

Fondamenti di Programmazione (A)

Lecture 8 - Assegnamenti ed espressioni

Vincenzo Arceri - Università di Parma - vincenzo.arceri@unipr.it

Puntate precedenti

- Assegnamento
- Espressioni
- Tipo di un'espressione
- Precedenza e associatività degli operatori

Espressioni booleane

- Espressioni di tipo `bool`

Operatori relazionali: `==` `!=` `>` `<` `>=` `<=` `bool`

Operatori booleani: `&&` `||` `!` `bool`

- Mentre la semantica degli operatori relazionali è intuitiva, quella degli operatori booleani non è standard

Espressioni booleane

&& e ||

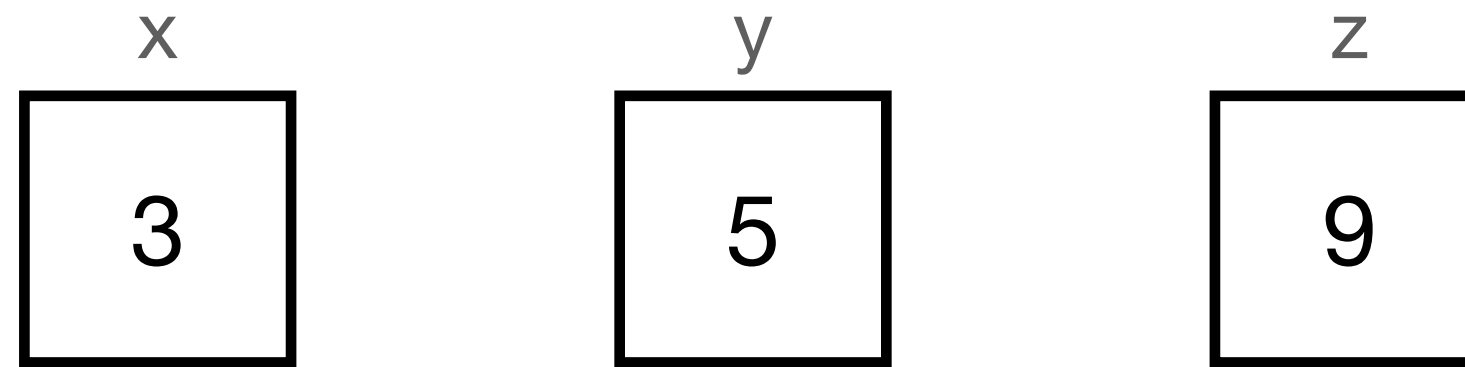
- $exp_1 \& \& exp_2$
 - ◆ sia exp_1 che exp_2 devono valutare a true (entrambe)
- $exp_1 || exp_2$
 - ◆ exp_1 oppure exp_2 deve valutare a true (una delle due)

&& (and)	true	false
true	true	false
false	false	false

 (or)	true	false
true	true	true
false	true	false

Espressioni booleane

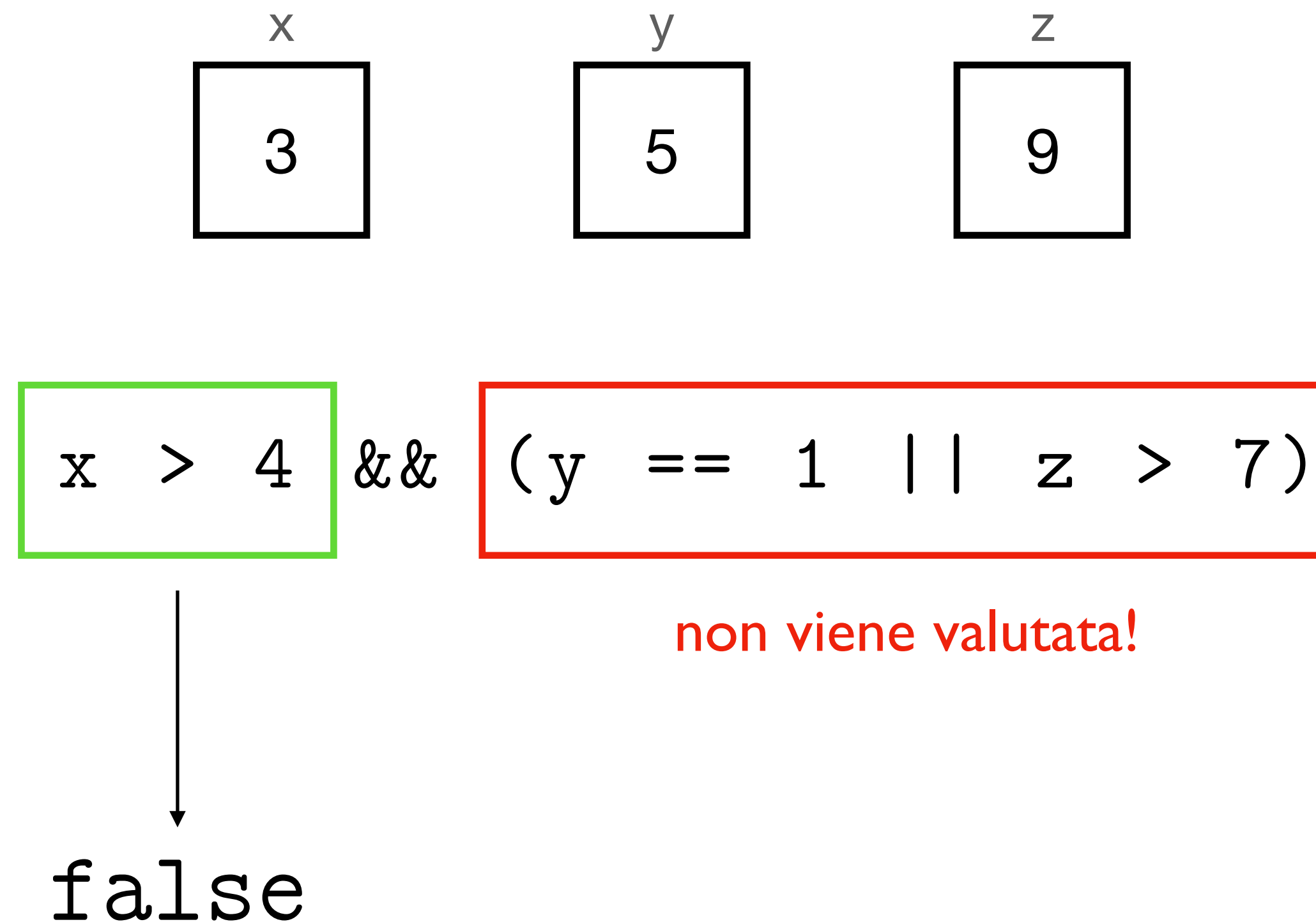
Valutazione *lazy/short-circuit*



`x > 4 && (y == 1 || z > 7)`

Espressioni booleane

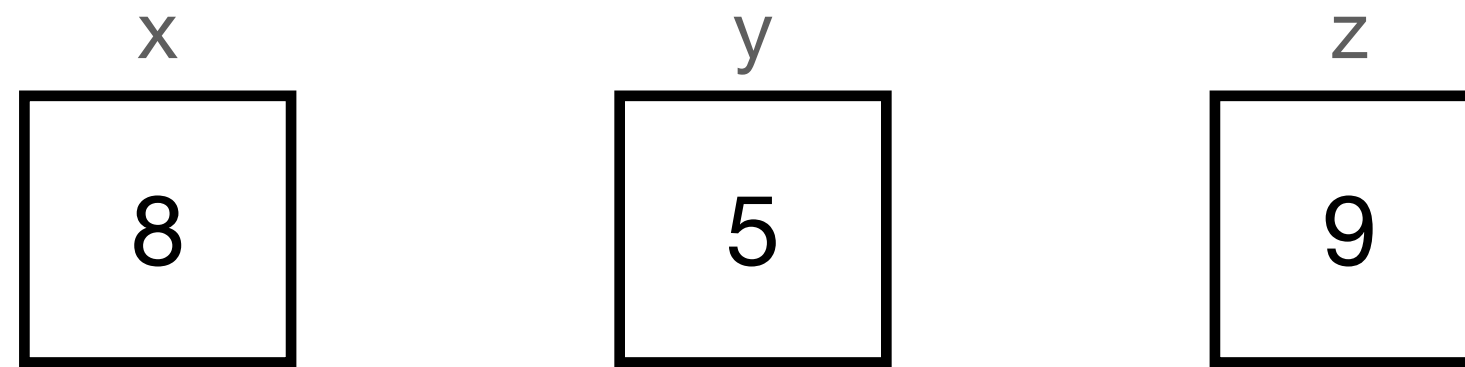
Valutazione *lazy/short-circuit*



- **Valutazione *lazy*:** dopo la valutazione della prima espressione (`x > 4`) posso già determinare il valore dell'espressione intera (`&&`), senza valutare il resto dell'espressione

Espressioni booleane

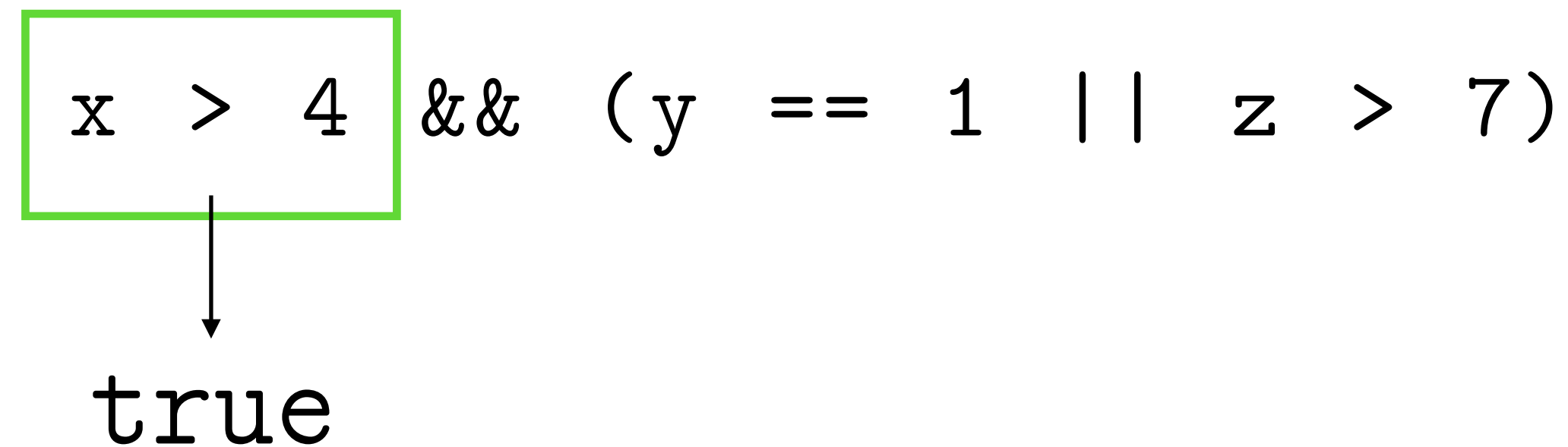
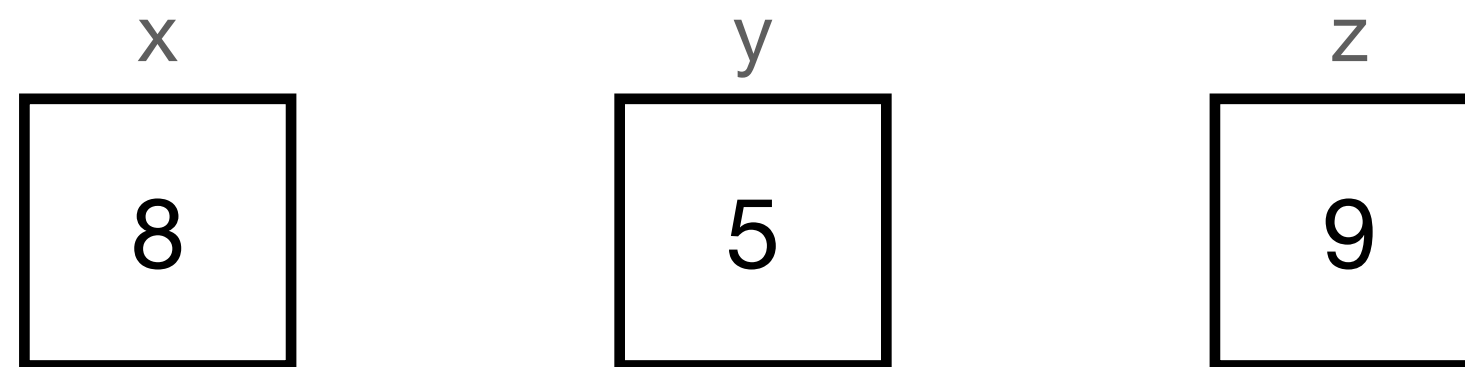
Valutazione *lazy/short-circuit*



`x > 4 && (y == 1 || z > 7)`

Espressioni booleane

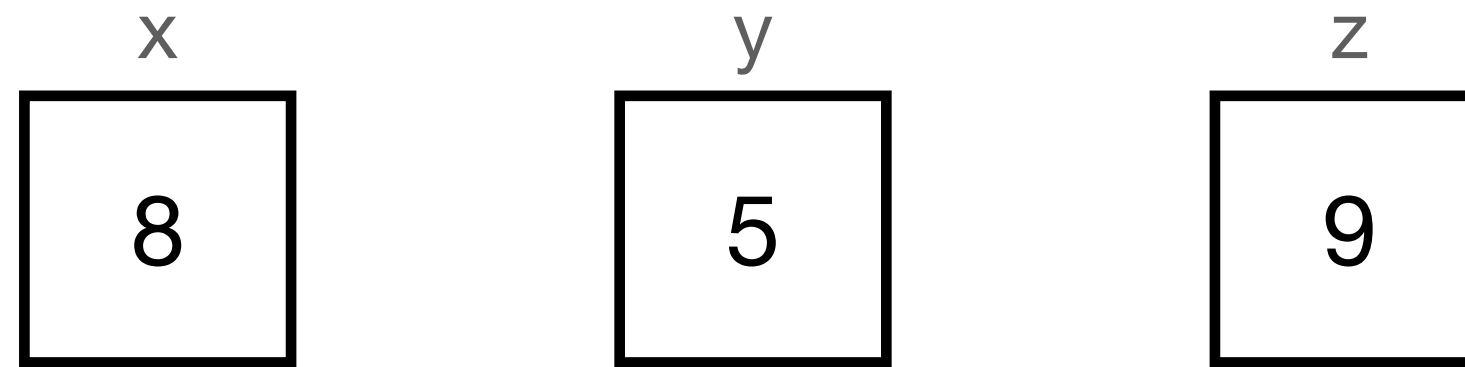
Valutazione *lazy/short-circuit*



- Dopo la valutazione della prima espressione (`x > 4`) **non** sono ancora in grado di determinare il valore dell'espressione intera (`&&`), devo necessariamente valutare il resto

Espressioni booleane

Valutazione *lazy/short-circuit*



`x > 4 &&`

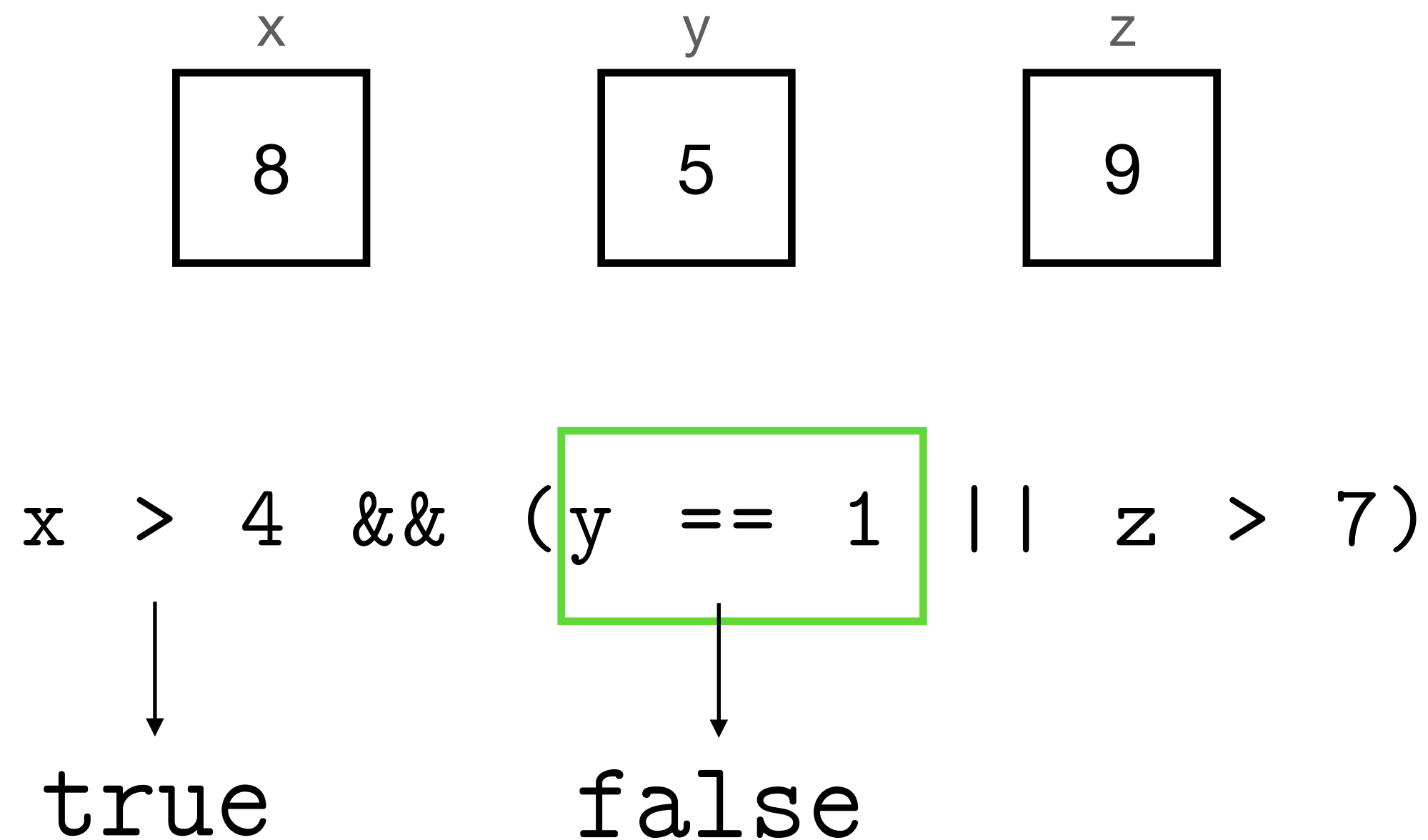


`true`

`(y == 1 || z > 7)`

Espressioni booleane

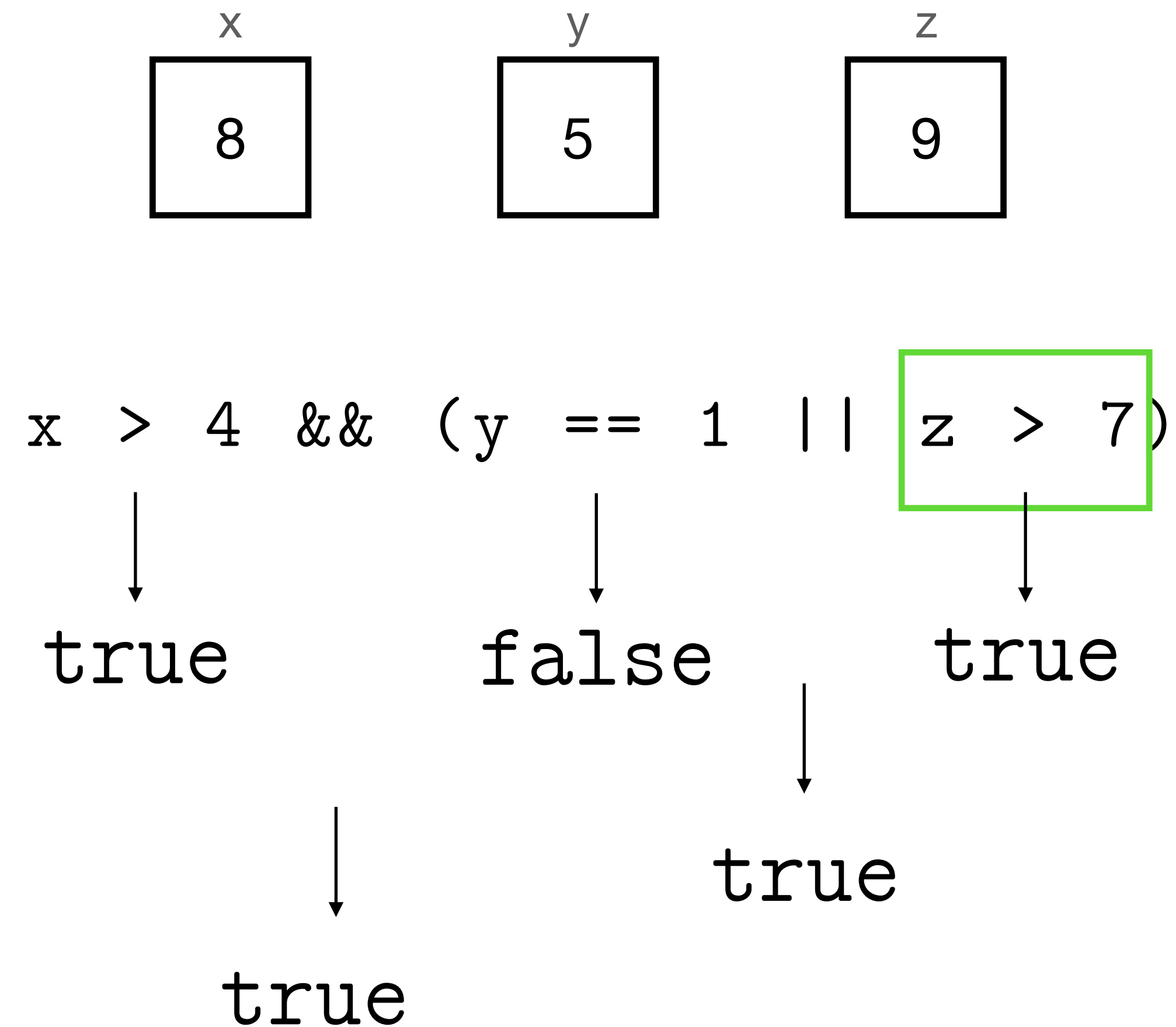
Valutazione *lazy/short-circuit*



- Dopo la valutazione dell'espressione `y == 1` **non** sono ancora in grado di determinare il valore dell'espressione intera, devo necessariamente valutare il resto

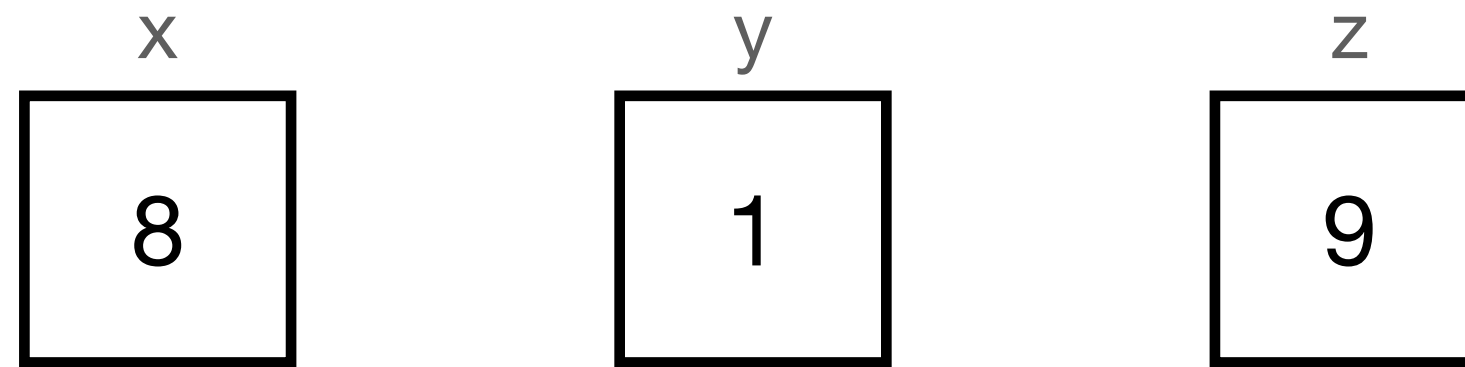
Espressioni booleane

Valutazione *lazy/short-circuit*



Espressioni booleane

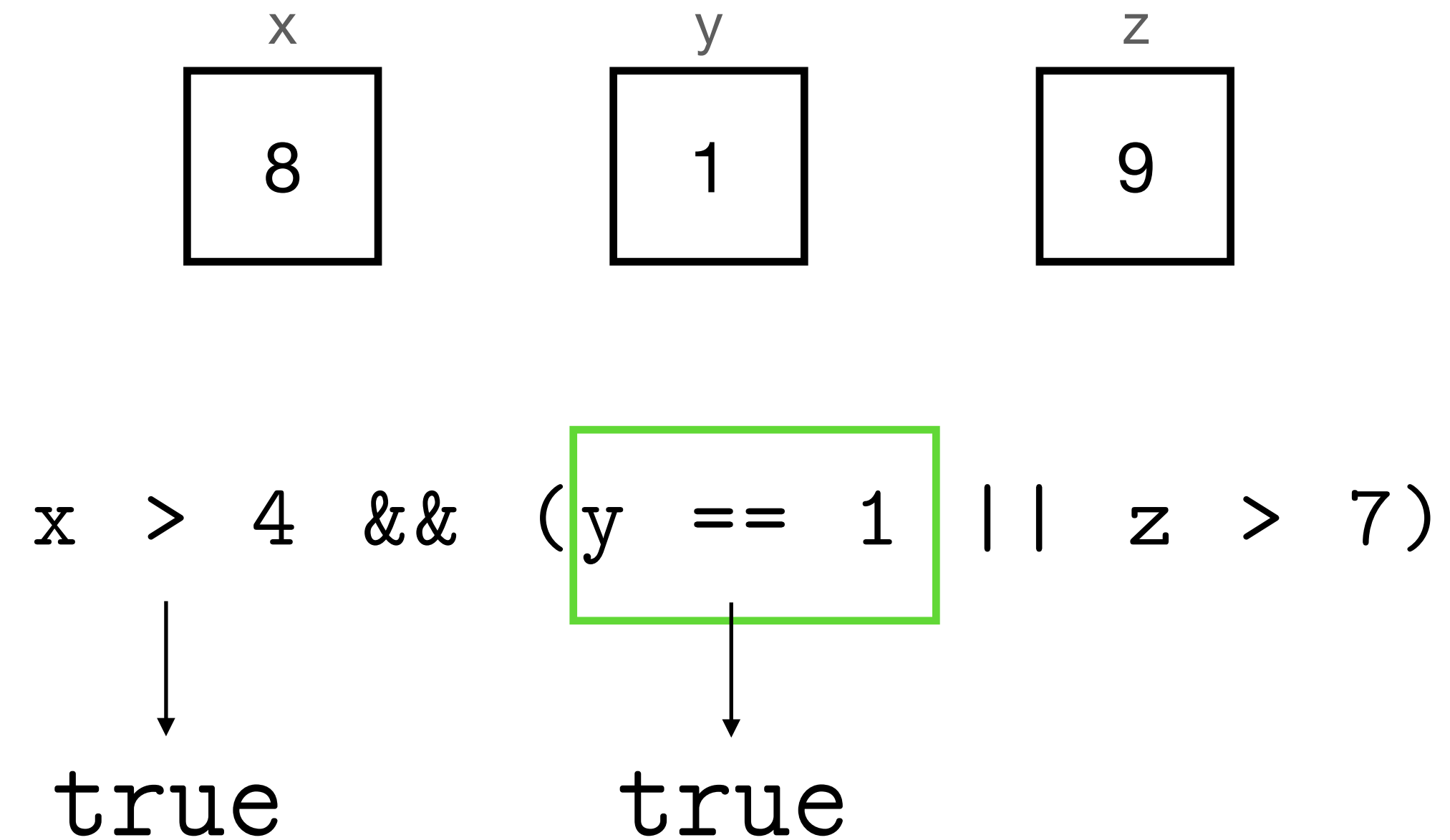
Valutazione *lazy/short-circuit*



`x > 4 && (y == 1 || z > 7)`

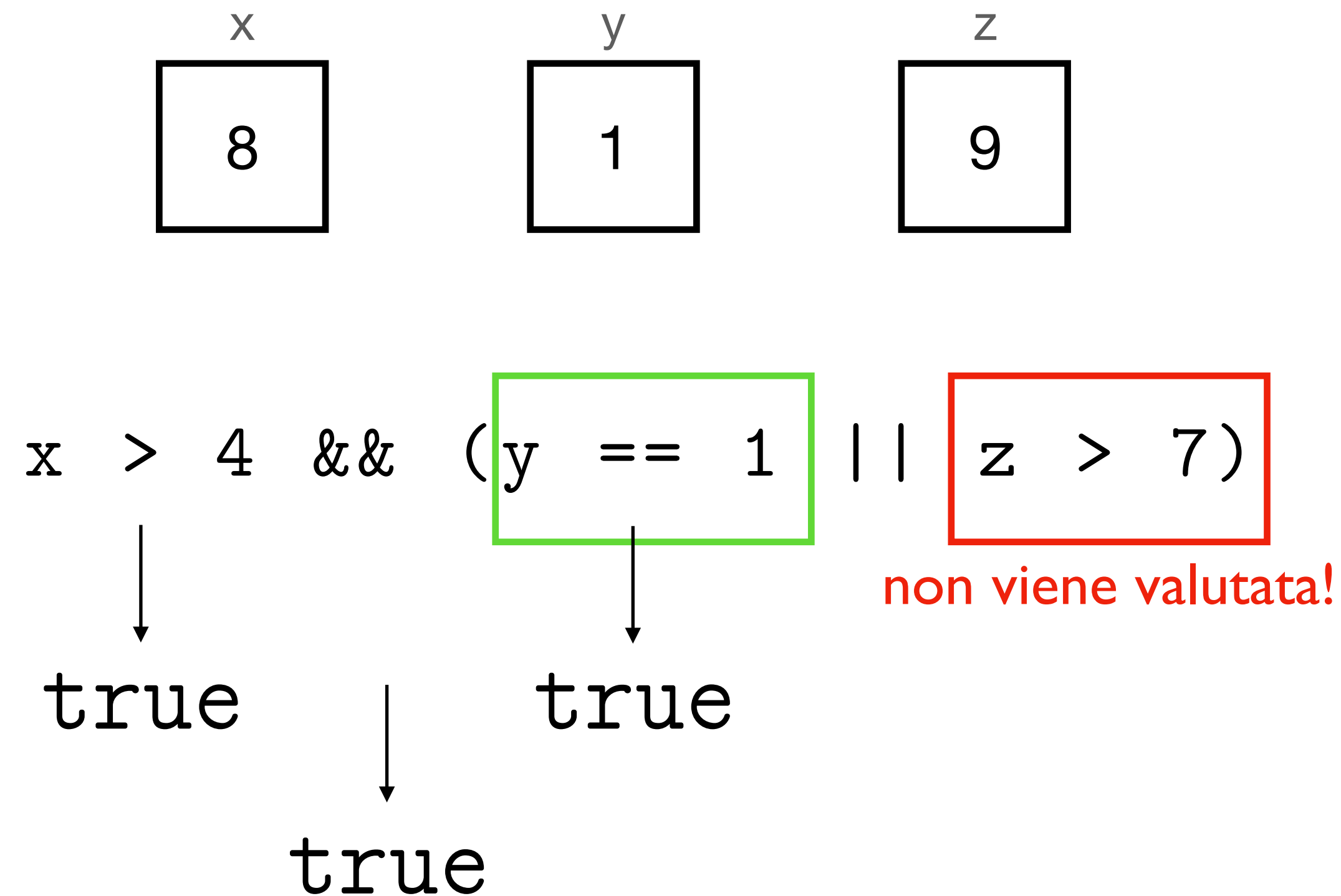
Espressioni booleane

Valutazione *lazy/short-circuit*



Espressioni booleane

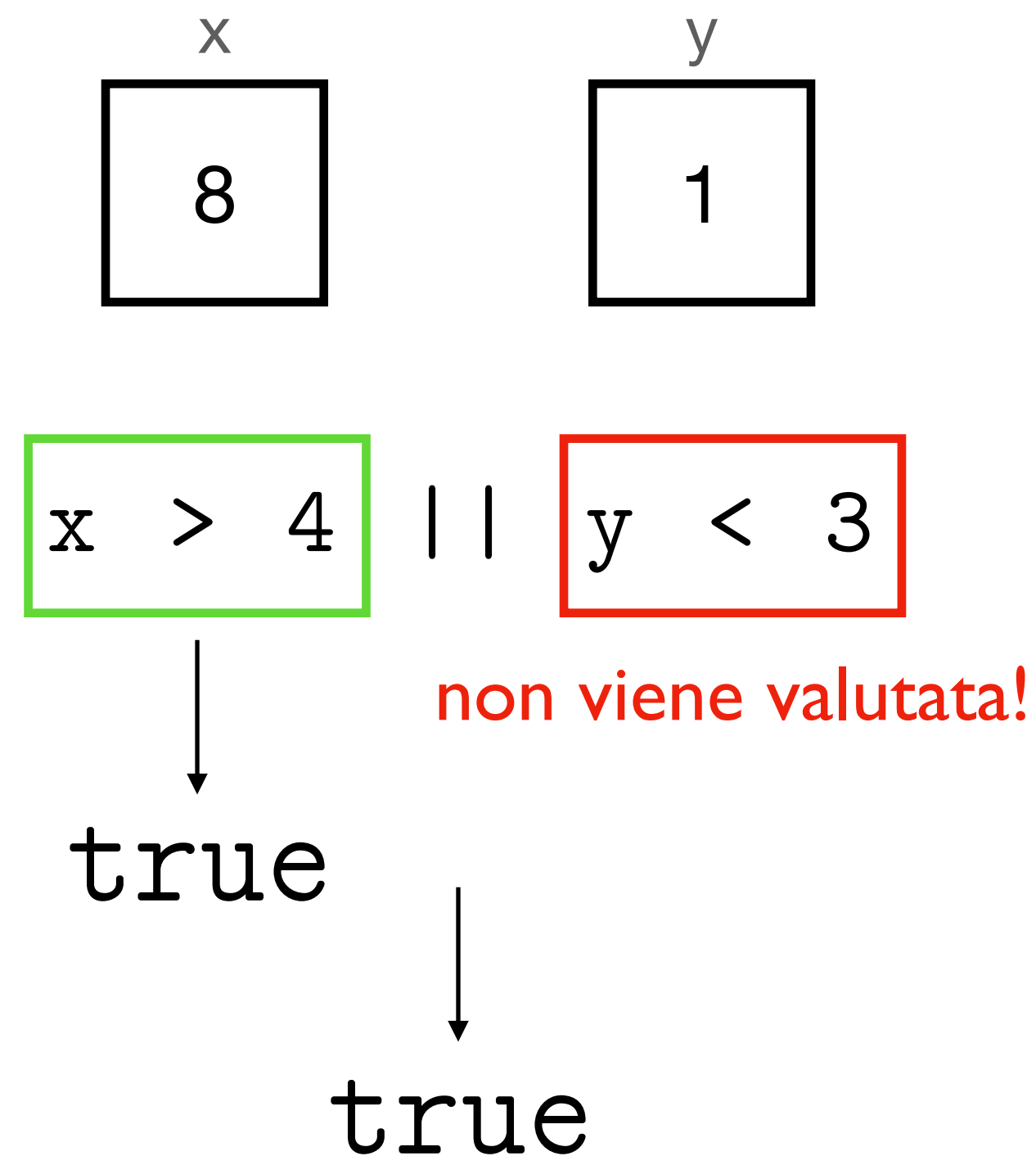
Valutazione *lazy/short-circuit*



- **Valutazione *lazy*:** dopo la valutazione dell'espressione `y == 1` sono già in grado determinare il valore dell'espressione intera (`||`), senza valutare il resto dell'espressione

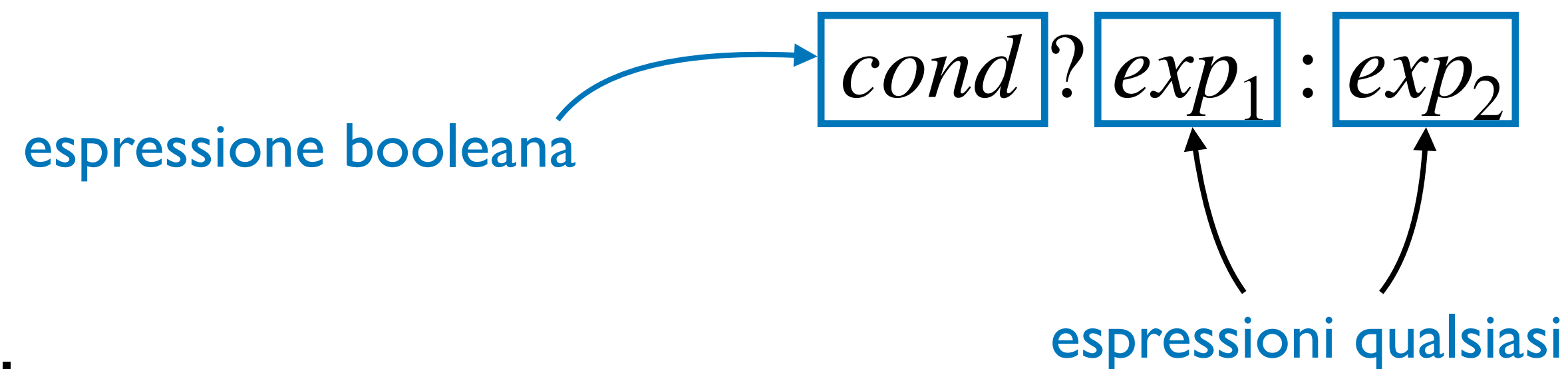
Espressioni booleane

Valutazione *lazy/short-circuit*



- **Valutazione *lazy*:** dopo la valutazione dell'espressione `x > 4` sono già in grado di determinare il valore dell'espressione intera (`||`), senza valutare il resto dell'espressione

Espressione condizionale



- Informalmente
 - ◆ $cond$ è vera? Allora valuta exp_1 e il valore dell'espressione condizionale sarà quello di exp_1
 - ◆ $cond$ è falsa? Allora valuta exp_2 e il valore dell'espressione condizionale sarà quello di exp_2
- Unico operatore ternario di C++

Espressione condizionale

Esempio

- **Problema:** dato in input un numero intero, stampare a video il suo valore assoluto

Assegnamento

Side-effects

left_expression = right_expression

- *left_expression*: espressione che denota una locazione di memoria di cui vogliamo aggiornarne il contenuto
- *right_expression*: espressione il cui risultato della valutazione deve essere assegnato a *left_expression*
- L'assegnamento agisce tramite **side-effects**: la sua esecuzione ha un effetto che persiste dopo la sua esecuzione (modifica il valore di *left_expression*)

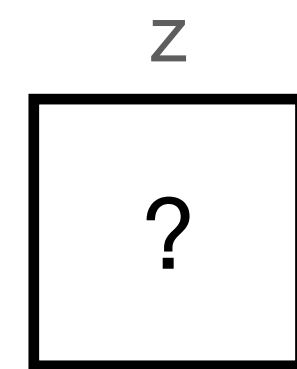
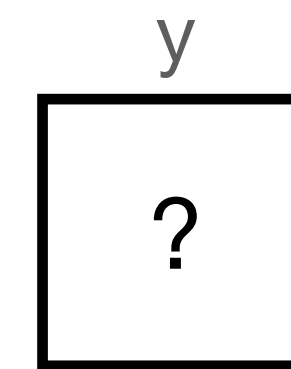
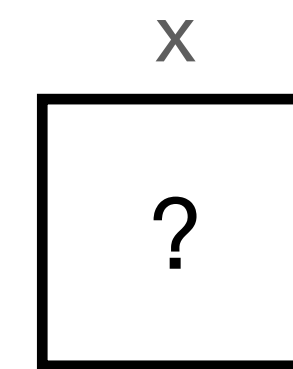
L'assegnamento è un'espressione!

Assegnamento

Side-effects e valore dell'assegnamento

- Il risultato della valutazione dell'assegnamento è il valore assegnato alla variabile

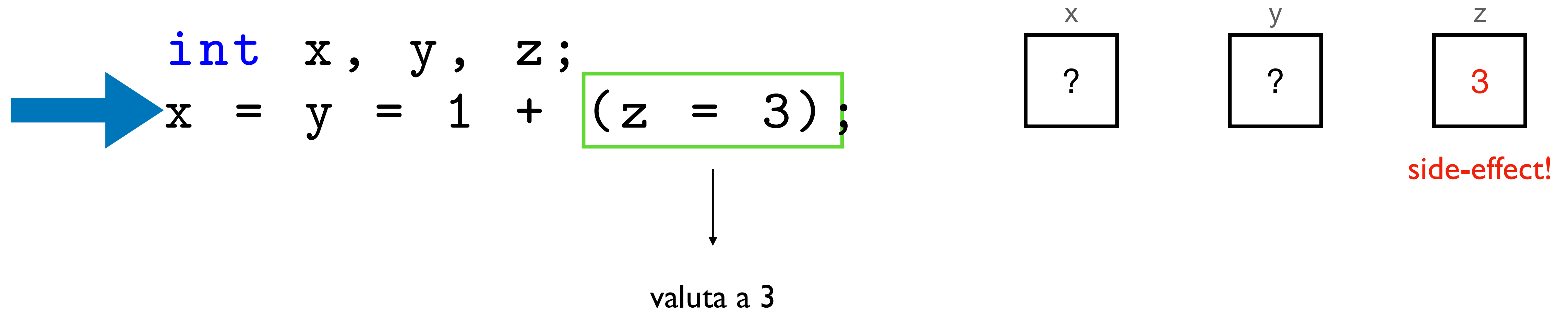
→ `int x, y, z;`
`x = y = 1 + (z = 3);`



Assegnamento

Side-effects e valore dell'assegnamento

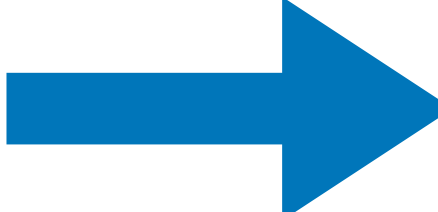
- Il risultato della valutazione dell'assegnamento è il valore assegnato alla variabile



Assegnamento

Side-effects e valore dell'assegnamento

- Il risultato della valutazione dell'assegnamento è il valore assegnato alla variabile

 `int x, y, z;`
`x = y = 1 + (z = 3);`

↓
valuta a 4

x
?

y
?

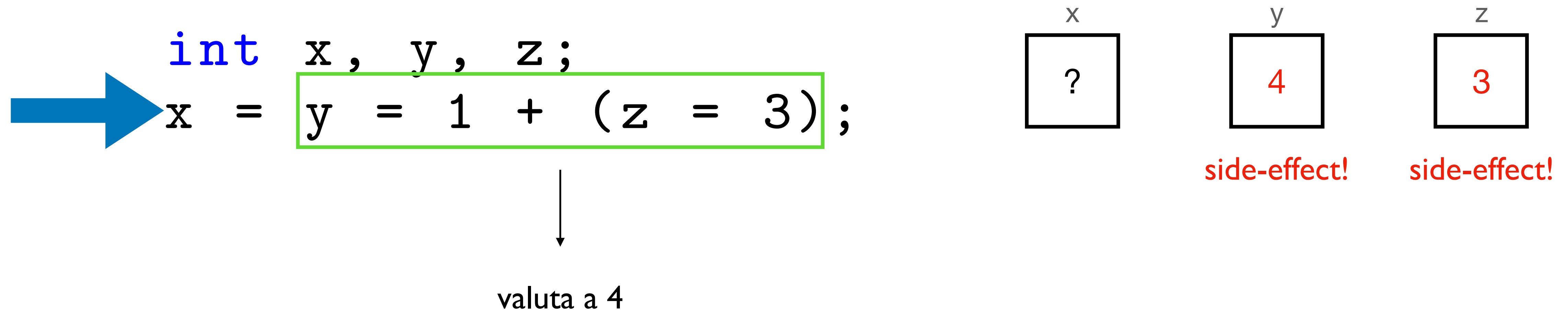
z
3

side-effect!

Assegnamento

Side-effects e valore dell'assegnamento

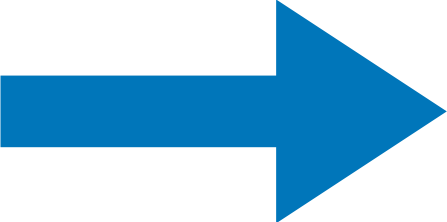
- Il risultato della valutazione dell'assegnamento è il valore assegnato alla variabile



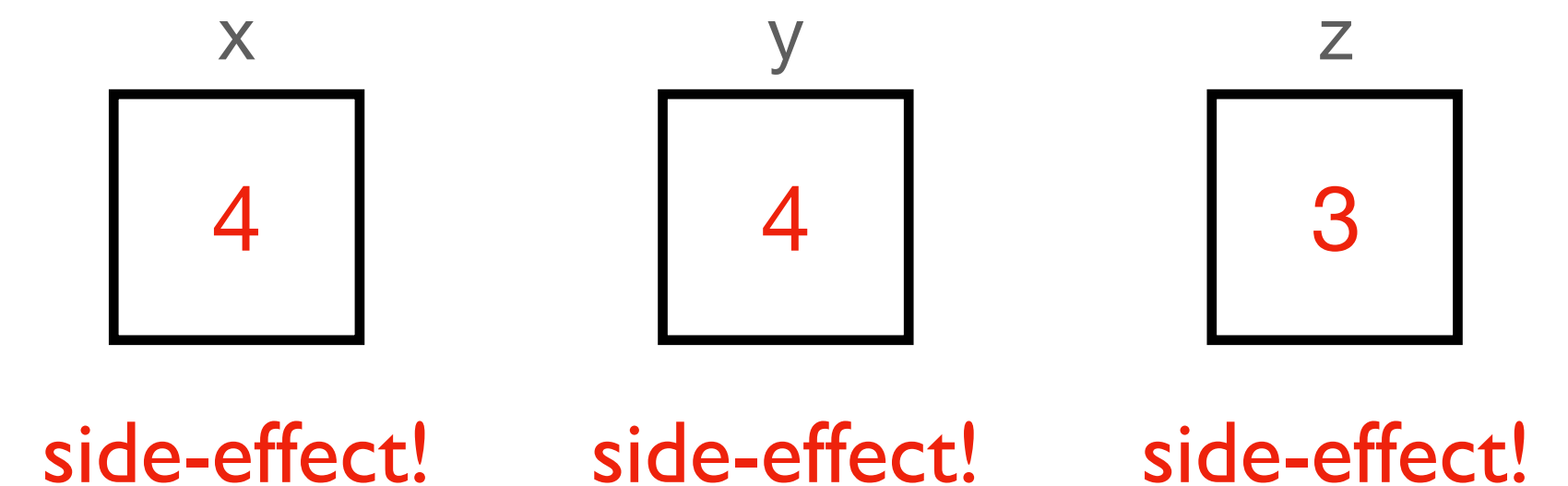
Assegnamento

Side-effects e valore dell'assegnamento

- Il risultato della valutazione dell'assegnamento è il valore assegnato alla variabile

 `int x, y, z;`
`x = y = 1 + (z = 3);`

↓
valuta a 4



Assegnamento

Associatività e tipo dell'assegnamento

- L'operatore di assegnamento è associativo a **destra**

$x = y = z = w = 1;$

come se fosse

$x = (y = (z = (w = 1))) ;$

- Il tipo dell'assegnamento è il tipo dell'espressione a sinistra (*left_expression*) e il tipo di *left_expression* e *right_expression* devono essere compatibili

Assegnamento

Tipo

```
int x, y, z;  
x = y = 1 + (z = 3);
```

int int

Assegnamento

Tipo

```
int x, y, z;  
x = y = 1 + (z = 3);
```

↓
int

Assegnamento

Tipo

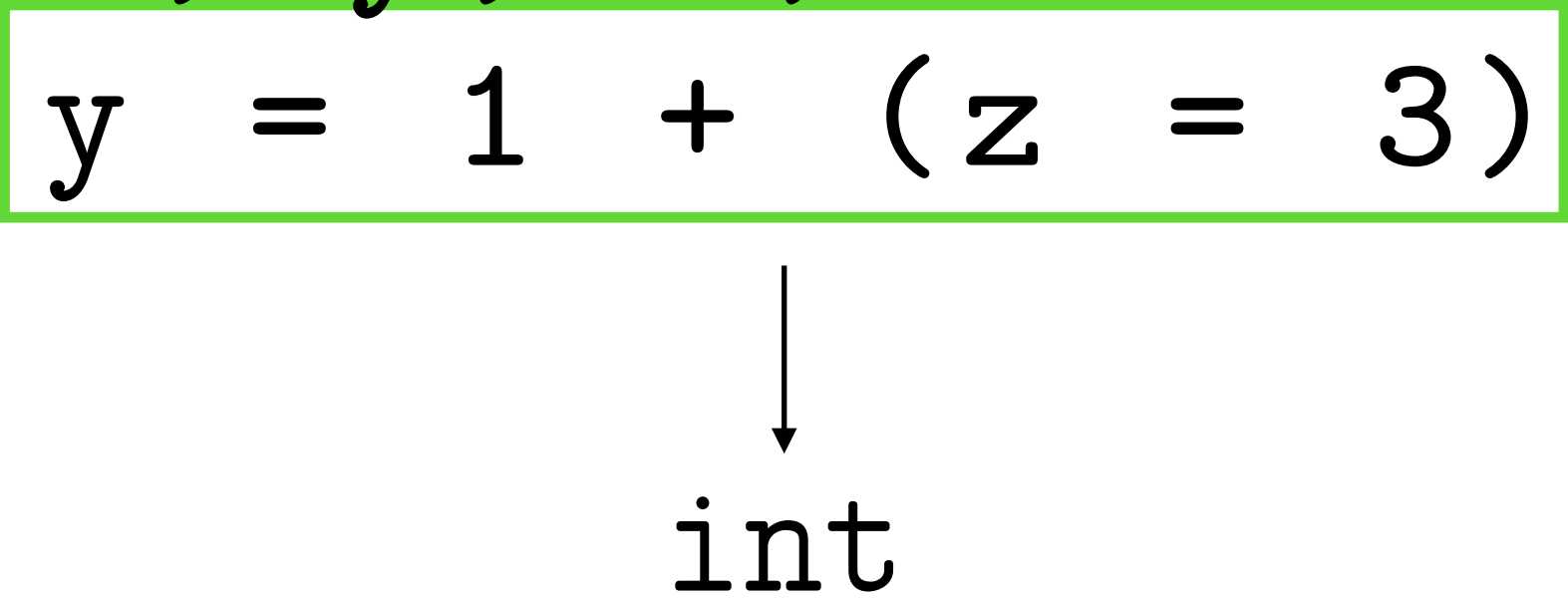
```
int x, y, z;  
x = y = 1 + (z = 3);
```

The diagram illustrates the type inference process for the assignment statement `x = y = 1 + (z = 3);`. The variable `y` is highlighted with a green box, and an arrow points from it to the text `int`, indicating its type. Similarly, the expression `1 + (z = 3)` is highlighted with a green box, and an arrow points from it to the text `int`, indicating its type. The variable `x` is also declared as `int` in the first line.

Assegnamento

Tipo

```
int x, y, z;  
x = y = 1 + (z = 3);
```



int

Assegnamento

Tipo

```
int x, y, z;  
x = y = 1 + (z = 3);
```

↓
int

Assegnamento

Conversioni e troncamenti

```
float f;  
f = 5
```

↓ ↓
float int

5 viene *implicitamente convertito* a float (5.0)

f

5.0

```
int i;  
i = 2.3;
```

↓ ↓
int float

2.3 viene *implicitamente convertito* a int (2)

Troncamento! Un float
non può essere
memorizzato dentro una
variabile int quindi devo
perdere informazione
(parte decimale del float)

i

2

Operatori di assegnamento

$left_expression = right_expression$

- Altri operatori di assegnamento

$left_expression\ op = right_expression$

Zuccherò sintattico!

- op può essere $+$ $-$ $*$ $\%$ $/$

$<<$ $>>$ $\&$ $|$ \wedge

operazioni *bit-wise* (non ancora viste)

Zucchero sintattico

- Costrutti sintattici che **non aumentano le funzionalità/espressività** del linguaggio ma sono modi alternativi (solitamente più facili/pratici/*dolci*) per scrivere il codice

left_expression op= right_expression

è del tutto equivalente a

left_expression = left_expression op right_expression

- Esempi
 - $x += 2$ è equivalente a $x = x + 2$
 - $y *= 3$ è equivalente a $y = y * 3$

Operatori di incremento/decremento

Incremento
 $++exp \quad exp++$

zucchero sintattico!

$exp = exp + 1$

Decremento
 $--exp \quad exp--$

zucchero sintattico!

$exp = exp - 1$

- $++$ e $--$ sono operatori di assegnamento, quindi exp deve denotare una locazione di memoria

$x++$



$(a + 1)++$



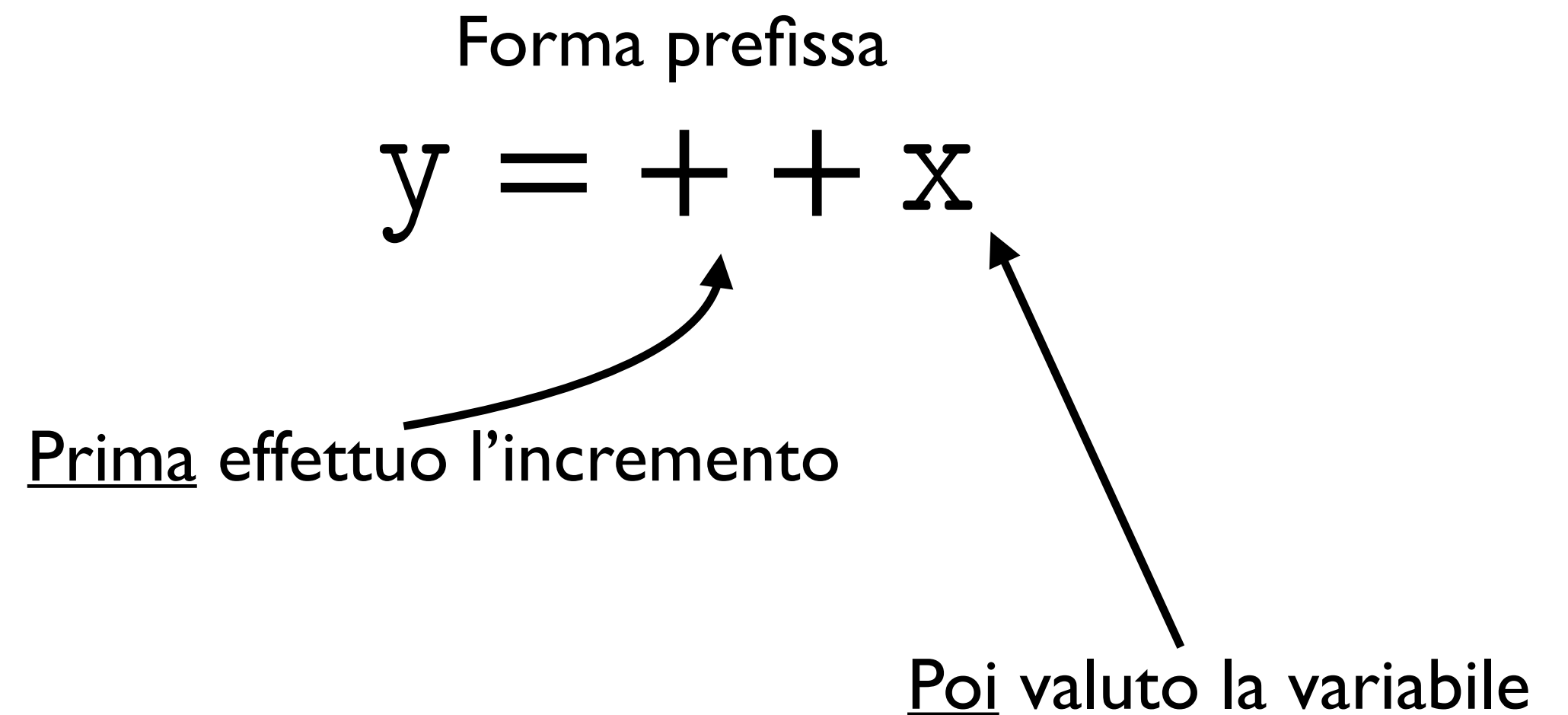
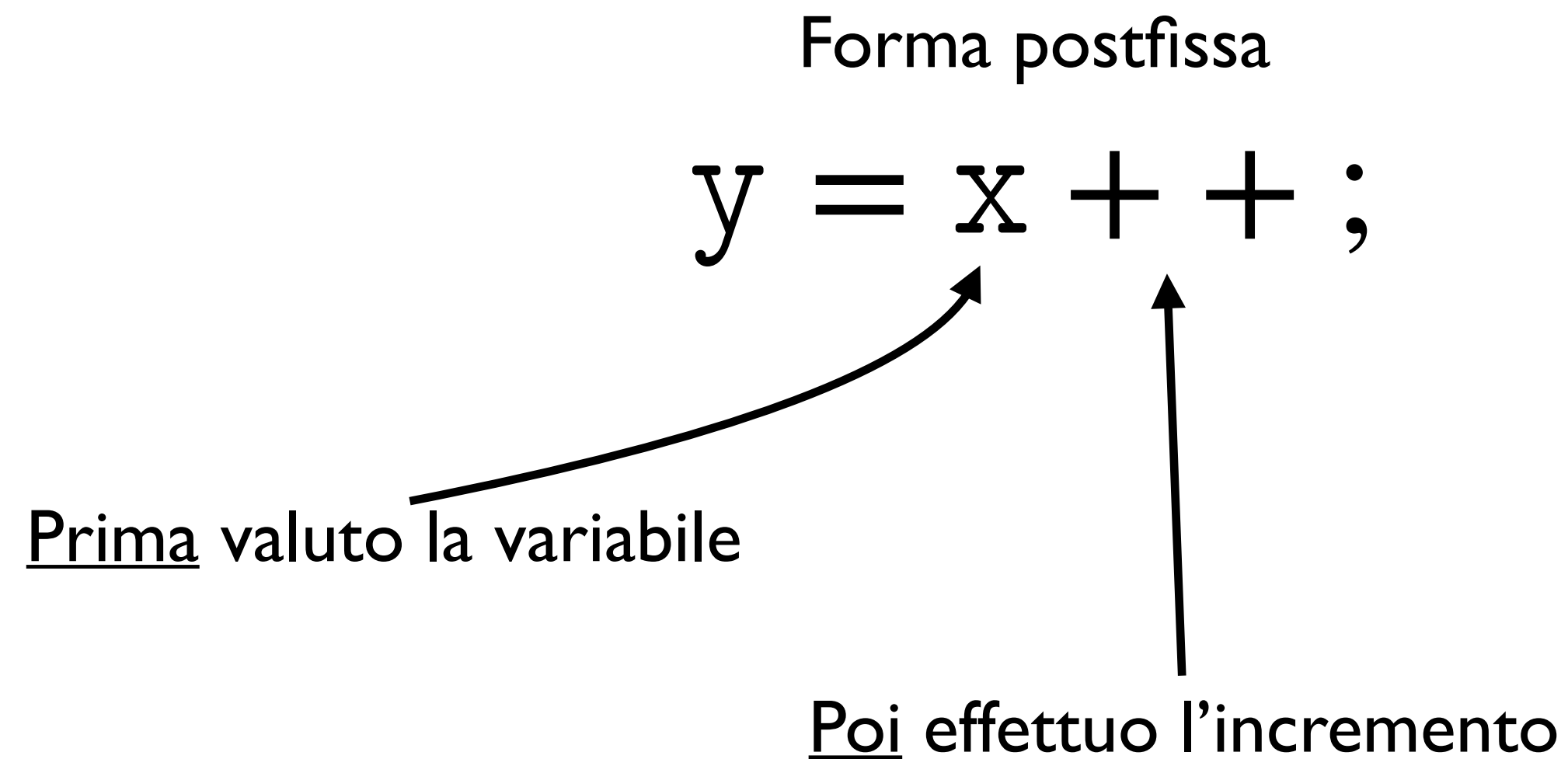
Operatori di incremento/decremento

Forma prefissa vs postfissa

- Se vengono usati come statement, non c'è differenza fra forma prefissa e suffissa

$X ++;$ equivalente a $++ X;$

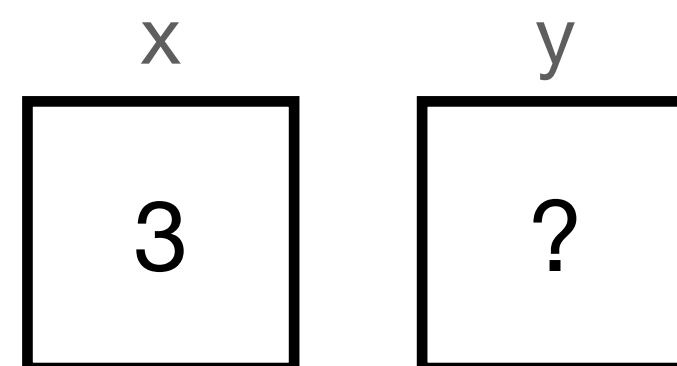
- Se vengono usati come espressioni, il risultato della valutazione dell'espressione è diverso



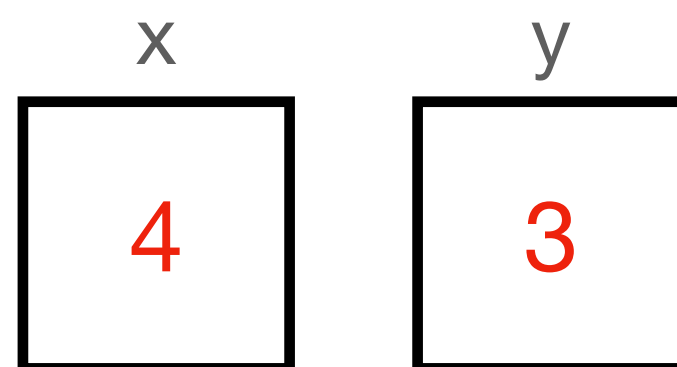
Operatori di incremento/decremento

Forma prefissa vs postfissa

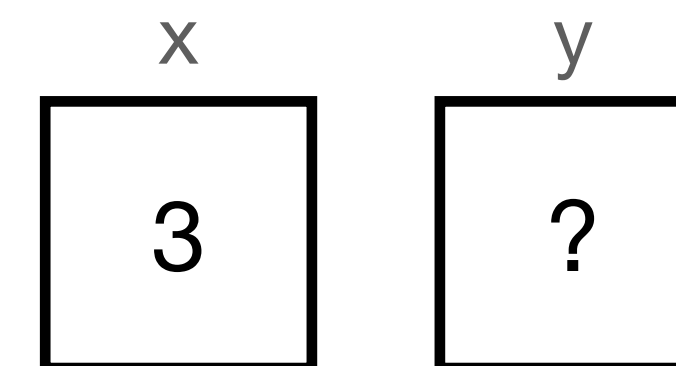
Forma postfissa



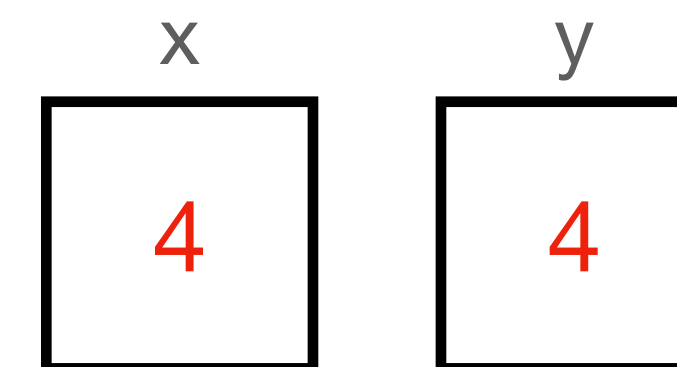
$y = x++;$



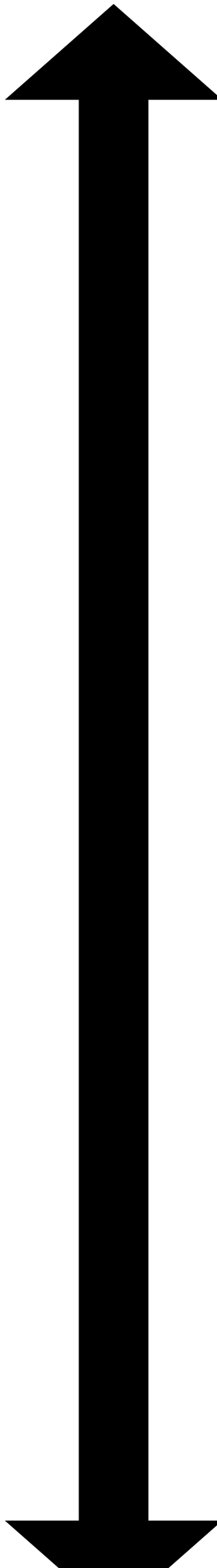
Forma prefissa



$y = ++x;$



Precedenza degli operatori (aggiornato)

Precedenza più alta		$++$	$--$	(postfissi)			
		$!$	$+$	$-$	$++$	$--$	(prefissi)
		$*$	$/$	$\%$			
		$+$	$-$				
		$<$	$>$	$< =$	$> =$		
		$= =$	$! =$				
		$\&\&$					
		$ $					
		$=$	$+=$	$- =$	$* =$	\dots	
Precedenza più bassa							

Associatività degli operatori d'assegnamento

$+$ $=$ $-$ $=$ $*$ $=$ \dots

associano da destra verso sinistra (right-to-left)

x $+$ $+$ x $-$ $-$

associano da sinistra verso destra (left-to-right)

$+$ $+$ x $-$ $-$ x

associano da destra verso sinistra (right-to-left)

Esercizio

- **Problema:** dato in input il raggio di un cerchio (in cm), calcolare e stampare a video la sua circonferenza e la sua area

$$A = r^2 \pi$$

$$C = 2r\pi$$

Costanti

via `#define`

```
#define s1 s2
```

- Associazione di un nome simbolico `s1` ad un valore `s2`
 - `s1` deve essere un identificatore
 - `s2` qualsiasi sequenze di caratteri

Costanti

via #define

#define s1 s2

- #define s1 s2 viene gestita dal **pre-processor** che si occuperà di rimpiazzare nel codice ogni occorrenza di s1 con s2

```
#include <iostream>
#define PI_GRECO 3.14159
using namespace std;
```

```
int main() {
    cout << "Inserire la lunghezza del raggio (cm): ";
    float r;
    cin >> r;
    float area = r * r * PI_GRECO;
    float circ = 2 * r * PI_GRECO;
    cout << "Circonferenza: " << circ << "cm" << endl;
    cout << "Area: " << area << "cm" << endl;
    return 0;
}
```

Replacement

```
#include <iostream>
#define PI_GRECO 3.14159
using namespace std;
```

```
int main() {
    cout << "Inserire la lunghezza del raggio (cm): ";
    float r;
    cin >> r;
    float area = r * r * 3.14159;
    float circ = 2 * r * 3.14159;
    cout << "Circonferenza: " << circ << "cm" << endl;
    cout << "Area: " << area << "cm" << endl;
    return 0;
}
```


Costanti

via `const` (dichiarazione di costante)

$$\text{const } \textit{type} \textit{id} = \textit{exp};$$

- Sintassi simile alla dichiarazione di variabile
 - ◆ *type* è un tipo
 - ◆ *id* è un identificatore
 - ◆ *exp* è un'espressione
- Crea un'associazione fra *id* e il valore di *exp*
- MA non può essere acceduta in scrittura (solo in lettura)

Costanti

#define vs const

#define

- Gestita dal preprocessore
- Effettua un replacement nel codice

const

- Gestita a *run-time*
- Dichiarata una variabile costante utilizzabile **solo in lettura**