



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

6. Ottimizzazione locale e Local Value Numbering

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

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

- Sampson, Cornell University, “Advanced Compilers”

Scope dell'Ottimizzazione

- Lo scope dell'ottimizzazione è influenzato da come viene gestito il flusso di controllo in un programma

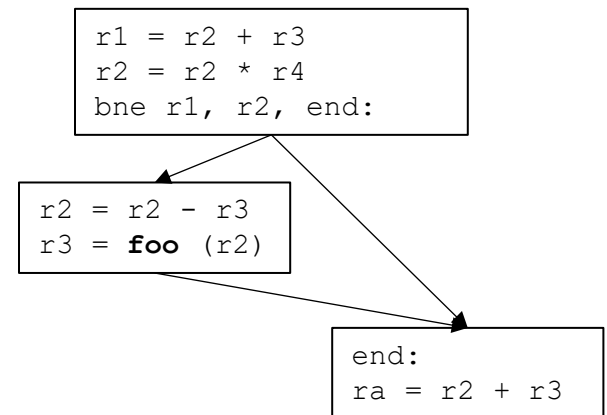
Scope dell'Ottimizzazione

- Lo scope dell'ottimizzazione è influenzato da come viene gestito il flusso di controllo in un programma
 - **OTTIMIZZAZIONE LOCALE**
 - Lavora entro un singolo basic block
 - Non si preoccupa del flusso di controllo

```
r1 = r2 + r3  
r2 = r2 * r4  
bne r1, r2, end:
```

Scope dell'Ottimizzazione

- Lo scope dell'ottimizzazione è influenzato da come viene gestito il flusso di controllo in un programma
 - **OTTIMIZZAZIONE LOCALE**
 - Lavora entro un singolo basic block
 - Non si preoccupa del flusso di controllo
 - **OTTIMIZZAZIONE GLOBALE**
 - Lavora al livello dell'intero CFG



Scope dell'Ottimizzazione

- Lo scope dell'ottimizzazione è influenzato da come viene gestito il flusso di controllo in un programma

- **OTTIMIZZAZIONE LOCALE**

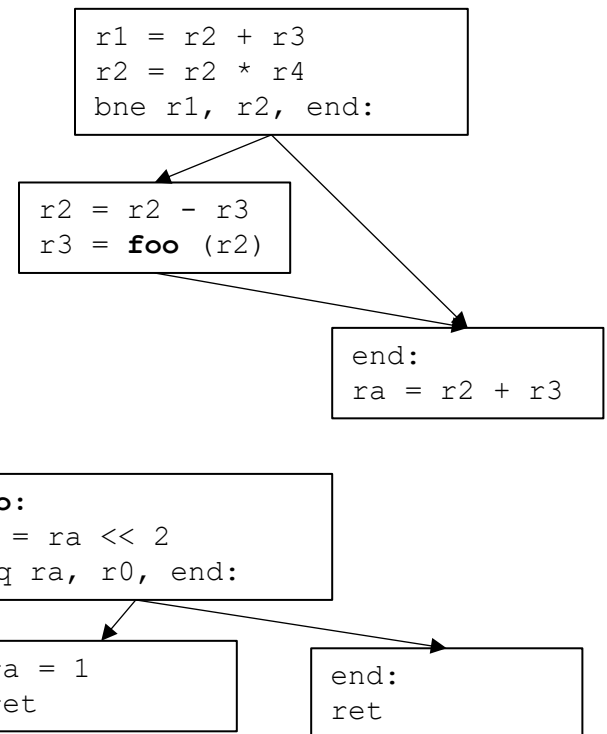
- Lavora entro un singolo basic block
- Non si preoccupa del flusso di controllo

- **OTTIMIZZAZIONE GLOBALE**

- Lavora al livello dell'intero CFG

- **OTTIMIZZAZIONE INTERPROCEDURALE**

- Lavora a livello del call graph
- Lavora sul CFG di più funzioni



Dead code elimination

- Cominciamo col ragionare su **ottimizzazioni locali**, e prendiamo ad esempio la Dead Code Elimination
- Ricorda: Rappresentano dead code le istruzioni che definiscono (assegnano un valore a) una variabile che non è mai utilizzata

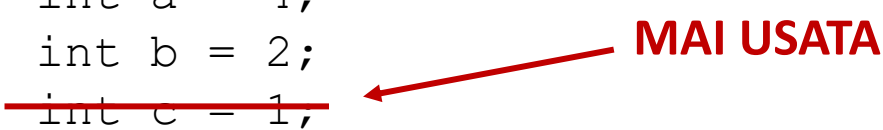
```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
    print d;  
}
```

**Dov'è l'opportunità di
ottimizzazione in questo codice?**

Dead code elimination

- Cominciamo col ragionare su **ottimizzazioni locali**, e prendiamo ad esempio la Dead Code Elimination
- Ricorda: Rappresentano dead code le istruzioni che definiscono (assegnano un valore a) una variabile che non è mai utilizzata

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
    print d;  
}
```



MAI USATA

Dead code elimination

- Cominciamo col ragionare su **ottimizzazioni locali**, e prendiamo ad esempio la Dead Code Elimination
- Ricorda: Rappresentano dead code le istruzioni che definiscono (assegnano un valore a) una variabile che non è mai utilizzata

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
    print d;  
}
```

E questa? Non scrive su nessuna variabile, secondo la definizione è dead code



Dead code elimination

- Cominciamo col ragionare su **ottimizzazioni locali**, e prendiamo ad esempio la Dead Code Elimination
- Ricorda: Rappresentano dead code le istruzioni **prive di *side effects*** che definiscono (assegnano un valore a) una variabile che non è mai utilizzata

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
    print d;  
}
```

HA SIDE EFFECTS. Non è dead code



Algoritmo per la DCE

- Proviamo a scrivere un algoritmo che implementi la DCE così come la abbiamo definita
 1. Per ogni istruzione nel BB
 - a) Aggiungi i suoi operandi ad un metadato array *used*
 2. Per ogni istruzione nel BB
 - a) Se l'istruzione non ha una destinazione e non ha *side effects* rimuovi l'istruzione
 - b) Altrimenti se la destinazione non corrisponde a nessuno degli elementi nell'array *used* rimuovi l'istruzione

Algoritmo per la DCE

- Proviamo a scrivere un algoritmo che implementi la DCE così come la abbiamo definita

```
used = {};  
  
for instr in BB:  
    used += instr.args;  
  
for instr in BB:  
    if instr.dest &&  
        instr.dest not in used:  
        delete instruction
```

Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
    int e = c + d;  
    print d;  
}
```

Quali istruzioni vengono rimosse dal nostro algoritmo?


Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
int e = c + d;  
    print d;  
}
```

Quali istruzioni vengono rimosse dal nostro algoritmo?

Solo questa. e non viene mai utilizzata



Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
int e = c + d;  
    print d;  
}
```

Quali istruzioni vengono rimosse dal nostro algoritmo?

Solo questa. e non viene mai utilizzata

MA CI SONO ALTRE ISTRUZIONI CHE POSSONO ESSERE ELIMINATE?

Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 4;  
    int b = 2;  
    int c = 1;  
    int d = a + b;  
int e = c + d;  
    print d;  
}
```

**Sì, questa, una volta
eliminata la precedente**

**Quali istruzioni vengono rimosse dal
nostro algoritmo?**

**Solo questa. e non viene mai
utilizzata**

**MA CI SONO ALTRE ISTRUZIONI CHE
POSSONO ESSERE ELIMINATE?**

Algoritmo per la DCE

- Perché il nostro algoritmo non elimina l'istruzione $c = 1$?
- *Come possiamo estendere l'algoritmo perché elimini anche quella istruzione?*

Algoritmo per la DCE

- Perché il nostro algoritmo non elimina l'istruzione $c = 1$?
- *Come possiamo estendere l'algoritmo perché elimini anche quella istruzione?*
- *Una maniera molto semplice è quella di ripetere l'algoritmo iterativamente finché non converge*

Algoritmo per la DCE

- Aggiungiamo la parte iterativa

```
while prog changed:
{
    used = {};

    for instr in BB:
        used += instr.args;

    for instr in BB:
        if instr.dest &&
           instr.dest not in used:
            delete instruction
}
```

Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

Quali istruzioni vengono rimosse dal nostro algoritmo?

Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

Quali istruzioni vengono rimosse dal nostro algoritmo?

NESSUNA!

Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

Quali istruzioni vengono rimosse dal nostro algoritmo?

NESSUNA!

Ma ci sono istruzioni che potrebbero essere rimosse?

Algoritmo per la DCE

- Consideriamo quest'altro esempio:

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

Sì, questa



Quali istruzioni vengono rimosse dal nostro algoritmo?

NESSUNA!

Ma ci sono istruzioni che potrebbero essere rimosse?

Algoritmo per la DCE

- Perché il nostro algoritmo non elimina nessuna istruzione?
 - Non è capace di identificare le dead stores

Algoritmo per la DCE

- Perché il nostro algoritmo non elimina nessuna istruzione?
 - Non è capace di identificare le dead stores
- *Come possiamo estendere l'algoritmo perché gestisca una situazione come questa?*

Algoritmo per la DCE

- Perché il nostro algoritmo non elimina nessuna istruzione?
 - Non è capace di identificare le dead stores
- *Come possiamo estendere l'algoritmo perché gestisca una situazione come questa?*
 - *Dovremmo essere in grado di rilevare i casi in cui assegniamo ad una variabile più volte prima di usarla*
 - *Più complicato, perché dobbiamo preoccuparci dell'ordine delle istruzioni*

Algoritmo per la DCE

- Dopo ogni istruzione teniamo traccia delle variabili che sono state definite, ma non usate
- Se vediamo un altro assegnamento alla stessa variabile prima di aver raggiunto la fine del blocco sappiamo che l'istruzione prima può essere eliminata

Algoritmo per la DCE

- Possibile pseudocodice

```
last_def = {};    // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

Algoritmo per la DCE

- Possibile pseudocodice

**Lista di variabili definite ma non usate
(puntatore alla definizione più recente
per le sole variabili mai usate)**

```
last_def = {}; // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

Algoritmo per la DCE

Controlliamo prima gli usi,
poi le definizioni



- Possibile pseudocodice

```
last_def = {};    // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

Es. in una situazione tipo
 $x = 2$
 $x = x + 5$
Voglio evitare che la
seconda istruzione venga
eliminata perché non ho
ancora realizzato che **x** è
usata

Algoritmo per la DCE

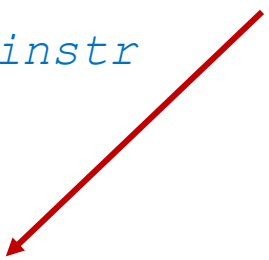
- Possibile pseudocodice

Gestione delle strutture dati:
rimuovo da *last_def* ogni argomento
(operando) dell'istruzione corrente (se
presente, è un uso della variabile)

```
last_def = {};    // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```



Algoritmo per la DCE

- Possibile pseudocodice

A questo punto, se l'argomento destinazione dell'istruzione corrente è presente in *last_def* posso eliminare l'istruzione con la precedente definizione

```
last_def = {};    // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

Ad esempio:
***x* = 2**
...
***x* = 5**



Algoritmo per la DCE

- Possibile pseudocodice

A questo punto, se l'argomento destinazione dell'istruzione corrente è presente in *last_def* posso eliminare l'istruzione con la precedente definizione

```
last_def = {};    // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

Ad esempio:

***x* = 2**

...

***x* = 5**

instr



Algoritmo per la DCE

- Possibile pseudocodice

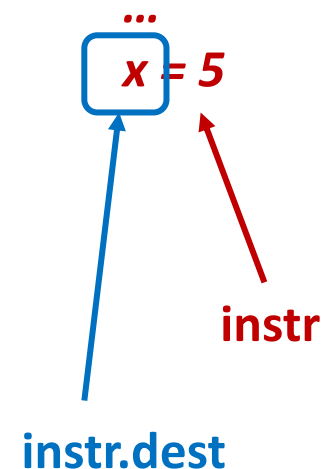
```
last_def = {};    // var → instr

for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

A questo punto, se l'argomento destinazione dell'istruzione corrente è presente in *last_def* posso eliminare l'istruzione con la precedente definizione

**Ad esempio:
x = 2**



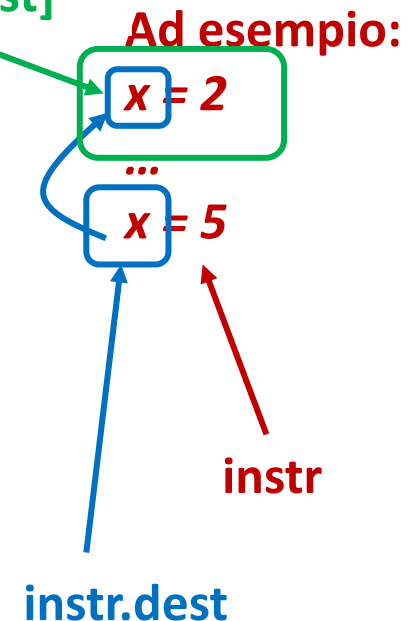
Algoritmo per la DCE

- Possibile pseudocodice

```
last_def = {}; // var → instr |
                last_def[instr.dest]
for instr in BB:
    // Check for uses
    last_def -= instr.args;

    // Check for definitions
    if instr.dest in last_def:
        delete last_def[instr.dest];
    last_def[instr.dest] = instr;
```

A questo punto, se l'argomento destinazione dell'istruzione corrente è presente in *last_def* posso eliminare l'istruzione con la precedente definizione



Algoritmo per la DCE

- Possibile pseudocodice

```
last_def = {};    // var → instr
```

```
for instr in BB:
```

```
    // Check for uses
```

```
    last_def -= instr.args;
```


```
    // Check for definitions
```

```
    if instr.dest in last_def:
```

```
        delete last_def[instr.dest];
```

```
    last_def[instr.dest] = instr;
```

**Gestione delle strutture dati:
aggiorno in *last_def* l'istruzione
che contiene la definizione più
recente di *instr.dest***



Algoritmo per la DCE

- Come per l'esempio precedente, abbiamo bisogno di ripetere iterativamente questo algoritmo fino a convergenza
 - Ovvero finché non ci sono più cambiamenti al BB



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Local Value Numbering

Diverse forme di ridondanza

- Estendiamo il nostro esempio a diverse forme di ottimizzazione locale. Cos'hanno in comune i problemi?

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

Dead Code Elimination

```
main {  
    int x = 4;  
    int copy1 = x;  
    int copy2 = copy1;  
    int copy3 = copy2;  
    print copy3;  
}
```

Copy Propagation

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

Common Subexpression Elimination

Diverse forme di ridondanza

- Sono tre diverse forme di ridondanza
- Qual è il fenomeno comune che rende questi programmi ridondanti?

Diverse forme di ridondanza

- Sono tre diverse forme di ridondanza
- Qual è il fenomeno comune che rende questi programmi ridondanti?
- *Il motivo per cui questi tre programmi hanno ridondanza è che la computazione si focalizza sulle **variabili***
- *Se ci focalizziamo, invece, sui **valori** possiamo eliminare tutte e tre le forme di ridondanza*

Diverse forme di ridondanza

- Estendiamo il nostro esempio a diverse forme di ottimizzazione locale. Cos'hanno in comune i problemi?

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

una variabile (pointing to the first `int a`)
due valori (bracketing the two assignments)

Dead Code Elimination

```
main {  
    int x = 4;  
    int copy1 = x;  
    int copy2 = copy1;  
    int copy3 = copy2;  
    print copy3;  
}
```

quattro variabili (bracketing the four `int` declarations)
un valore (pointing to the value `4` in the first assignment)

Copy Propagation

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

un valore (pointing to the value `4` in the first assignment)
due variabili (bracketing the two `sum` declarations)

Common Subexpression Elimination

Local Value Numbering

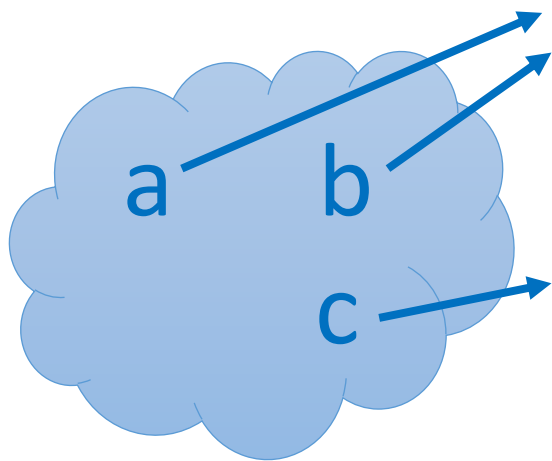
- La tecnica del *Local Value Numbering* (LVN) ci aiuta a reimpostare diversi problemi di ottimizzazione con un focus sui loro valori, piuttosto che sulle variabili
- L'idea si basa sulla costruzione di una struttura dati (metadato) di tipo tabella che ci aiuta a riscrivere le espressioni (istruzioni) in funzione dei valori già osservati
- Evitando di riassegnare lo stesso valore a più variabili si elimina la ridondanza

Local Value Numbering

Build a table to track unique canonical sources for every value we compute

One place and one place only where that value is stored

Step through the code, keeping track of the value number for each variable at a given point in time



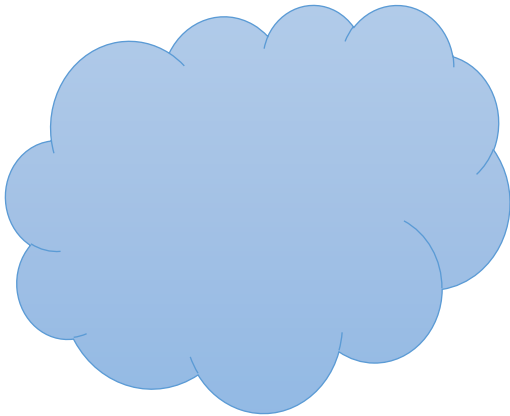
#	CANONICAL	
	VALUE (EXPRESSION)	VARIABLE NAME
1		x
2	#1 + #1	a
3	#2 * #1	d
4	#3 + #2	c

Local Value Numbering

- Proviamo a costruire la tabella per il nostro esempio di CSE

```
main {
    int a = 4;
    int b = 2;
    int sum1 = a + b;
    int sum2 = a + b;
    int prod = sum1 * sum2;
    print prod;
}
```

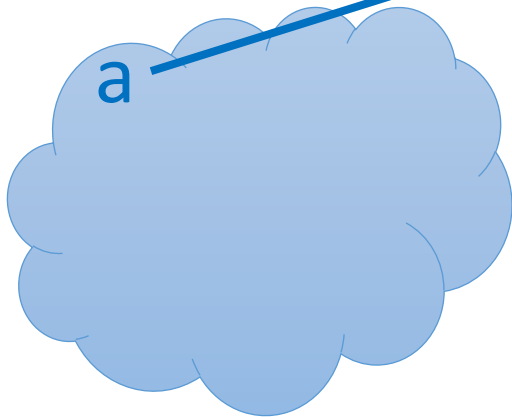
#	VALUE (EXPRESSION)	CANONICAL VARIABLE NAME



Local Value Numbering

- Proviamo a costruire la tabella per il nostro esempio di CSE

```
main {  
  int a = 4;  
  int b = 2;  
  int sum1 = a + b;  
  int sum2 = a + b;  
  int prod = sum1 * sum2;  
  print prod;  
}
```

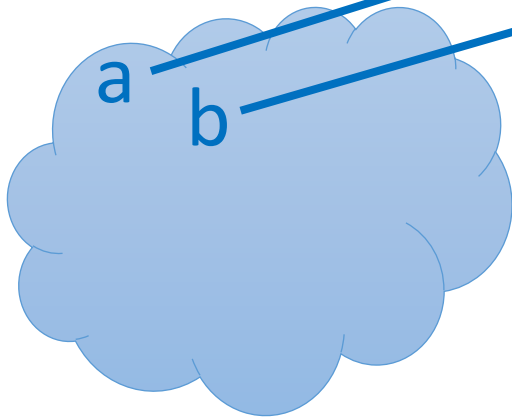


#	VALUE (EXPRESSION)	CANONICAL VARIABLE NAME
1	4	a

Local Value Numbering

- Proviamo a costruire la tabella per il nostro esempio di CSE

```
main {  
  int a = 4;  
  int b = 2;  
  int sum1 = a + b;  
  int sum2 = a + b;  
  int prod = sum1 * sum2;  
  print prod;  
}
```

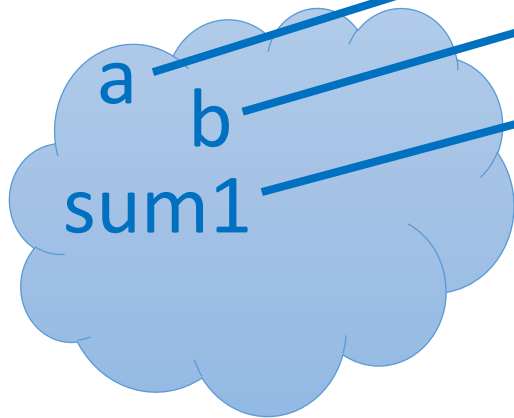


#	VALUE (EXPRESSION)	CANONICAL VARIABLE NAME
1	4	a
2	2	b

Local Value Numbering

- Proviamo a costruire la tabella per il nostro esempio di CSE

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

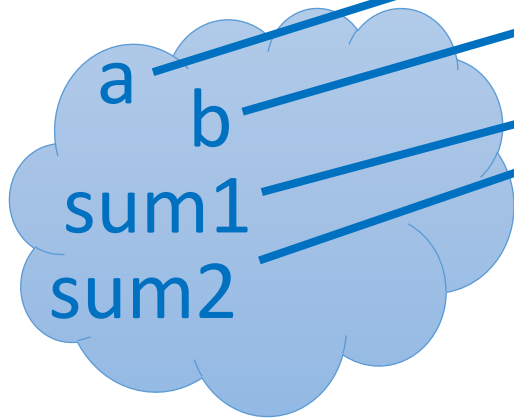


#	VALUE	CANONICAL
	(EXPRESSION)	VARIABLE NAME
1	4	a
2	2	b
3	#1 + #2	sum1

Local Value Numbering

- Proviamo a costruire la tabella per il nostro esempio di CSE

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

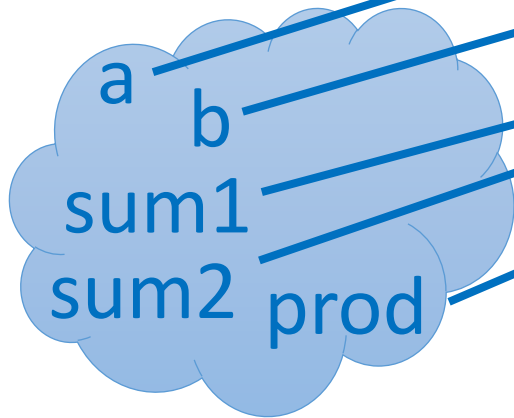


#	VALUE	CANONICAL
	(EXPRESSION)	VARIABLE NAME
1	4	a
2	2	b
3	#1 + #2	sum1

Local Value Numbering

- Proviamo a costruire la tabella per il nostro esempio di CSE

```
main {  
  int a = 4;  
  int b = 2;  
  int sum1 = a + b;  
  int sum2 = a + b;  
  int prod = sum1 * sum2;  
  print prod;  
}
```



#	VALUE	CANONICAL
	(EXPRESSION)	VARIABLE NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#3 * #3	prod

Local Value Numbering

- A questo punto possiamo usare la tabella per ricostruire le istruzioni del programma

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

Programma originale

#	VALUE	CANONICAL VARIABLE
	(EXPRESSION)	NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#3 * #3	prod

Local Value Numbering

- A questo punto possiamo usare la tabella per ricostruire le istruzioni del programma

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

Programma originale

```
main {  
    int a = 4;
```

Programma ottimizzato

#	VALUE	CANONICAL
	(EXPRESSION)	VARIABLE NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#3 * #3	prod

Local Value Numbering

- A questo punto possiamo usare la tabella per ricostruire le istruzioni del programma

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

Programma originale

```
main {  
    int a = 4;  
    int b = 2;
```

Programma ottimizzato

#	VALUE	CANONICAL
	(EXPRESSION)	VARIABLE NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#3 * #3	prod

Local Value Numbering

- A questo punto possiamo usare la tabella per ricostruire le istruzioni del programma

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

Programma originale

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
}
```

Programma ottimizzato

#	VALUE	CANONICAL VARIABLE
	(EXPRESSION)	NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#3 * #3	prod

Local Value Numbering

- A questo punto possiamo usare la tabella per ricostruire le istruzioni del programma

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = a + b;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

Programma originale

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int prod = sum1 * sum1;  
    ...  
}
```

Programma ottimizzato

#	VALUE	CANONICAL VARIABLE
	(EXPRESSION)	NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#3 * #3	prod

Local Value Numbering

- Consideriamo questa semplice variante del nostro programma

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = b + a;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

- Cosa succede?
- E cosa dovrebbe succedere invece?

Local Value Numbering

- Il nostro algoritmo non sarebbe in grado di eliminare l'istruzione che assegna a `sum2`, perché non è a conoscenza della proprietà commutativa dell'addizione.

```
main {  
    int a = 4;  
    int b = 2;  
    int sum1 = a + b;  
    int sum2 = b + a;  
    int prod = sum1 * sum2;  
    print prod;  
}
```

#	VALUE (EXPRESSION)	CANONICAL VARIABLE NAME
1	4	a
2	2	b
3	#1 + #2	sum1
4	#2 + #1	sum2
5	#3 * #3	prod

- Posso **canonicalizzare** l'algoritmo perché usi sempre i valori in ordine numerico crescente per le operazioni commutative