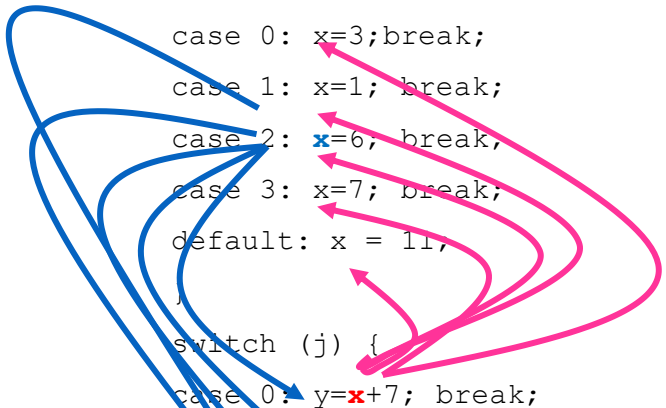
Use-Definition e Definition-Use Chains

- Use-Definition (UD) Chains:
  - Per una data definizione di una variabile X, quali sono tutti i suoi usi?
- Definition-Use (DU) Chains:
  - Per un dato uso di una variabile X, quali sono tutte le *reaching definitions* di X?



UD e DU Chain possono essere onerose

```
foo(int i, int j) {
…
switch (i) {
case 0: x=3;break;
case 1: x=1; break;
case 2: x=6; break;
case 3: x=7; break;
default: x = 11;
}
switch (j) {
case 0: y=x+7; break;
case 1: y=x+4; break;
case 2: y=x-2; break;
case 3: y=x+1; break;
default: y=x+9;
}
…
```



In generale,

N defs

M uses

$\Rightarrow$ O(NM) spazio e tempo

Una soluzione: limitiamo ogni variabile ad UNA definizione