

Fondamenti di Programmazione (A)

5 - Assegnamenti ed espressioni

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

Puntate precedenti

- Primo *statement*/comando: dichiarazione di variabile

```
int x, y = 1;  
float z;  
bool b = true;
```

- Tipi primitivi: `int`, `float`, `bool`, `char`
- Modificatori di tipo: `short`, `long` e `signed`, `unsigned`

Assegnamento

Assegnamento

- L'assegnamento è uno statement per settare/aggiornare il valore contenuto all'interno di una variabile (più in generale una locazione di memoria)

Assegnamento

- L'assegnamento è uno statement per settare/aggiornare il valore contenuto all'interno di una variabile (più in generale una locazione di memoria)
- In C++

Assegnamento

- L'assegnamento è uno statement per settare/aggiornare il valore contenuto all'interno di una variabile (più in generale una locazione di memoria)
- In C++

left_expression = right_expression

Assegnamento

- L'assegnamento è uno statement per settare/aggiornare il valore contenuto all'interno di una variabile (più in generale una locazione di memoria)
- In C++

left_expression = right_expression

- *left_expression*: espressione che denota una locazione di memoria di cui vogliamo aggiornarne il contenuto

Assegnamento

- L'assegnamento è uno statement per settare/aggiornare il valore contenuto all'interno di una variabile (più in generale una locazione di memoria)
- In C++

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*

Assegnamento

- L'assegnamento è uno statement per settare/aggiornare il valore contenuto all'interno di una variabile (più in generale una locazione di memoria)
- In C++

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*

“valuta *right_expression*, prendi il suo risultato e assegnalo a *left_expression*”

Assegnamento

Assegnamento

```
int x , y , z ;  
x = 1 ;  
y = x + 3 ;  
z = ( x * y ) - ( 2 / x ) ;
```

Assegnamento

```
int x , y , z ;  
x = 1 ;  
y = x + 3 ;  
z = ( x * y ) - ( 2 / x ) ;
```

left_expression = right_expression

Assegnamento

```
int x , y , z ;  
x = 1 ;  
y = x + 3 ;  
z = ( x * y ) - ( 2 / x ) ;
```

left_expression = right_expression

- *left_expression* non può essere una espressione qualsiasi, deve denotare una locazione di memoria

Assegnamento

```
int x , y , z ;  
x = 1 ;  
y = x + 3 ;  
z = ( x * y ) - ( 2 / x ) ;
```

left_expression = right_expression

- *left_expression* non può essere una espressione qualsiasi, deve denotare una locazione di memoria

```
x = 1 ;
```

Assegnamento

```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```

left_expression = right_expression

- *left_expression* non può essere una espressione qualsiasi, deve denotare una locazione di memoria

x = 1;



Assegnamento

```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```

left_expression = right_expression

- *left_expression* non può essere una espressione qualsiasi, deve denotare una locazione di memoria

x = 1;



x + 3 = 1;

Assegnamento

```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```

left_expression = right_expression

- *left_expression* non può essere una espressione qualsiasi, deve denotare una locazione di memoria

x = 1;



x + 3 = 1;



Espressioni

Espressioni

- Un'espressione è un costrutto sintattico che può essere *valutato* per ottenere e determinare il suo valore

Espressioni

- Un'espressione è un costrutto sintattico che può essere *valutato* per ottenere e determinare il suo valore

$$x + y - 3$$

Espressioni

- Un'espressione è un costrutto sintattico che può essere *valutato* per ottenere e determinare il suo valore

$$x + y - 3$$

$$5.2$$

Espressioni

- Un'espressione è un costrutto sintattico che può essere *valutato* per ottenere e determinare il suo valore

$x + y - 3$

5.2

$x > 5 \ \&\& \ (y < 3 \ || \ !b)$

Espressioni

Cosa può essere un'espressione?

Espressioni

Cosa può essere un'espressione?

exp può essere:

Espressioni

Cosa può essere un'espressione?

exp può essere:

◆ una costante: 1, 5, 3.3, 'a'

Espressioni

Cosa può essere un'espressione?

exp può essere:

- ◆ una costante: 1, 5, 3.3, 'a'
- ◆ una variabile: x, y

Espressioni

Cosa può essere un'espressione?

exp può essere:

- ◆ una costante: 1, 5, 3.3, 'a'
- ◆ una variabile: x, y
- ◆ un'espressione fra parentesi (*exp*): (5 + 3)

Espressioni

Cosa può essere un'espressione?

exp può essere:

- ◆ una costante: 1, 5, 3.3, 'a'
- ◆ una variabile: x, y
- ◆ un'espressione fra parentesi (*exp*): (5 + 3)
- ◆ un'espressione binaria $exp_1 \text{ op } exp_2$ dove:

- ⊙ exp_1 e exp_2 sono espressioni, chiamati argomenti/operandi

- ⊙ *op* è un operatore binario

- ➡ Operatore aritmetico: + − * % / (+ e - operatori binari)

- ➡ Operatore booleano: & & ||

- ➡ Operatore relazionale: == != > < >= <=

Espressioni

Cosa può essere un'espressione?

Espressioni

Cosa può essere un'espressione?

exp può essere:

Espressioni

Cosa può essere un'espressione?

exp può essere:

◆ un'espressione unaria *op exp* dove:

◎ *exp* è un'espressione, chiamato argomento/operando

◎ *op* è un operatore unario

➡ Operatore aritmetico: $+$ $-$ (operatori unari)

➡ Operatore booleano: $!$

◆ ... (nelle prossime lezioni)

Espressioni

Valore di un'espressione

Il valore di *exp* è:

Espressioni

Valore di un'espressione

Il valore di *exp* è:

◆ Costante: la costante stessa

Espressioni

Valore di un'espressione

Il valore di *exp* è:

- ◆ Costante: la costante stessa
- ◆ Variable: il valore contenuto nella variabile al momento della valutazione

Espressioni

Valore di un'espressione

Il valore di *exp* è:

- ◆ Costante: la costante stessa
- ◆ Variable: il valore contenuto nella variabile al momento della valutazione
- ◆ Espressione parentesizzata (*exp*): il valore di *exp*

Espressioni

Valore di un'espressione

Il valore di exp è:

- ◆ Costante: la costante stessa
- ◆ Variable: il valore contenuto nella variabile al momento della valutazione
- ◆ Espressione parentesizzata (exp): il valore di exp
- ◆ Espressione binaria $exp_1 \ op \ exp_2$: il valore ottenuto dall'applicazione dell'operatore op ai valori di exp_1 e exp_2

Espressioni

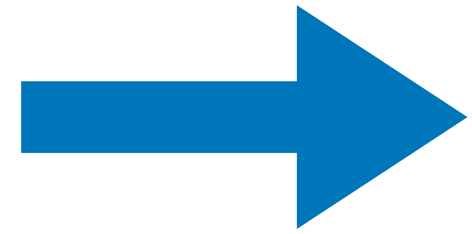
Valore di un'espressione

Il valore di exp è:

- ◆ Costante: la costante stessa
- ◆ Variable: il valore contenuto nella variabile al momento della valutazione
- ◆ Espressione parentesizzata (exp): il valore di exp
- ◆ Espressione binaria $exp_1 \ op \ exp_2$: il valore ottenuto dall'applicazione dell'operatore op ai valori di exp_1 e exp_2
- ◆ Espressione unaria $op \ exp$: il valore ottenuto dall'applicazione dell'operatore op al valore di exp

Espressioni

Valore di un'espressione



```
int x, y, z;
```

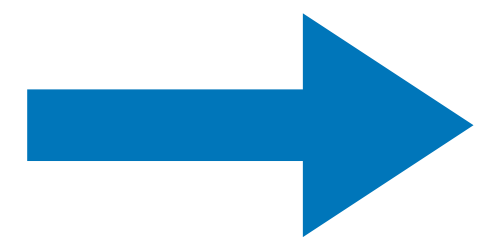
```
x = 1;
```

```
y = x + 3;
```

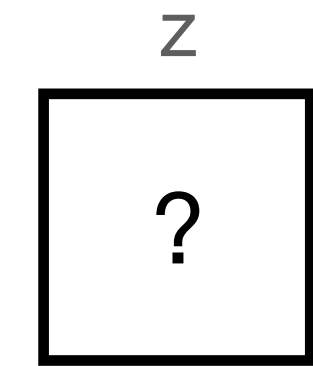
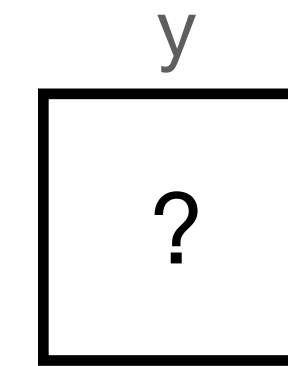
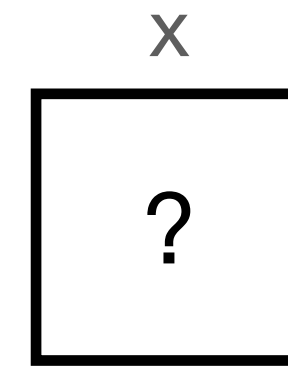
```
z = (x * y) - (2 / x);
```

Espressioni

Valore di un'espressione



```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```



Espressioni

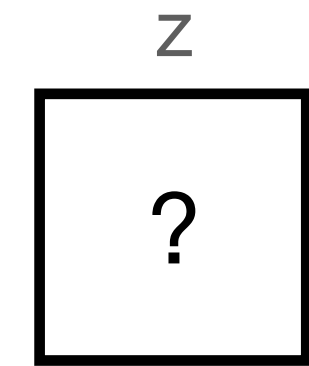
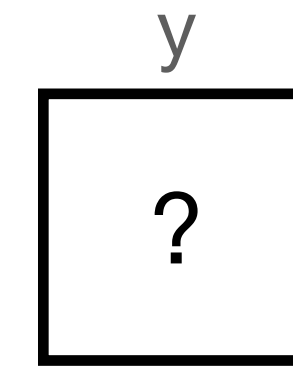
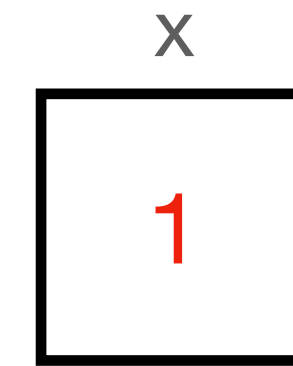
Valore di un'espressione

`int x, y, z;`

`x = 1;`

`y = x + 3;`

`z = (x * y) - (2 / x);`



Espressioni

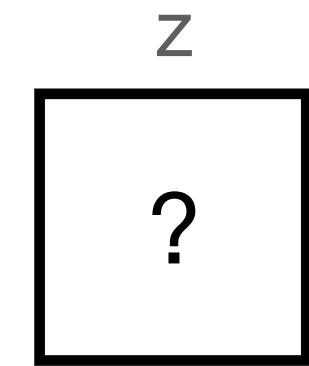
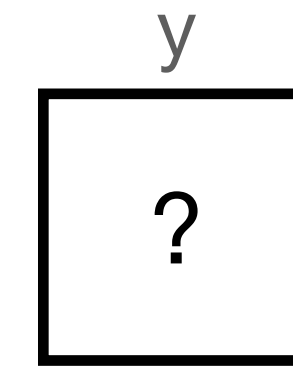
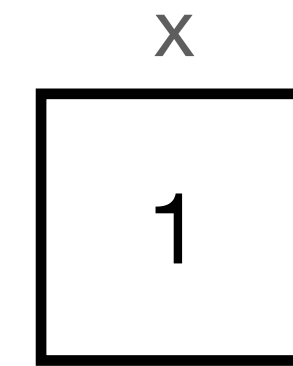
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```

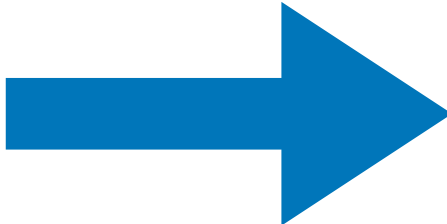


Espressioni

Valore di un'espressione

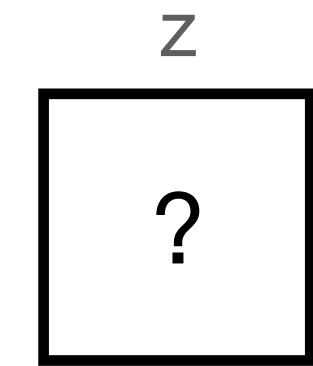
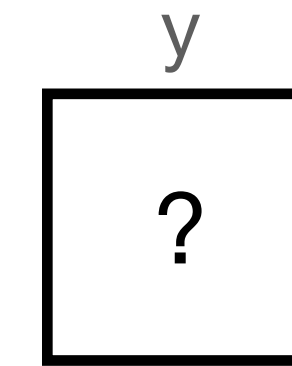
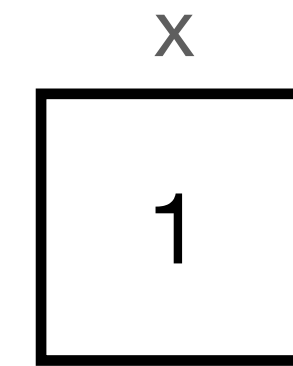
```
int x, y, z;
```

```
x = 1;
```



```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



$x + 3$

Espressioni

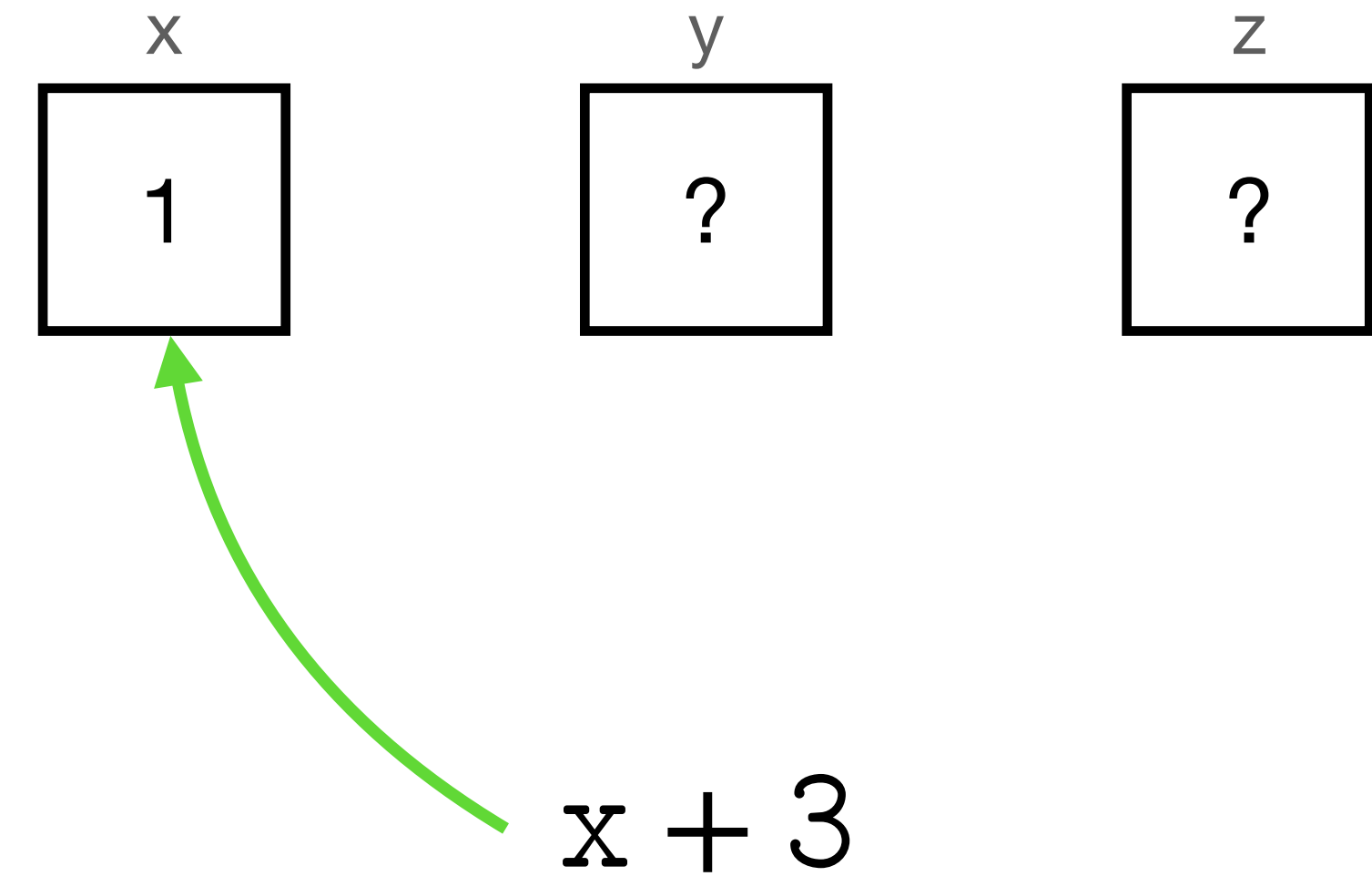
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

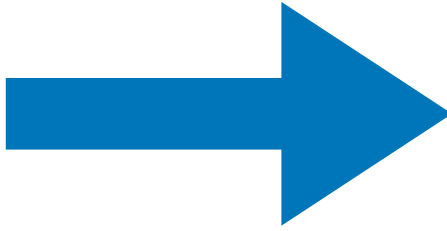
```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```

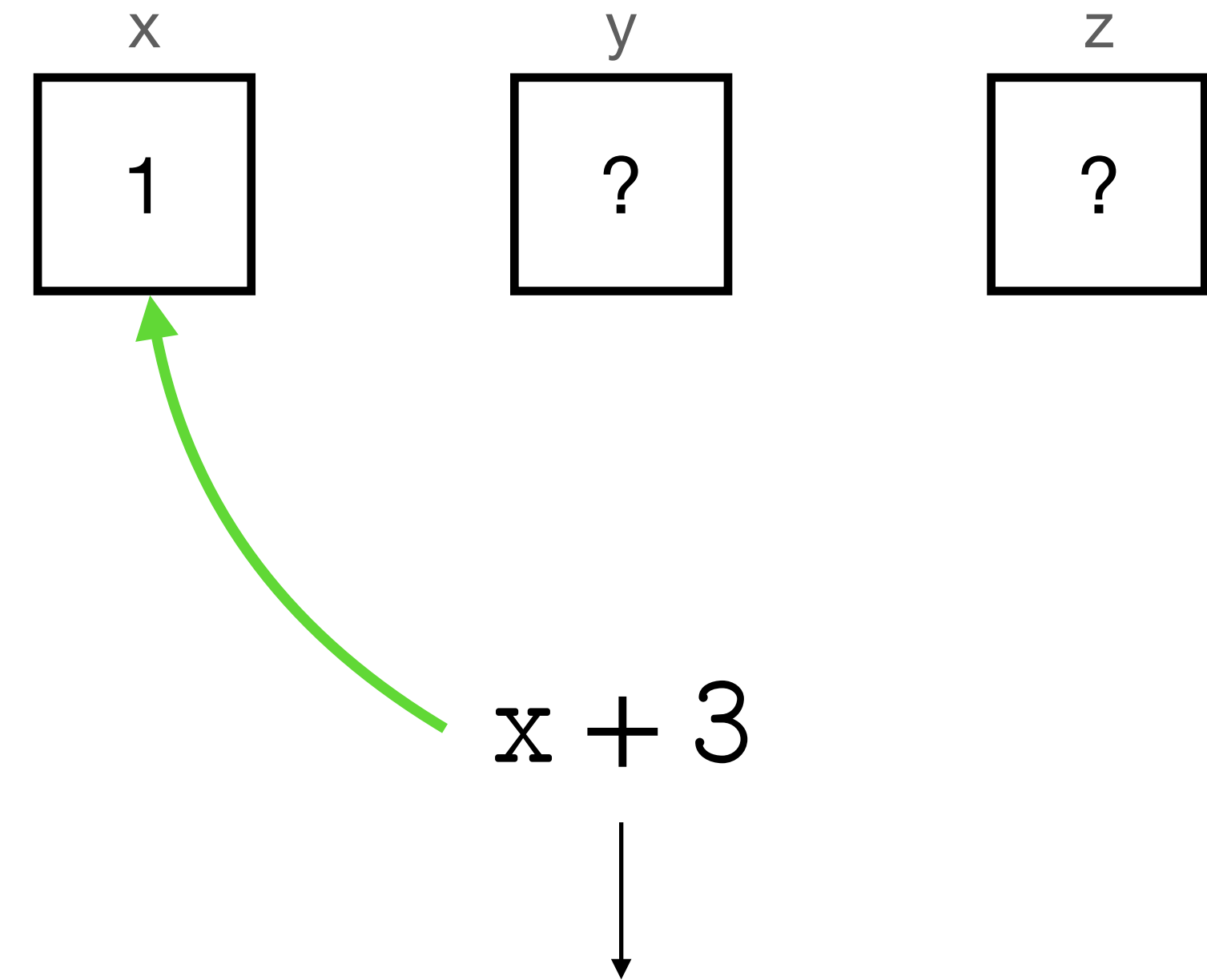


Espressioni

Valore di un'espressione

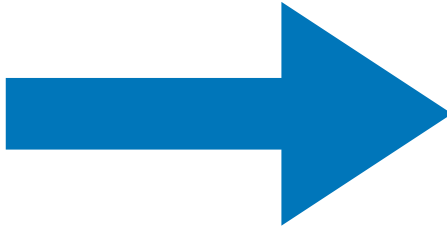


```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```

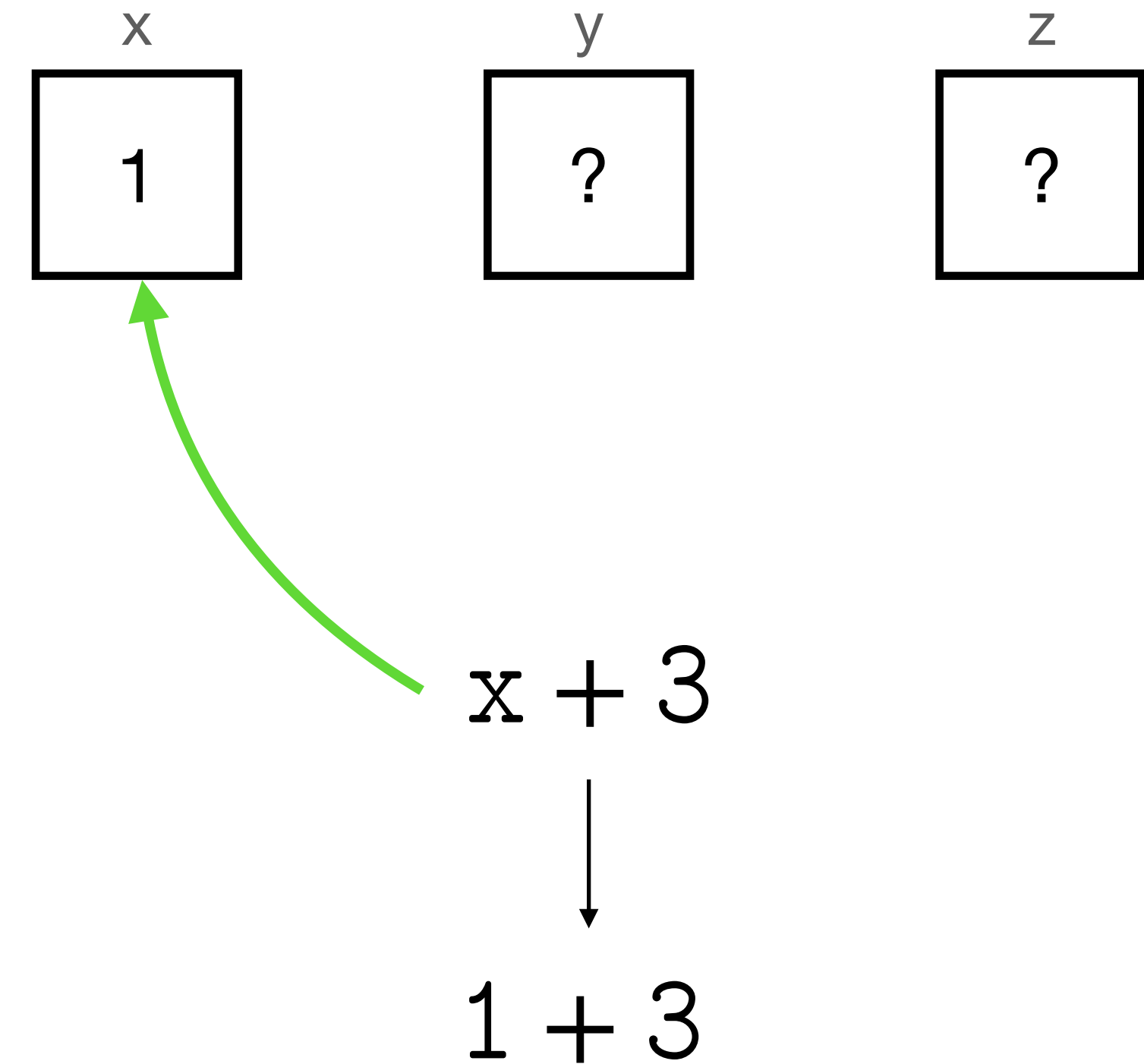


Espressioni

Valore di un'espressione



```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```



Espressioni

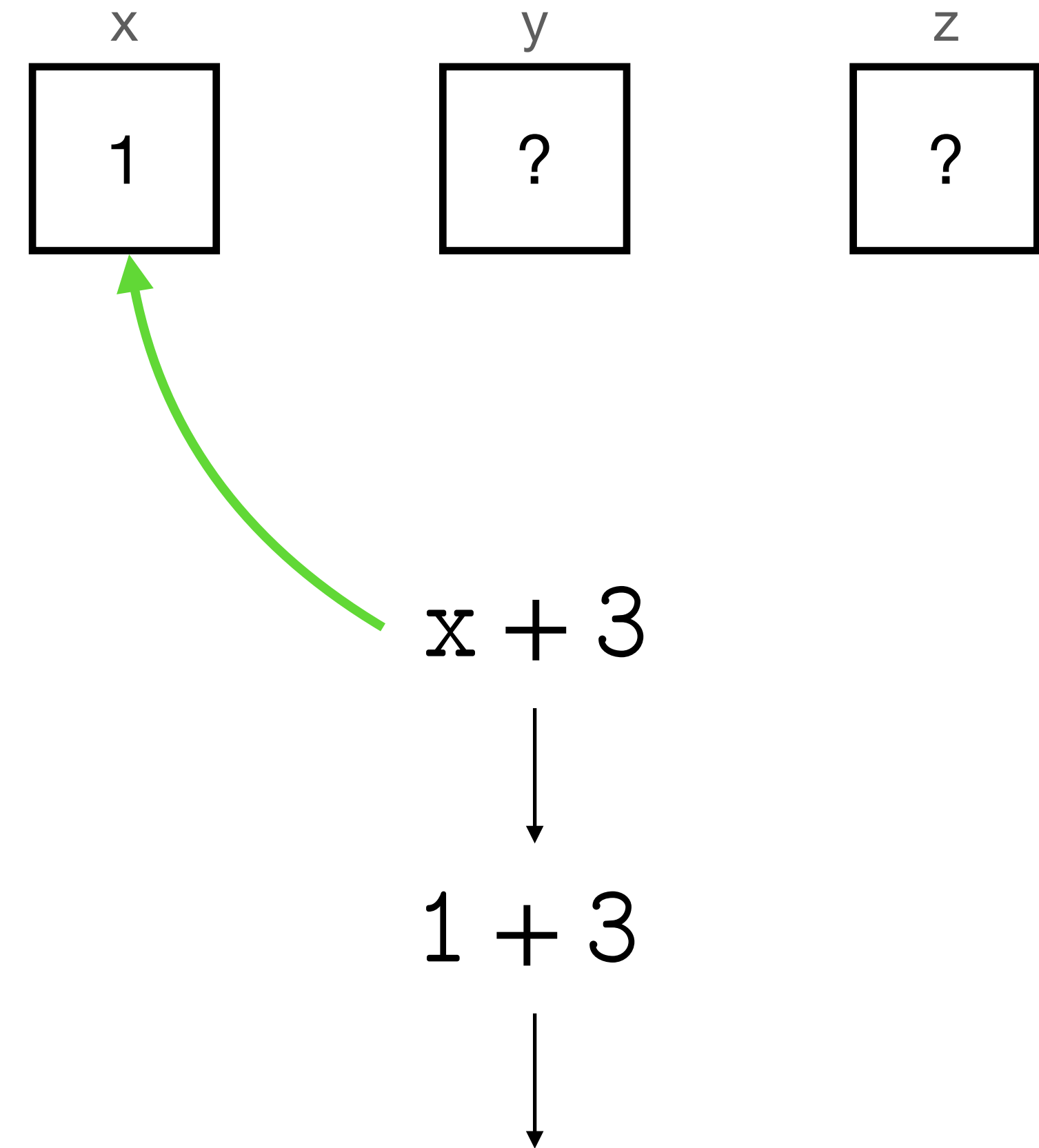
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

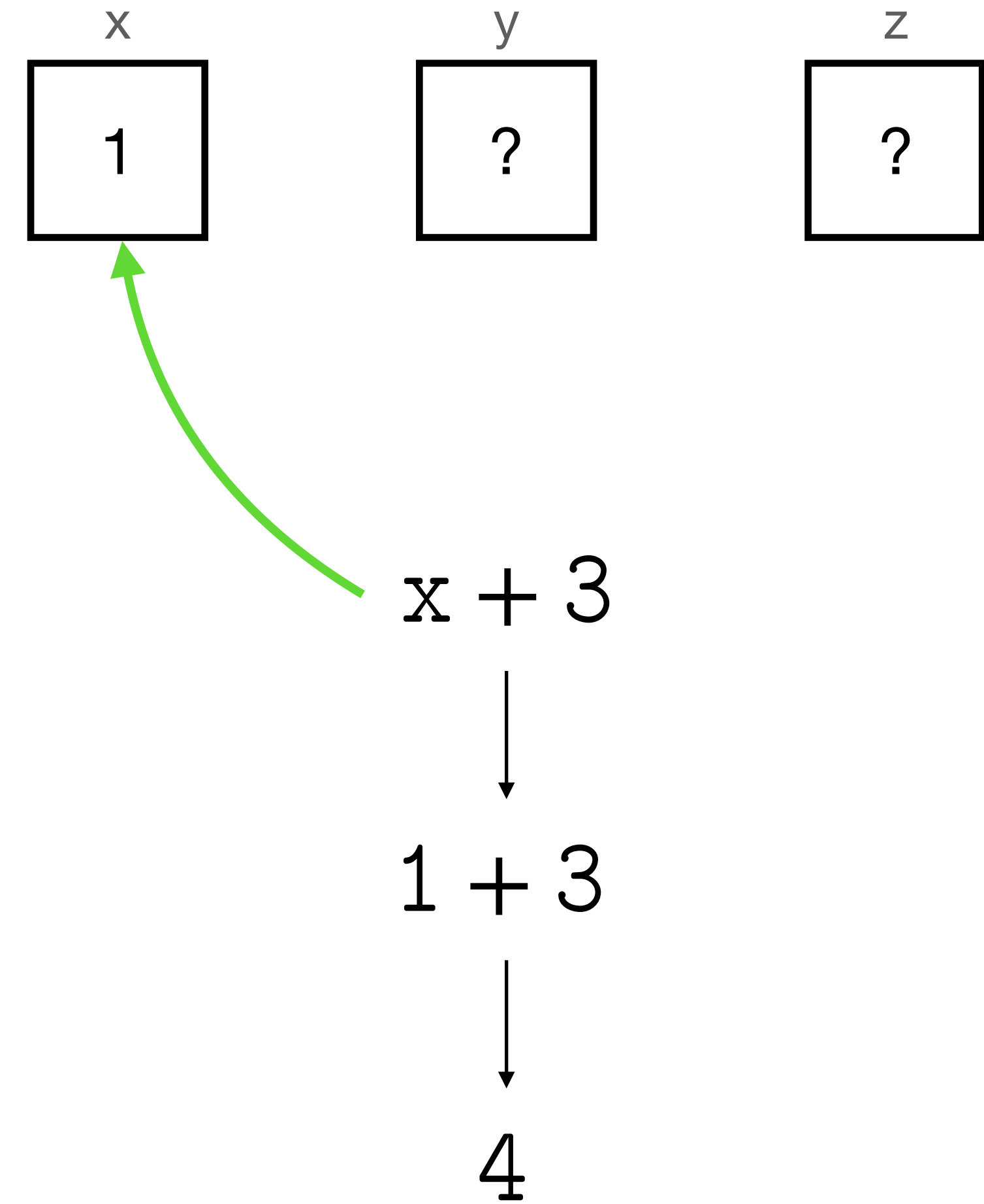
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

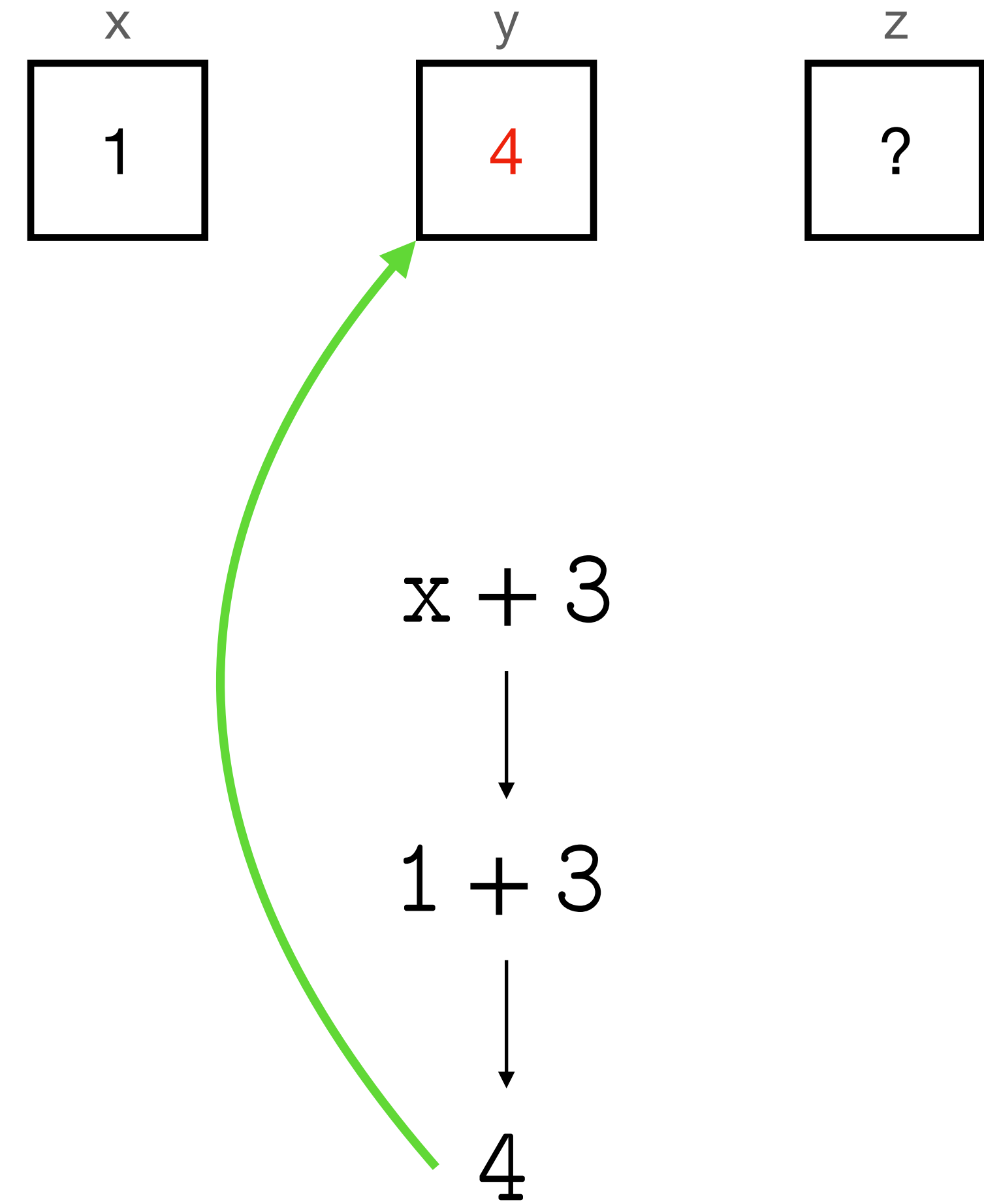
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

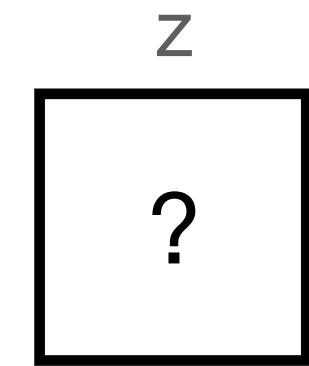
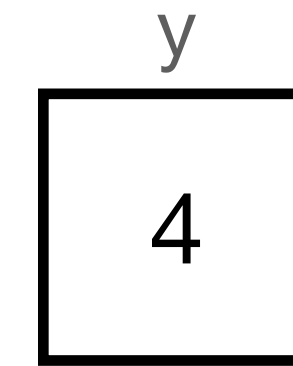
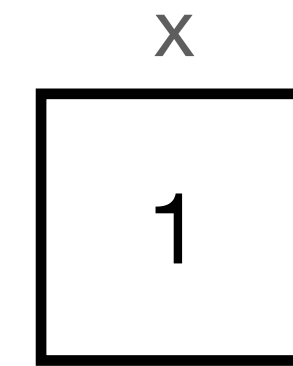
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

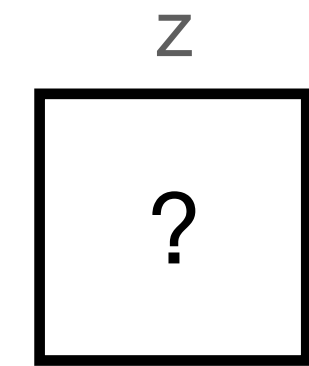
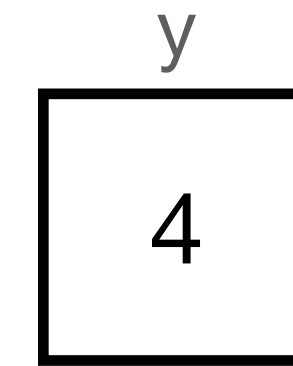
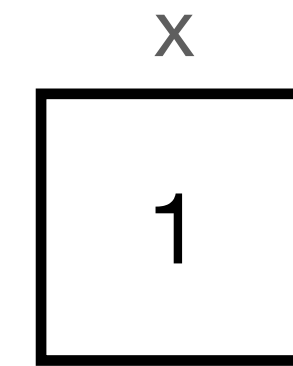
Valore di un'espressione

```
int x, y, z;
```

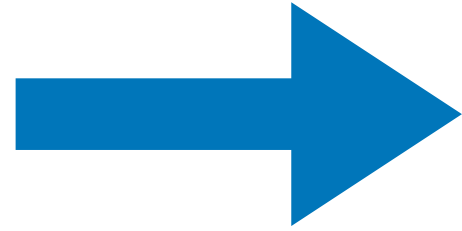
```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



$(x * y) - (2 / x)$



Espressioni

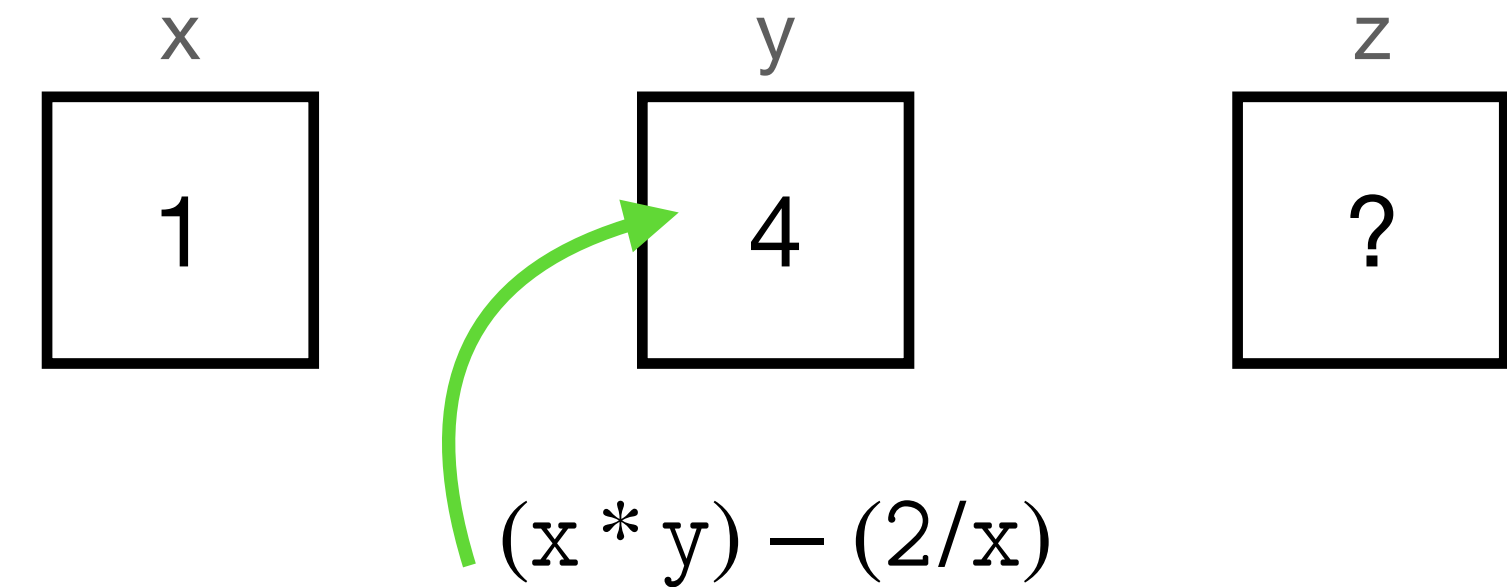
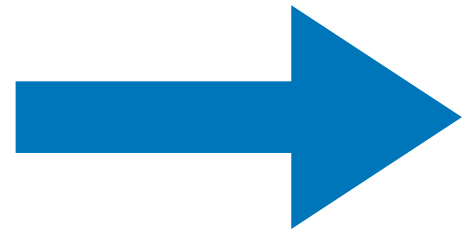
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

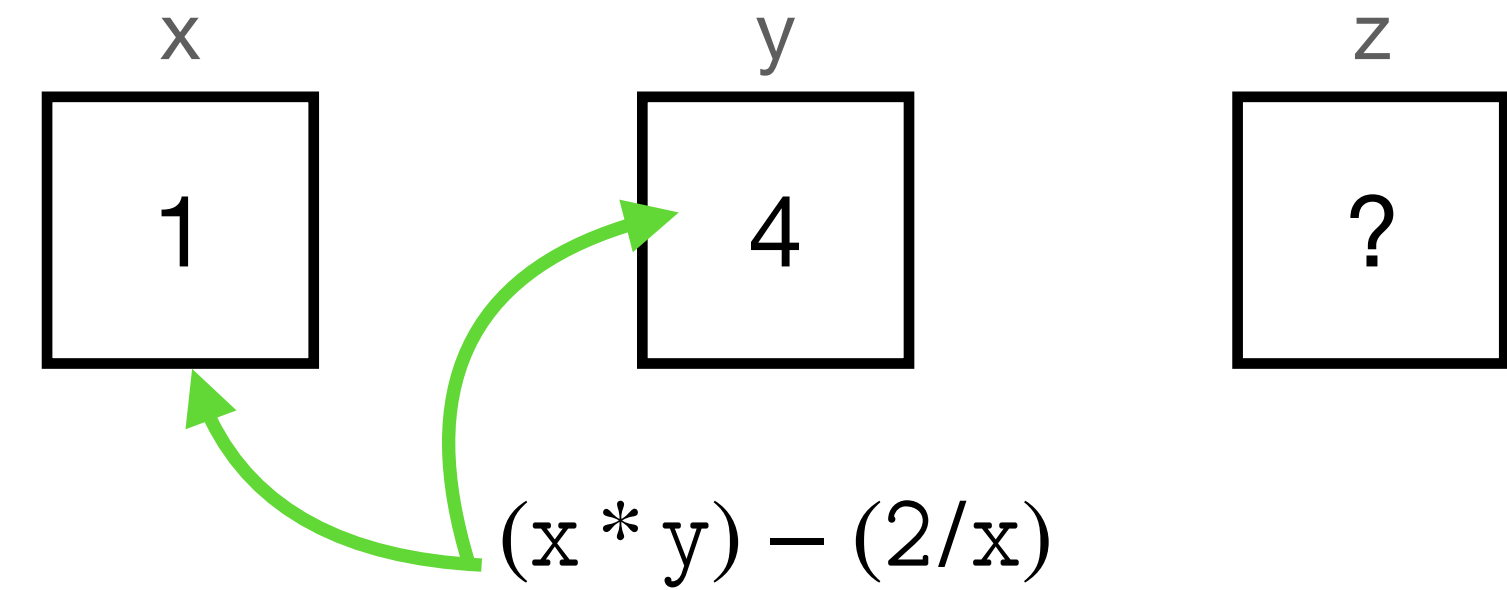
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

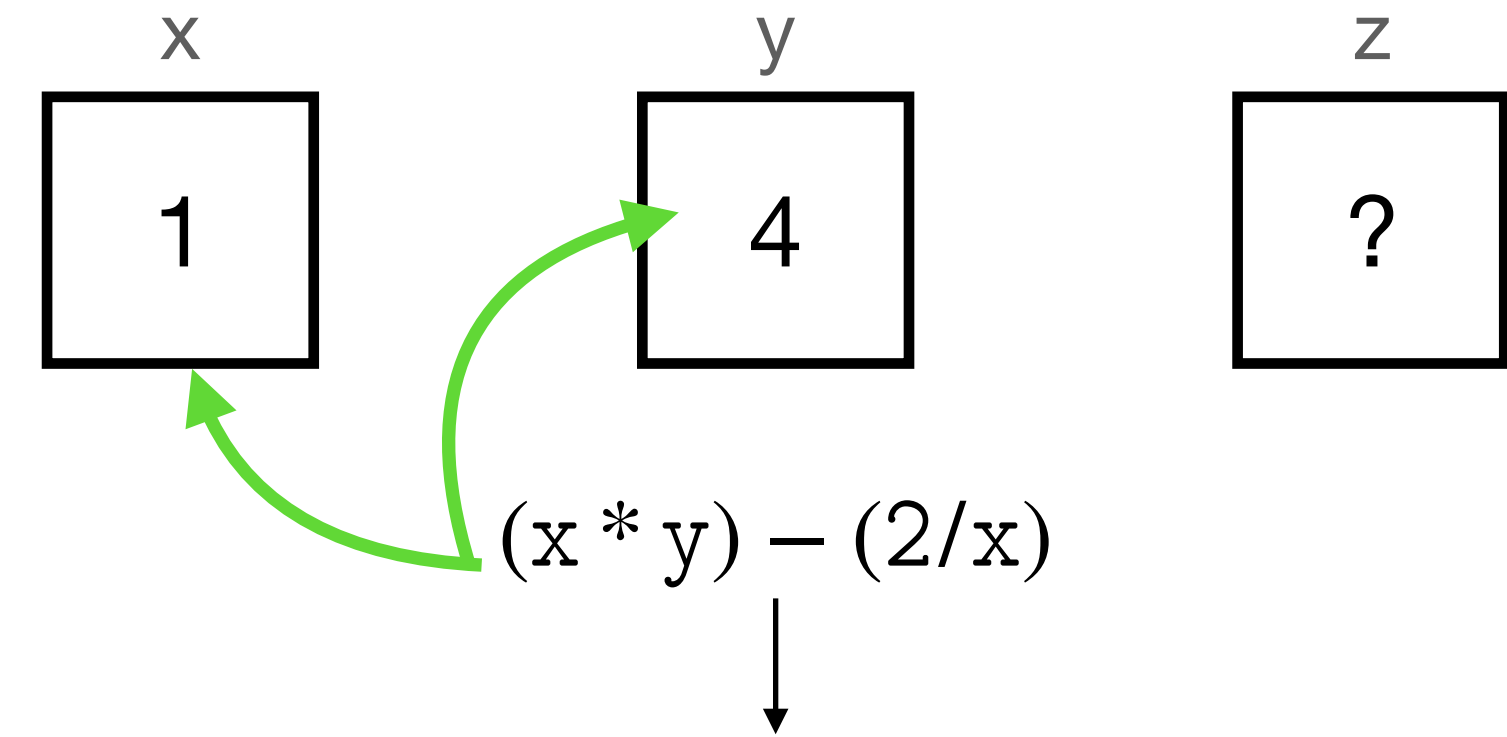
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

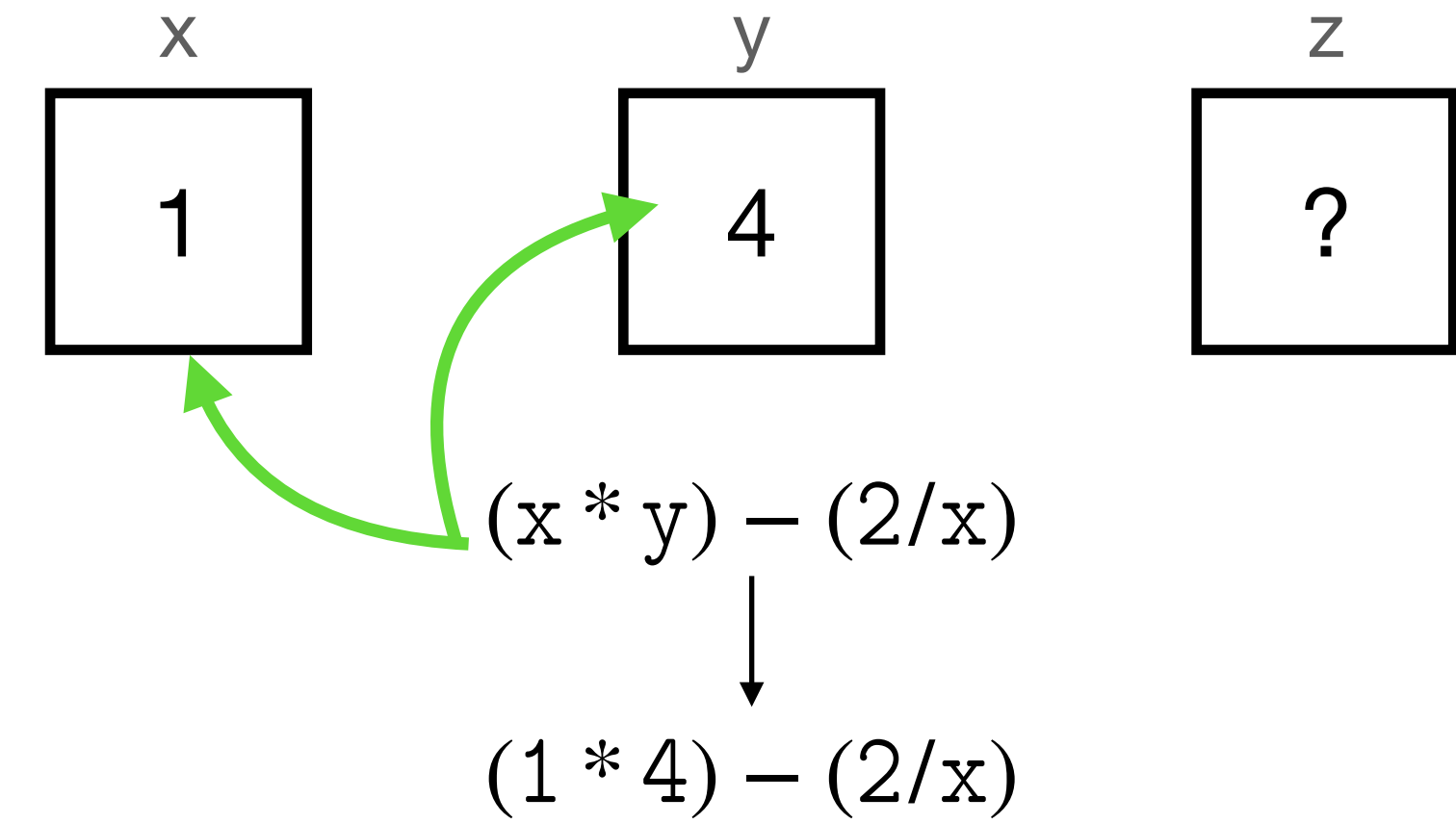
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

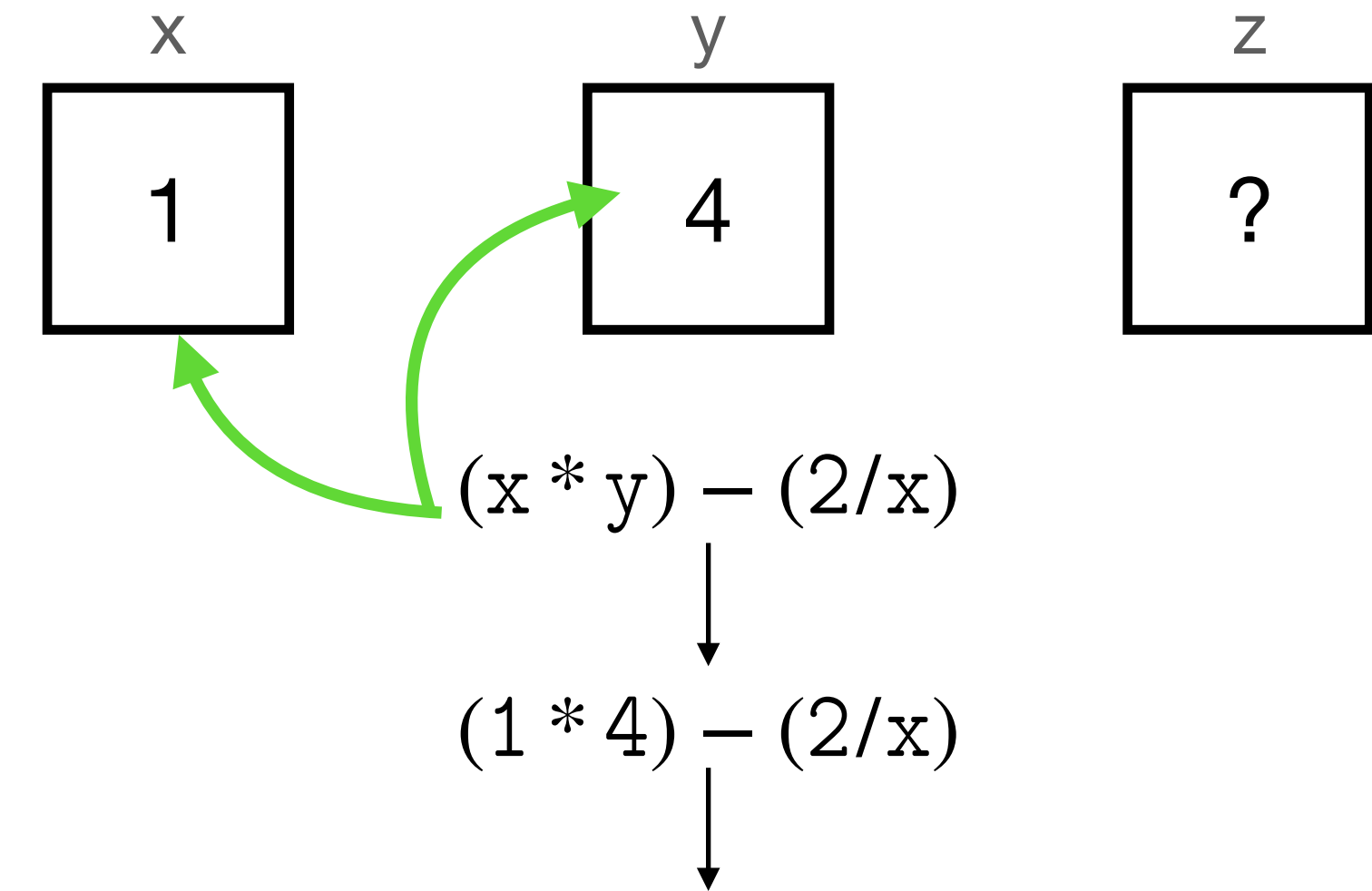
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

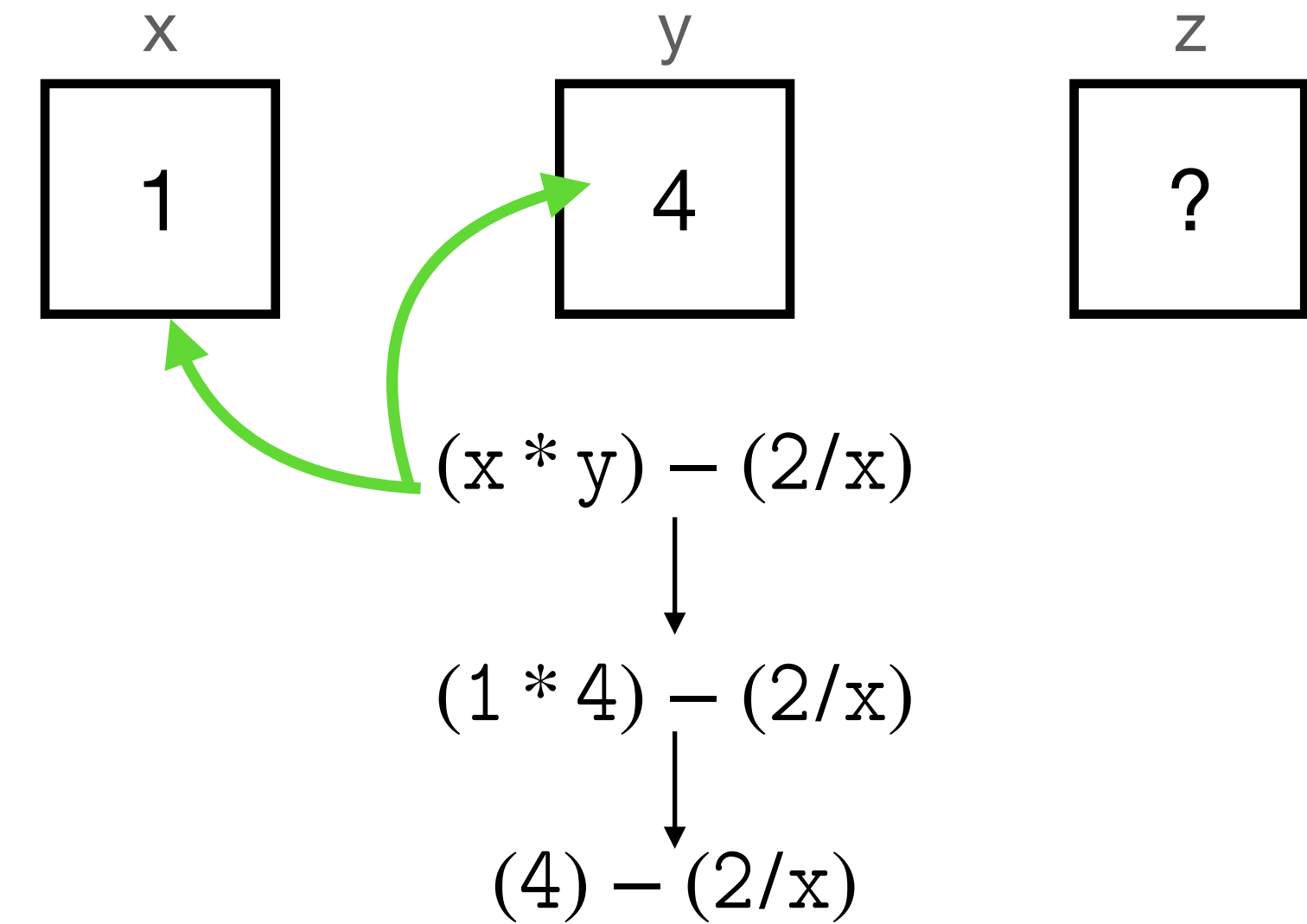
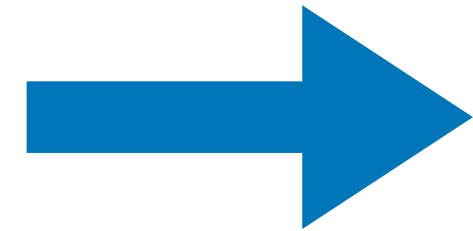
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

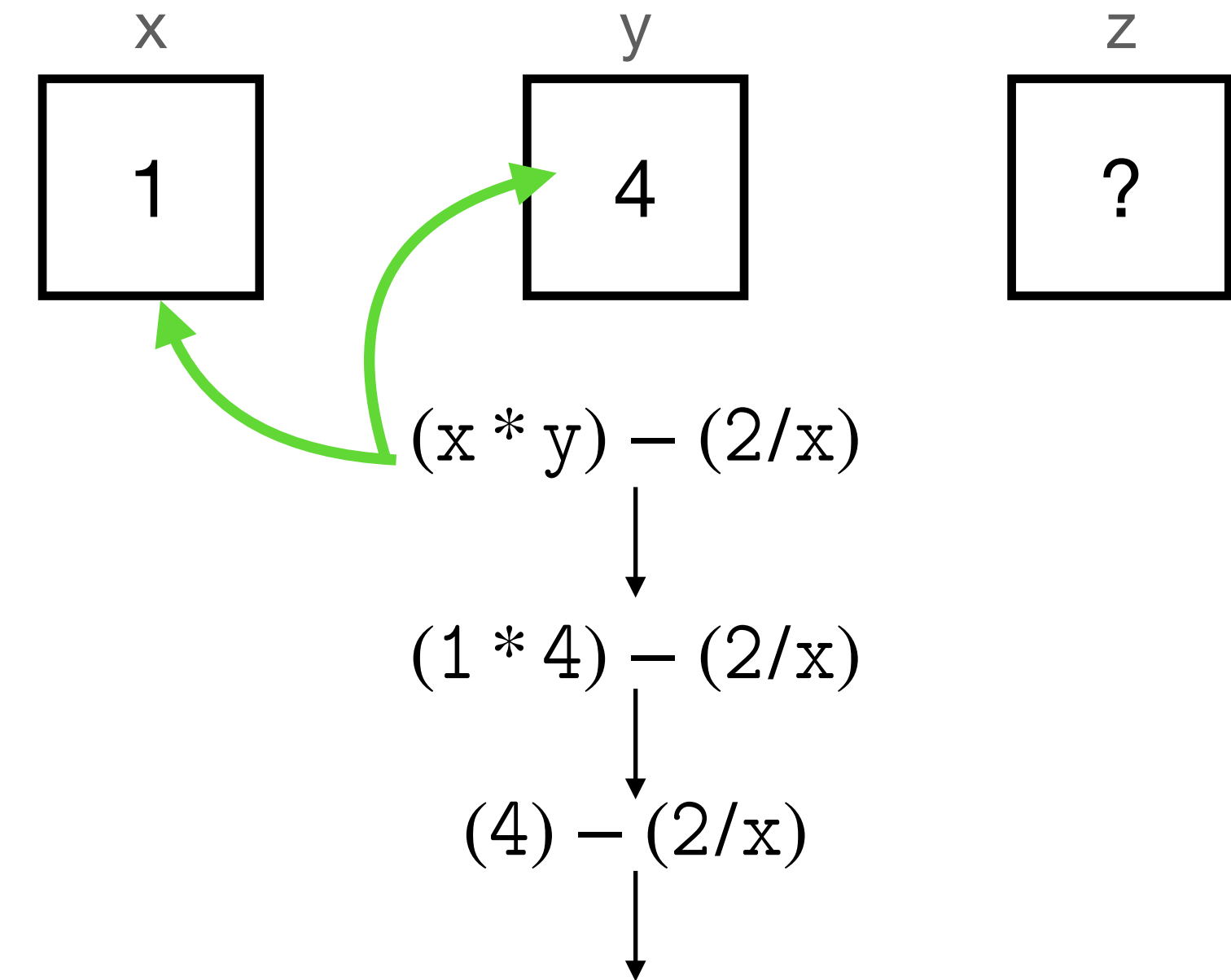
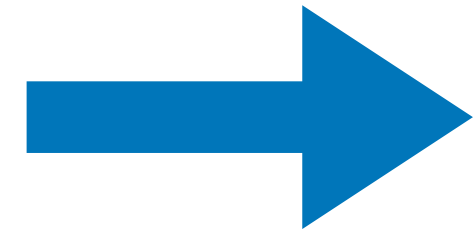
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

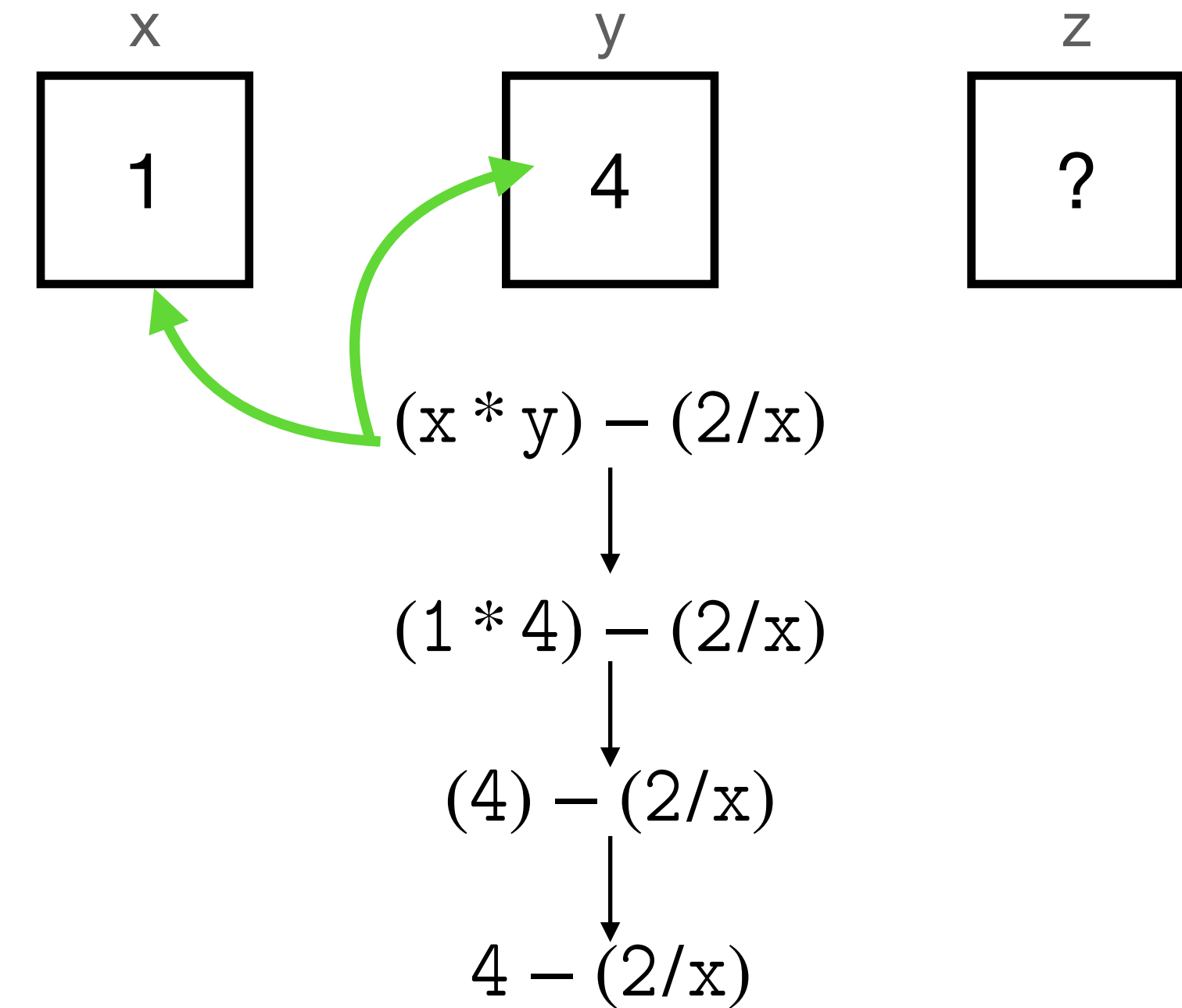
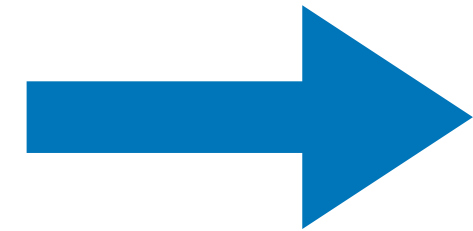
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

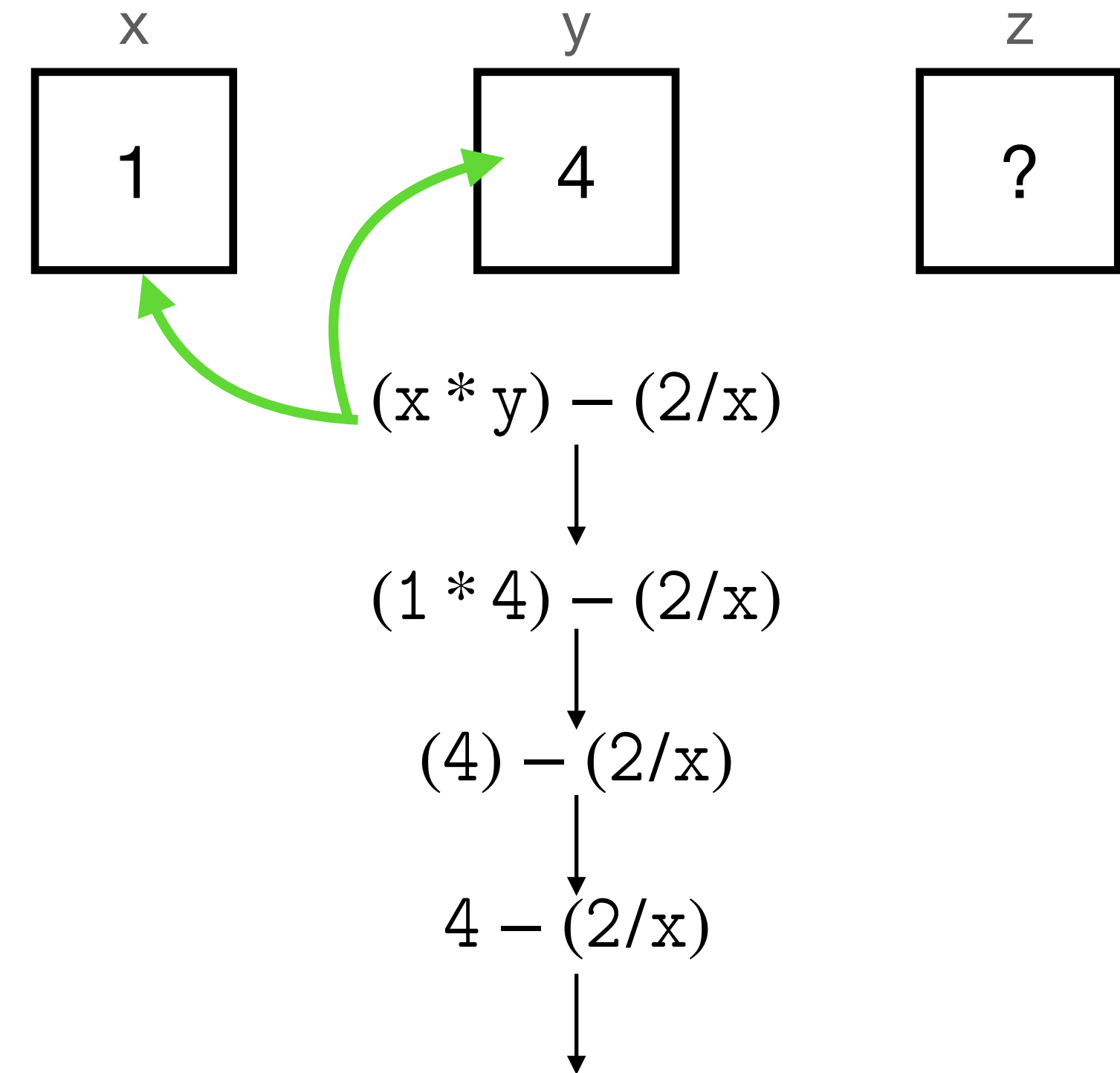
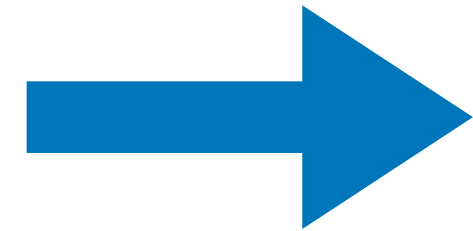
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

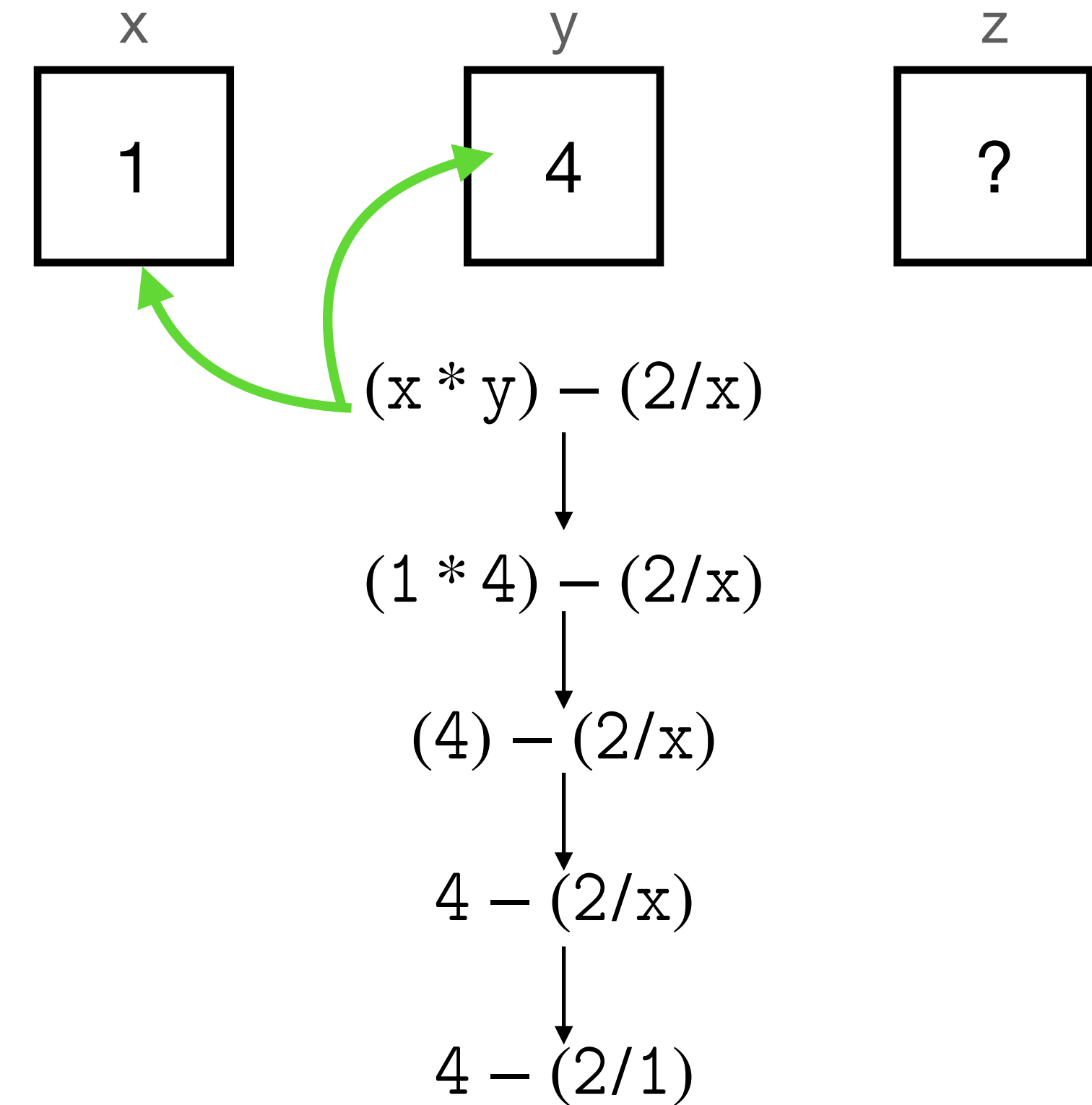
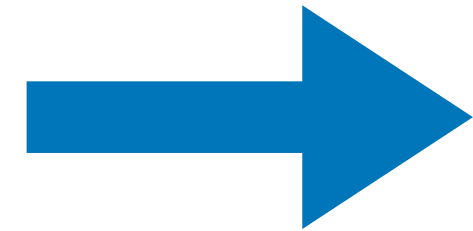
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

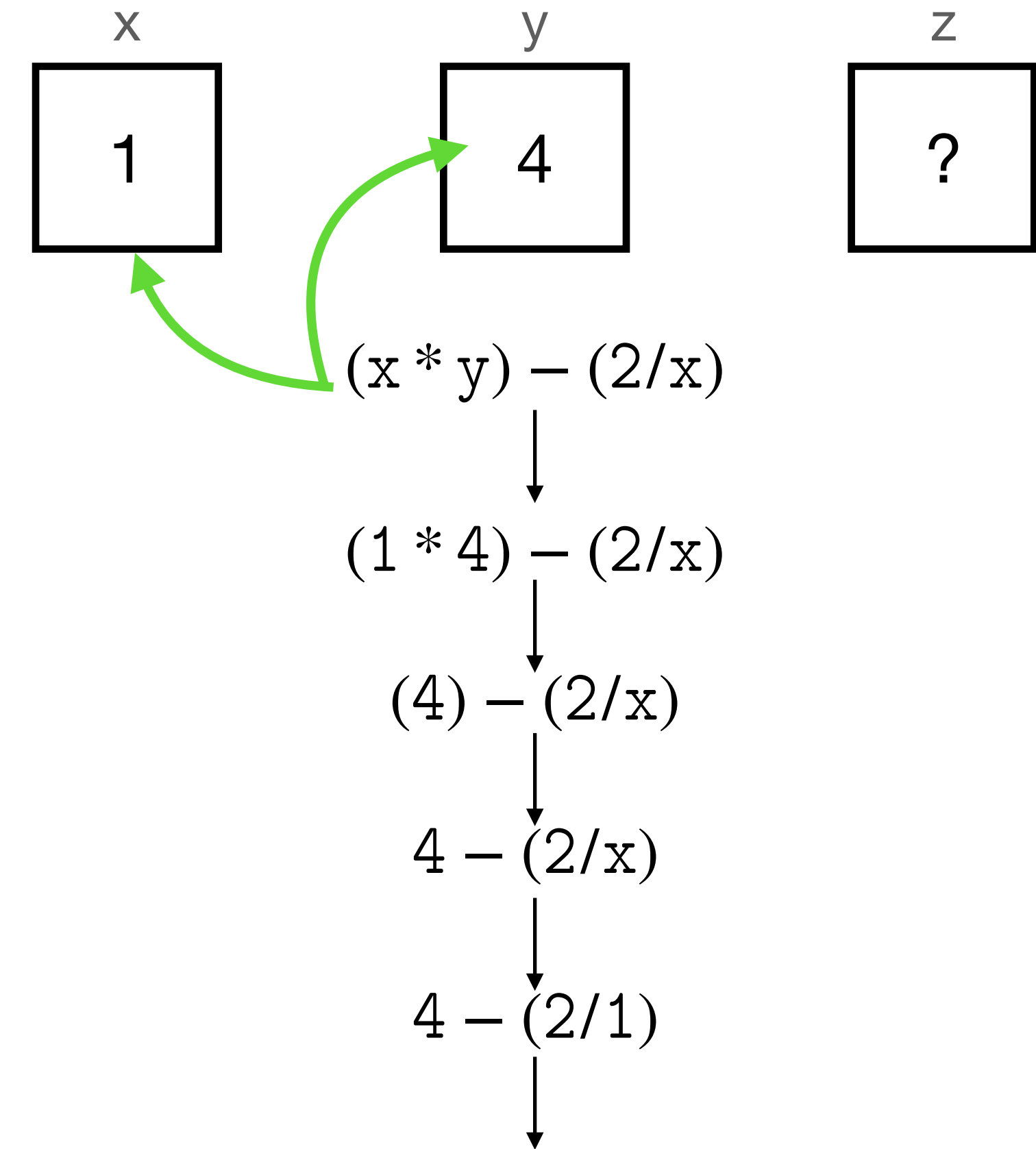
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

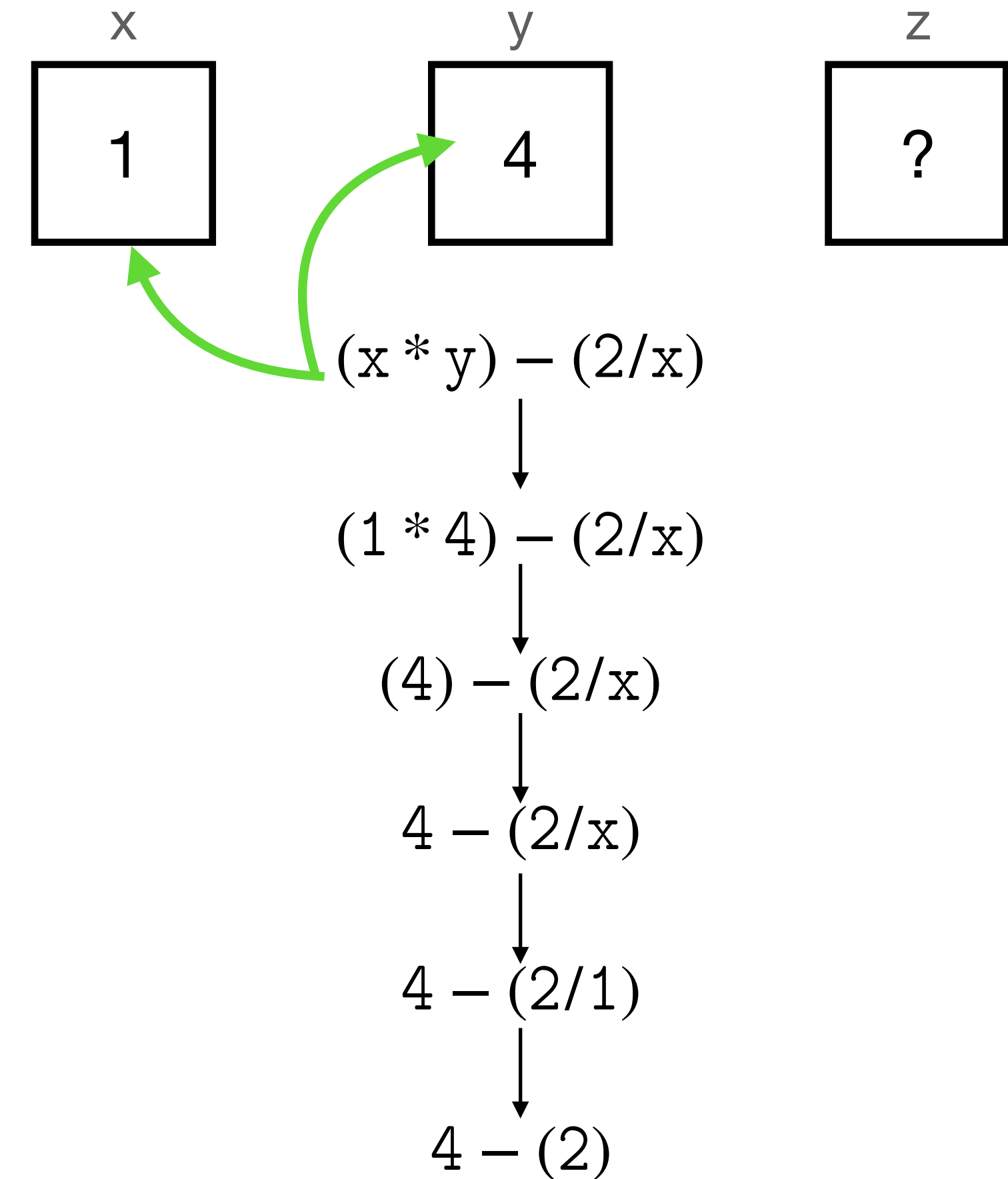
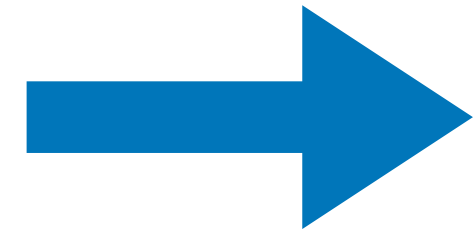
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

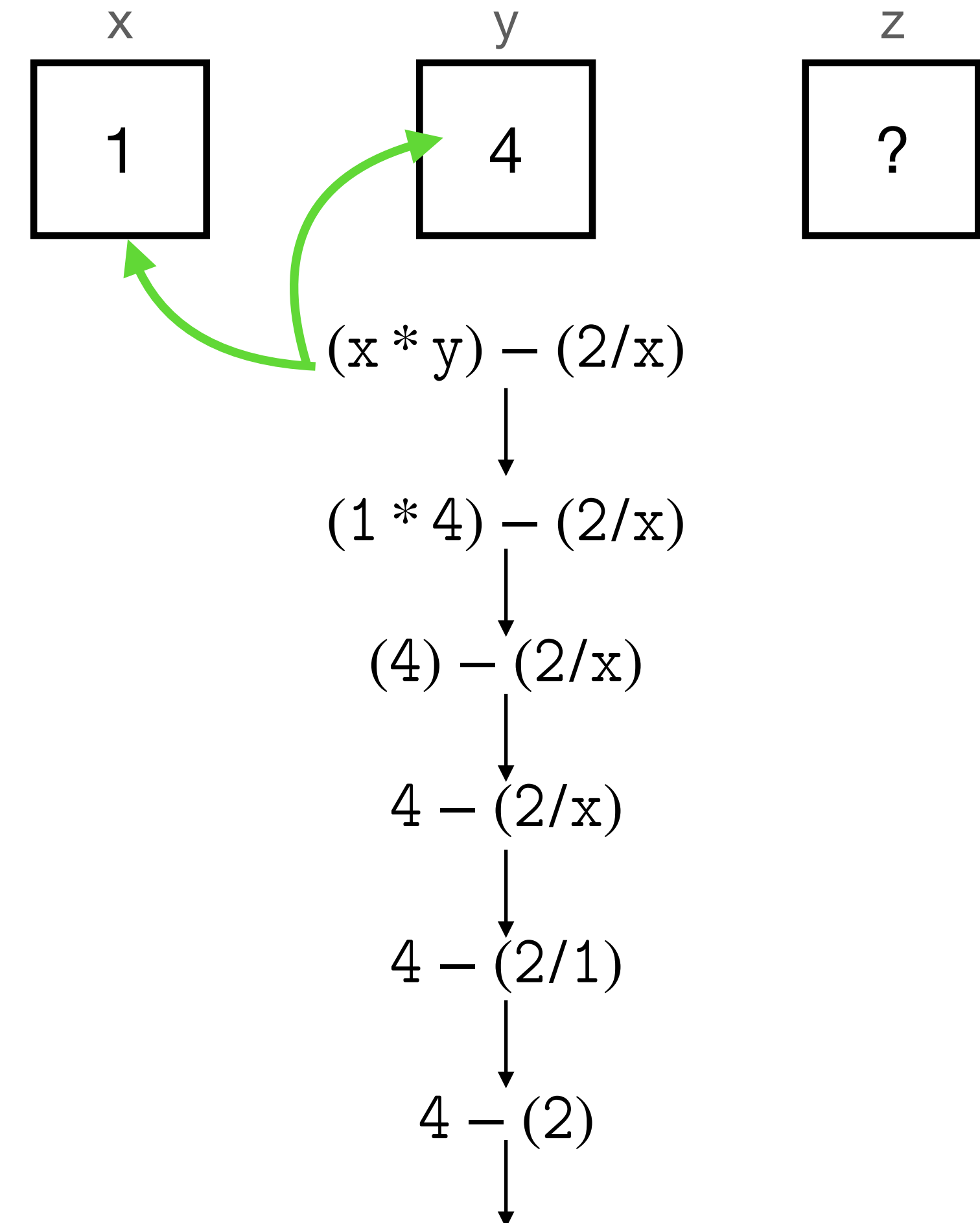
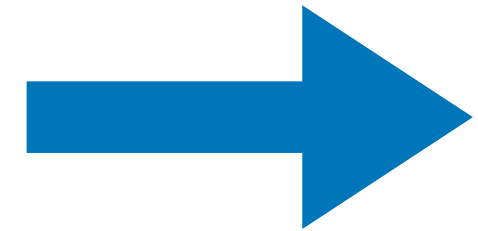
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

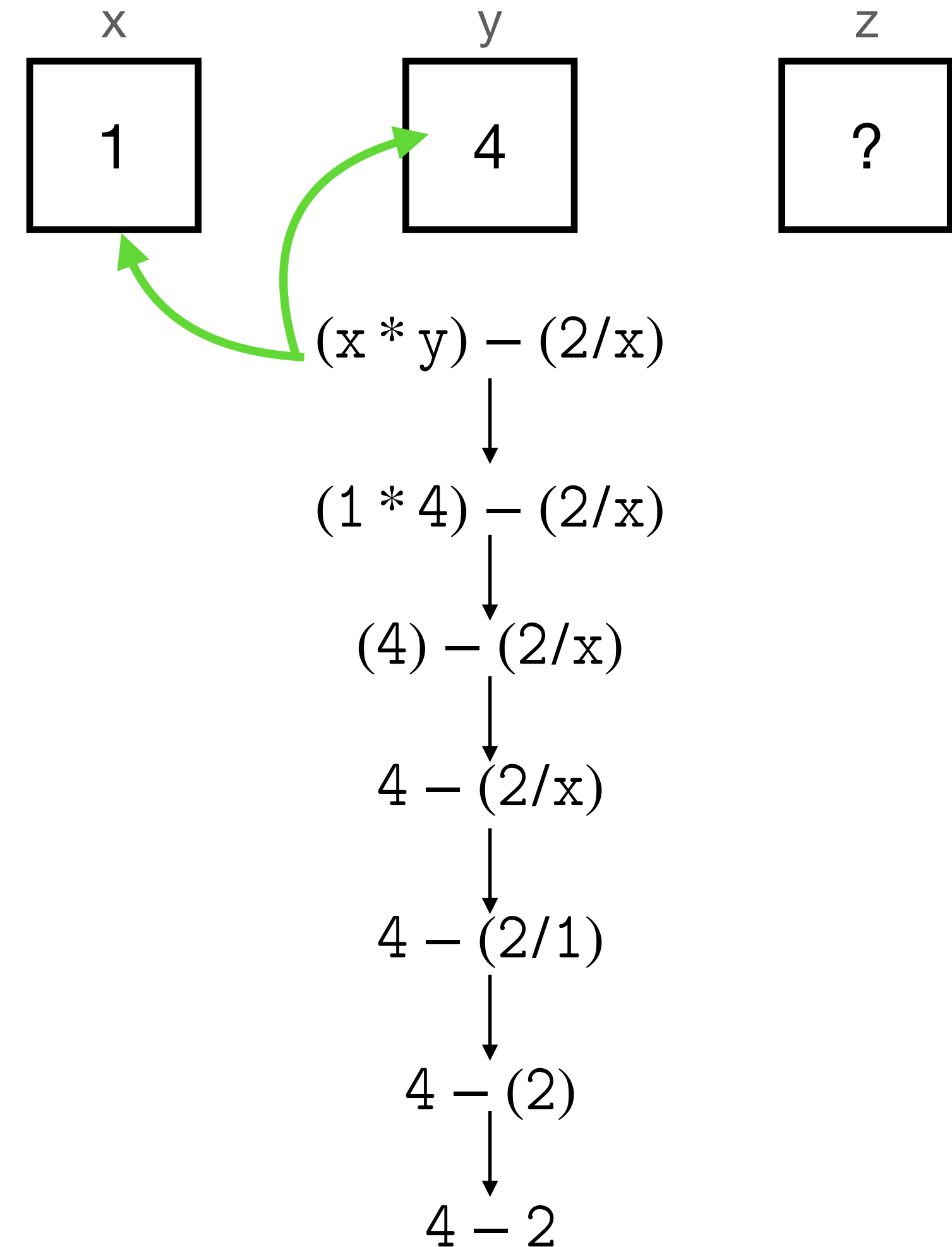
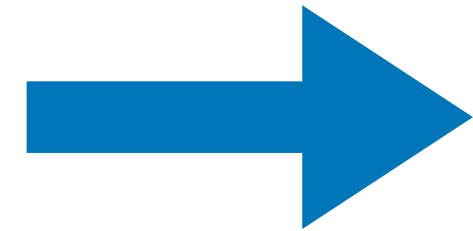
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

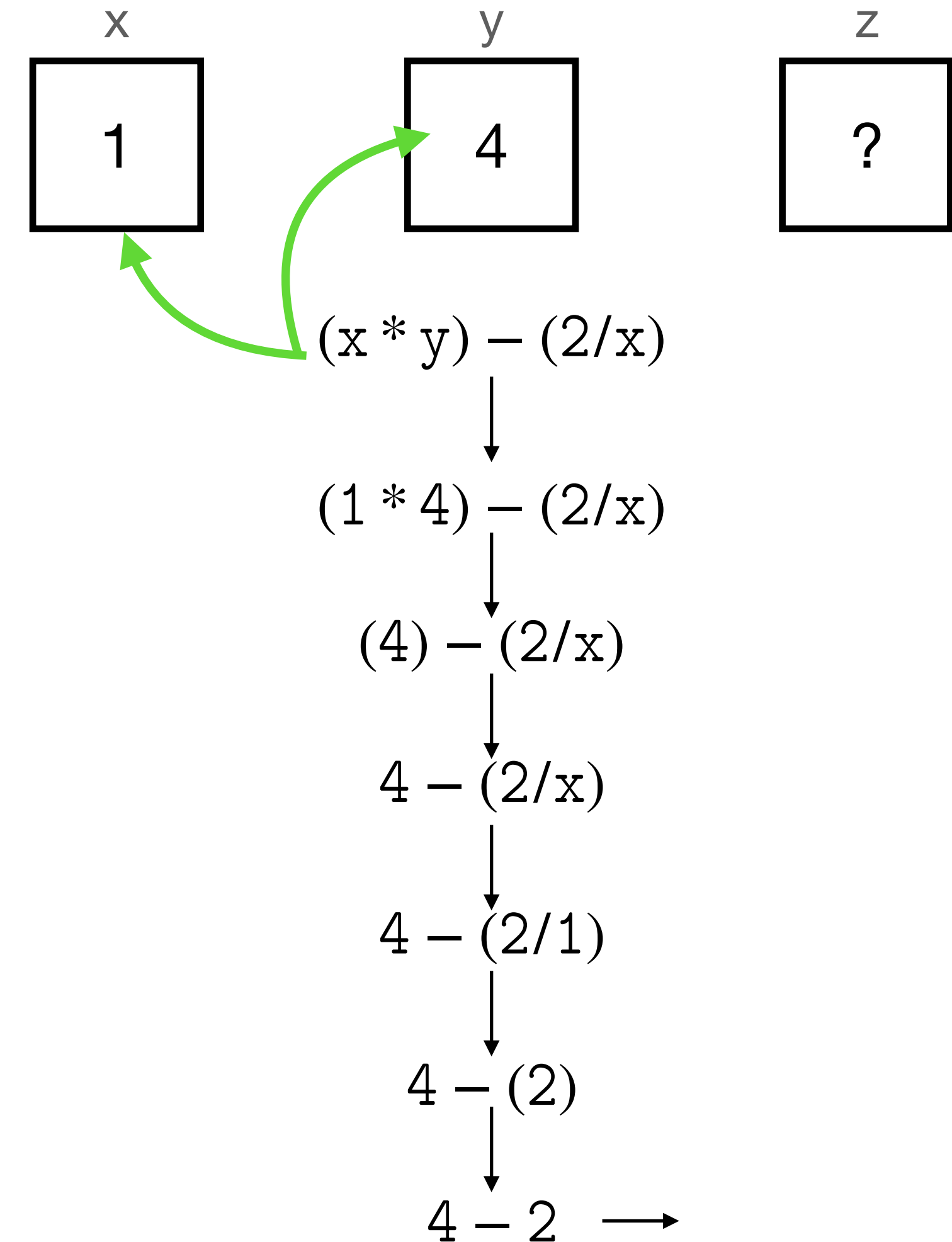
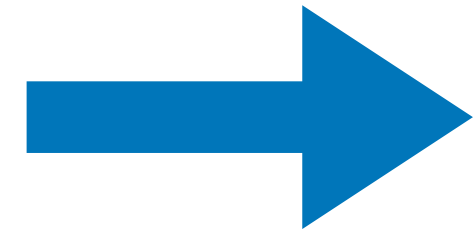
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```



Espressioni

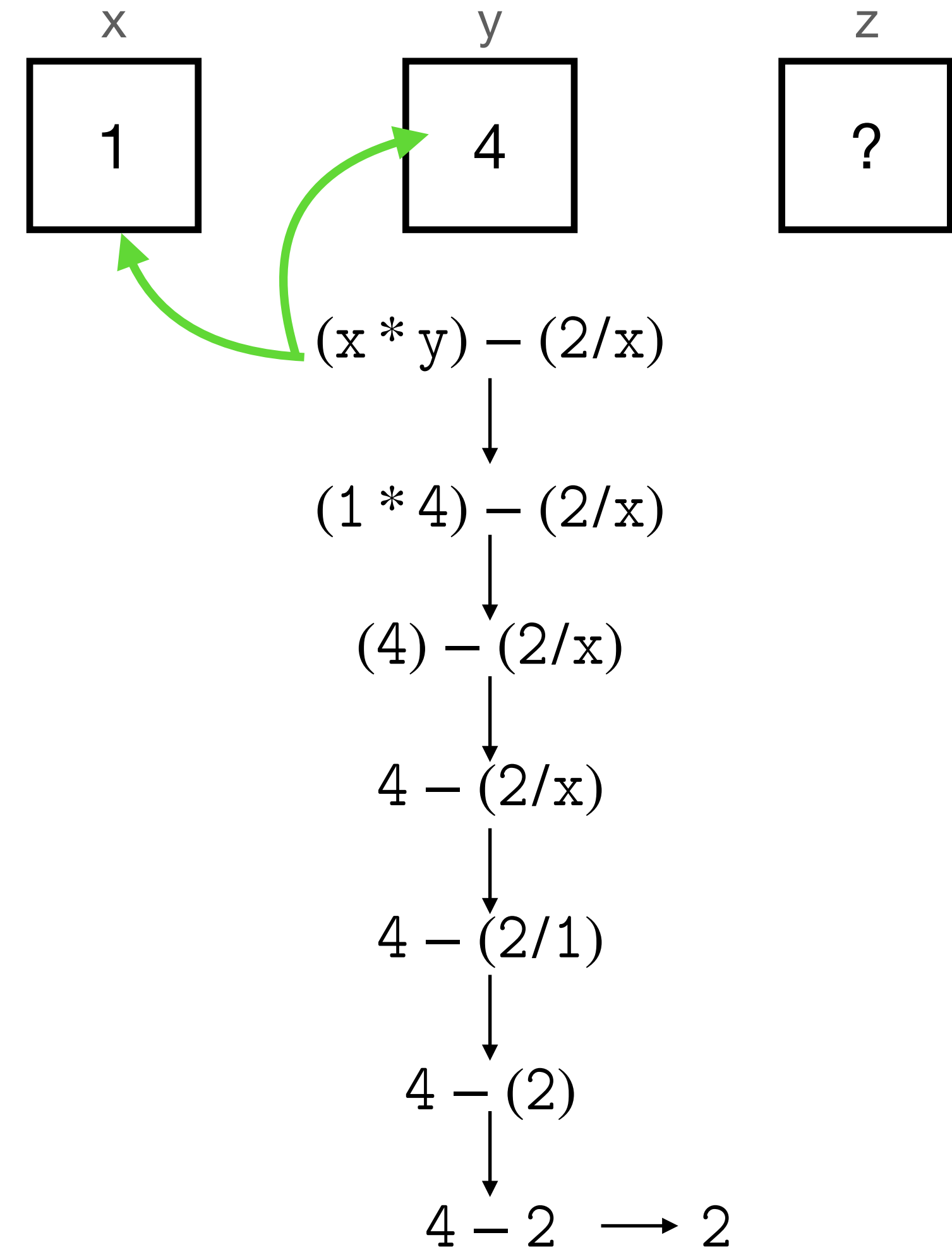
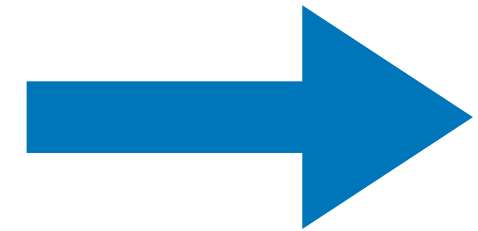
Valore di un'espressione

```
int x, y, z;
```

```
x = 1;
```

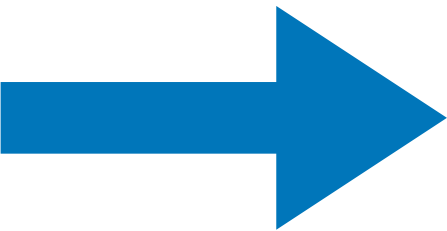
```
y = x + 3;
```

```
z = (x * y) - (2 / x);
```

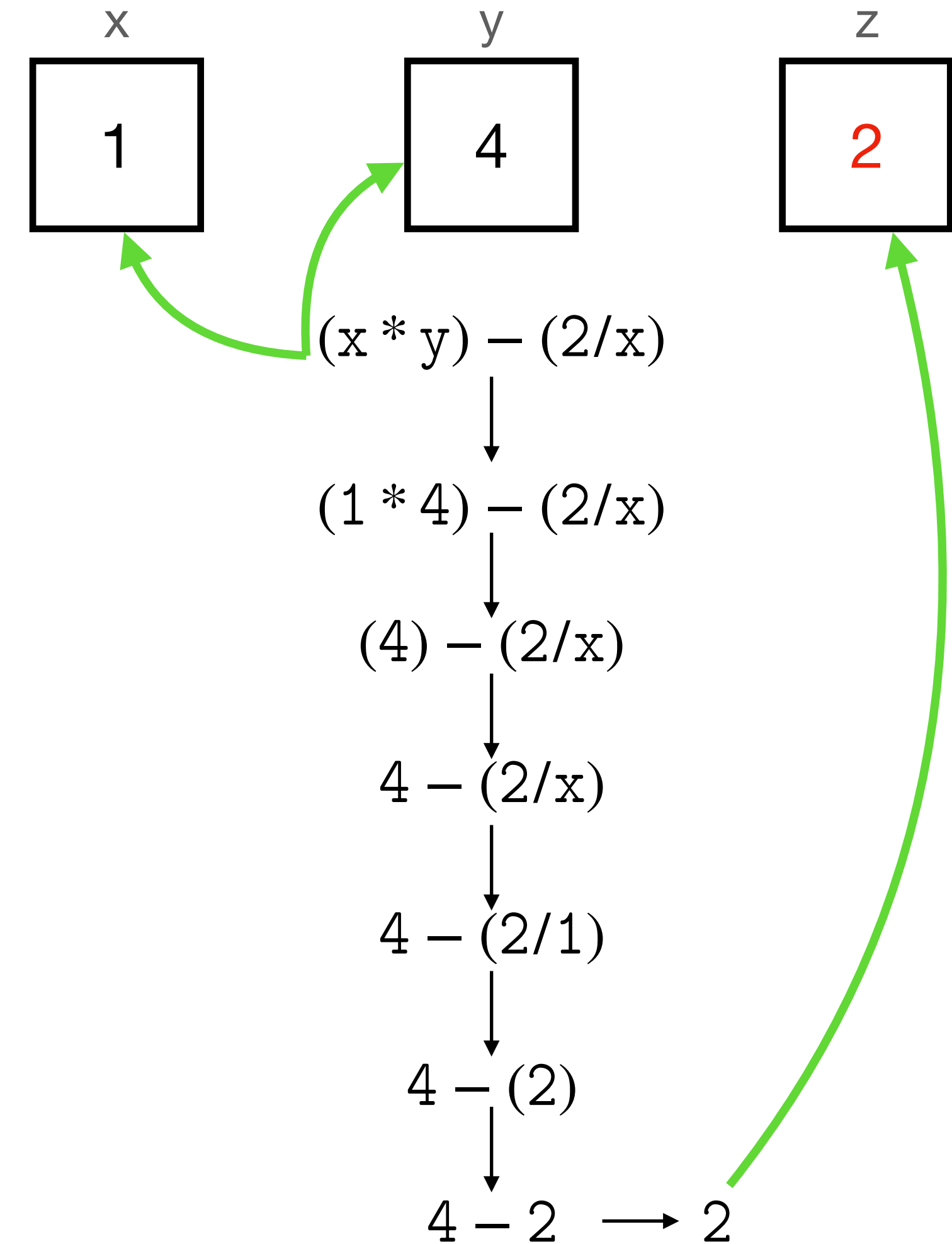


Espressioni

Valore di un'espressione



```
int x, y, z;  
x = 1;  
y = x + 3;  
z = (x * y) - (2 / x);
```



Precedenza degli operatori

$$9 + 3 * 5$$

Precedenza degli operatori

$$9 + 3 * 5$$

a cosa
valuta?

Precedenza degli operatori

$$9 + 3 * 5$$

a cosa
valuta?

24

60

Precedenza degli operatori

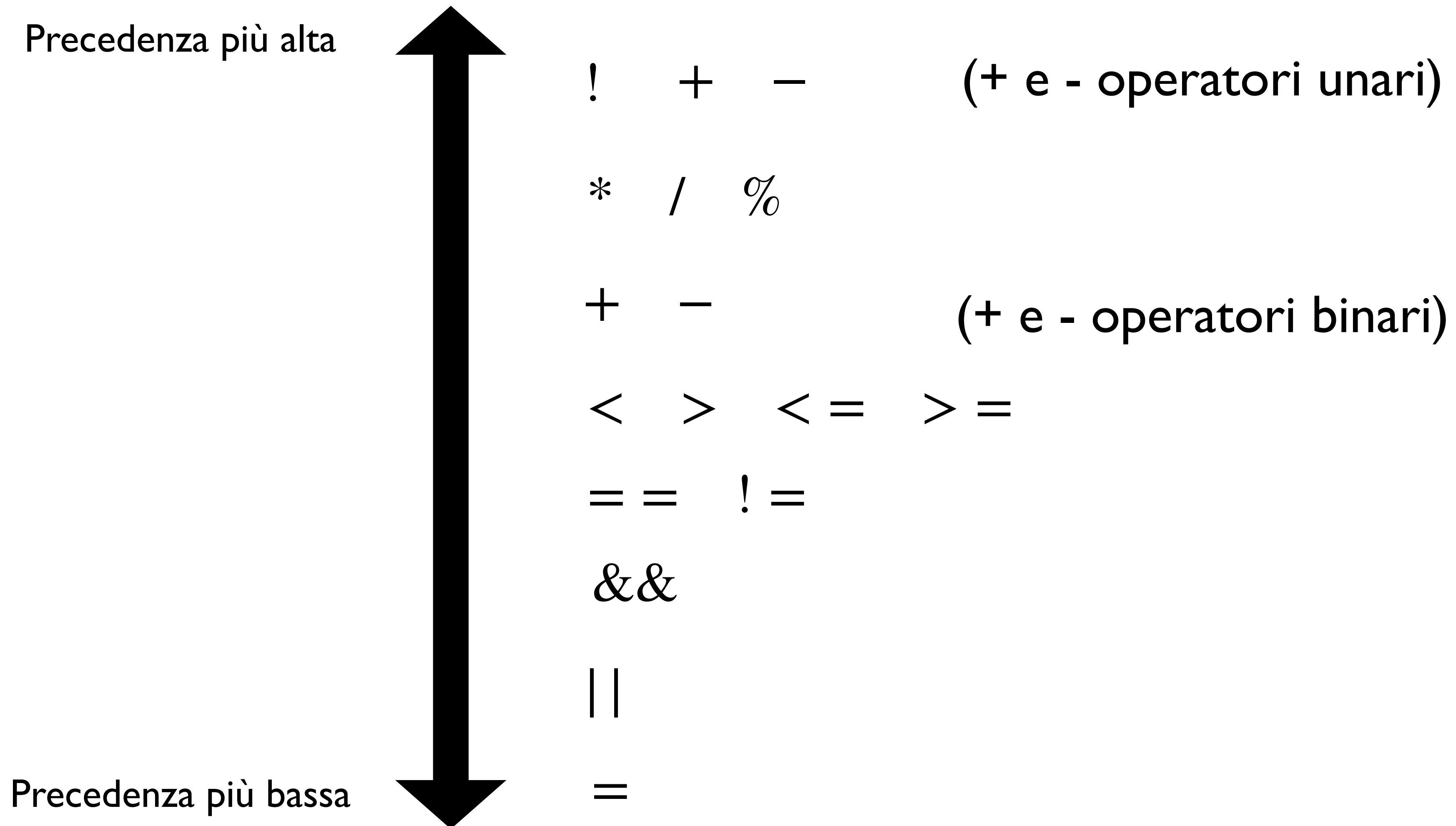
$$9 + 3 * 5$$

a cosa
valuta?

24

60

Precedenza degli operatori



Precedenza degli operatori

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a-- type() type{ } a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (type) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of ^[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <= > >=	For relational operators < and ≤ and > and ≥ respectively	
10	== !=	For equality operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13		Bitwise OR (inclusive or)	
14	&&	Logical AND	
15		Logical OR	
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= =	Ternary conditional ^[note 2] throw operator yield-expression (C++20) Direct assignment (provided by default for C++ classes) Compound assignment by sum and difference Compound assignment by product, quotient, and remainder Compound assignment by bitwise left shift and right shift Compound assignment by bitwise AND, XOR, and OR	Right-to-left
17	,	Comma	Left-to-right

from https://en.cppreference.com/w/cpp/language/operator_precedence

Precedenza degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

come se fosse

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

Come già visto, per controllare esplicitamente l'ordine di valutazione di un'espressione, possiamo usare la parentesi

Precedenza degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

come se fosse

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

Precedenza degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

come se fosse

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

Q: La somma e la sottrazione hanno la stessa precedenza!

Valuto prima $x + (y * z)$ oppure $(y * z) - ((-1)/x)$?

Precedenza degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

come se fosse

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

Q: La somma e la sottrazione hanno la stessa precedenza!

Valuto prima $x + (y * z)$ oppure $(y * z) - ((-1)/x)$?

A: Dipende da come associano gli operatori con stessa precedenza

Associatività degli operatori

- La maggior parte degli operatori associa *left-to-right*

$$1 + 4 - 7$$

associa left-to-right

$$((1 + 4) - 7)$$

Precedenza e associativi dei 60 operatori di C++: https://en.cppreference.com/w/cpp/language/operator_precedence

Associatività degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

Associatività degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

↓ applicando regole di precedenza

Associatività degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```



applicando regole di precedenza

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

Associatività degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

↓ applicando regole di precedenza

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

↓ applicando associatività degli operatori

Associatività degli operatori

```
int x = 1, y = 2, z = 3;  
int w = x + y * z - -1 / x;
```

↓ applicando regole di precedenza

```
int x = 1, y = 2, z = 3;  
int w = x + (y * z) - (-1 / x);
```

↓ applicando associatività degli operatori

```
int x = 1, y = 2, z = 3;  
int w = ((x + (y * z)) - (-1 / x));
```

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

$=$ $=$ $!$ $=$ $>$ $<$ $> =$ $< =$

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`== != > < >= <= bool`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!` `bool`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!` `bool`

`+` `-` `*` `/`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!` `bool`

`+` `-` `*` `/` `Dipende`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!` `bool`

`+` `-` `*` `/` `Dipende`

`int op int` → il tipo dell'espressione è `int`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!` `bool`

`+` `-` `*` `/` `Dipende`

`int op int` → il tipo dell'espressione è `int`

`float op float` → il tipo dell'espressione è `float`

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=`

`bool`

`&&` `||` `!`

`bool`

`+` `-` `*` `/`

Dipende

Tipo di un'espressione

- Il tipo di un'espressione è il tipo del valore risultante dalla sua valutazione e dipende dal tipo dell'operatore *principale* applicato

`==` `!=` `>` `<` `>=` `<=` `bool`

`&&` `||` `!` `bool`

`+` `-` `*` `/` `Dipende`

Overloaded operator: l'implementazione dell'operatore cambia a seconda dei tipi degli argomenti (polimorfismo ad hoc)

`5 + 3`: somma fra interi

`5.2 + 3.7`: somma fra float

Tipo di un'espressione

Conversione implicita

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

$$\text{int} < \text{float}$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

`int < float`

$$5 + 3.7$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

`int < float`

$$5 + 3.7 \longrightarrow$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

`int < float`

$$5 + 3.7 \longrightarrow 5.0 + 3.7$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

`int < float`

$$5 + 3.7 \longrightarrow 5.0 + 3.7 \longrightarrow$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

`int < float`

$$5 + 3.7 \longrightarrow 5.0 + 3.7 \longrightarrow 8.7$$

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

$$\text{int} < \text{float}$$

$$5 + 3.7 \longrightarrow 5.0 + 3.7 \longrightarrow 8.7$$

- Se gli argomenti di un'espressione sono incompatibili con la definizione dell'operatore, viene ritornato un errore a *compile-time*

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

$$\text{int} < \text{float}$$

$$5 + 3.7 \longrightarrow 5.0 + 3.7 \longrightarrow 8.7$$

- Se gli argomenti di un'espressione sono incompatibili con la definizione dell'operatore, viene ritornato un errore a *compile-time*

```
1 * "hello world"
```

Tipo di un'espressione

Conversione implicita

$$5 + 3.7$$

- L'operatore `+` si aspetta due `int` oppure due `float` come argomenti
- L'argomento di tipo “*inferiore*” viene *implicitamente convertito* a quello di livello superiore in modo tale da rendere l'operazione fattibile

`int < float`

$$5 + 3.7 \longrightarrow 5.0 + 3.7 \longrightarrow 8.7$$

- Se gli argomenti di un'espressione sono incompatibili con la definizione dell'operatore, viene ritornato un errore a *compile-time*

`1 * "hello world"`



Tipo di un'espressione

Conversione implicita

```
#include <iostream>
using namespace std;

int main() {
    int x, y, z;
    cout << "Inserisci 3 numeri interi" << endl;
    cin >> x >> y >> z;
    float m;
    m = (x + y + z) / 3.0;
    cout << "La media e " << m << endl;
    return 0;
}
```

forzata la divisione fra float