1. Ganze Zahlen

Auswertung arithmetischer Ausdrücke, Assoziativität und Präzedenz, arithmetische Operatoren, Wertebereich der Typen `int`, `unsigned int`

Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
  // Input
  std::cout << "Temperature in degrees Celsius =? ";
  int celsius;
  std::cin >> celsius;

  // Computation and output
  std::cout << celsius << " degrees Celsius are "
            << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
  return 0;
}
```

Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
  // Input
  std::cout << "Temperature in degrees Celsius =? ";
  int celsius;
  std::cin >> celsius;

  // Computation and output
  std::cout << celsius << " degrees Celsius are "
            << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
  return 0;
}
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

`9 * celsius / 5 + 32`

- Arithmetischer Ausdruck,
- enthält drei Literale, eine Variable, drei Operatorsymbole

Wie ist der Ausdruck geklammert?

Assoziativität und Präzedenz

Regel 1 : Punkt vor Strichrechnung

```
9 * celsius / 5 + 32
```

bedeutet

```
(9 * celsius / 5) + 32
```

Assoziativität und Präzedenz

Regel 2 : Von links nach rechts

```
9 * celsius / 5 + 32
```

bedeutet

```
((9 * celsius) / 5) + 32
```

Assoziativität und Präzedenz

Regel 1: Präzedenz

Multiplikative Operatoren (`*`, `/`, `%`) haben höhere Präzedenz ("binden stärker") als additive Operatoren (`+`, `-`)

Regel 2: Assoziativität

Arithmetische Operatoren (`*`, `/`, `%`, `+`, `-`) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Assoziativität und Präzedenz

Regel 3: Stelligkeit

Unäre Operatoren `+`, `-` vor binären `+`, `-`.

```
-3 - 4
```

bedeutet

```
(-3) - 4
```

Assoziativität und Präzedenz

Jeder Ausdruck kann mit Hilfe der

- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

der beteiligten Operatoren eindeutig geklammert werden (Details im Skript).

Ausdrucksbäume

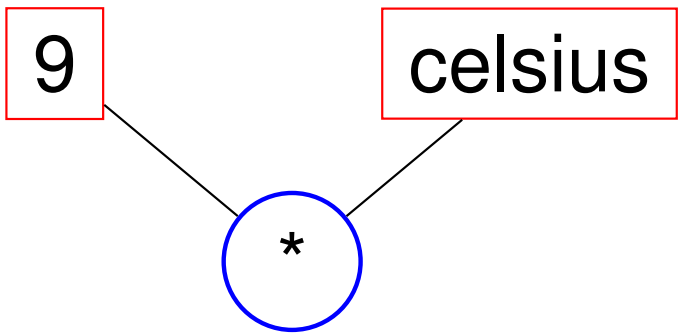
Klammerung ergibt Ausdrucksbaum

```
9 * celsius / 5 + 32
```

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

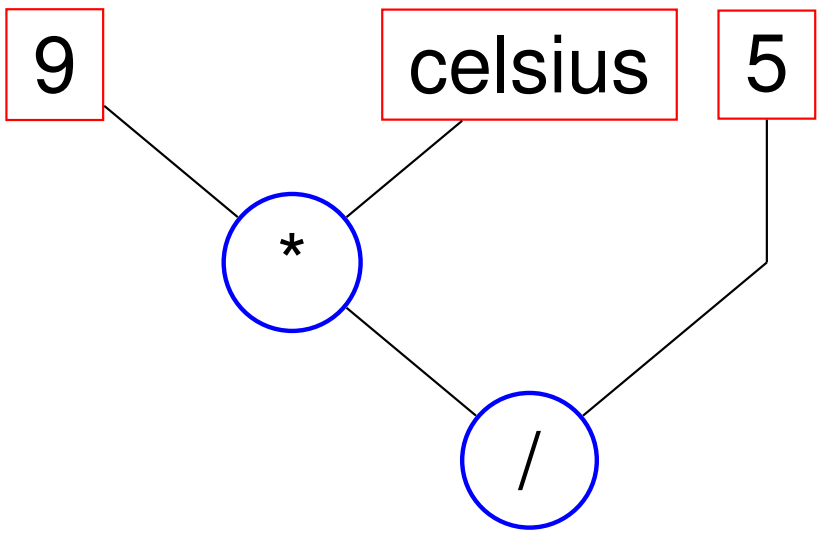
```
(9 * celsius) / 5 + 32
```



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

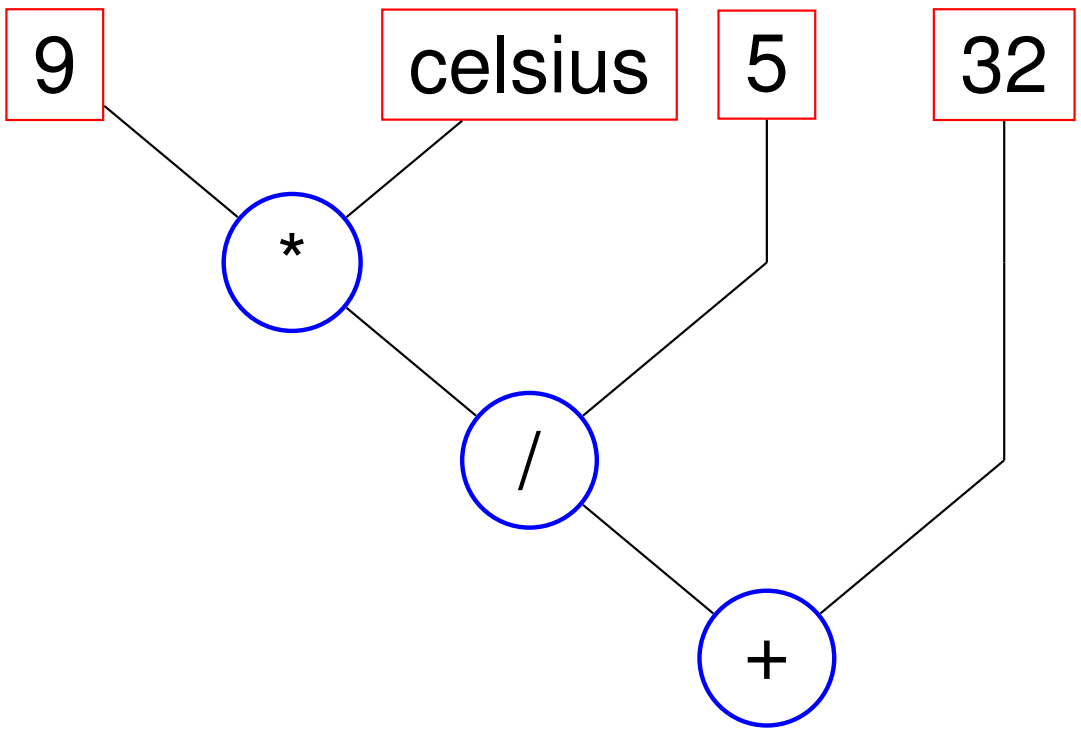
```
((9 * celsius) / 5) + 32
```



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

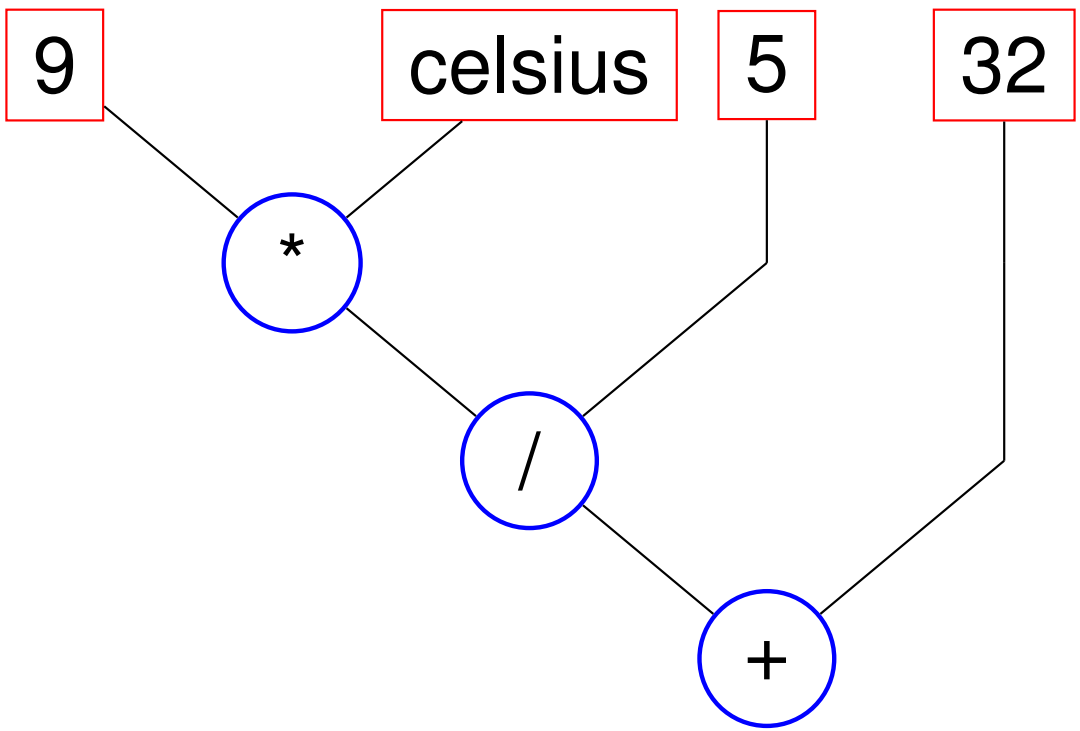
```
(((9 * celsius / 5) + 32)
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

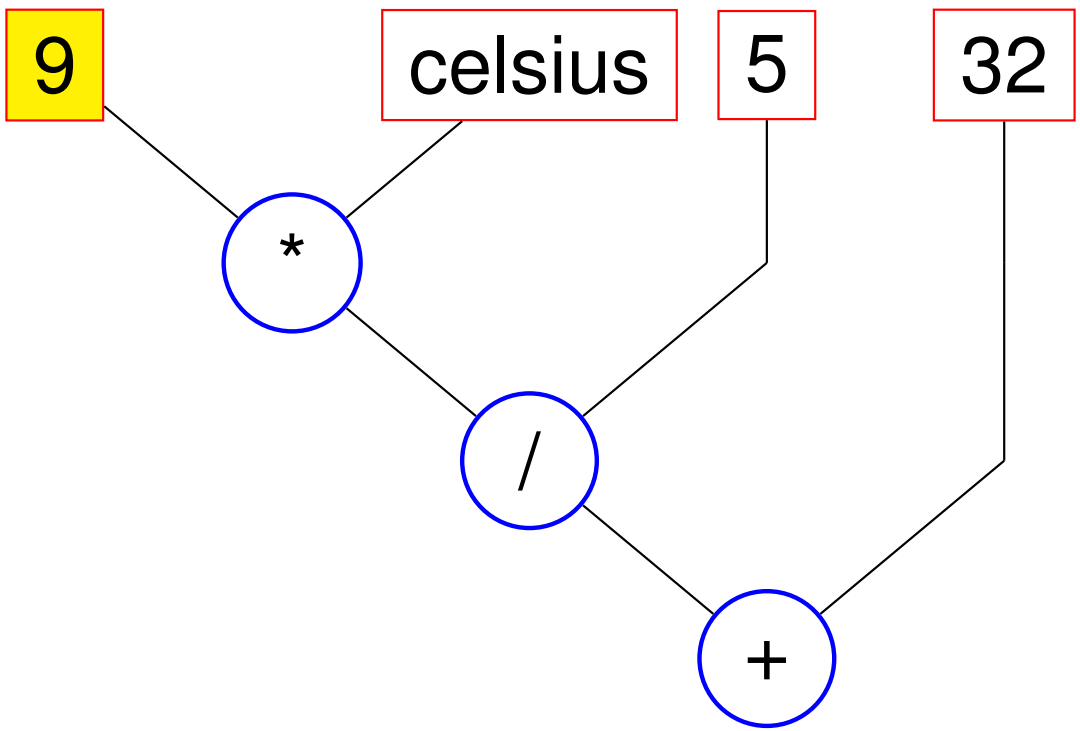
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

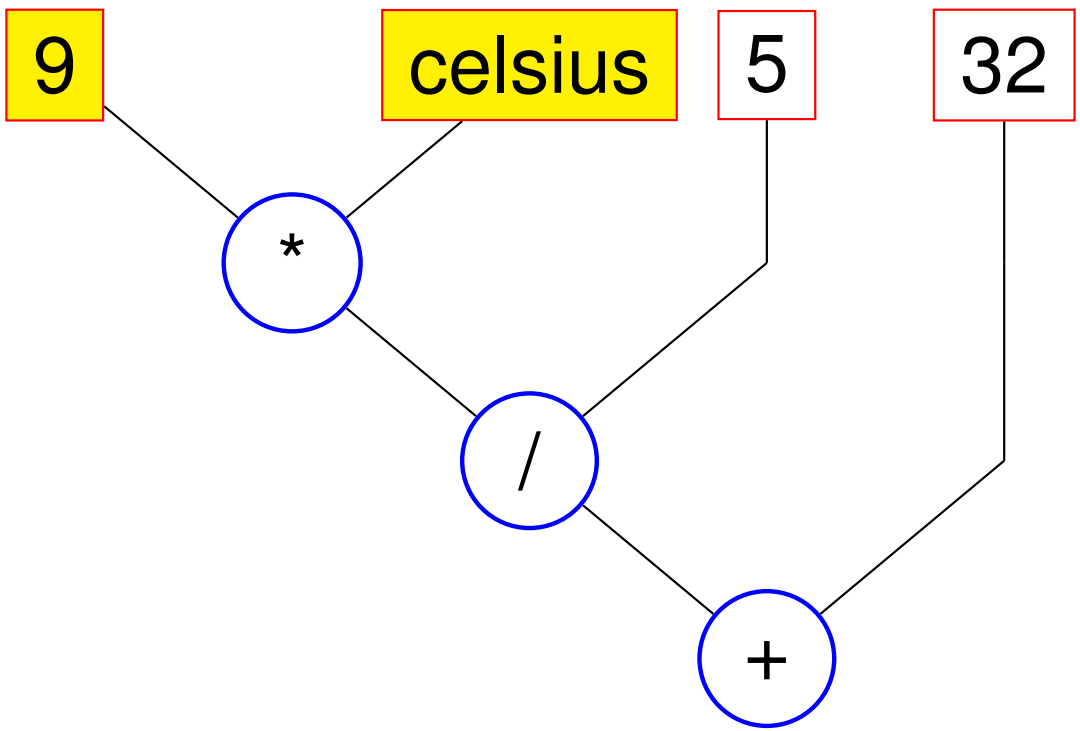
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

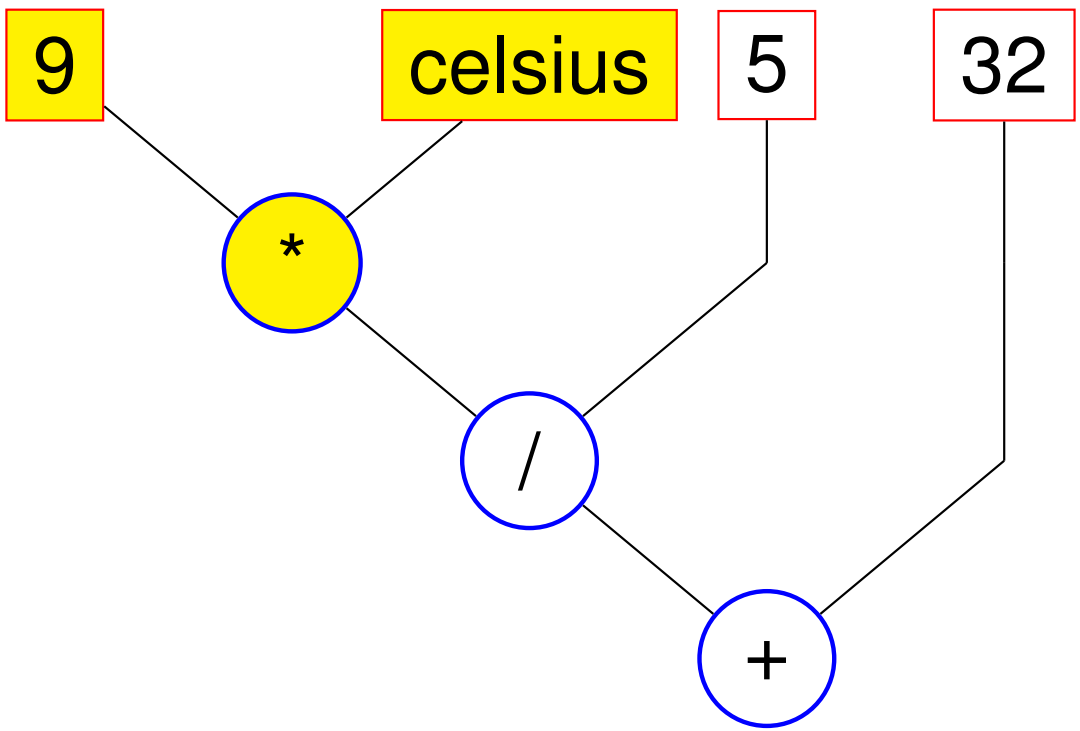
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

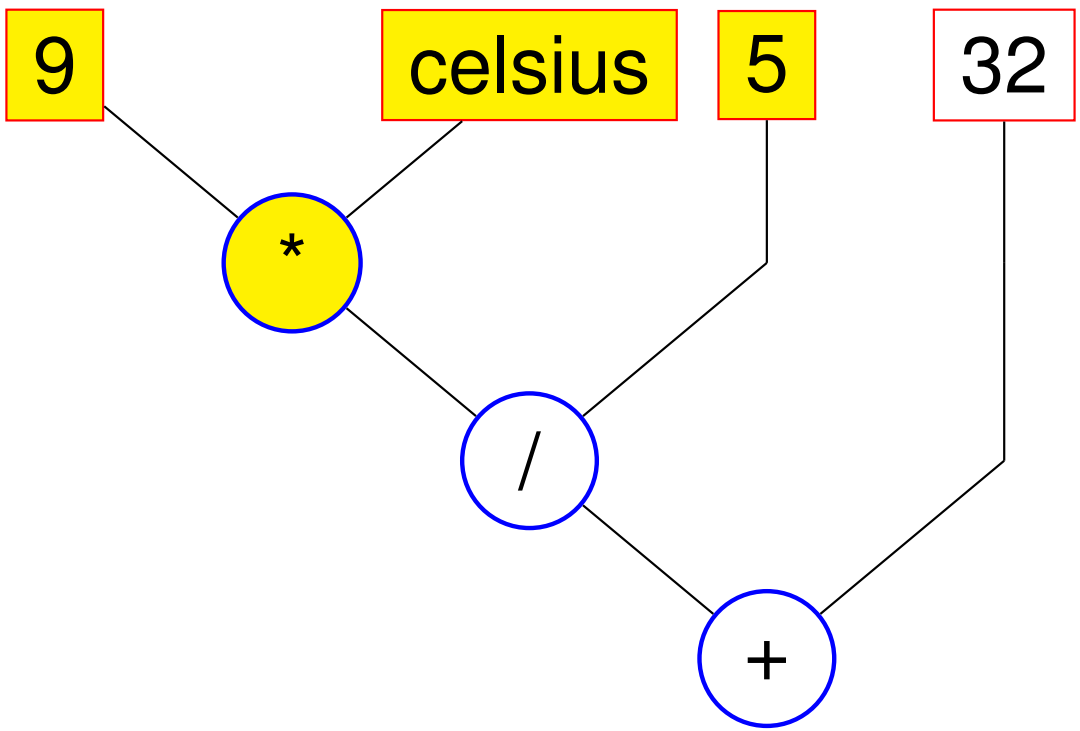
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

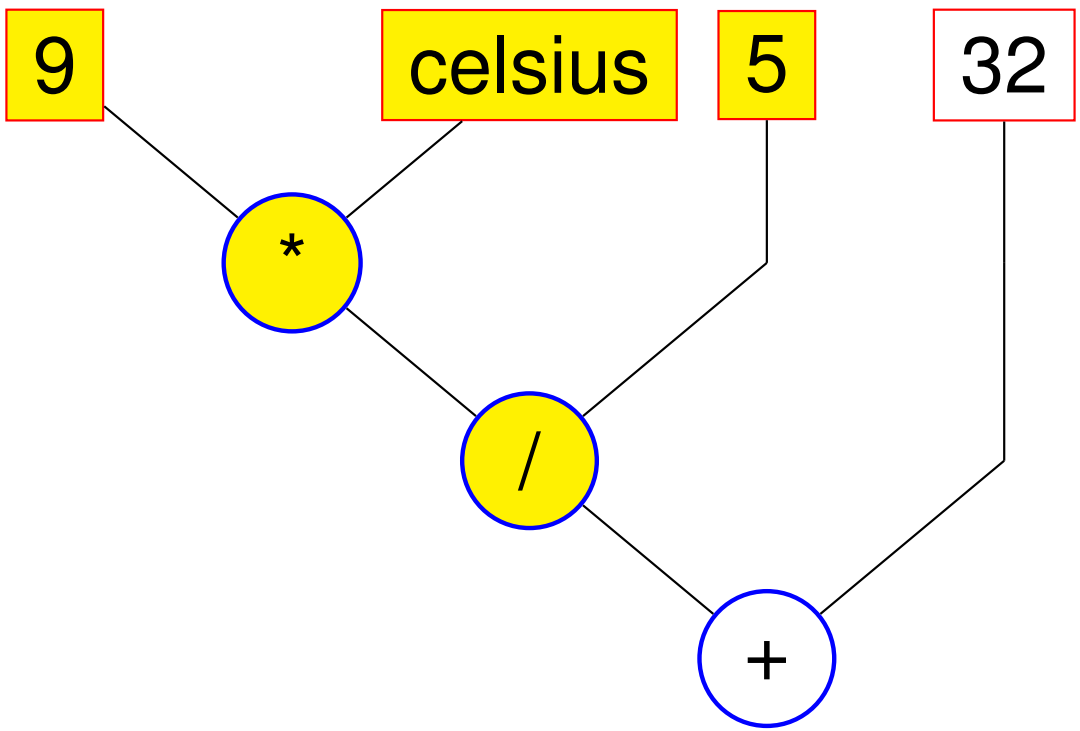
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

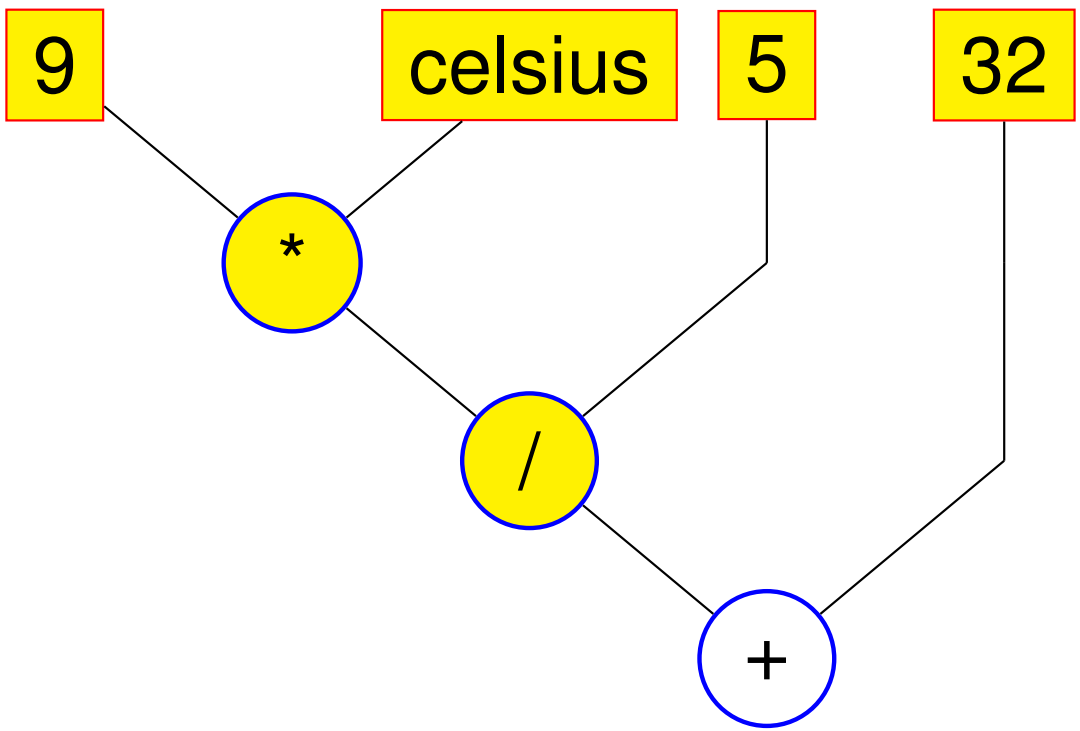
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

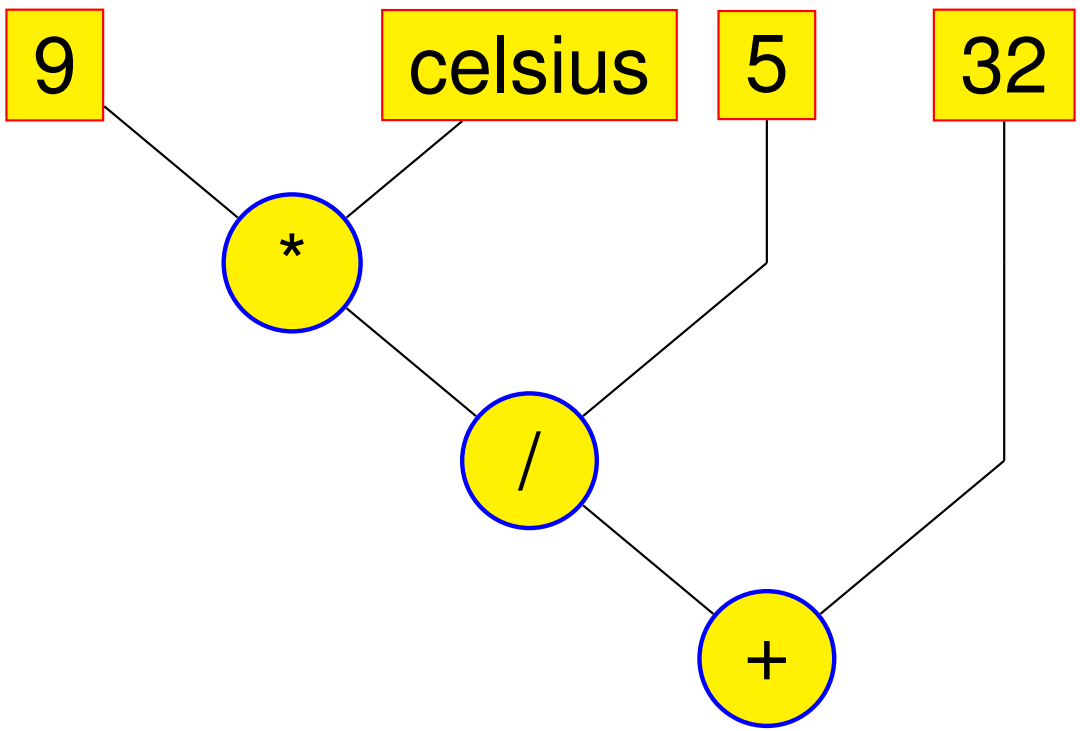
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

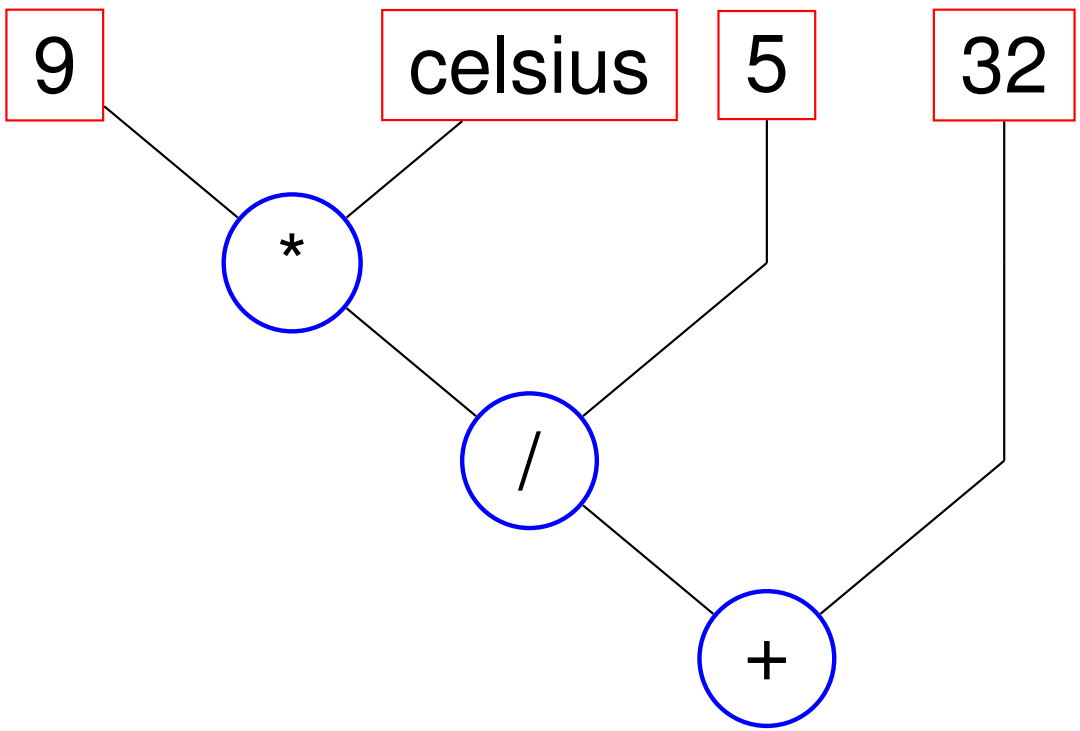
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

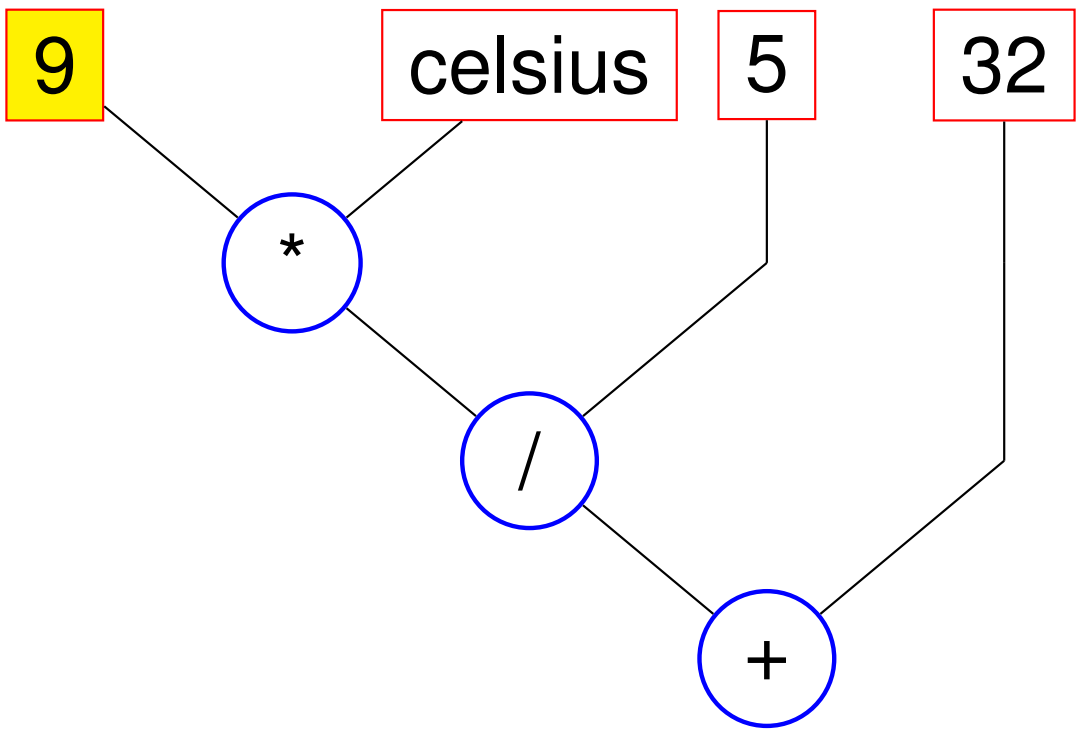
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

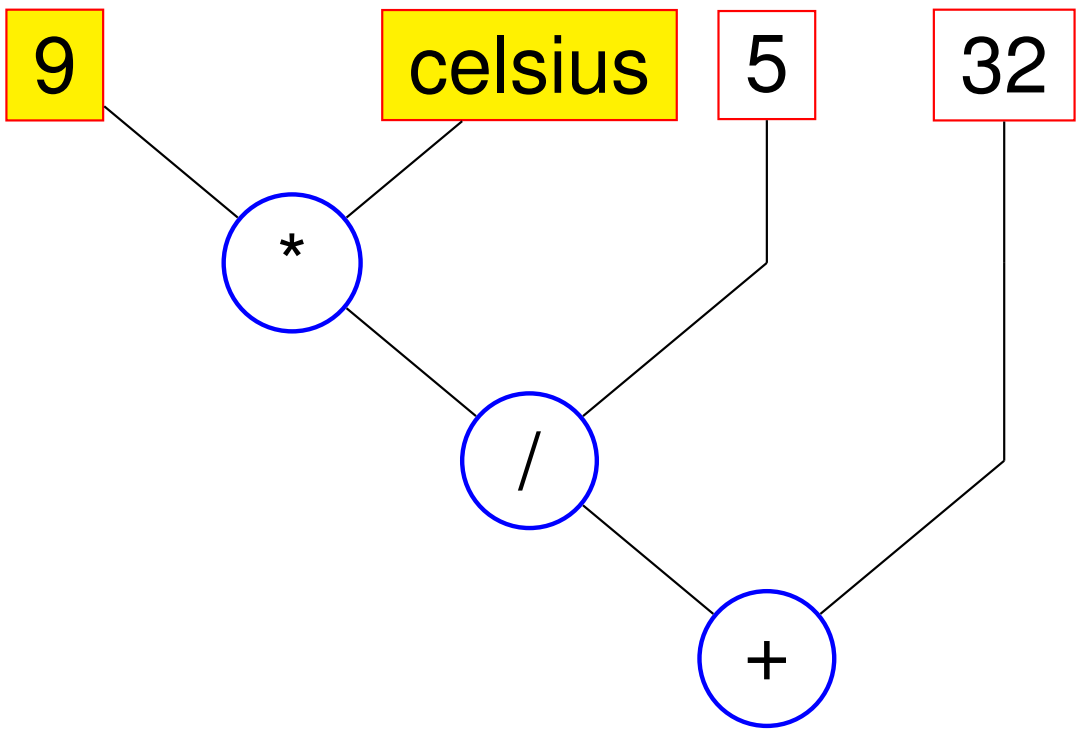
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

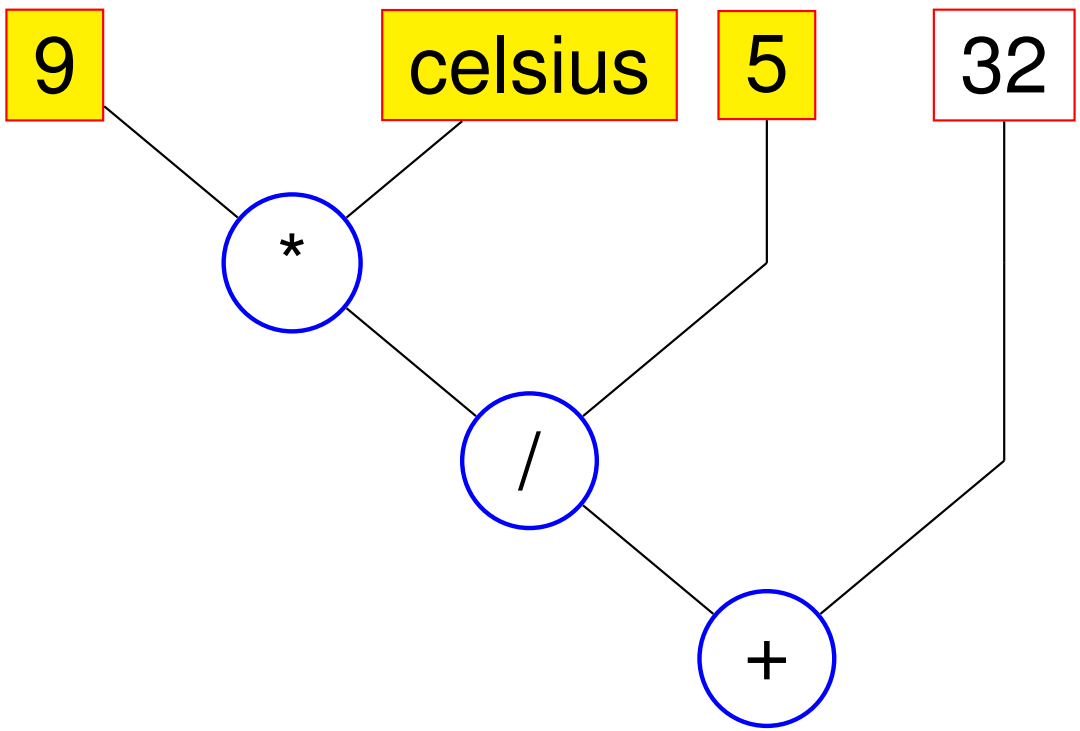
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

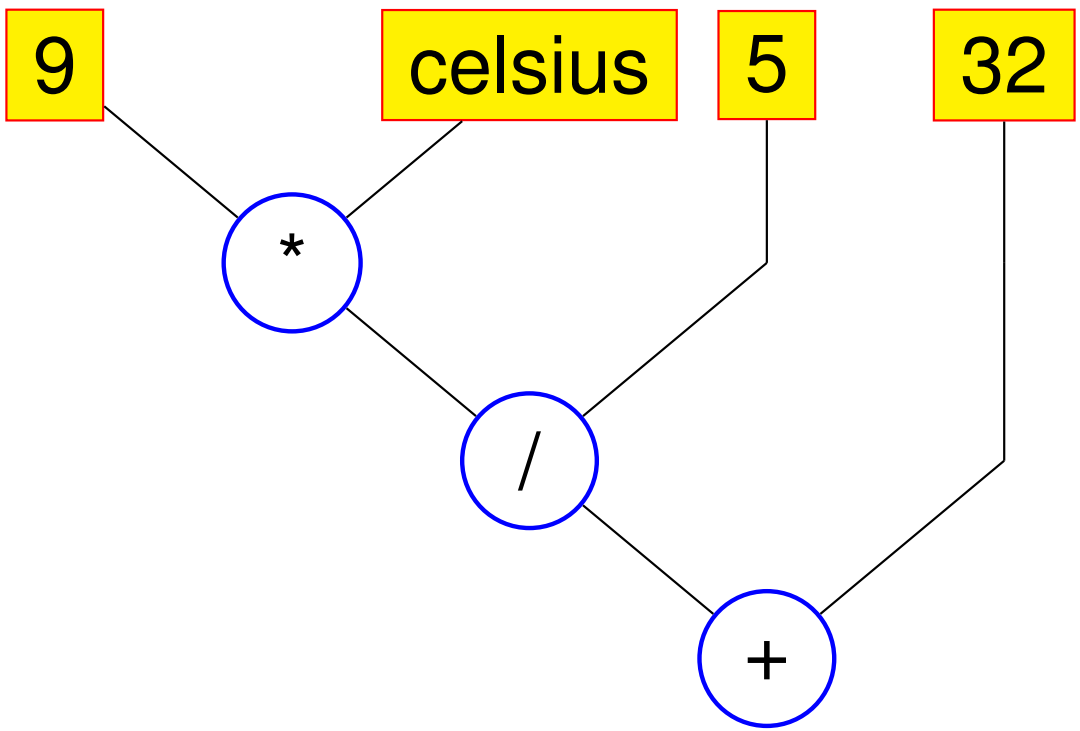
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

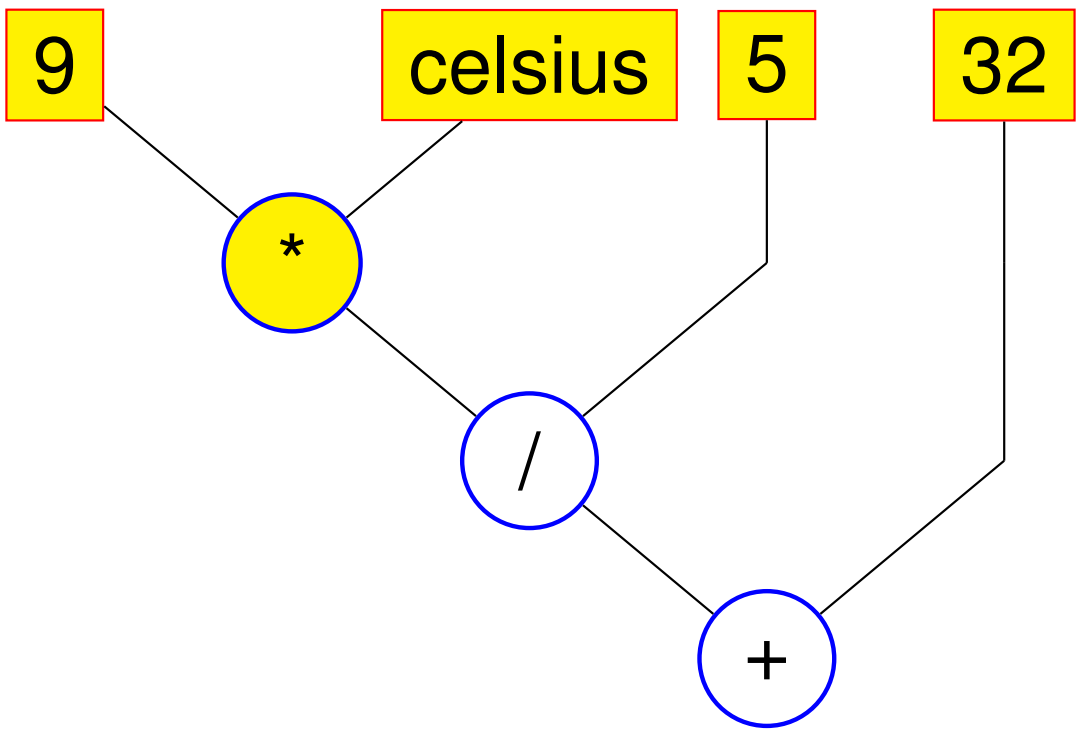
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

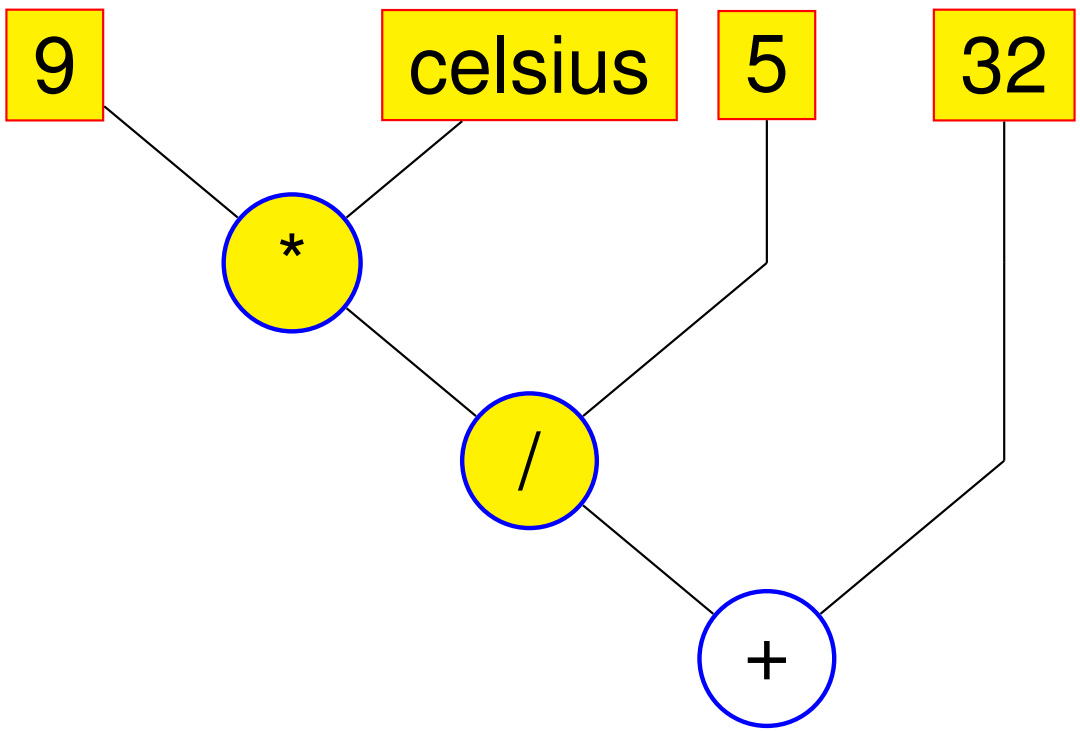
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

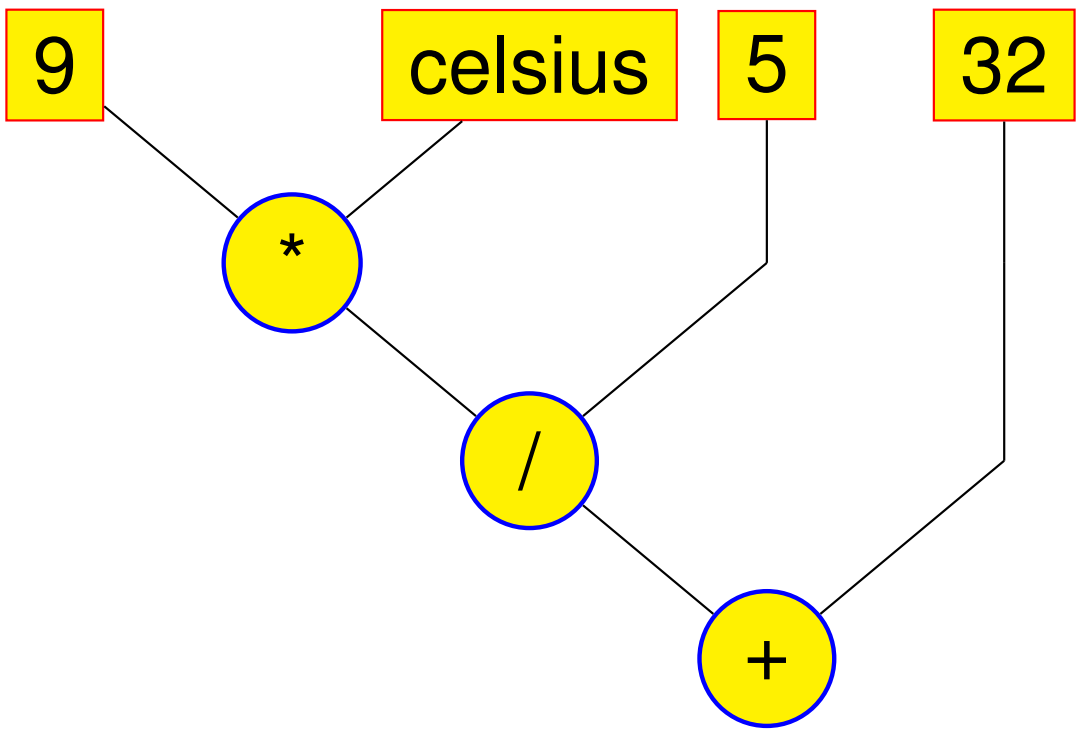
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

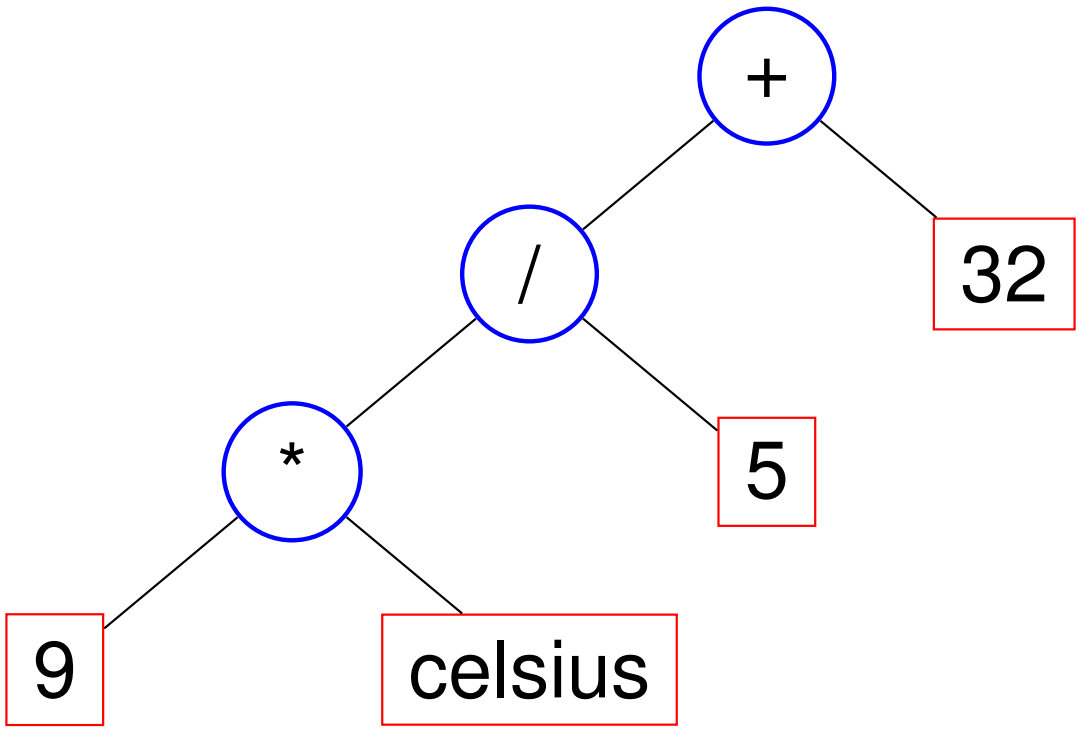
```
9 * celsius / 5 + 32
```



Ausdrucksbäume – Notation

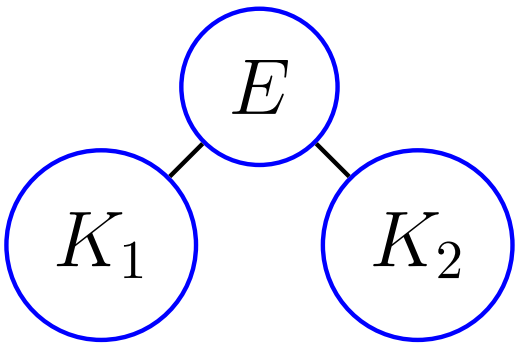
Üblichere Notation: Wurzel oben

```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.

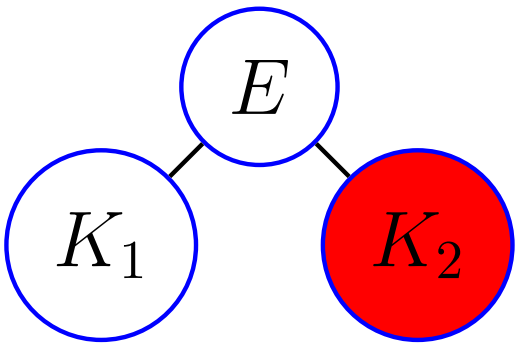


In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.

Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.

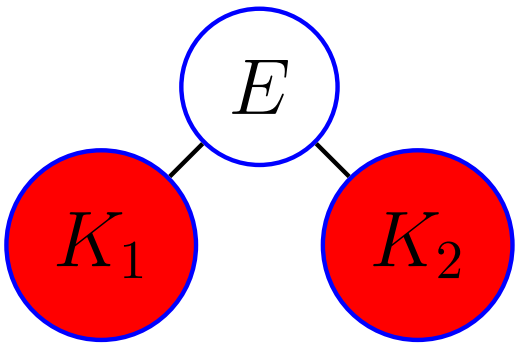


In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.

Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.

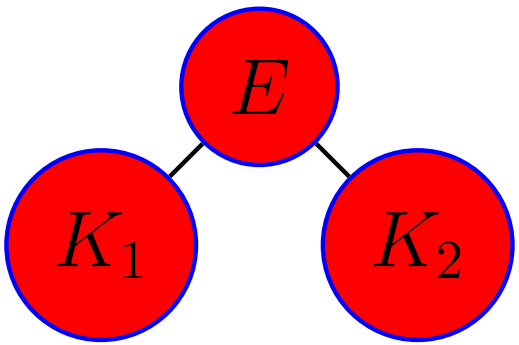


In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.

Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.

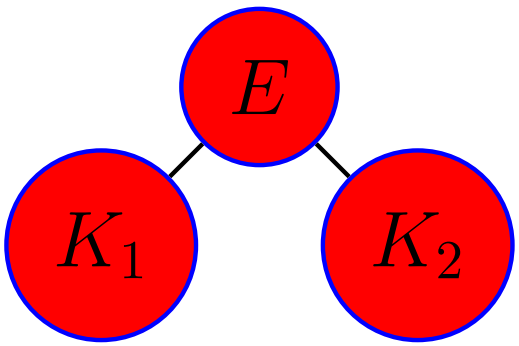


In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.

Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.
- Beispiel für schlechten Ausdruck": `(a=b)*(b=7)`

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Unäres -	–	1	16	rechts
Multiplikation	*	2	14	links
Division	/	2	14	links
Modulus	%	2	14	links
Addition	+	2	13	links
Subtraktion	–	2	13	links

Alle Operatoren: R-Wert $\times$ R-Wert $\rightarrow$ R-Wert

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In **`fahrenheit.cpp`**

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
9 / 5 * celsius + 32
```

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In **`fahrenheit.cpp`**

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
1 * celsius + 32
```

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In **`fahrenheit.cpp`**

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
15 + 32
```

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In **`fahrenheit.cpp`**

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
47
```

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In **`fahrenheit.cpp`**

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent. . . aber nicht in C++!

```
9 / 5 * celsius + 32

15 degrees Celsius are  47 degrees Fahrenheit
```

Division und Modulus

- Modulus-Operator berechnet Rest der ganzzahligen Division

`5 / 2` hat Wert 2, `5 % 2` hat Wert 1.

- Es gilt immer:

`(a / b) * b + a % b` hat den Wert von `a`.

Inkrement und Dekrement

- Erhöhen / Erniedrigen einer Zahl um 1 ist eine häufige Operation
- geht für einen L-Wert so:

```
expr = expr + 1.
```

- Nachteile:
  - relativ lang
  - expr wird zweimal ausgewertet (Effekte!)

In-/Dekrement Operatoren

	Gebrauch	Stelligkeit	Präz	Assoz.	L/R-Werte
Post-Inkrement	`expr++`	1	17	links	L-Wert → R-Wert
Prä-Inkrement	`++expr`	1	16	rechts	L-Wert → L-Wert
Post-Dekrement	`expr--`	1	17	links	L-Wert → R-Wert
Prä-Dekrement	`--expr`	1	16	rechts	L-Wert → L-Wert

	Gebrauch	Effekt und Wert
Post-Inkrement	`expr++`	Wert von expr wird um 1 erhöht, der *alte* Wert von expr wird (als R-Wert) zurückgegeben
Prä-Inkrement	`++expr`	Wert von expr wird um 1 erhöht, der *neue* Wert von expr wird (als L-Wert) zurückgegeben
Post-Dekrement	`expr--`	Wert von expr wird um 1 erniedrigt, der *alte* Wert von expr wird (als R-Wert) zurückgegeben
Prä-Dekrement	`--expr`	Wert von expr wird um 1 erniedrigt, der *neue* Wert von expr wird (als L-Wert) zurückgegeben

In-/Dekrement Operatoren

Beispiel

```
int a = 7;
std::cout << ++a << "\n";
std::cout << a++ << "\n";
std::cout << a << "\n";
```

In-/Dekrement Operatoren

Beispiel

```
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n";
std::cout << a << "\n";
```

In-/Dekrement Operatoren

Beispiel

```
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n"; // 8
std::cout << a << "\n";
```

In-/Dekrement Operatoren

Beispiel

```
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n"; // 8
std::cout << a << "\n"; // 9
```

In-/Dekrement Operatoren

Ist die Anweisung

```
++expr;
```

äquivalent zu

```
expr++;
```
?

In-/Dekrement Operatoren

Ist die Anweisung

`++expr;` ← wir bevorzugen dies

äquivalent zu

`expr++;`?

Ja, aber

- Prä-Inkrement ist effizienter (alter Wert muss nicht gespeichert werden)
- Post-In/Dekrement sind die einzigen linksassoziativen unären Operatoren (nicht sehr intuitiv)

C++ **vs.** ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,

C++ **vs.** ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,
- während C++ ja immer noch das alte C liefert.

Arithmetische Zuweisungen

	Gebrauch	Bedeutung
`+=`	`expr1 += expr2`	`expr1 = expr1 + expr2`
`-=`	`expr1 -= expr2`	`expr1 = expr1 - expr2`
`*=`	`expr1 *= expr2`	`expr1 = expr1 * expr2`
`/=`	`expr1 /= expr2`	`expr1 = expr1 / expr2`
`%=`	`expr1 %= expr2`	`expr1 = expr1 % expr2`

Arithmetische Zuweisungen werten expr1 nur einmal aus.

Wertebereich des Typs `int`

- Repräsentation mit b Bits. Wertebereich umfasst die 2^b ganzen Zahlen:

$$\{-2^{b-1}, -2^{b-1}+1, \ldots, -1, 0, 1, \ldots, 2^{b-1}-2, 2^{b-1}-1\}$$

- Auf den meisten Plattformen $b = 32$, C++ garantiert für den Typ **`int`** $b \geq 16$

Wertebereich des Typs `int`

- Repräsentation mit b Bits. Wertebereich umfasst die 2^b ganzen Zahlen:

$$\{-2^{b-1}, -2^{b-1}+1, \ldots, -1, 0, 1, \ldots, 2^{b-1}-2, 2^{b-1}-1\}$$

- Auf den meisten Plattformen $b = 32$, C++ garantiert für den Typ **`int`** $b \geq 16$
- Hintergrund: Abschnitt 2.2.8 (Binary Representation) im Skript

Wertebereich des Typs `int`

```
// Program: limits.cpp
// Output the smallest and the largest value of type int.

#include <iostream>
#include <limits>

int main()
{
  std::cout << "Minimum int value is "
            << std::numeric_limits<int>::min() << ".\n"
            << "Maximum int value is "
            << std::numeric_limits<int>::max() << ".\n";
  return 0;
}
```

Wertebereich des Typs `int`

```
// Program: limits.cpp
// Output the smallest and the largest value of type int.

#include <iostream>
#include <limits>

int main()
{
  std::cout << "Minimum int value is "
            << std::numeric_limits<int>::min() << ".\n"
            << "Maximum int value is "
            << std::numeric_limits<int>::max() << ".\n";
  return 0;
}
```

Zum Beispiel

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```

Überlauf und Unterlauf

- Arithmetische Operationen (`+`, `-`, `*`) können aus dem Wertebereich herausführen.
- Ergebnisse können inkorrekt sein.

`power8.cpp`: $15^8 = -1732076671$

`power20.cpp`: $3^{20} = -808182895$

- Es gibt *keine Fehlermeldung!*

Der Typ `unsigned int`

- Wertebereich

$$\{0, 1, \ldots, 2^b - 1\}$$

- Alle arithmetischen Operationen gibt es auch für **`unsigned int`**.
- Literale: **`1u`**, **`17u`** ...

Gemischte Ausdrücke

- Operatoren können Operanden verschiedener Typen haben (z.B. **`int`** und **`unsigned int`**).

```
17 + 17u
```

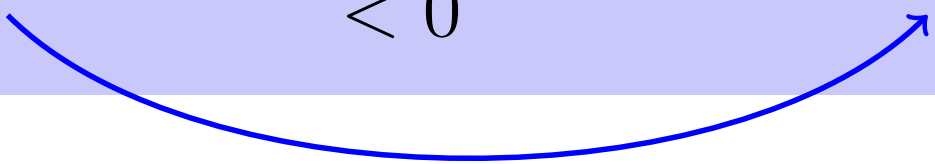
- Solche gemischten Ausdrücke sind vom „allgemeineren” Typ **`unsigned int`**.
- **`int`**-Operanden werden *konvertiert* nach **`unsigned int`**.

Konversion

`int` Wert	Vorzeichen	`unsigned int` Wert
x	≥ 0	x
x	< 0	$x + 2^b$

Konversion

`int` Wert	Vorzeichen	`unsigned int` Wert
x	≥ 0	x
x	< 0	$x + 2^b$



Bei Zweierkomplementdarstellung passiert dabei intern gar nichts

Konversion “andersherum”

Die Deklaration

```
int a = 3u;
```

konvertiert `3u` nach `int`.

Der Wert bleibt erhalten, weil er im Wertebereich von `int` liegt; andernfalls ist das Ergebnis implementierungsabhängig.