

5 Mehr Java

Java ist **statisch typisiert**, d.h., **Variablen**, **Ergebnisse von Ausdrücken**, etc. haben einen **Datentyp**, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- ▶ Basistypen / Primitive Datentypen
byte, char, short, int, long, float, double, boolean
- ▶ Referenzdatentypen
kann man auch selber definieren

5 Mehr Java

Java ist **statisch typisiert**, d.h., **Variablen**, **Ergebnisse von Ausdrücken**, etc. haben einen **Datentyp**, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- ▶ Basistypen / Primitive Datentypen
`byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
- ▶ Referenzdatentypen
kann man auch selber definieren

Beispiel – Statische Typisierung

```
a = 5
a = a + 1
a = "Hello World." # a is now a string
a = a + 1           # runtime error
```

Python

```
int a;
a = 5;
a = "Hello World." // will not compile
```

Java

5.1 Basistypen

Primitive Datentypen

- ▶ Zu jedem Basistypen gibt es eine Menge möglicher **Werte**.
- ▶ Jeder Wert eines Basistyps benötigt den gleichen **Platz**, um ihn im Rechner zu repräsentieren.
- ▶ Der Platz wird in **Bit** gemessen.

Wie viele Werte kann man mit n Bit darstellen?

Primitive Datentypen – Ganze Zahlen

Es gibt **vier** Sorten ganzer Zahlen:

<i>Typ</i>	<i>Platz</i>	<i>kleinster Wert</i>	<i>größter Wert</i>
byte	8	-128	127
short	16	-32 768	32 767
int	32	-2 147 483 648	2 147 483 647
long	64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie **byte** oder **short** spart Platz.

Primitive Datentypen – Ganze Zahlen

Literale:

- ▶ dezimale Notation
- ▶ hexadezimale Notation (Präfix `0x` oder `0X`)
- ▶ oktale Notation (Präfix `0`)
- ▶ binäre Notation (Präfix `0b` oder `0B`)
- ▶ Suffix `l` ☹️ oder `L` für `long`
- ▶ `'_'` um Ziffern zu gruppieren

Beispiele

- ▶ `192, 0b11000000, 0xC0, 0300` sind alle gleich
- ▶ `20_000L, 0xABFF_0078L`
- ▶ `09, 0x_FF` sind ungültig

Primitive Datentypen – Ganze Zahlen

Literale:

- ▶ dezimale Notation
- ▶ hexadezimale Notation (Präfix `0x` oder `0X`)
- ▶ oktale Notation (Präfix `0`)
- ▶ binäre Notation (Präfix `0b` oder `0B`)
- ▶ Suffix `l` ☹️ oder `L` für `long`
- ▶ `'_'` um Ziffern zu gruppieren

Beispiele

- ▶ `192`, `0b11000000`, `0xC0`, `0300` sind alle gleich
- ▶ `20_000L`, `0xABFF_0078L`
- ▶ `09`, `0xFF` sind ungültig

Primitive Datentypen – Ganze Zahlen

Achtung: Java warnt nicht vor Überlauf/Unterlauf!!!

Beispiel:

```
1 int x = 2147483647; // groesstes int
2 x = x + 1;
3 write(x);
```

liefert: -2147483648

Primitive Datentypen – Gleitkommazahlen

Es gibt **zwei** Sorten von Gleitkommazahlen:

<i>Typ</i>	<i>Platz</i>	<i>kleinster Wert</i>	<i>größter Wert</i>	<i>signifikante Stellen</i>
float	32	ca. $-3.4 \cdot 10^{38}$	ca. $3.4 \cdot 10^{38}$	ca. 7
double	64	ca. $-1.7 \cdot 10^{308}$	ca. $1.7 \cdot 10^{308}$	ca. 15

$$x = s \cdot m \cdot 2^e \quad \text{mit } 1 \leq m < 2$$

- ▶ Vorzeichen s : 1 bit
- ▶ reduzierte Mantisse $m - 1$: 23 bit (float), 52 bit (double)
- ▶ Exponent e : 8 bit (float), 11 bit (double)

Primitive Datentypen – Gleitkommazahlen

Literale:

- ▶ dezimale Notation.
- ▶ dezimale Exponentialschreibweise (e, E für Exponent)
- ▶ hexadezimale Exponentialschreibweise. (Präfix 0x oder 0X, p oder P für Exponent)
- ▶ Suffix f oder F für float, Suffix d oder D für double (default is double)

Beispiele

- ▶ 640.5F == 0x50.1p3f
- ▶ 3.1415 == 314.15E-2
- ▶ 0x1e3_dp0, 1e3d
- ▶ 0x1e3d, 1e3_d, 0x50.1

Primitive Datentypen – Gleitkommazahlen

- ▶ Überlauf/Unterlauf bei Berechnungen liefert **Infinity**, bzw. **-Infinity**
- ▶ Division Null durch Null, Wurzel aus einer negativen Zahl etc. liefert **NaN**

Weitere Basistypen

<i>Typ</i>	<i>Platz</i>	<i>Werte</i>
<code>boolean</code>	1	<code>true</code> , <code>false</code>
<code>char</code>	16	alle(?) <code>Unicode</code> -Zeichen

`Unicode` ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

- ▶ die Zeichen unserer Tastatur (inklusive Umlaute);
- ▶ die chinesischen Schriftzeichen;
- ▶ die ägyptischen Hieroglyphen ...

Literale:

- ▶ `char`-Literale schreibt man in Hochkommas: `'A'`, `'\u00ED'`, `';'`, `'\n'`.
- ▶ `boolean`-Literale sind `true` und `false`.

5.2 Strings

Der Datentyp `String` für Wörter ist ein Referenzdatentyp (genauer eine `Klasse` (dazu kommen wir später)).

Hier nur drei Eigenschaften:

- ▶ Literale vom Typ `String` haben die Form `"Hello World!"`;
- ▶ Man kann Wörter in Variablen vom Typ `String` abspeichern;
- ▶ Man kann Wörter mithilfe des Operators `'+'` konkatenieren.

Beispiel

```
String s0 = "";  
String s1 = "Hel";  
String s2 = "lo Wo";  
String s3 = "rld!";  
  
write(s0 + s1 + s2 + s3);
```

...liefert: Hello World!

5.3 Auswertung von Ausdrücken

Funktionen in **Java** bekommen **Parameter**/Argumente als Input, und liefern als Output den Wert eines vorbestimmten Typs. Zum Beispiel könnte man eine Funktion

```
int min(int a, int b)
```

implementieren, die das Minimum ihrer Argumente zurückliefert.

Operatoren sind spezielle vordefinierte Funktionen, die in **Infix**-Notation geschrieben werden (wenn sie binär sind):

```
a + b = +(a,b)
```

5.3 Auswertung von Ausdrücken

Ein **Ausdruck** ist eine Kombination von Literalen, Operatoren, Funktionen, Variablen und Klammern, die verwendet wird, um einen Wert zu berechnen.

Beispiele: (x z.B. vom Typ `int`)

- ▶ `7 + 4`
- ▶ `3 / 5 + 3`
- ▶ `min(3,x) + 20`
- ▶ `x = 7`
- ▶ `x *= 2`

Unäre Operatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
++	Post-inkrement	(var) Zahl, char	keine	2
--	Post-dekrement	(var) Zahl, char	keine	2
++	Pre-inkrement	(var) Zahl, char	rechts	3
--	Pre-dekrement	(var) Zahl, char	rechts	3
+	unäres Plus	Zahl, char	rechts	3
-	unäres Minus	Zahl, char	rechts	3
!	Negation	boolean	rechts	3

Achtung



Prefix- und Postfixoperator

- ▶ Die Operatoranwendungen `++x` und `x++` inkrementieren beide den Wert der Variablen `x` (als **Seiteneffekt**).
- ▶ `++x` tut das, **bevor** der Wert des Ausdrucks ermittelt wird (**Pre-Inkrement**).
- ▶ `x++` tut das, **nachdem** der Wert ermittelt wurde (**Post-Inkrement**).

- ▶ `b = x++;` entspricht:

```
b = x;  
x = x + 1;
```

- ▶ `b = ++x;` entspricht:

```
x = x + 1;  
b = x;
```

Operatoren

Binäre arithmetische Operatoren:

byte, short, char werden nach int konvertiert

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
*	Multiplikation	Zahl, char	links	4
/	Division	Zahl, char	links	4
%	Modulo	Zahl, char	links	4
+	Addition	Zahl, char	links	5
-	Subtraktion	Zahl, char	links	5

Konkatenation

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
+	Konkatenation	String	links	5

Vergleichsoperatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
>	größer	Zahl, char	keine	7
>=	größergleich	Zahl, char	keine	7
<	kleiner	Zahl, char	keine	7
<=	kleinergleich	Zahl, char	keine	7
==	gleich	alle	links	8
!=	ungleich	alle	links	8

Boolsche Operatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
&&	Und-Bedingung	boolean	links	12
	Oder-Bedingung	boolean	links	13

Zuweisungsoperatoren:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
=	Zuweisung	(links var) alle	rechts	15
*=, /=, %= +=, -=	Zuweisung	(links var) alle	rechts	15

Für die letzte Form gilt:

$$v \Leftarrow a \iff v = (\text{type}(v)) (v \circ a)$$

Warnung:

- ▶ Eine Zuweisung $x = y$; ist in Wahrheit ein **Ausdruck**.
- ▶ Der Wert ist der Wert der rechten Seite.
- ▶ Die Modifizierung der Variablen x erfolgt als **Seiteneffekt**.
- ▶ Das Semikolon ';' hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;  
if (x = true)  
    write("Sorry! This must be an error ...");
```

5.3 Auswertung von Ausdrücken

Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Assoziativität

- ▶ Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- ▶ Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- ▶ Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- ▶ unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- ▶ unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- ▶ Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- ▶ eine Variable (**var**),
- ▶ einen reinen Wert (**val**) oder
- ▶ void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- ▶ Typ, z.B.: **int**, und einen
- ▶ Wert, z.B.: **42**

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

5.3 Auswertung von Ausdrücken

In **Java** werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- ▶ werte (rekursiv) alle Operanden von links nach rechts aus
- ▶ führe die Operation auf den Resultaten aus

Ausnahmen: `||`, `&&`, und der ternäre Bedingungsoperator `?:`, werten nicht alle Operanden aus (**Kurzschlussauswertung**).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

5.3 Auswertung von Ausdrücken

In **Java** werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- ▶ werte (rekursiv) alle Operanden von links nach rechts aus
- ▶ führe die Operation auf den Resultaten aus

Ausnahmen: `||`, `&&`, und der ternäre Bedingungsoperator `?:`, werten nicht alle Operanden aus (**Kurzschlussauswertung**).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

5.3 Auswertung von Ausdrücken

Im Folgenden betrachten wir Klammern als einen Operator der nichts tut:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
()	Klammerung	alle	links	0

Beispiel: $2 + x * (z - d)$

2 + x * (z - d)

Beispiel: $2 + x * (z - d)$



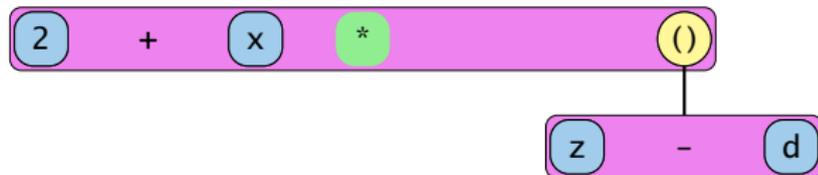
Beispiel: $2 + x * (z - d)$



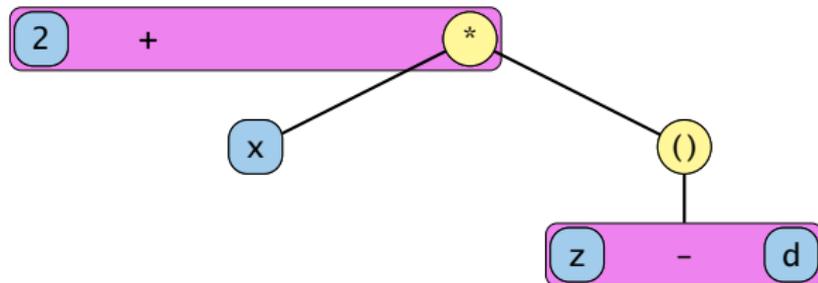
Beispiel: $2 + x * (z - d)$



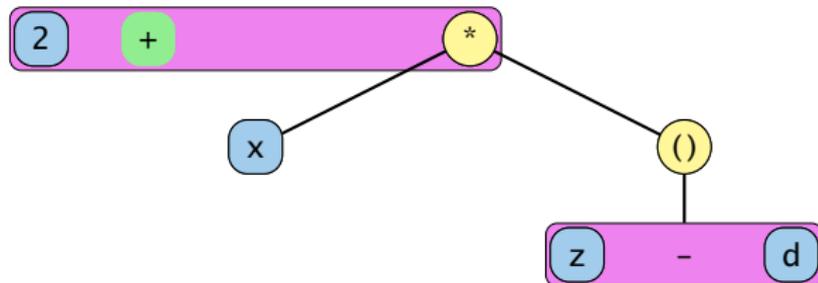
Beispiel: $2 + x * (z - d)$



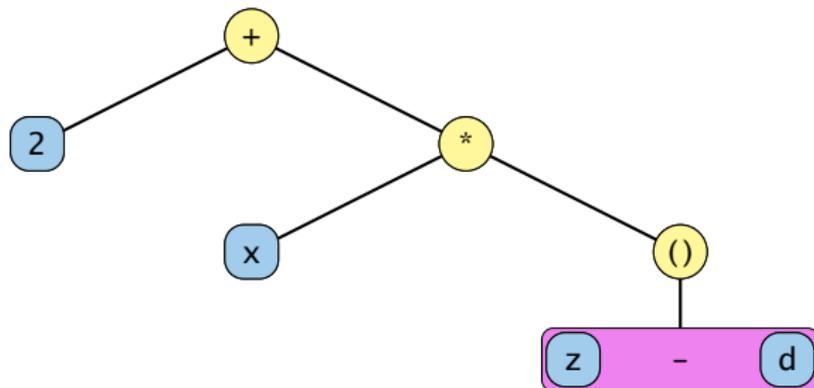
Beispiel: $2 + x * (z - d)$



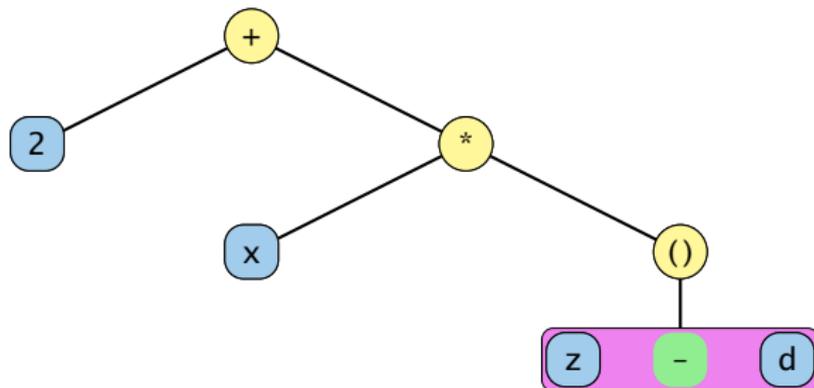
Beispiel: $2 + x * (z - d)$



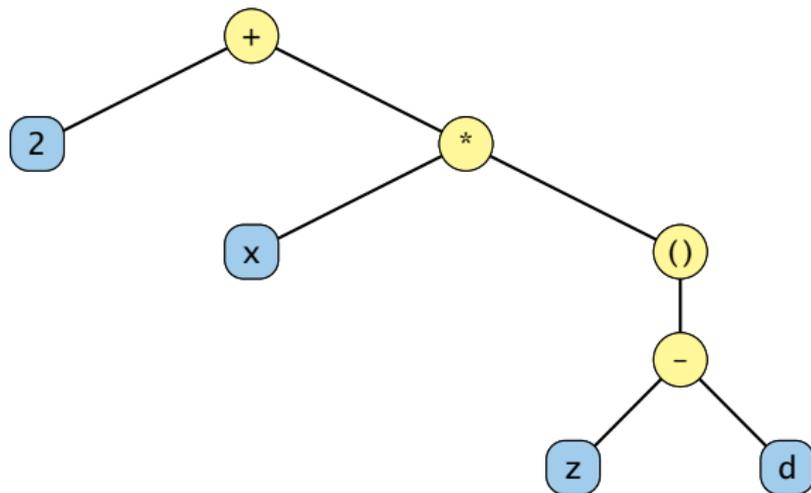
Beispiel: $2 + x * (z - d)$



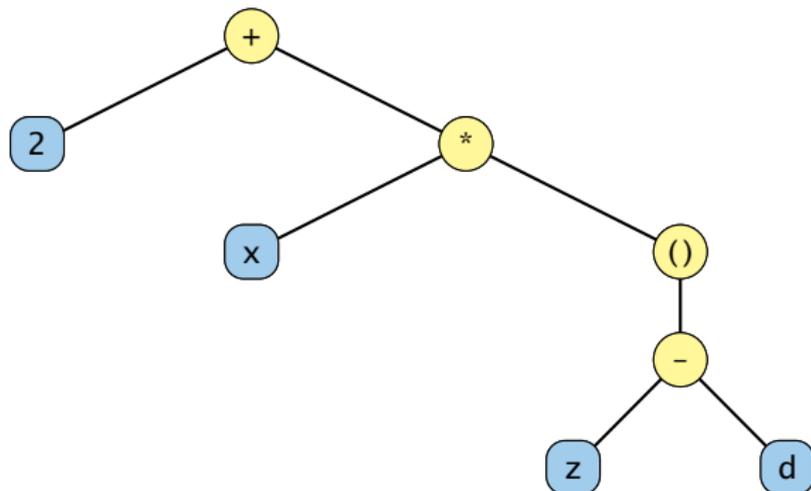
Beispiel: $2 + x * (z - d)$



Beispiel: $2 + x * (z - d)$



Beispiel: $2 + x * (z - d)$

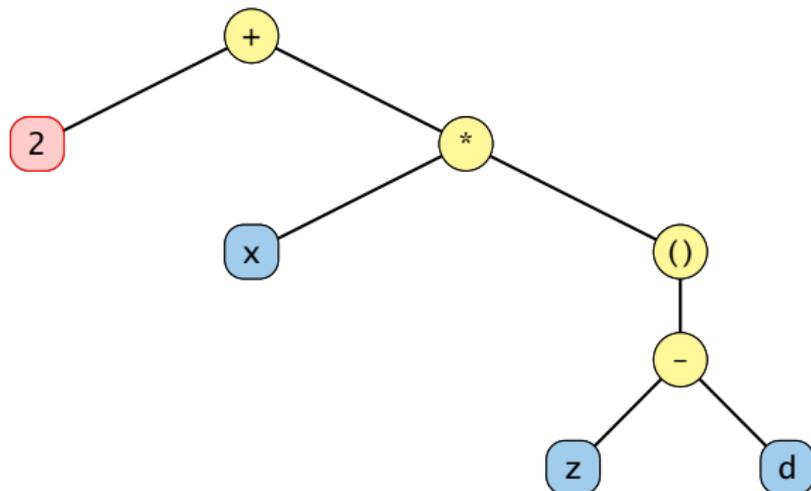


x

d

z

Beispiel: $2 + x * (z - d)$

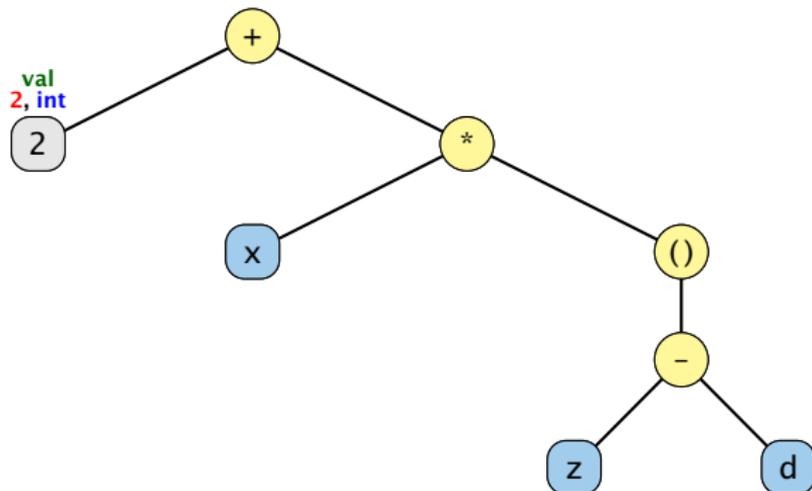


x

d

z

Beispiel: $2 + x * (z - d)$

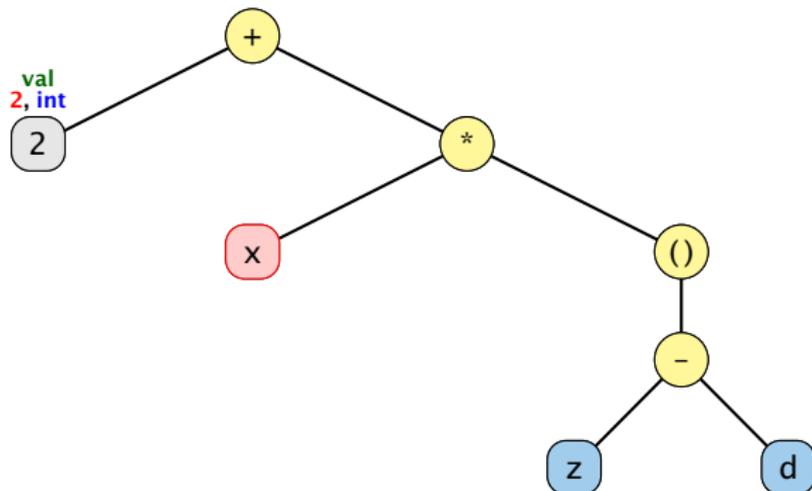


x

d

z

Beispiel: $2 + x * (z - d)$

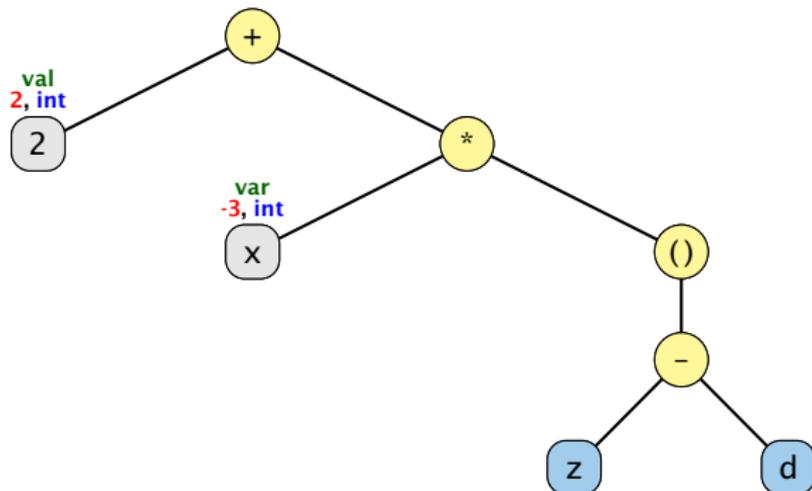


x

d

z

Beispiel: $2 + x * (z - d)$

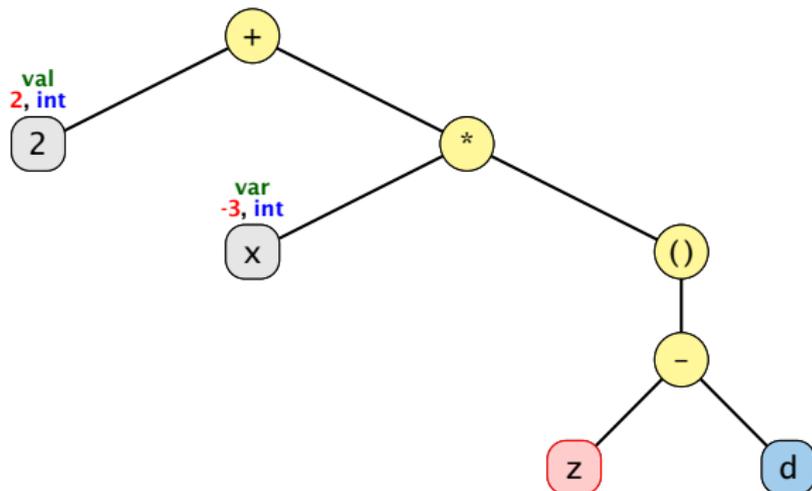


x

d

z

Beispiel: $2 + x * (z - d)$

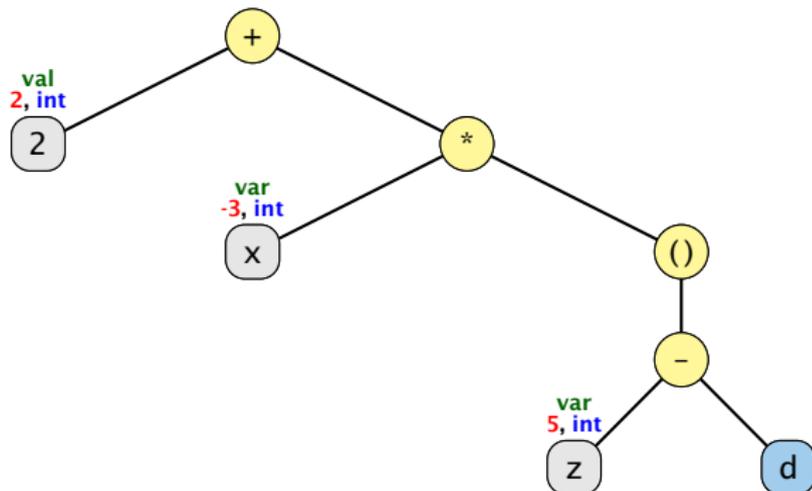


x

d

z

Beispiel: $2 + x * (z - d)$

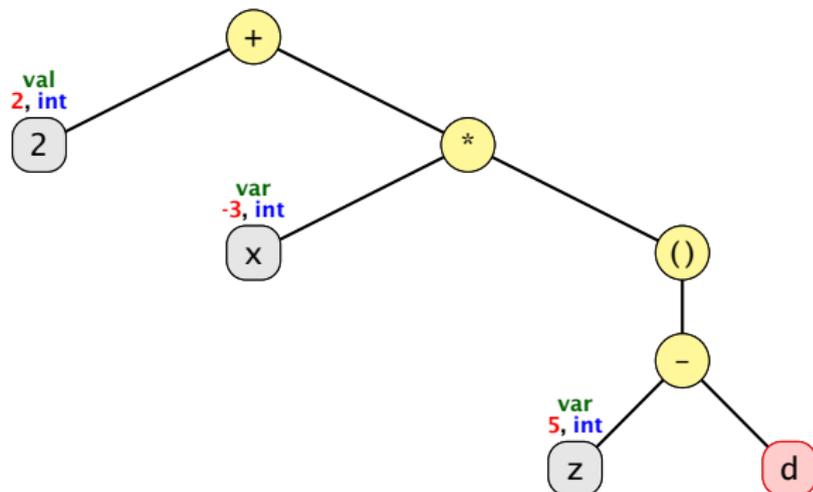


x

d

z

Beispiel: $2 + x * (z - d)$

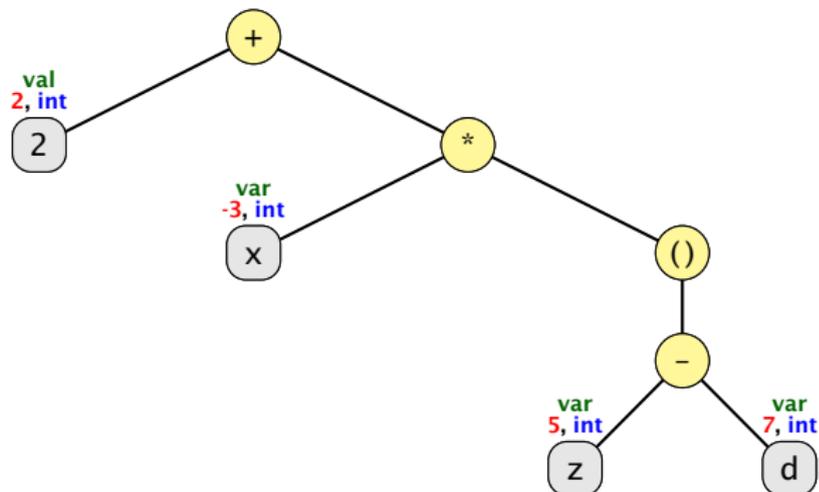


x

d

z

Beispiel: $2 + x * (z - d)$

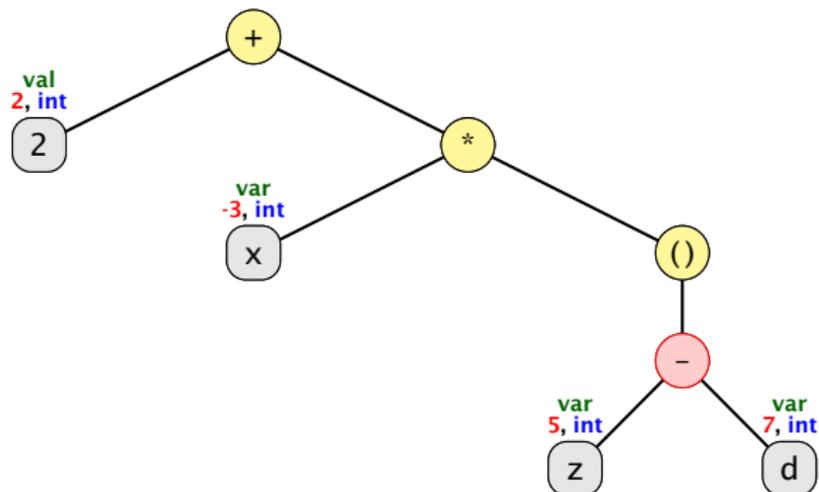


x

d

z

Beispiel: $2 + x * (z - d)$

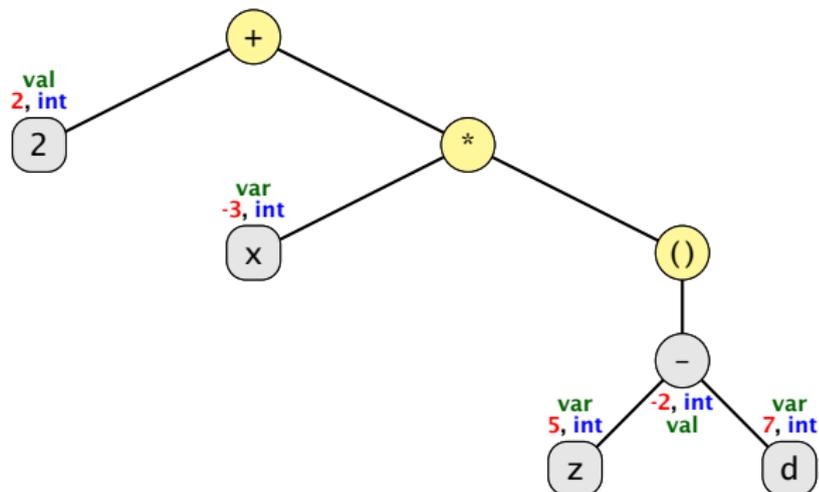


x

d

z

Beispiel: $2 + x * (z - d)$

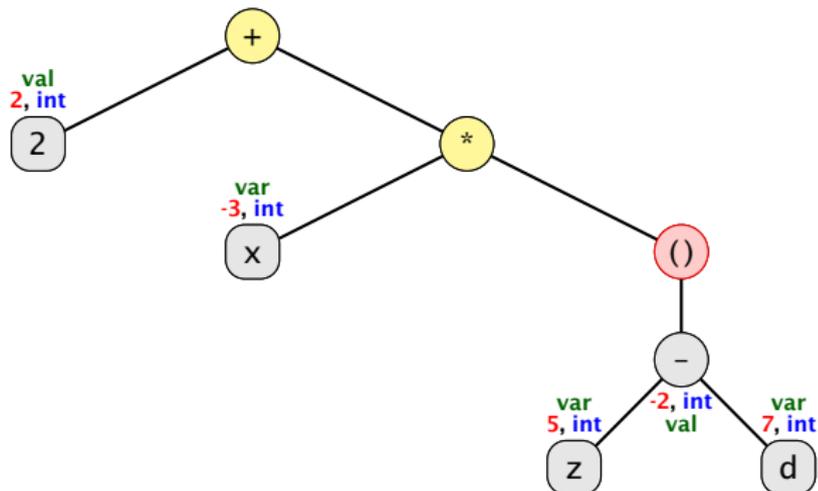


x

d

z

Beispiel: $2 + x * (z - d)$

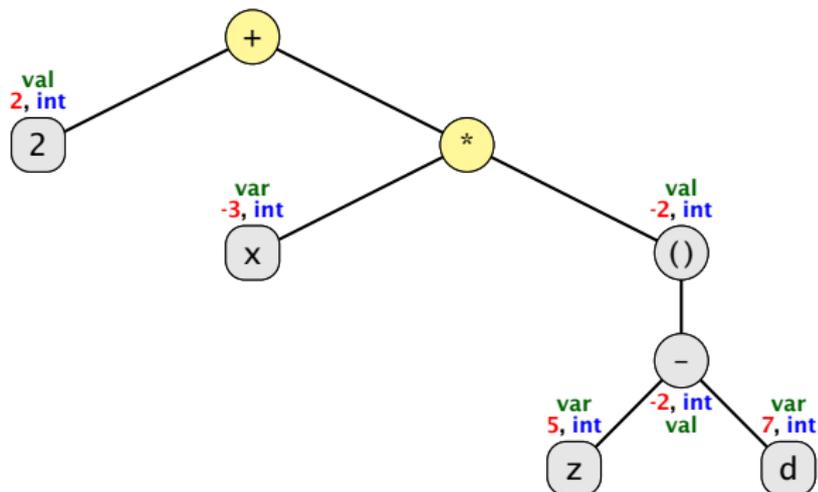


x

d

z

Beispiel: $2 + x * (z - d)$

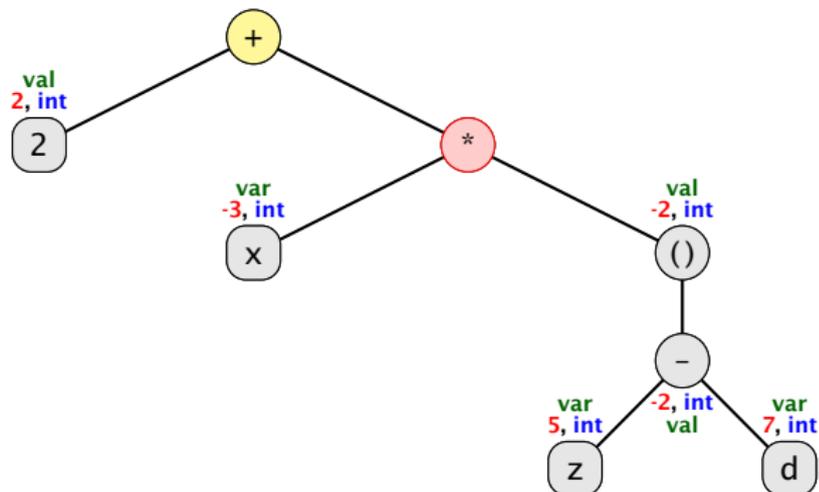


x

d

z

Beispiel: $2 + x * (z - d)$

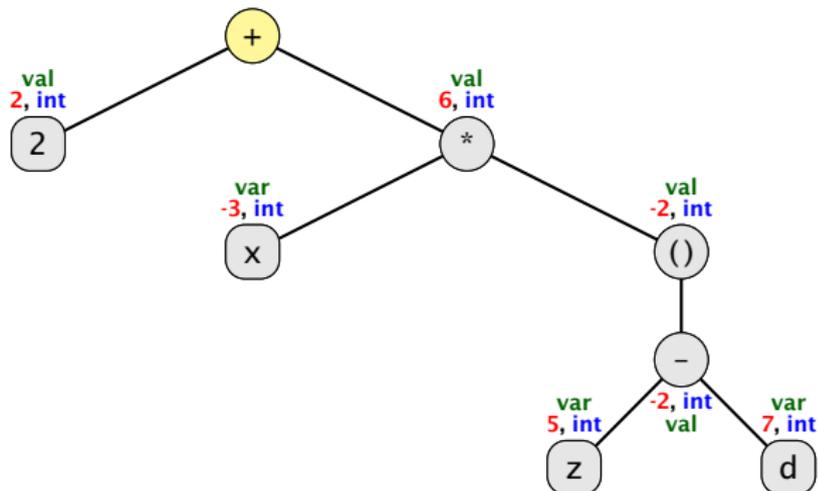


x

d

z

Beispiel: $2 + x * (z - d)$

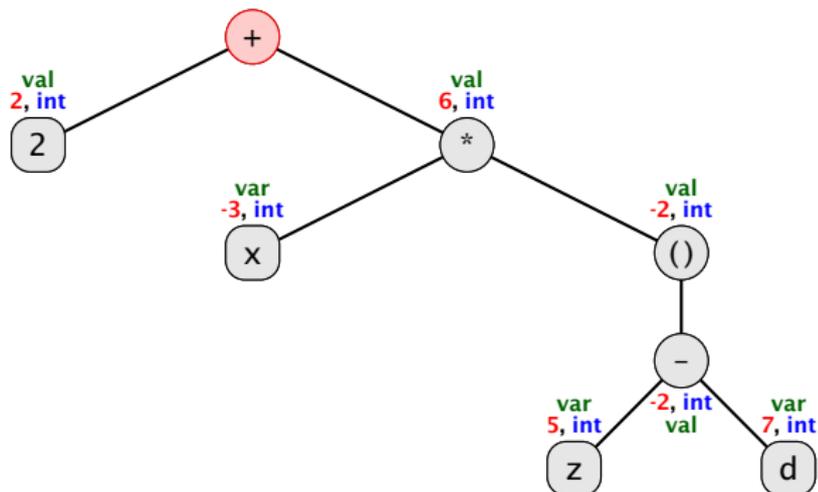


x

d

z

Beispiel: $2 + x * (z - d)$

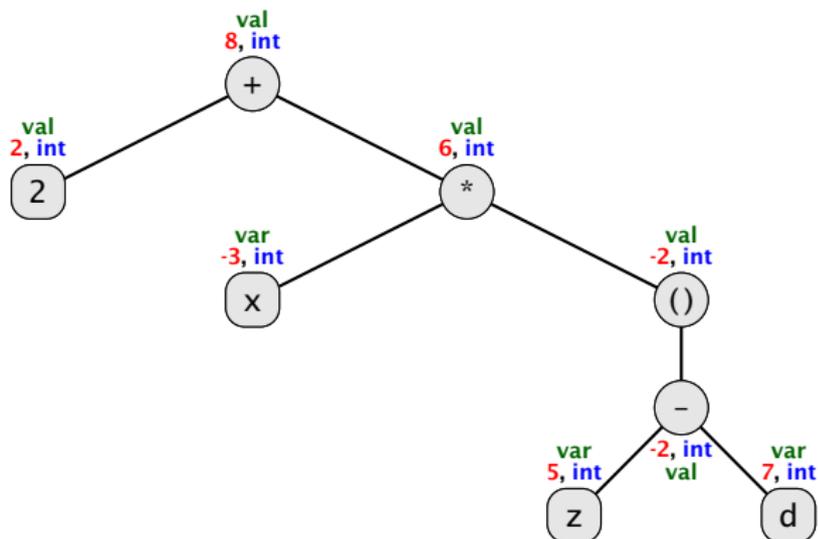


x

d

z

Beispiel: $2 + x * (z - d)$



x

d

z

Beispiel: $a = b = c = d = 0$

$a = b = c = d = 0$

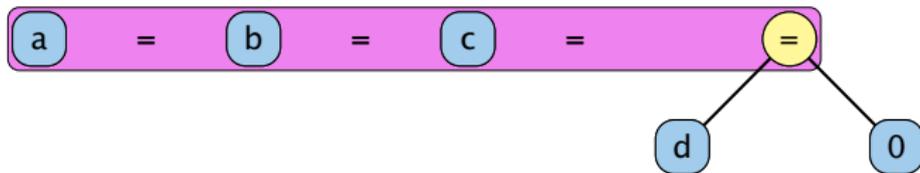
Beispiel: $a = b = c = d = 0$



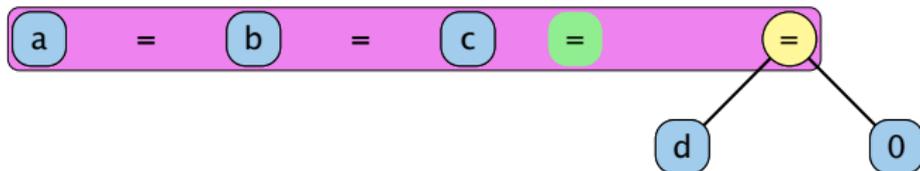
Beispiel: $a = b = c = d = 0$



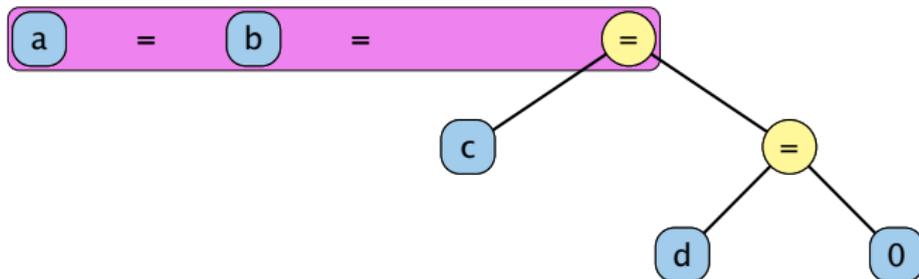
Beispiel: $a = b = c = d = 0$



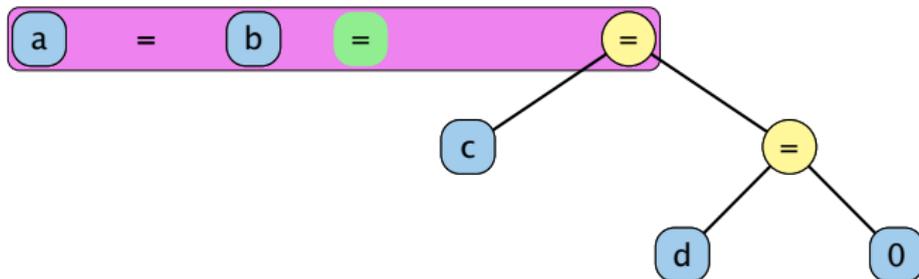
Beispiel: $a = b = c = d = 0$



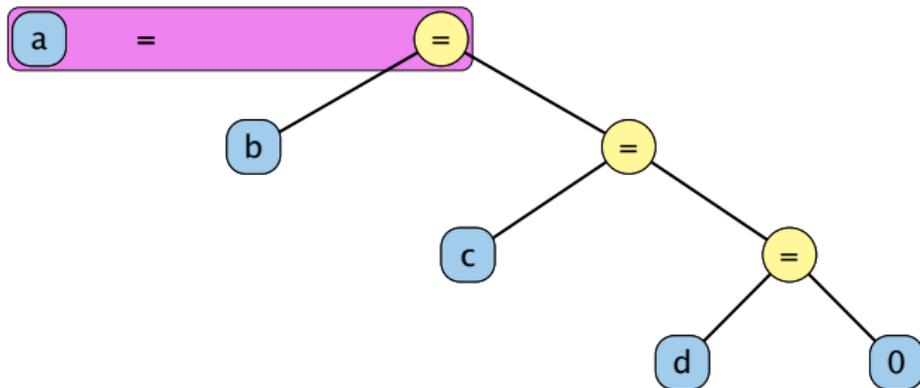
Beispiel: $a = b = c = d = 0$



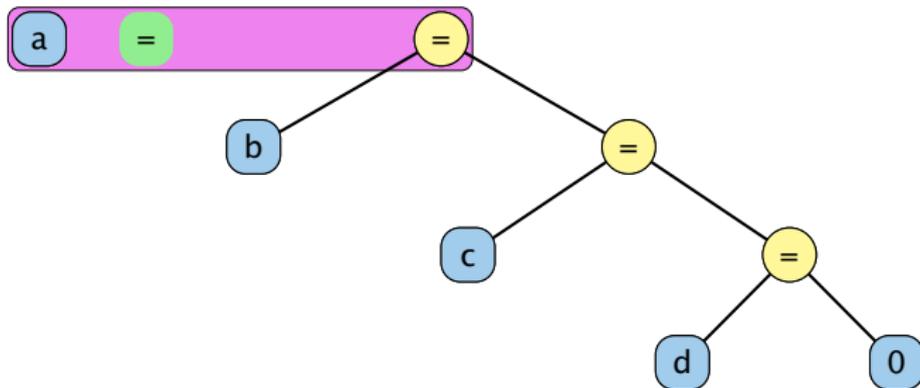
Beispiel: $a = b = c = d = 0$



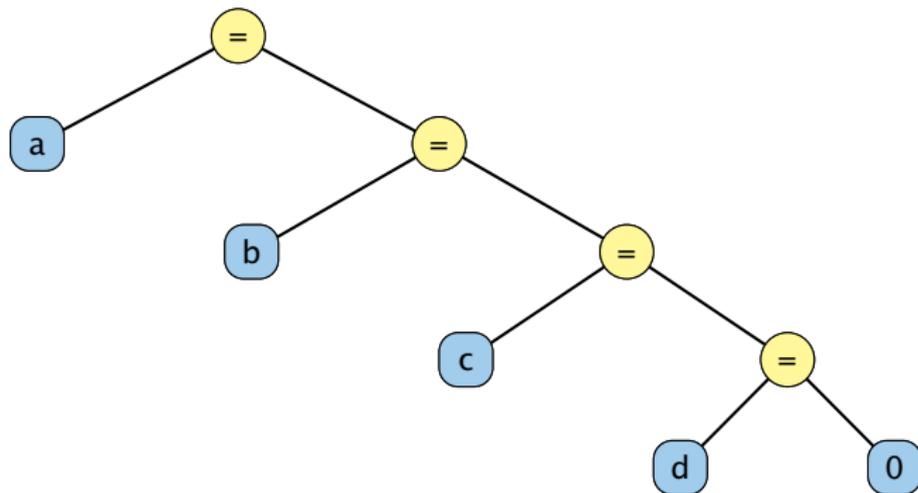
Beispiel: $a = b = c = d = 0$



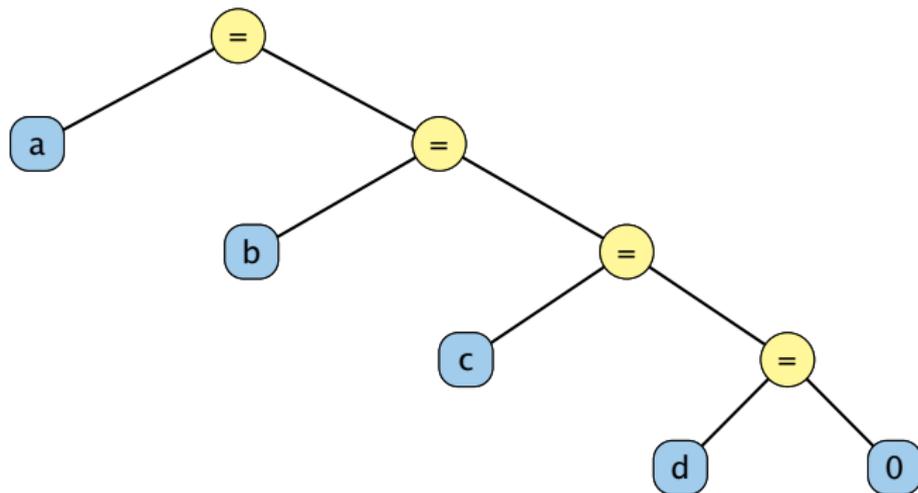
Beispiel: $a = b = c = d = 0$



Beispiel: $a = b = c = d = 0$



Beispiel: $a = b = c = d = 0$



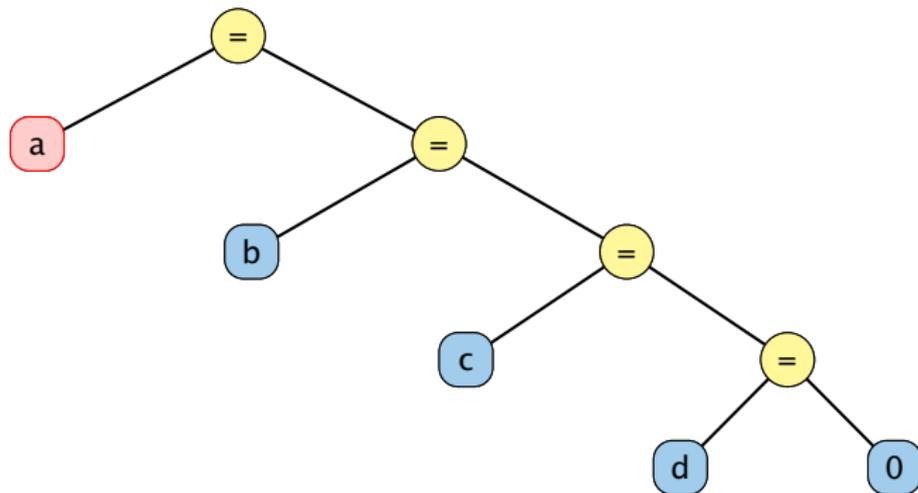
a

b

c

d

Beispiel: $a = b = c = d = 0$



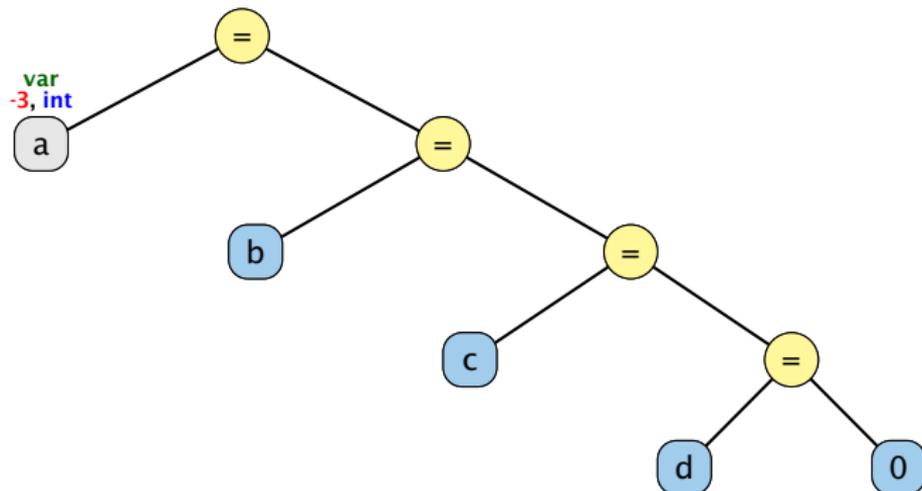
a

b

c

d

Beispiel: $a = b = c = d = 0$



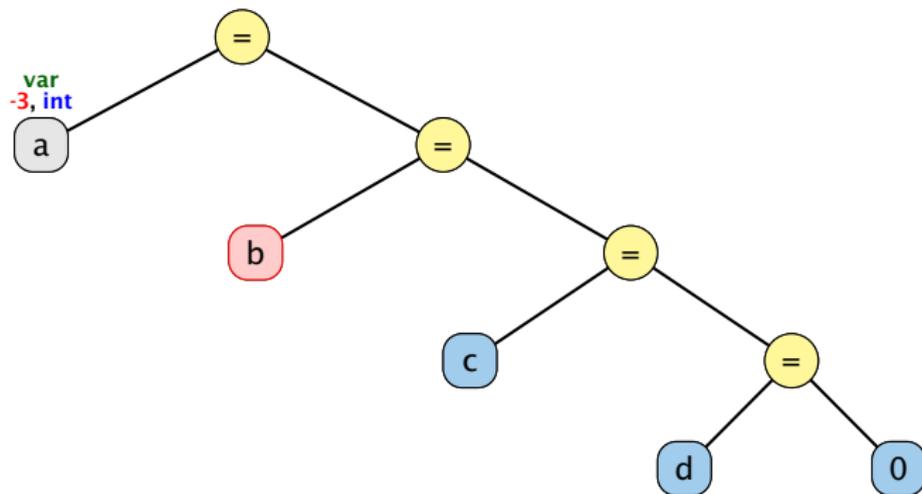
a

b

c

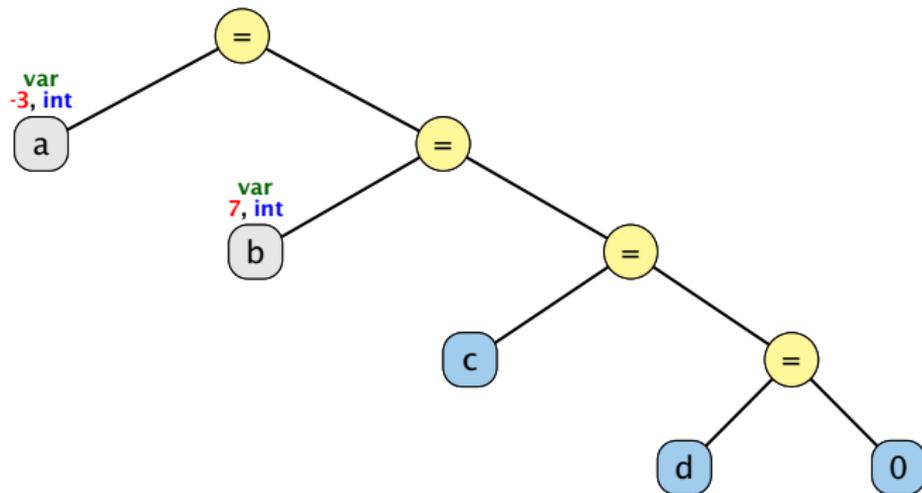
d

Beispiel: $a = b = c = d = 0$



a b c d

Beispiel: $a = b = c = d = 0$



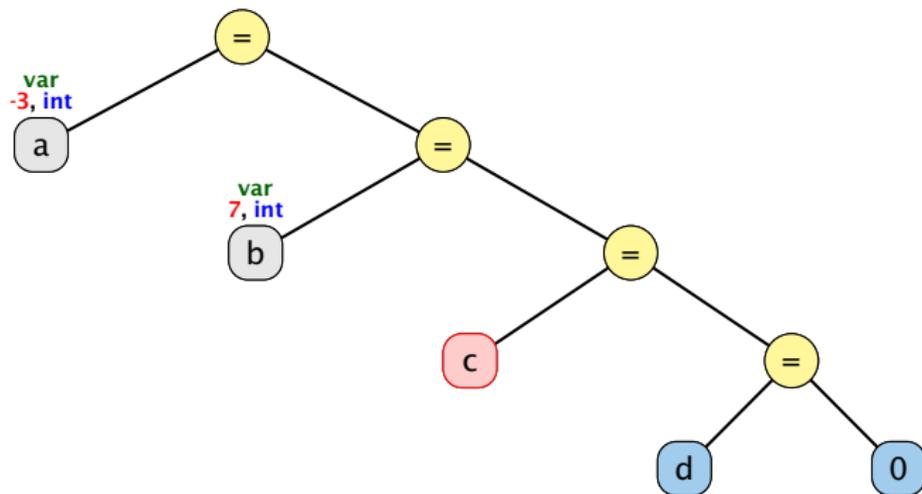
a [-3]

b [7]

c [5]

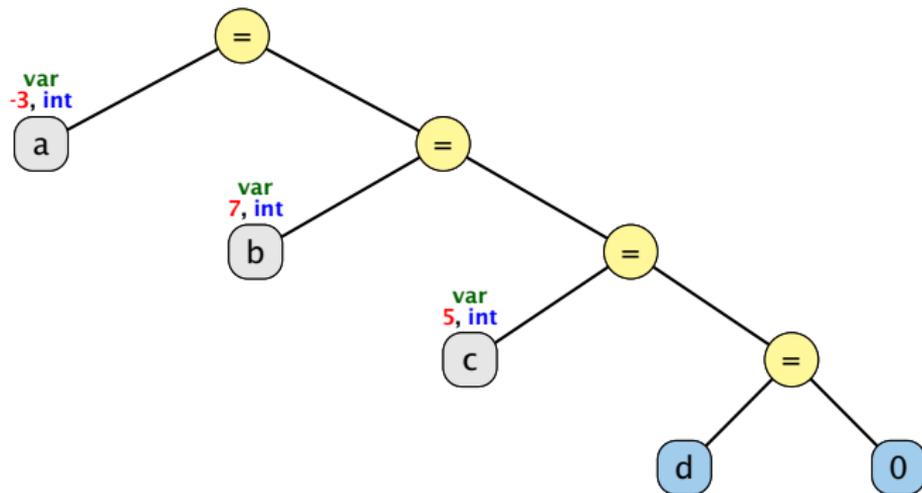
d [2]

Beispiel: $a = b = c = d = 0$



a b c d

Beispiel: $a = b = c = d = 0$



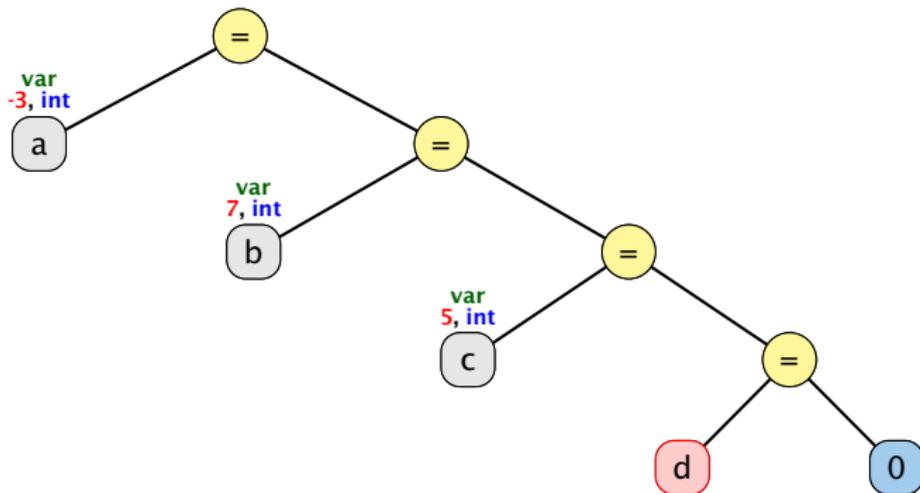
a

b

c

d

Beispiel: $a = b = c = d = 0$



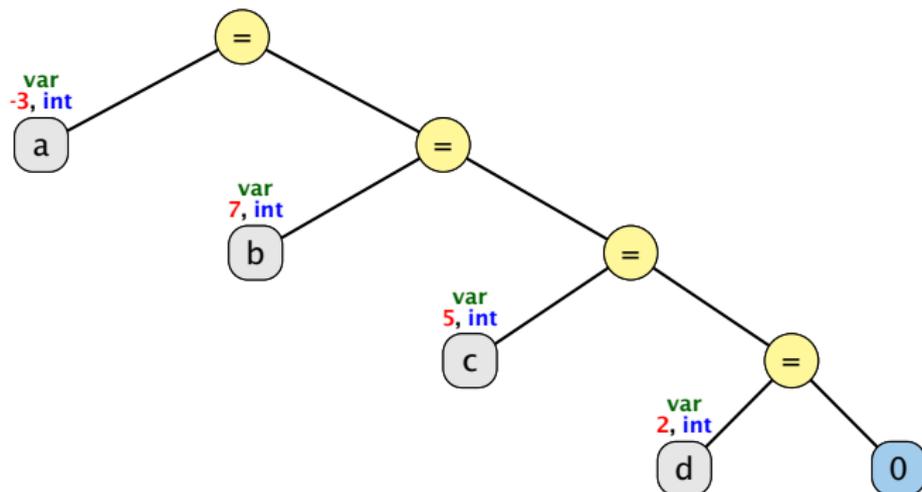
a [-3]

b [7]

c [5]

d [2]

Beispiel: $a = b = c = d = 0$



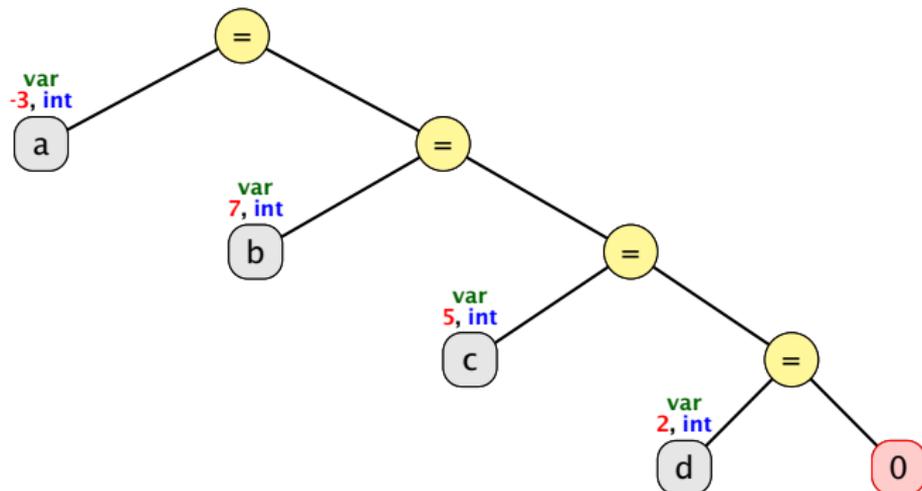
`a` `-3`

`b` `7`

`c` `5`

`d` `2`

Beispiel: $a = b = c = d = 0$



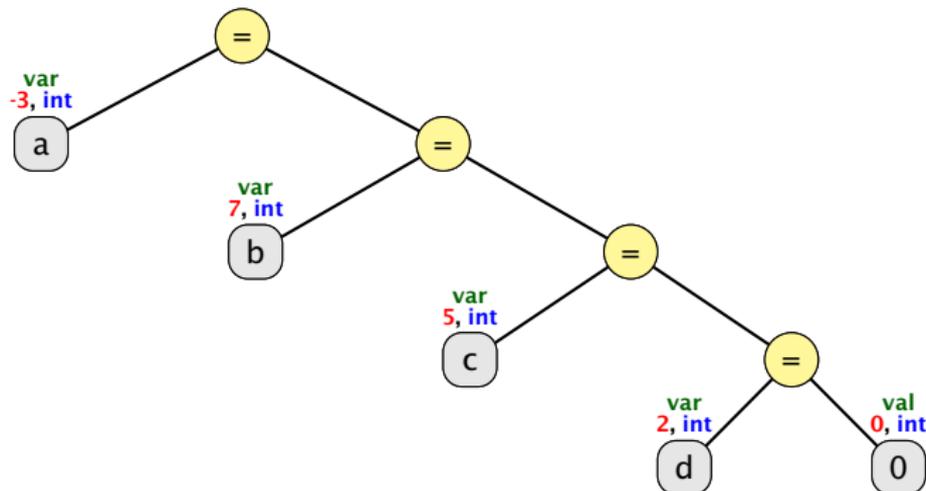
a [-3]

b [7]

c [5]

d [2]

Beispiel: $a = b = c = d = 0$



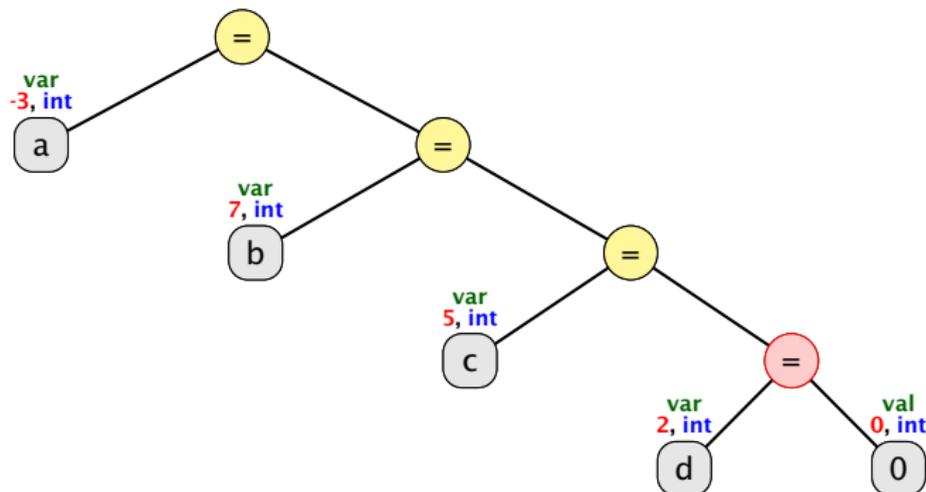
a [-3]

b [7]

c [5]

d [2]

Beispiel: $a = b = c = d = 0$



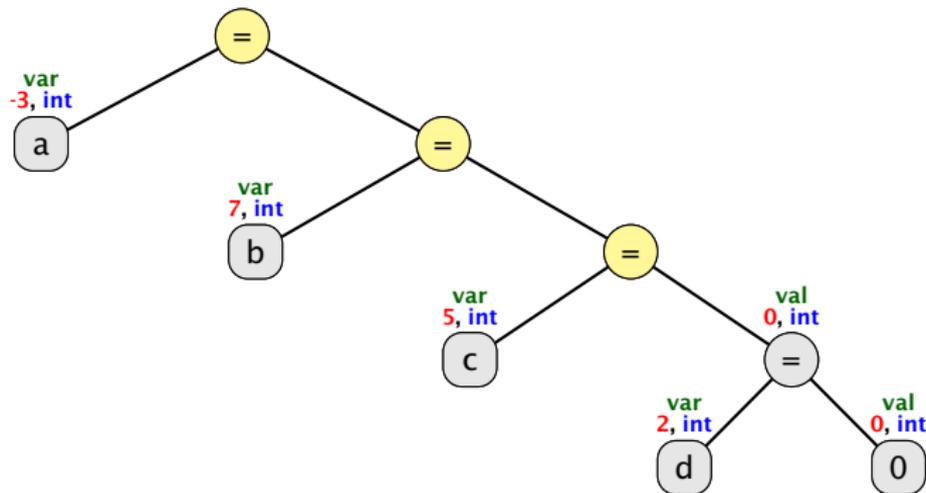
a [-3]

b [7]

c [5]

d [2]

Beispiel: $a = b = c = d = 0$



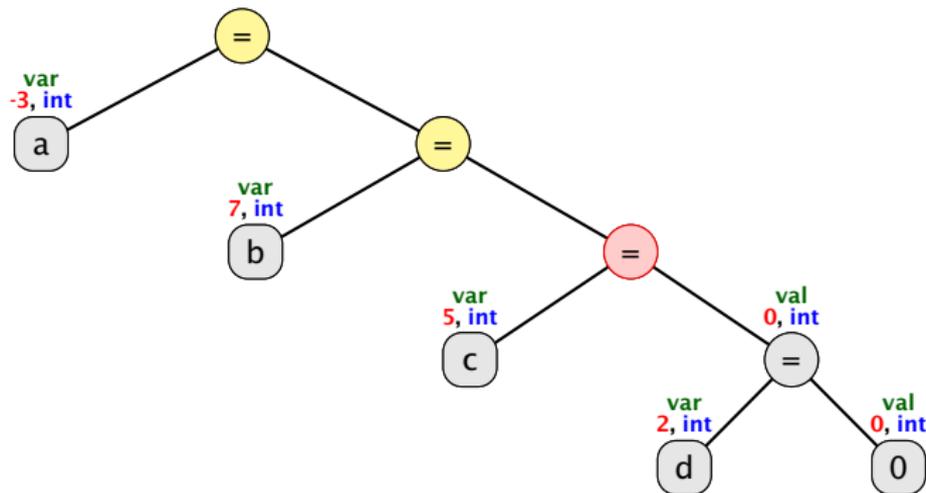
a

b

c

d

Beispiel: $a = b = c = d = 0$



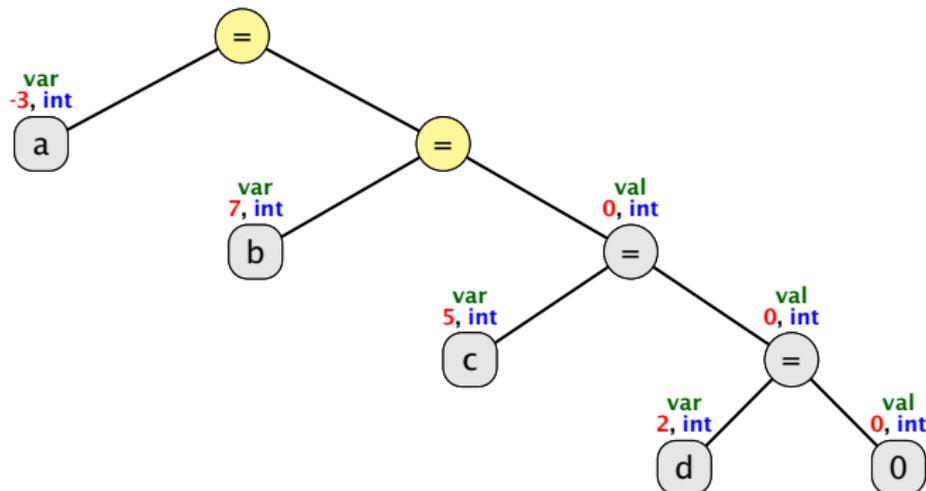
a [-3]

b [7]

c [5]

d [0]

Beispiel: $a = b = c = d = 0$



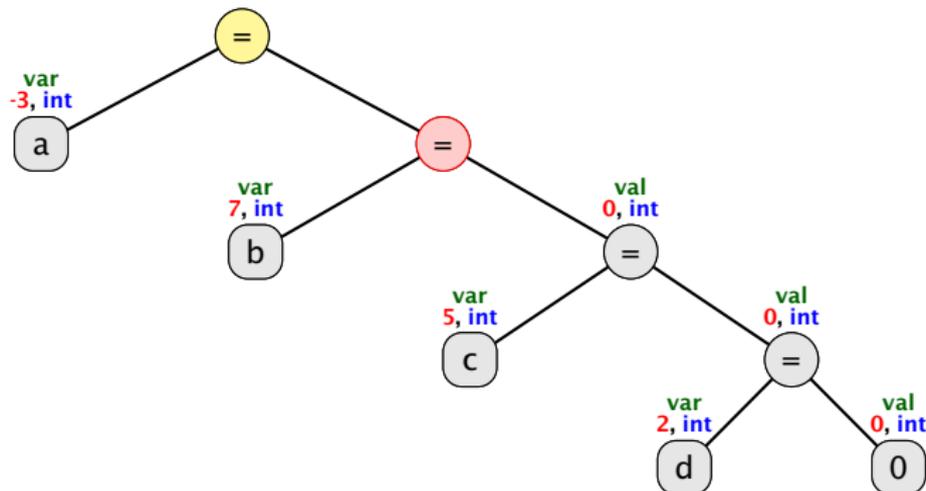
a

b

c

d

Beispiel: $a = b = c = d = 0$



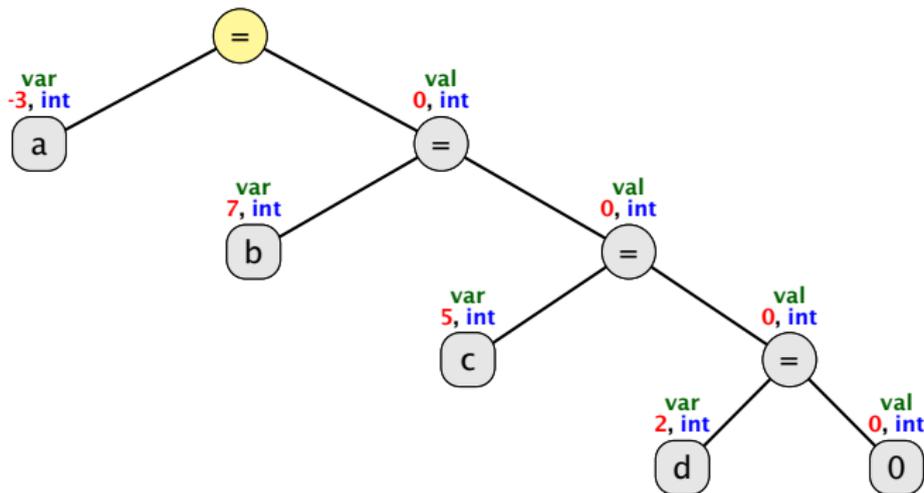
a

b

c

d

Beispiel: $a = b = c = d = 0$



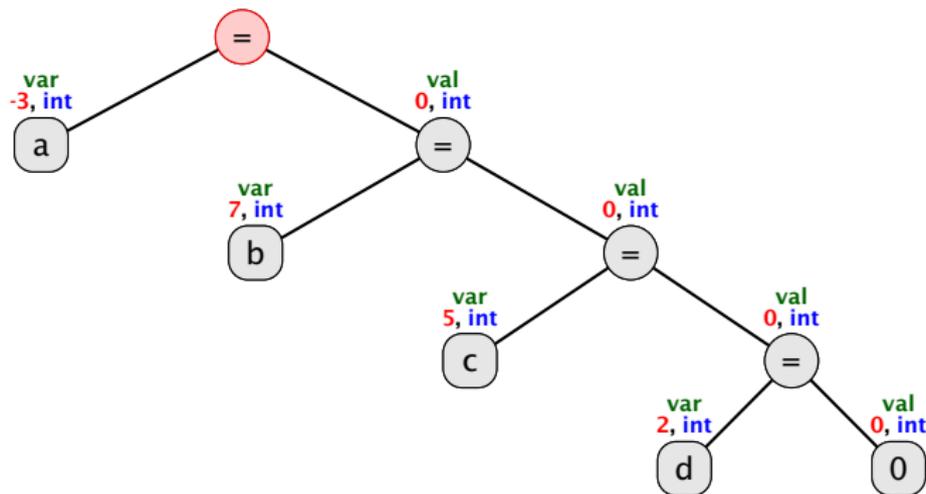
a [-3]

b [0]

c [0]

d [0]

Beispiel: $a = b = c = d = 0$



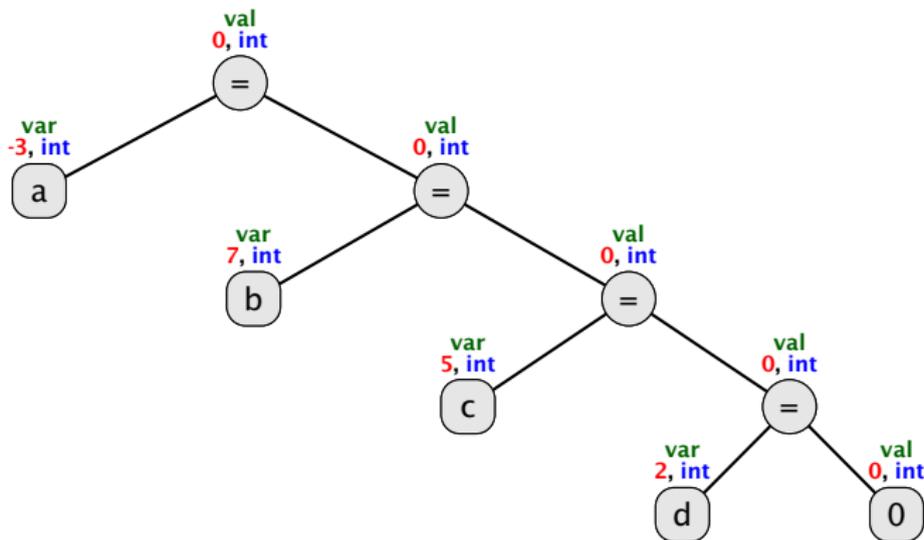
a [-3]

b [0]

c [0]

d [0]

Beispiel: $a = b = c = d = 0$



a 0

b 0

c 0

d 0

Beispiel: $a \neq 0 \ \&\& \ b/a < 10$

a != 0 && b / a < 10

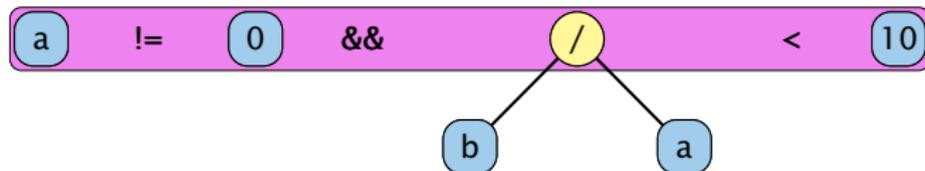
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



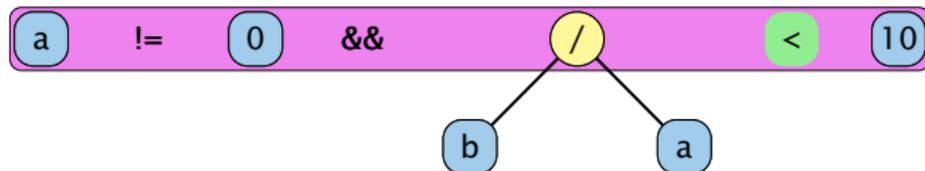
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



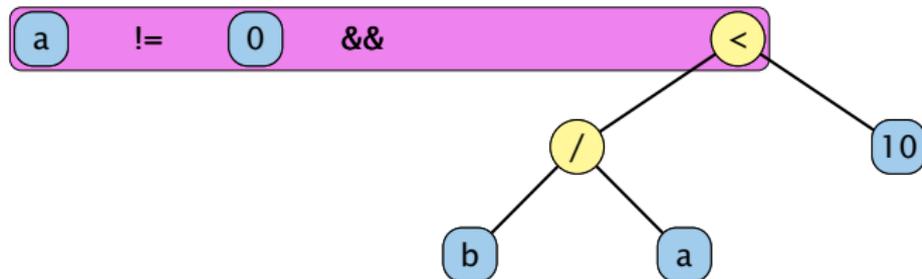
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



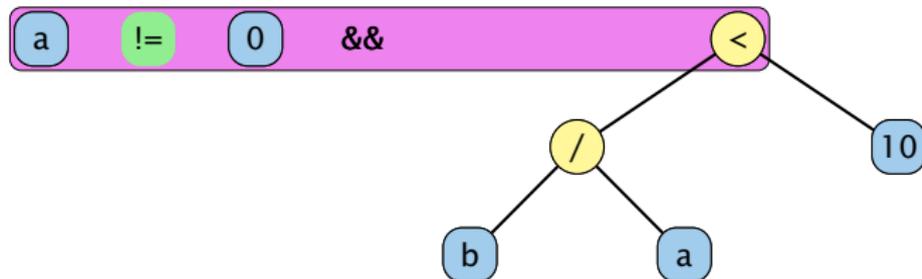
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



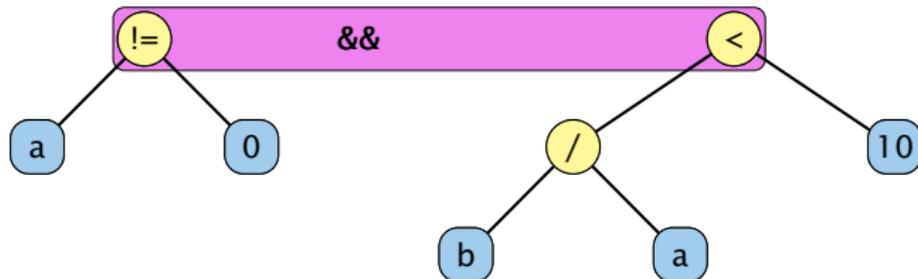
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



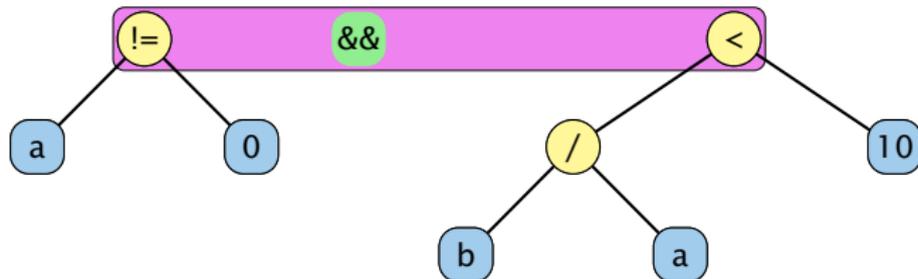
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



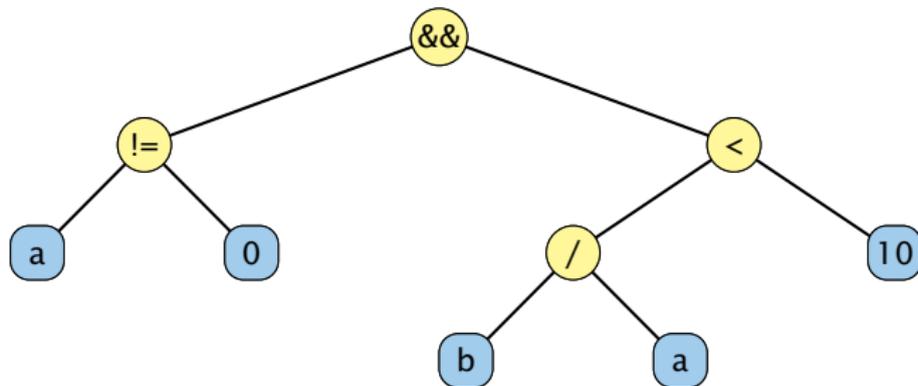
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



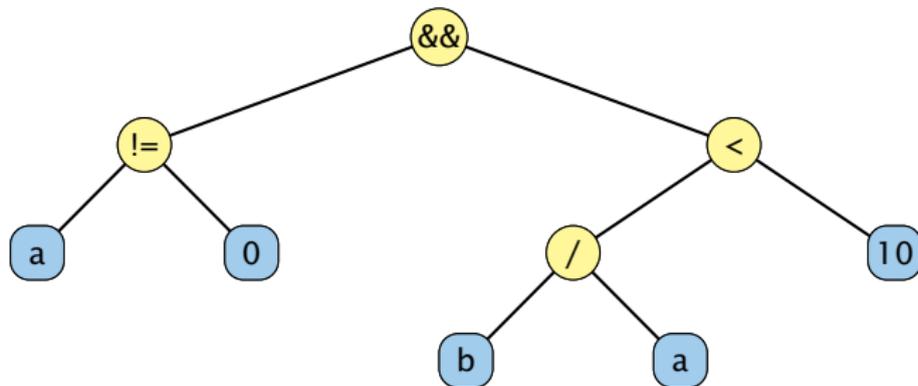
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



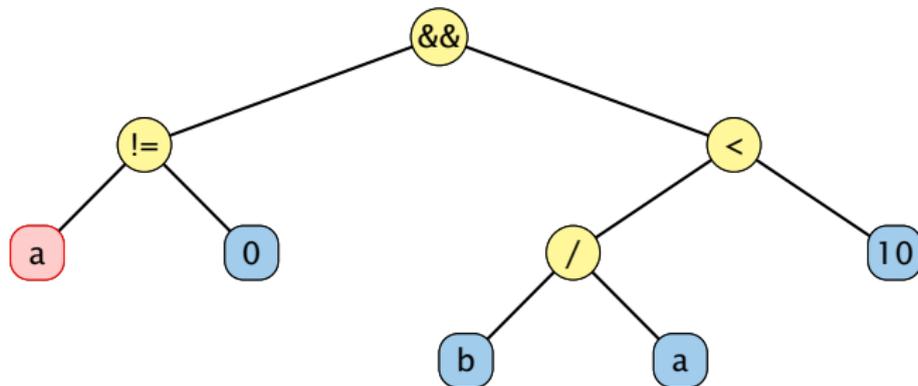
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a

b

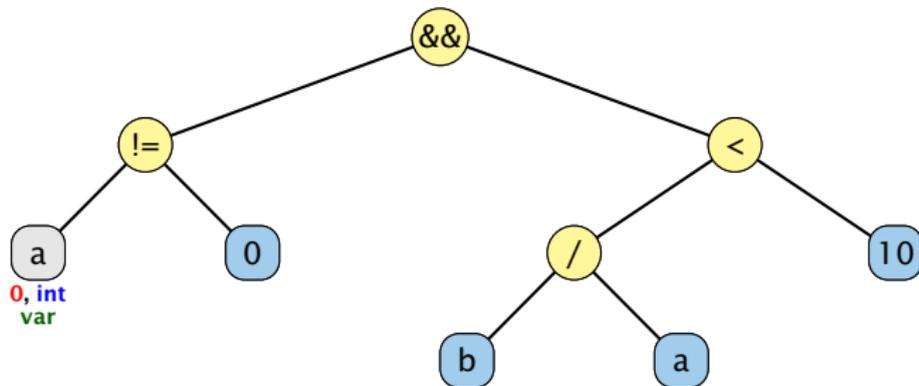
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

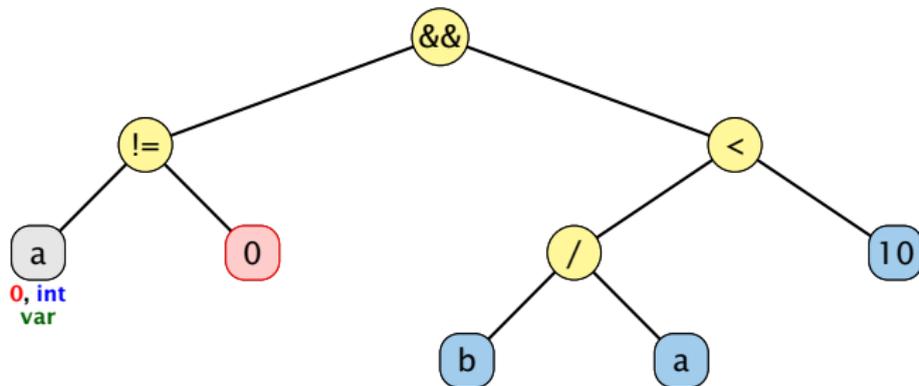
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

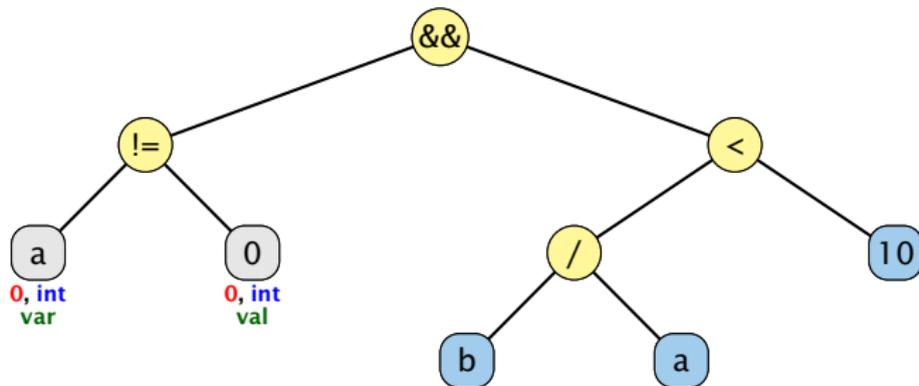
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

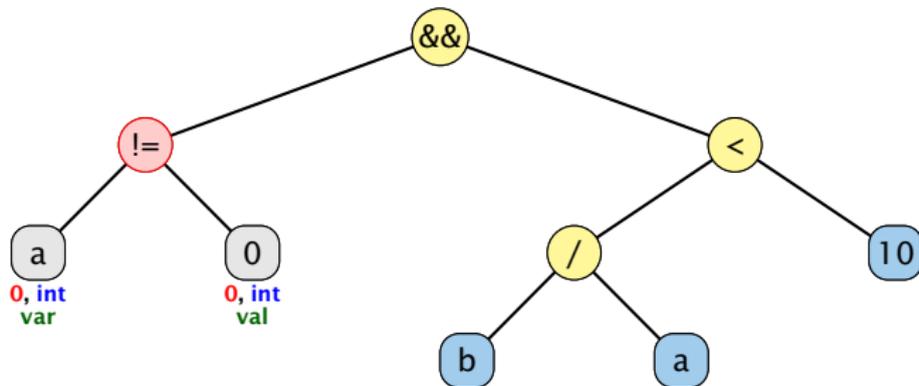
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

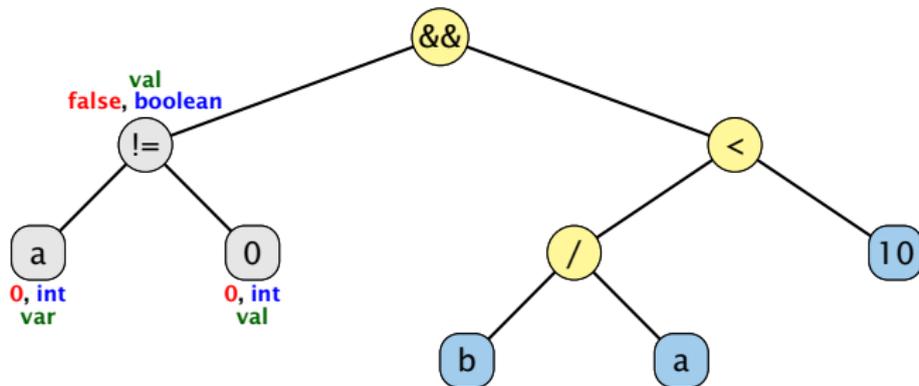
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

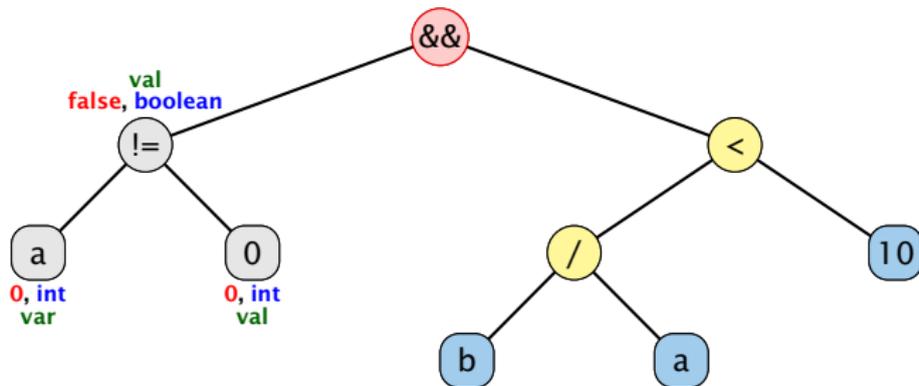
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

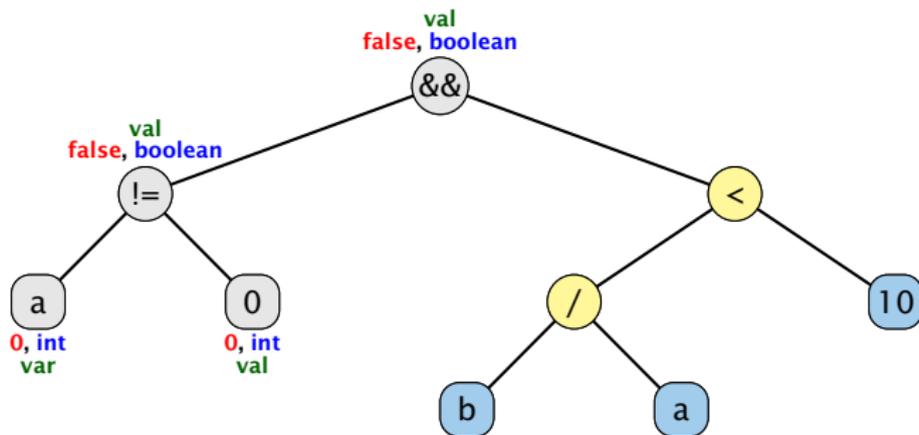
Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

Beispiel: $a \neq 0 \ \&\& \ b/a < 10$



a 0

b 4

Beispiel: $y = x + ++x$

y = x + ++x

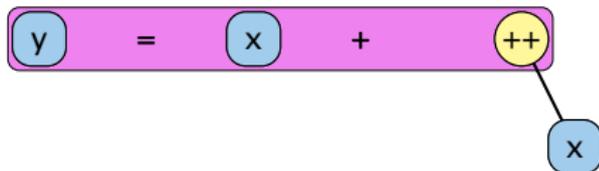
Beispiel: $y = x + ++x$



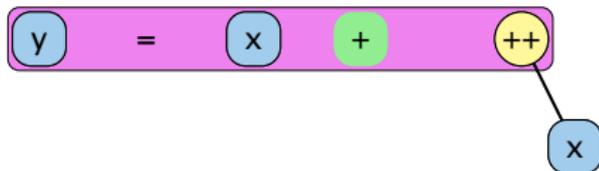
Beispiel: $y = x + ++x$



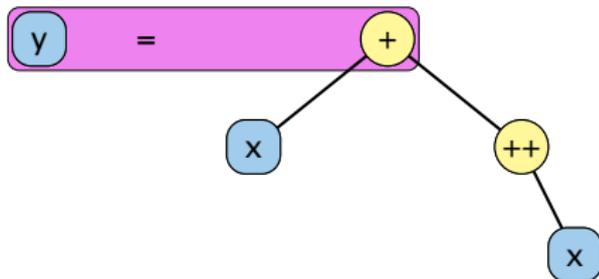
Beispiel: $y = x + ++x$



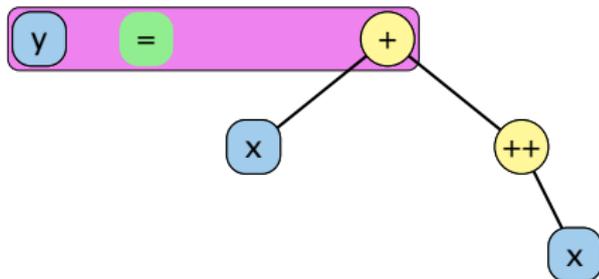
Beispiel: $y = x + ++x$



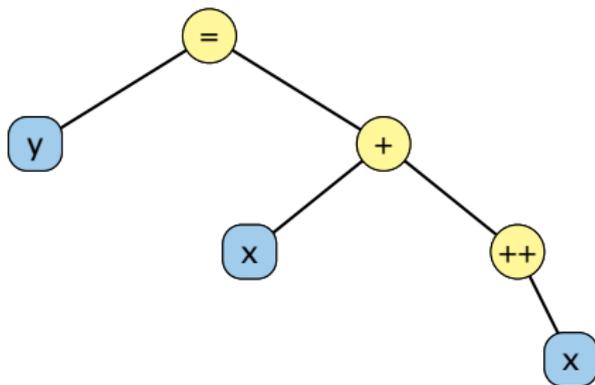
Beispiel: $y = x + ++x$



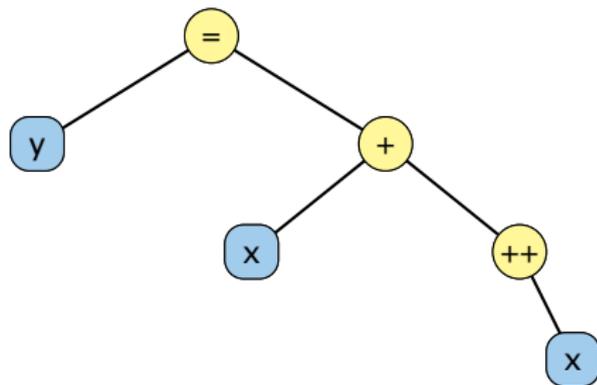
Beispiel: $y = x + ++x$



Beispiel: $y = x + ++x$



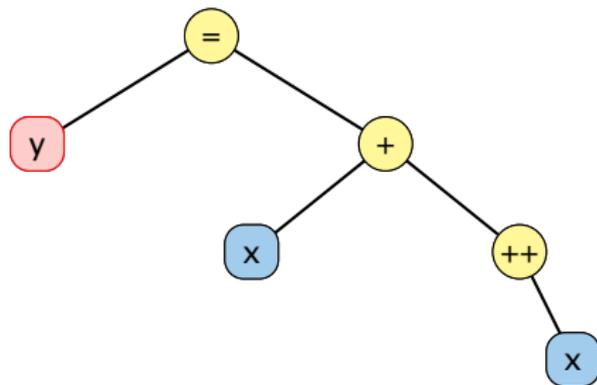
Beispiel: $y = x + ++x$



x 4

y 0

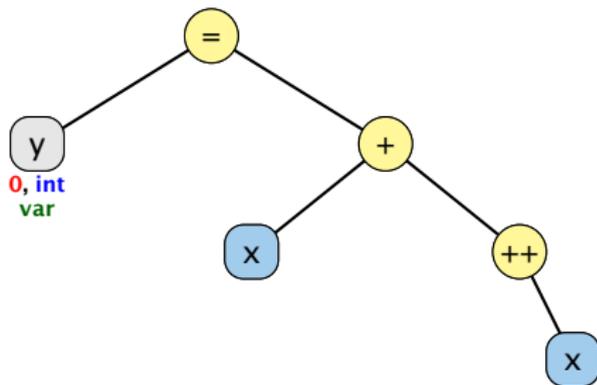
Beispiel: $y = x + ++x$



x 4

y 0

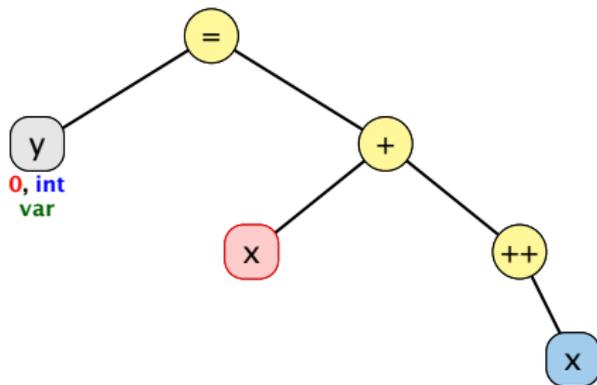
Beispiel: $y = x + ++x$



x 4

y 0

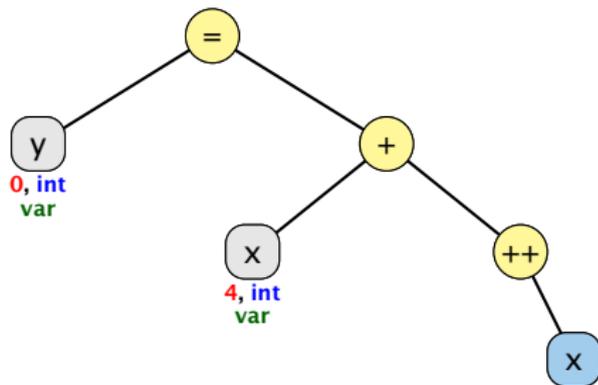
Beispiel: $y = x + ++x$



x 4

y 0

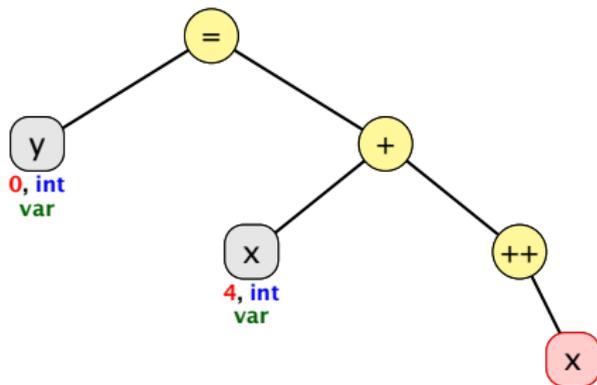
Beispiel: $y = x + ++x$



x 4

y 0

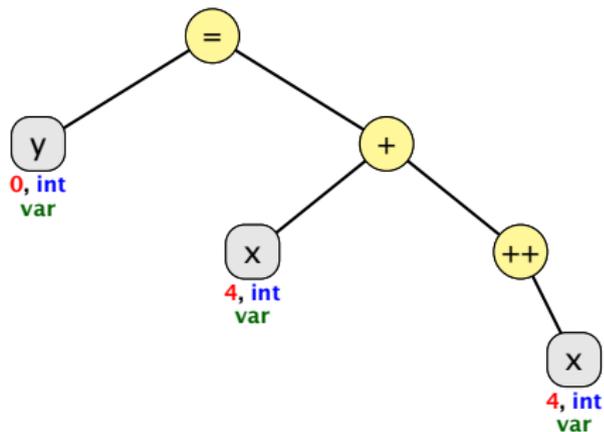
Beispiel: $y = x + ++x$



x 4

y 0

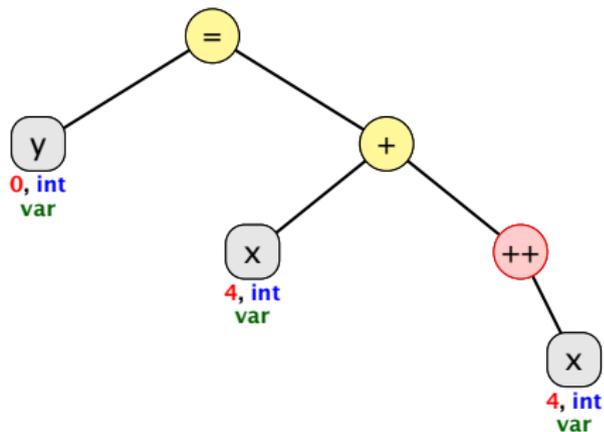
Beispiel: $y = x + ++x$



x 4

y 0

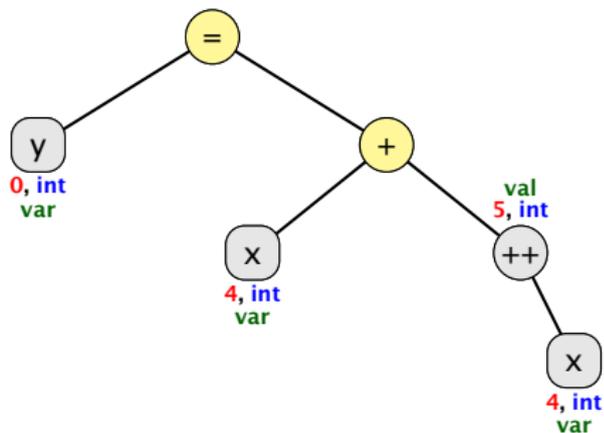
Beispiel: $y = x + ++x$



x 4

y 0

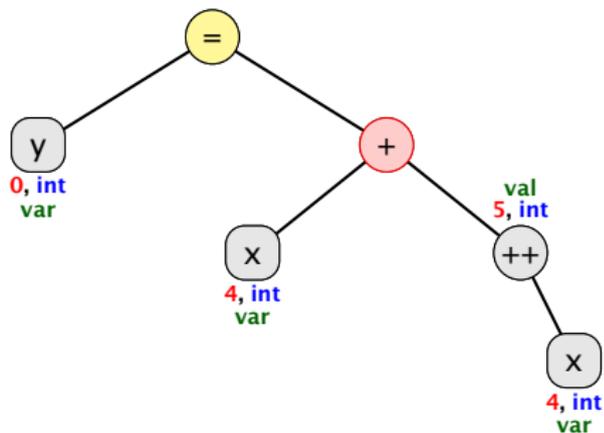
Beispiel: $y = x + ++x$



x 5

y 0

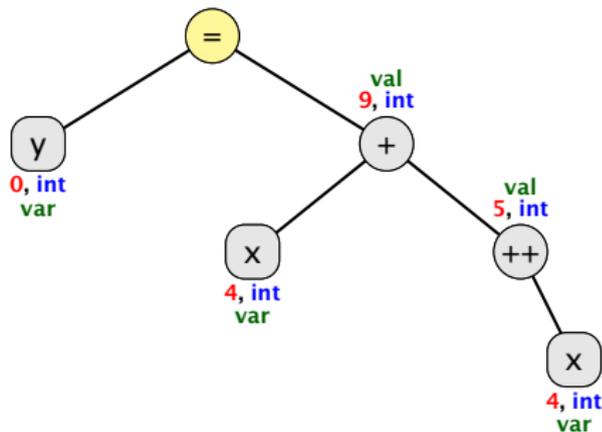
Beispiel: $y = x + ++x$



x 5

y 0

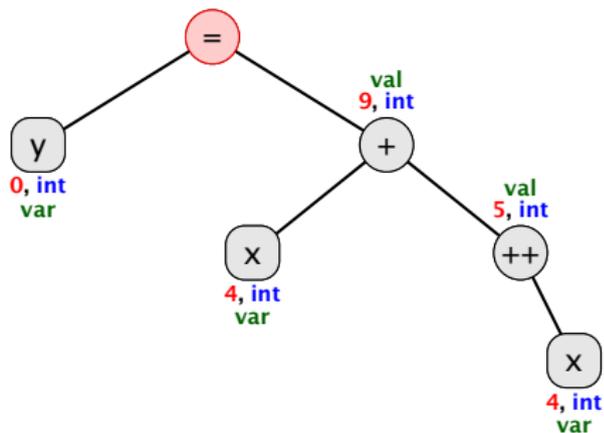
Beispiel: $y = x + ++x$



x 5

y 0

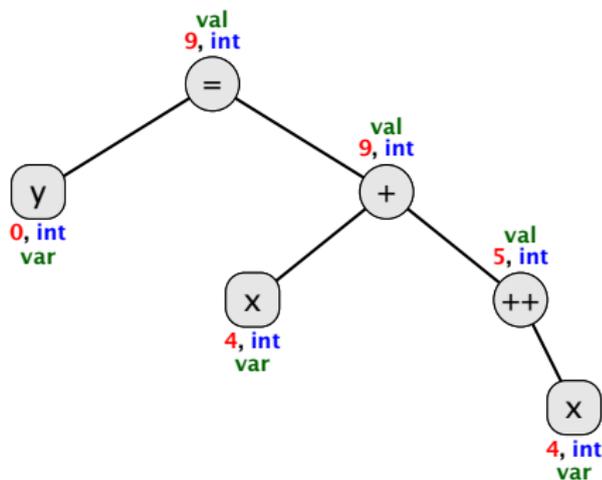
Beispiel: $y = x + ++x$



x 5

y 0

Beispiel: $y = x + ++x$



x 5

y 9

Beispiel: $y = x++ + x$

y = x ++ + x

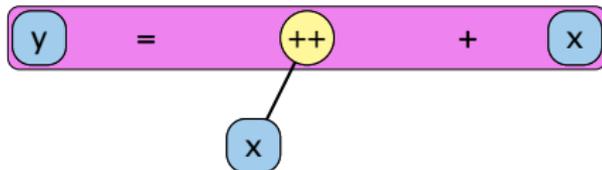
Beispiel: $y = x++ + x$



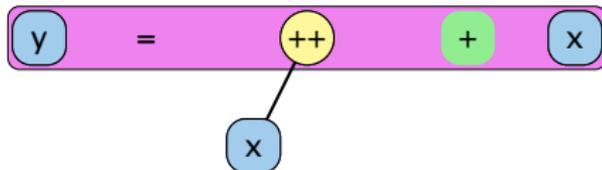
Beispiel: $y = x++ + x$



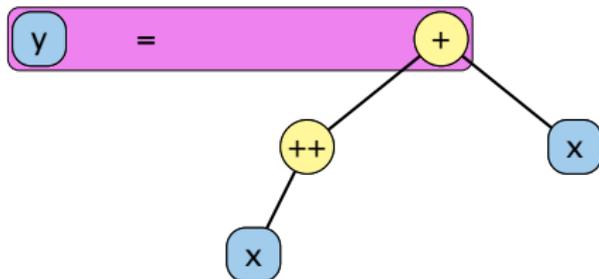
Beispiel: $y = x++ + x$



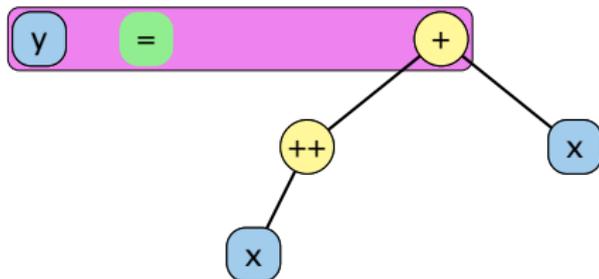
Beispiel: $y = x++ + x$



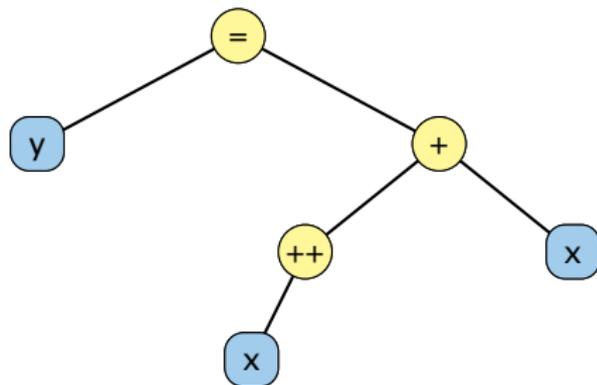
Beispiel: $y = x++ + x$



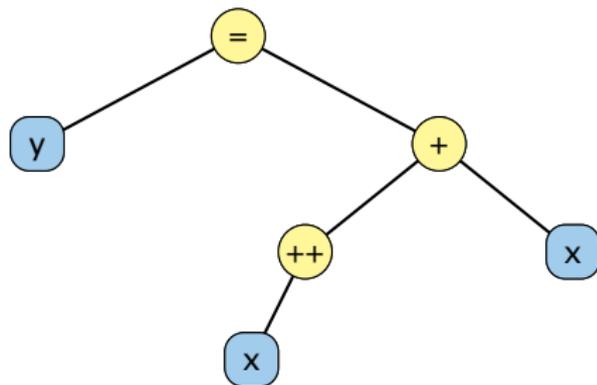
Beispiel: $y = x++ + x$



Beispiel: $y = x++ + x$



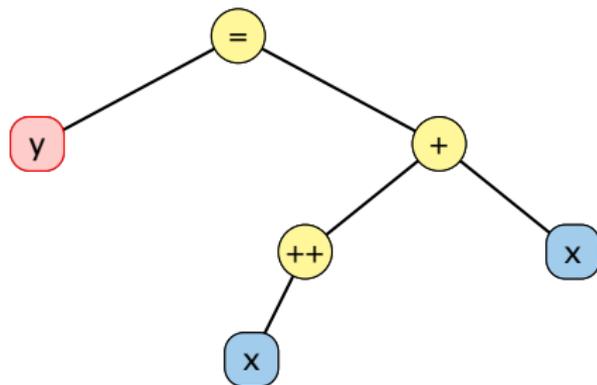
Beispiel: $y = x++ + x$



x 4

y 0

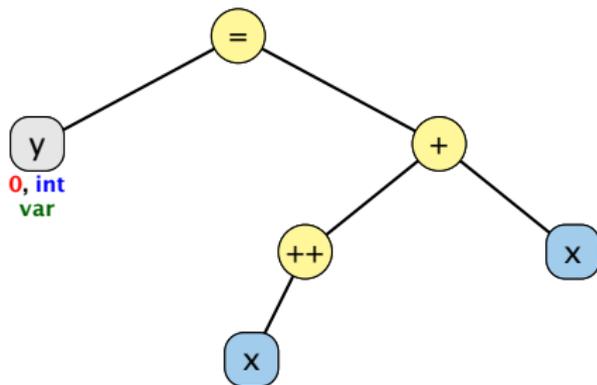
Beispiel: $y = x++ + x$



x

y

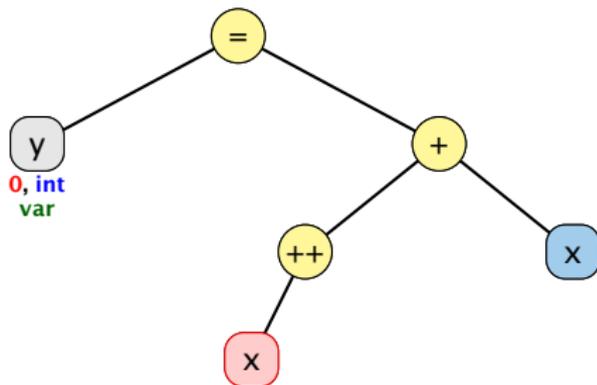
Beispiel: $y = x++ + x$



x

y

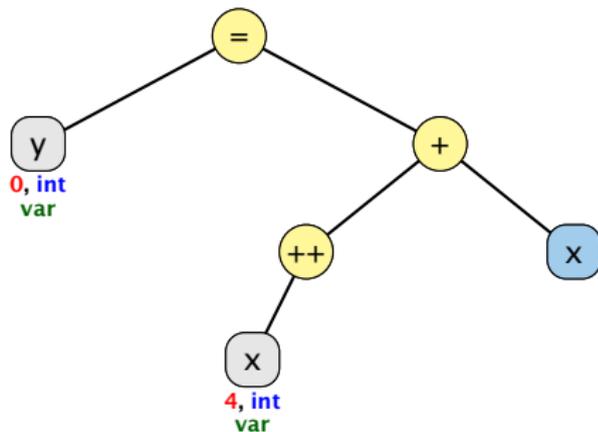
Beispiel: $y = x++ + x$



x 4

y 0

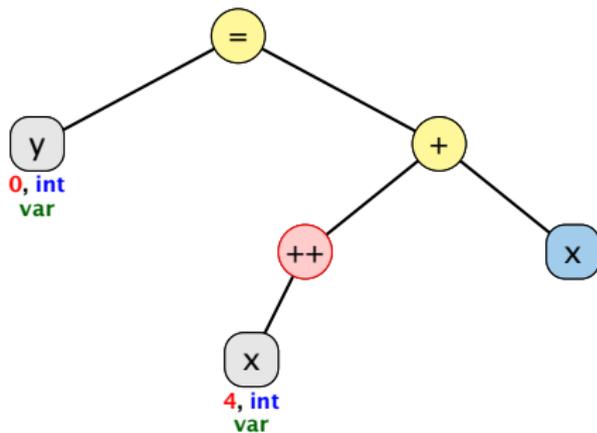
Beispiel: $y = x++ + x$



x 4

y 0

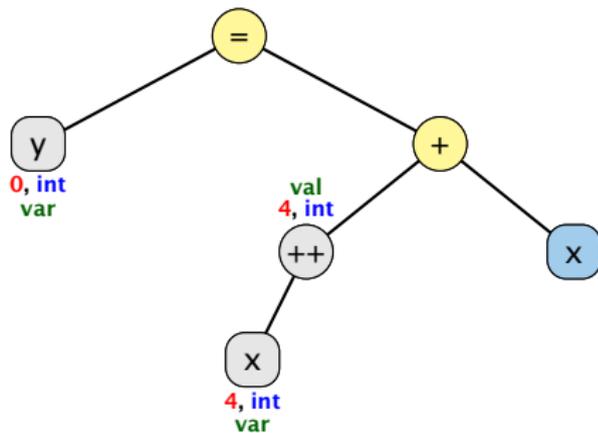
Beispiel: $y = x++ + x$



x 4

y 0

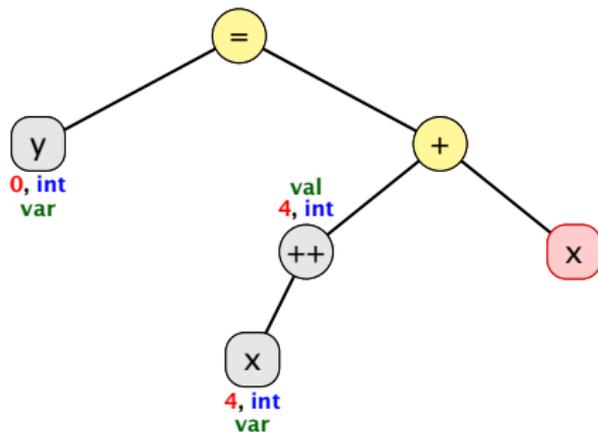
Beispiel: $y = x++ + x$



x 5

y 0

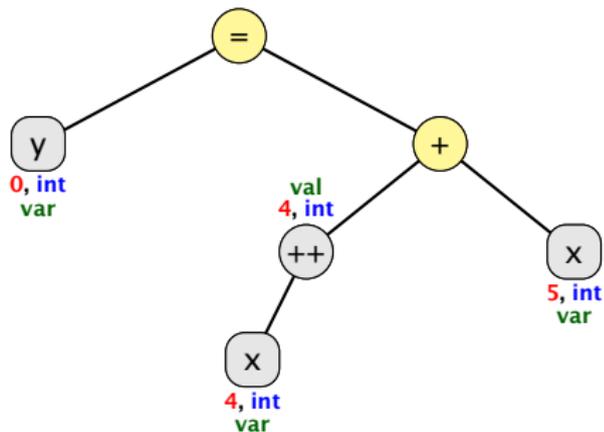
Beispiel: $y = x++ + x$



x 5

y 0

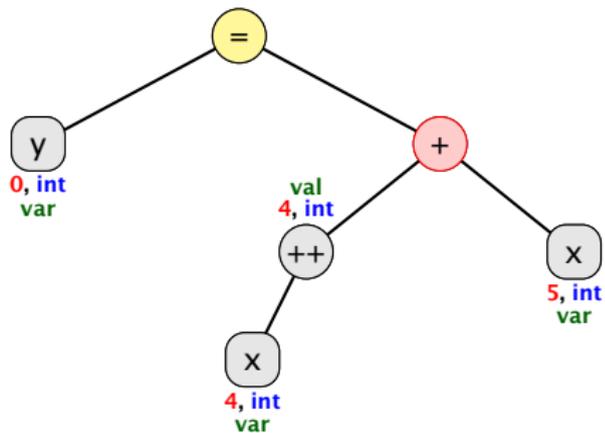
Beispiel: $y = x++ + x$



x 5

y 0

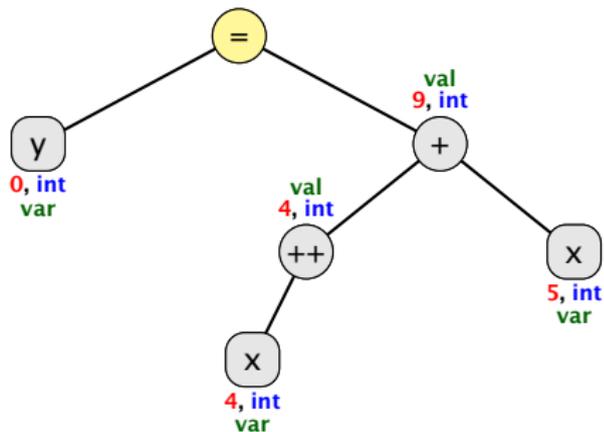
Beispiel: $y = x++ + x$



x 5

y 0

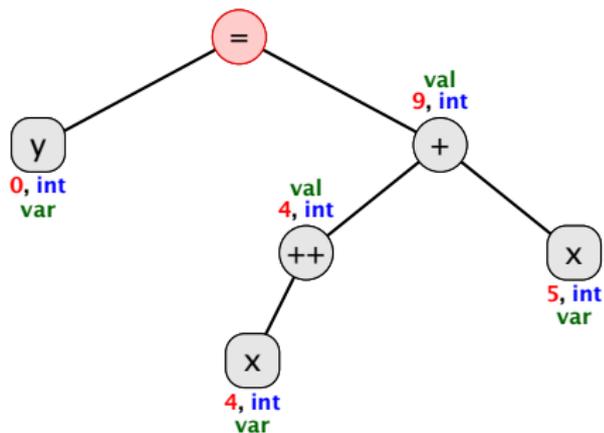
Beispiel: $y = x++ + x$



x 5

y 0

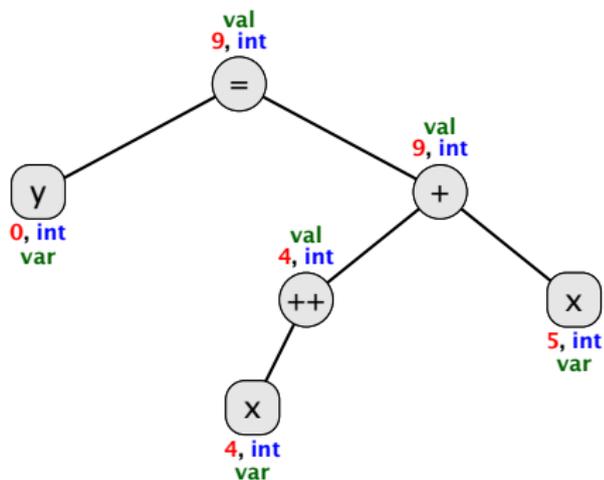
Beispiel: $y = x++ + x$



x

y

Beispiel: $y = x++ + x$



x 5

y 9

Impliziter Typecast

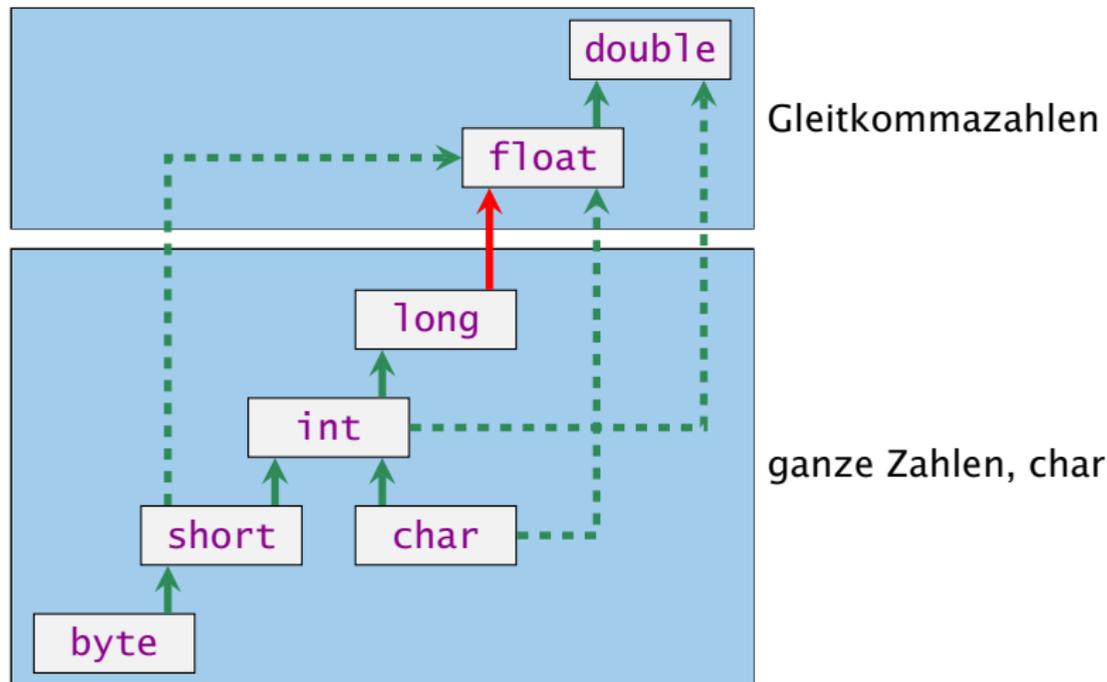
Wenn ein Ausdruck vom **TypA** an einer Stelle verwendet wird, wo ein Ausdruck vom **TypB** erforderlich ist, wird

- ▶ entweder der Ausdruck vom **TypA** in einen Ausdruck vom **TypB** **gecastet** (**impliziter Typecast**),
- ▶ oder ein Compilerfehler erzeugt, falls dieser Cast nicht (automatisch) erlaubt ist.

Beispiel: Zuweisung

```
long x = 5;  
int y = 3;  
x = y; // impliziter Cast von int nach long
```

Erlaubte Implizite Typecasts - Numerische Typen



Konvertierung von `long` nach `double` oder von `int` nach `float` kann Information verlieren wird aber **automatisch** durchgeführt.

Welcher Typ wird benötigt?

Operatoren sind üblicherweise **überladen**, d.h. ein Symbol (+, -, ...) steht in Abhängigkeit der Parameter (Argumente) für unterschiedliche Funktionen.

+ : int \rightarrow int

+ : long \rightarrow long

+ : float \rightarrow float

+ : double \rightarrow double

+ : int \times int \rightarrow int

+ : long \times long \rightarrow long

+ : float \times float \rightarrow float

+ : double \times double \rightarrow double

+ : String \times String \rightarrow String

Der Compiler muss in der Lage sein **während der Compilierung** die richtige Funktion zu bestimmen.

Impliziter Typecast

Der Compiler wertet nur die Typen des Ausdrucksbaums aus.

- ▶ Für jeden inneren Knoten wählt er dann die geeignete Funktion (z.B. $+ : \text{long} \times \text{long} \rightarrow \text{long}$ falls ein $+$ -Knoten zwei long -Argumente erhält).
- ▶ Falls keine passende Funktion gefunden wird, versucht der Compiler durch **implizite Typecasts** die Operanden an eine Funktion anzupassen.
- ▶ Dies geschieht auch für selbstgeschriebene Funktionen (z.B. $\text{min}(\text{int } a, \text{int } b)$ und $\text{min}(\text{long } a, \text{long } b)$).
- ▶ Der Compiler nimmt die Funktion mit der speziellsten **Signatur**.

Speziellste Signatur



Ordnungsrelationen

Relation \preceq : $\text{TypA} \preceq \text{TypB}$ falls TypA nach TypB (implizit) gecasted werden kann:

- ▶ **reflexiv:** $T \preceq T$
- ▶ **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- ▶ **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., \preceq definiert **Halbordnung auf der Menge der Typen**.

Relation \preceq_k : $(T_1, \dots, T_k) \preceq_k (T'_1, \dots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \dots, k\}$:

- ▶ **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- ▶ **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- ▶ **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., \preceq_k definiert **Halbordnung auf Menge der k -Tupel von Typen**

Ordnungsrelationen

Relation \preceq : $\text{TypA} \preceq \text{TypB}$ falls TypA nach TypB (implizit) gecasted werden kann:

- ▶ **reflexiv:** $T \preceq T$
- ▶ **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- ▶ **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., \preceq definiert **Halbordnung auf der Menge der Typen**.

Relation \preceq_k : $(T_1, \dots, T_k) \preceq_k (T'_1, \dots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \dots, k\}$:

- ▶ **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- ▶ **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- ▶ **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

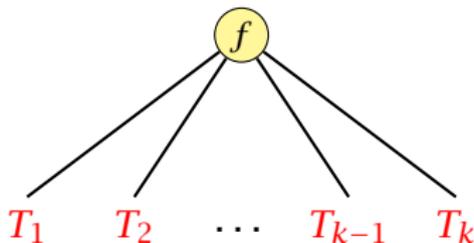
d.h., \preceq_k definiert **Halbordnung auf Menge der k -Tupel von Typen**

$R_1 \quad f(\mathcal{T}_1)$

$R_2 \quad f(\mathcal{T}_2)$

\vdots

$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \dots, \mathcal{T}_\ell$ sind k -Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \dots, T_k)$ ist das k -tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i \mid \mathcal{T} \preceq_k \mathcal{T}_i\} .$$

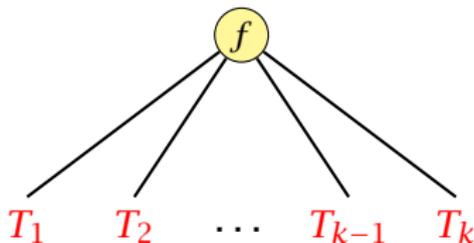
Wähle **kleinstes** Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

$R_1 \quad f(\mathcal{T}_1)$

$R_2 \quad f(\mathcal{T}_2)$

\vdots

$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \dots, \mathcal{T}_\ell$ sind k -Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \dots, T_k)$ ist das k -tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i \mid \mathcal{T} \preceq_k \mathcal{T}_i\} .$$

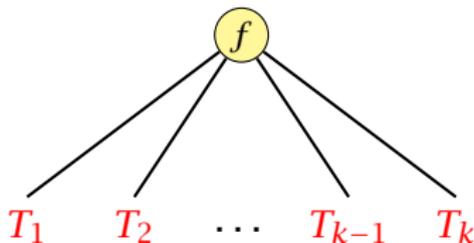
Wähle **kleinstes** Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

$R_1 \quad f(\mathcal{T}_1)$

$R_2 \quad f(\mathcal{T}_2)$

\vdots

$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \dots, \mathcal{T}_\ell$ sind k -Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (\mathcal{T}_1, \dots, \mathcal{T}_k)$ ist das k -tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i \mid \mathcal{T} \preceq_k \mathcal{T}_i\} .$$

Wähle **kleinstes** Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Impliziter Typecast – Numerische Typen

Angenommen wir haben Funktionen

```
int min(int a, int b)
```

```
float min(float a, float b)
```

```
double min(double a, double b)
```

definiert.

```
1 long a = 7, b = 3;  
2 double d = min(a, b);
```

würde die Funktion `float min(float a, float b)` aufrufen.

Impliziter Typecast

Bei Ausdrücken mit Seiteneffekten (Zuweisungen, ++ , --) gelten andere Regeln:

Beispiel: Zuweisungen

= : $\text{byte}^* \times \text{byte} \rightarrow \text{byte}$
= : $\text{char}^* \times \text{char} \rightarrow \text{char}$
= : $\text{short}^* \times \text{short} \rightarrow \text{short}$
= : $\text{int}^* \times \text{int} \rightarrow \text{int}$
= : $\text{long}^* \times \text{long} \rightarrow \text{long}$
= : $\text{float}^* \times \text{float} \rightarrow \text{float}$
= : $\text{double}^* \times \text{double} \rightarrow \text{double}$

Es wird nur der Parameter konvertiert, der nicht dem Seiteneffekt unterliegt.

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
()	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: $x = \min(a, \min(a,b) + 4L)$

$$x = \min (a , \min (a , b) + 4L)$$

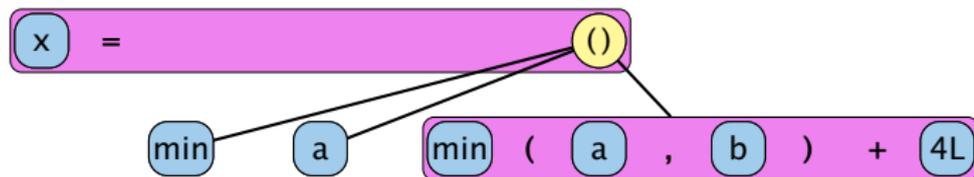
Beispiel: $x = \min(a, \min(a,b) + 4L)$

$$x = \min(a, \min(a, b) + 4L)$$

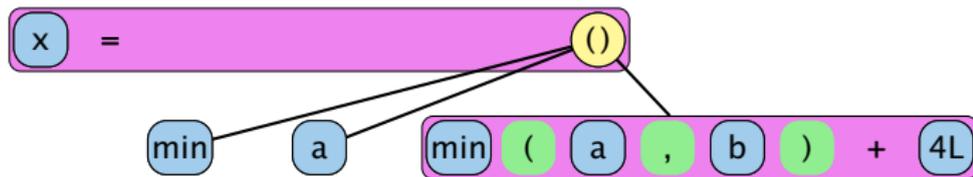
Beispiel: $x = \min(a, \min(a,b) + 4L)$

$$x = \min(a, \min(a, b) + 4L)$$

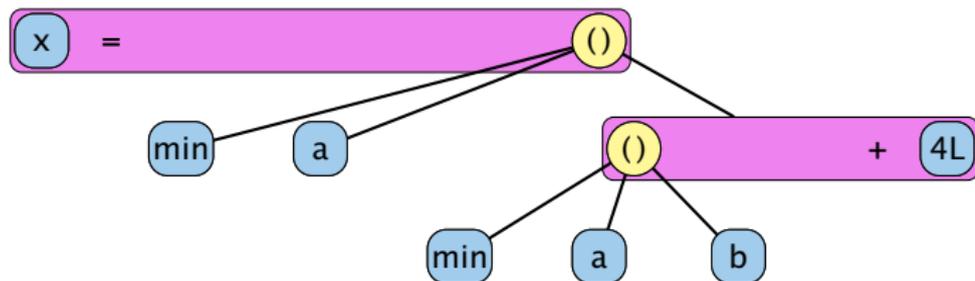
Beispiel: $x = \min(a, \min(a,b) + 4L)$



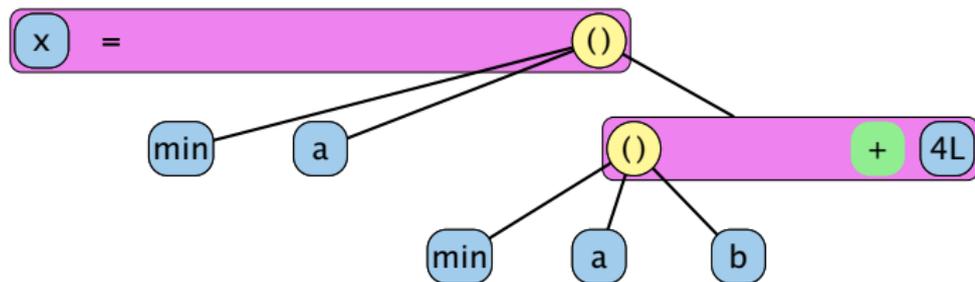
Beispiel: $x = \min(a, \min(a, b) + 4L)$



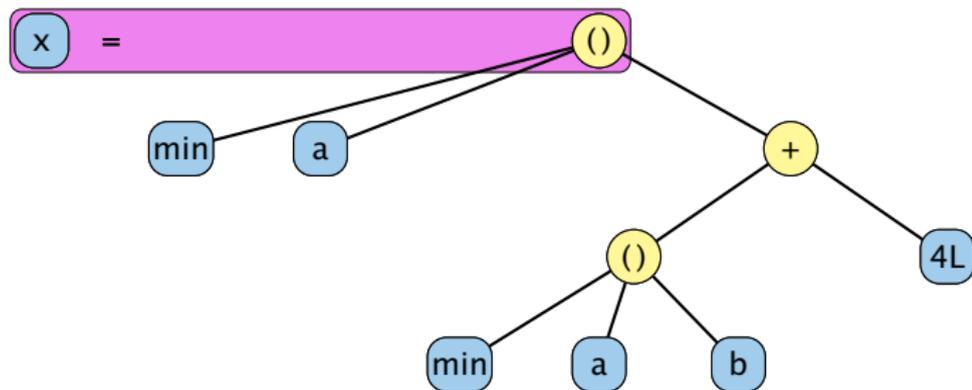
Beispiel: $x = \min(a, \min(a,b) + 4L)$



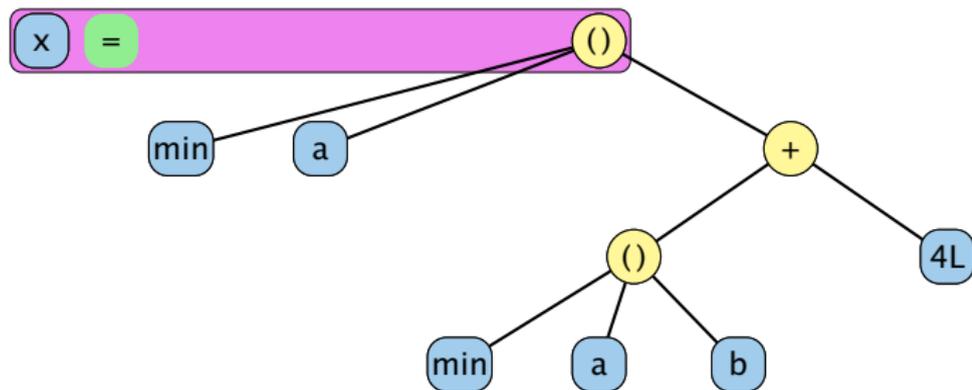
Beispiel: $x = \min(a, \min(a,b) + 4L)$



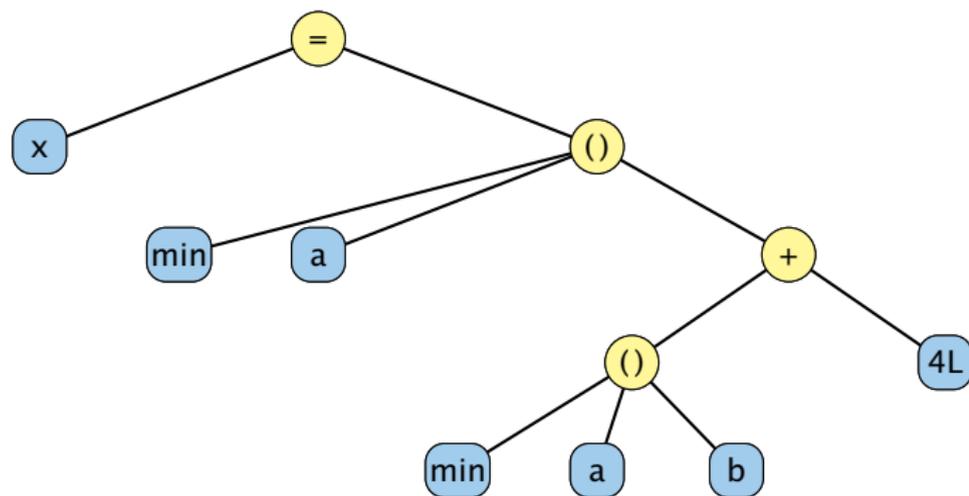
Beispiel: $x = \min(a, \min(a,b) + 4L)$



Beispiel: $x = \min(a, \min(a,b) + 4L)$



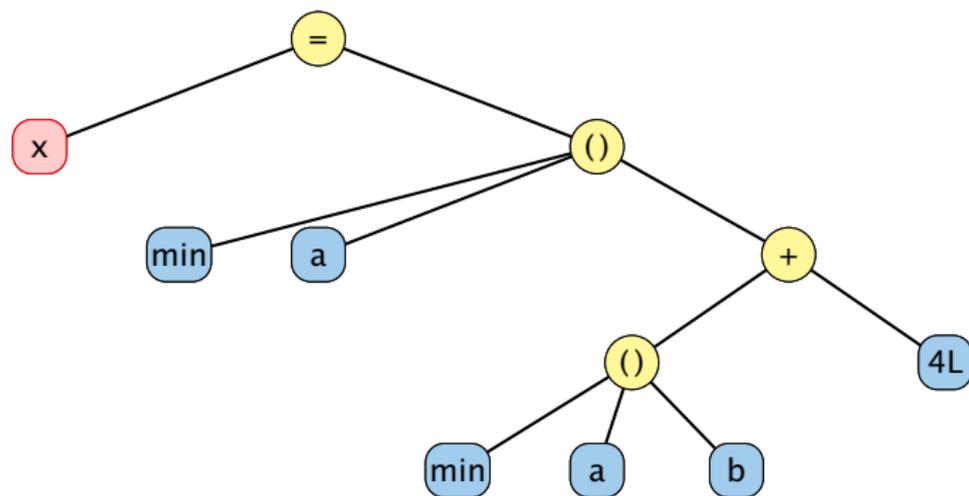
Beispiel: $x = \min(a, \min(a,b) + 4L)$



```
int min(int,int)
float min(float,float)
double min(double,double)
```

long x int a int b

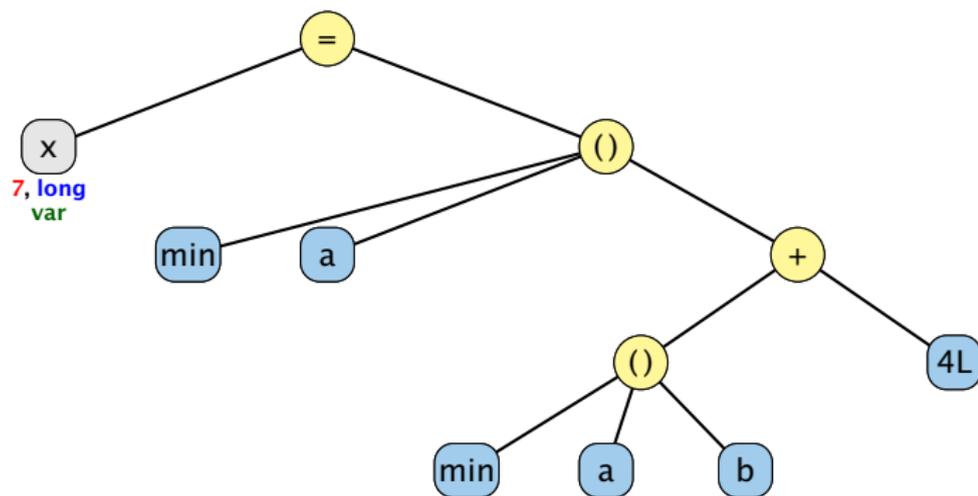
Beispiel: $x = \min(a, \min(a,b) + 4L)$



```
int min(int,int)
float min(float,float)
double min(double,double)
```

long x int a int b

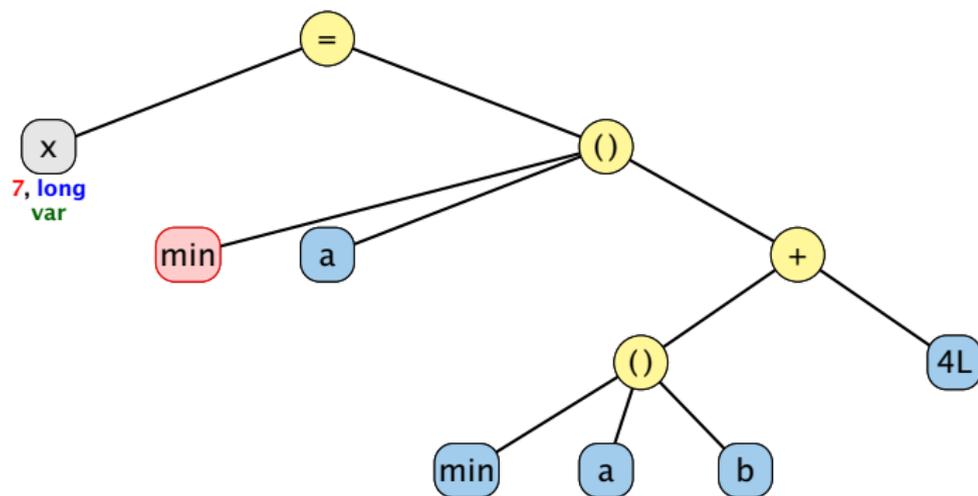
Beispiel: $x = \min(a, \min(a,b) + 4L)$



```
int min(int,int)
float min(float,float)
double min(double,double)
```

long x int a int b

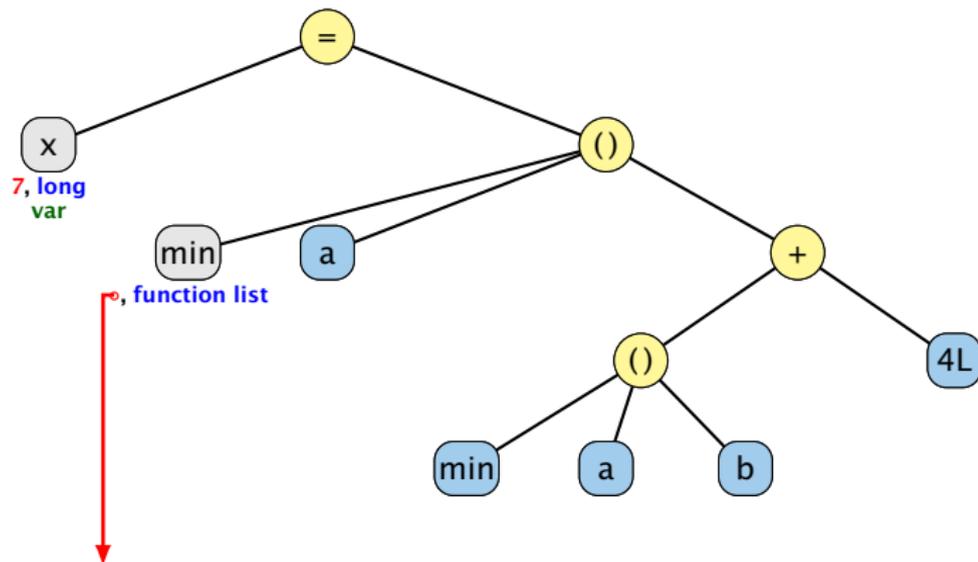
Beispiel: $x = \min(a, \min(a,b) + 4L)$



```
int min(int,int)
float min(float,float)
double min(double,double)
```

long x int a int b

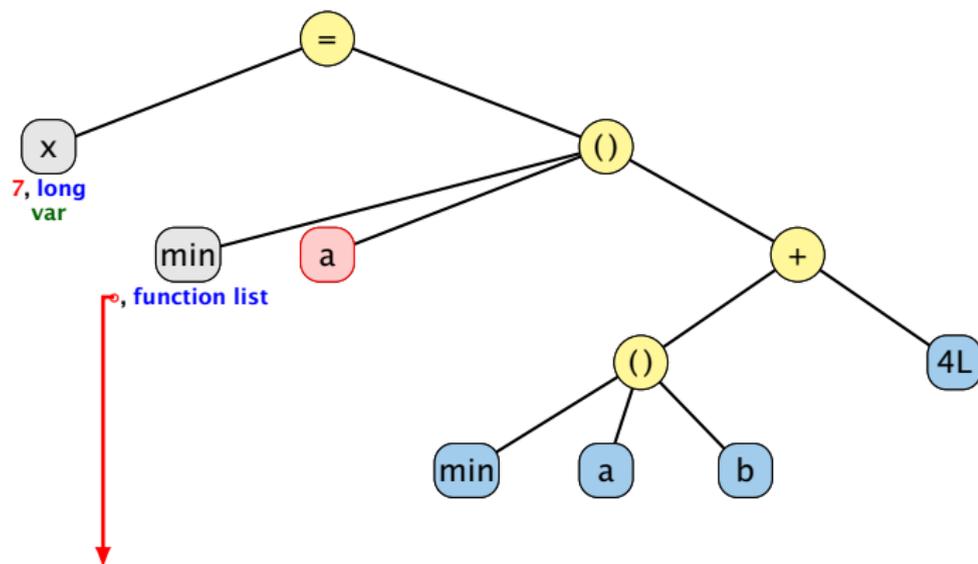
Beispiel: $x = \min(a, \min(a,b) + 4L)$



int min(int,int)
float min(float,float)
double min(double,double)

long x int a int b

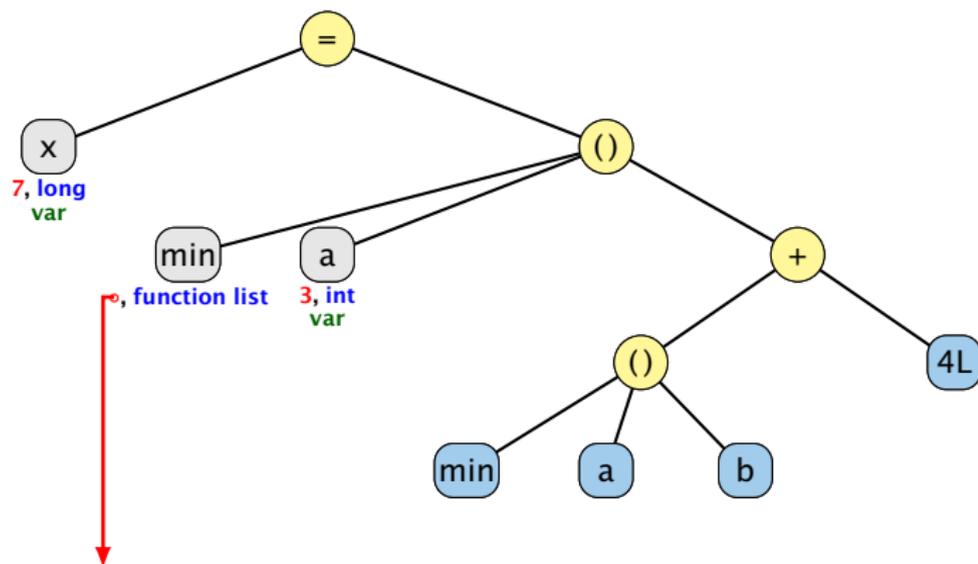
Beispiel: $x = \min(a, \min(a,b) + 4L)$



`int min(int,int)`
`float min(float,float)`
`double min(double,double)`

`long x` `int a` `int b`

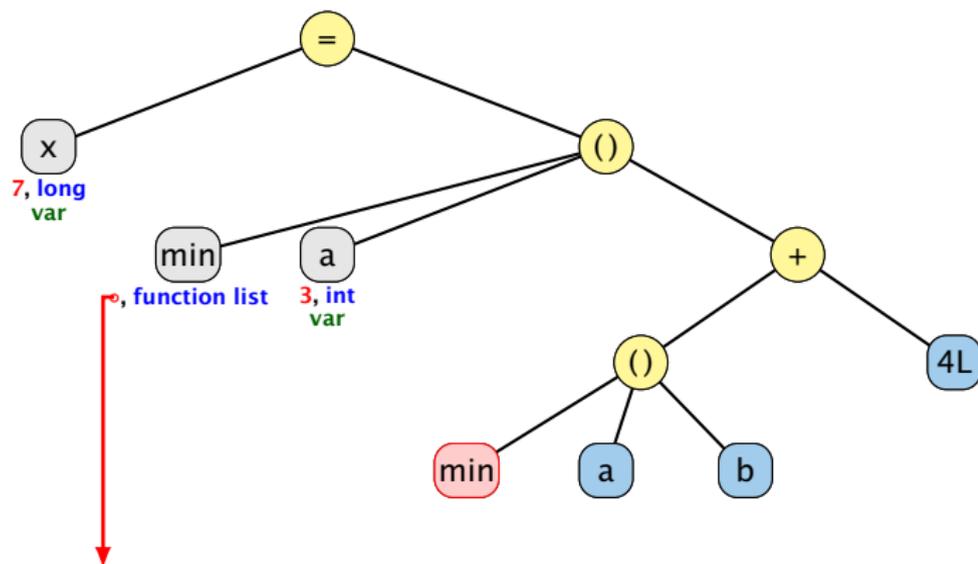
Beispiel: $x = \min(a, \min(a,b) + 4L)$



int min(int,int)
float min(float,float)
double min(double,double)

long x int a int b

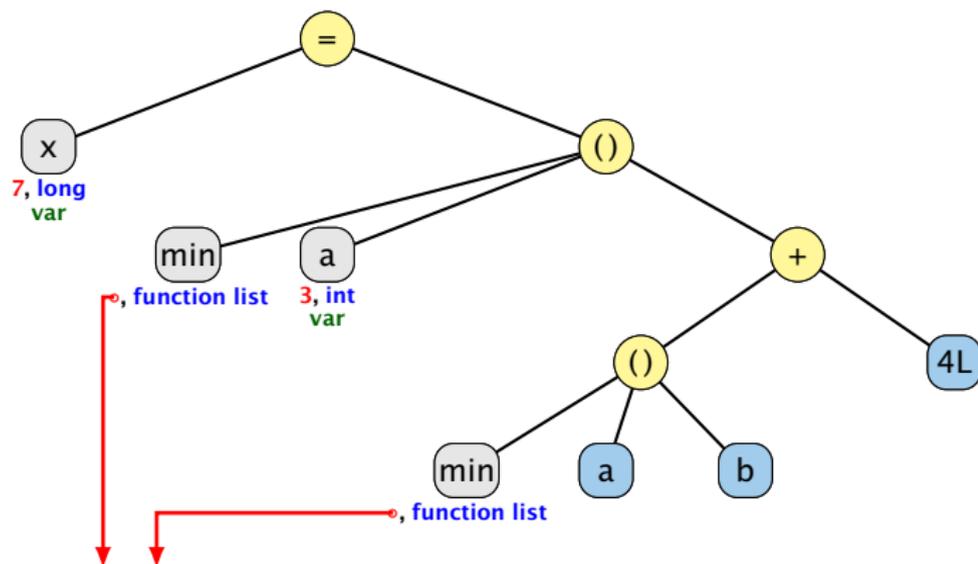
Beispiel: $x = \min(a, \min(a,b) + 4L)$



`int min(int,int)`
`float min(float,float)`
`double min(double,double)`

`long x` `int a` `int b`

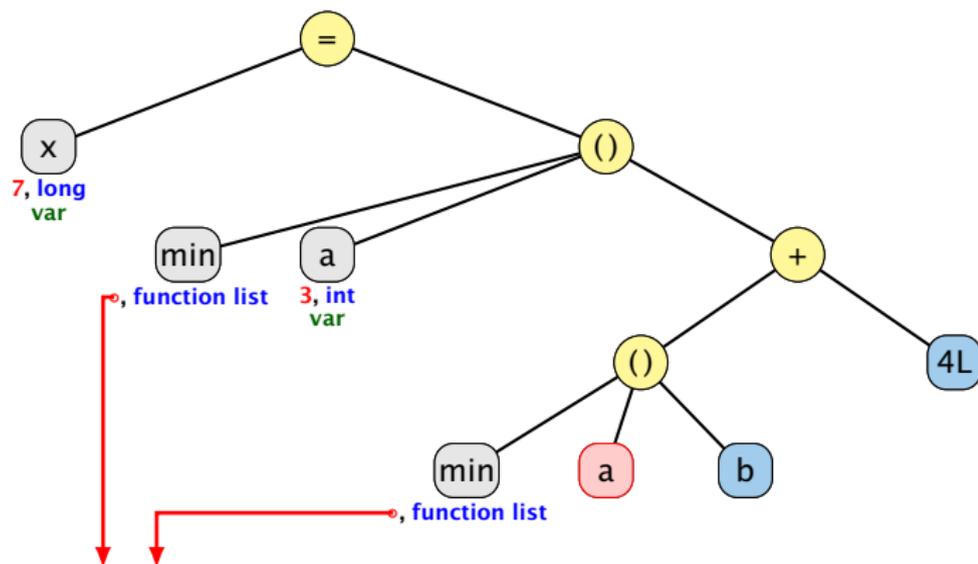
Beispiel: $x = \min(a, \min(a,b) + 4L)$



int min(int,int)
float min(float,float)
double min(double,double)

long x int a int b

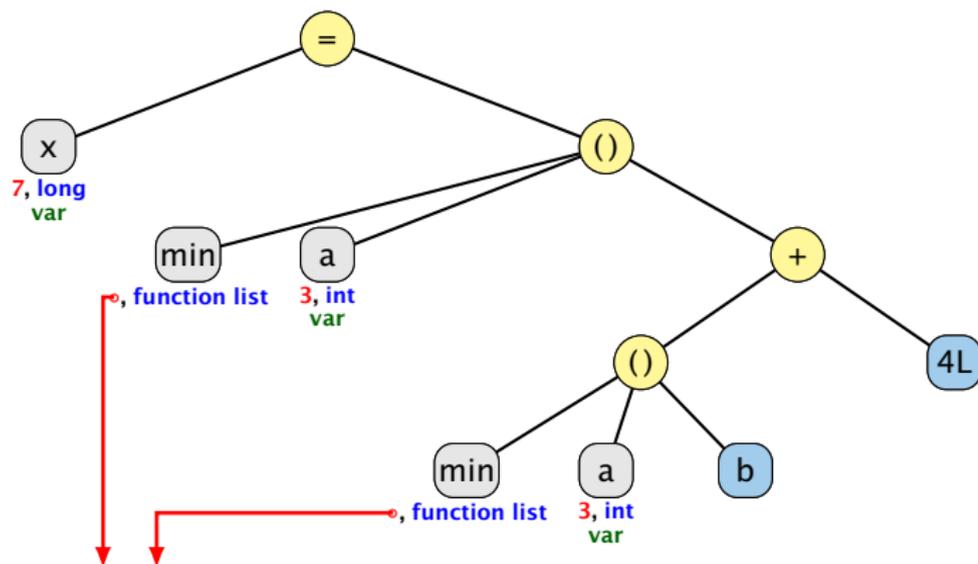
Beispiel: $x = \min(a, \min(a,b) + 4L)$



`int min(int,int)`
`float min(float,float)`
`double min(double,double)`

`long x` `7` `int a` `3` `int b` `5`

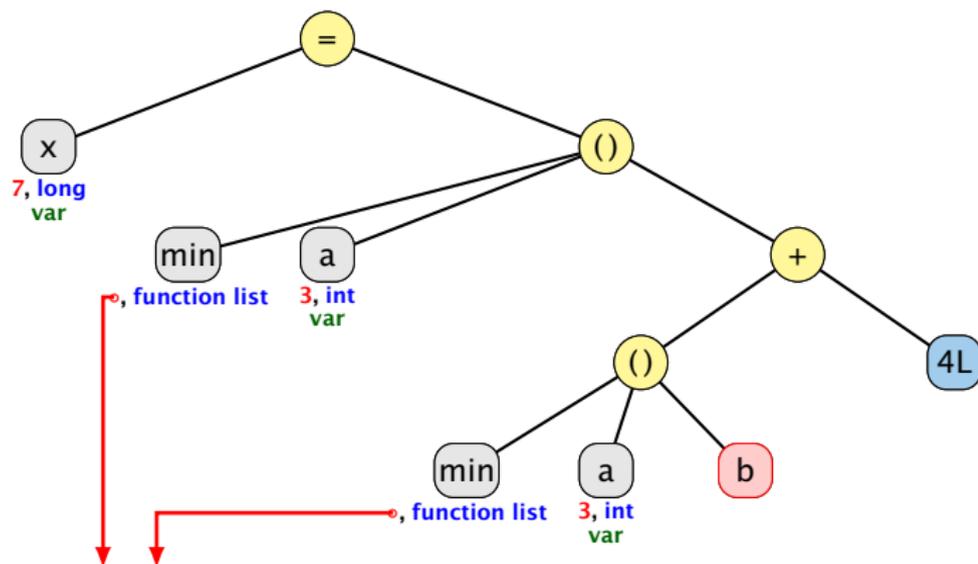
Beispiel: $x = \min(a, \min(a,b) + 4L)$



```
int min(int,int)
float min(float,float)
double min(double,double)
```

long x int a int b

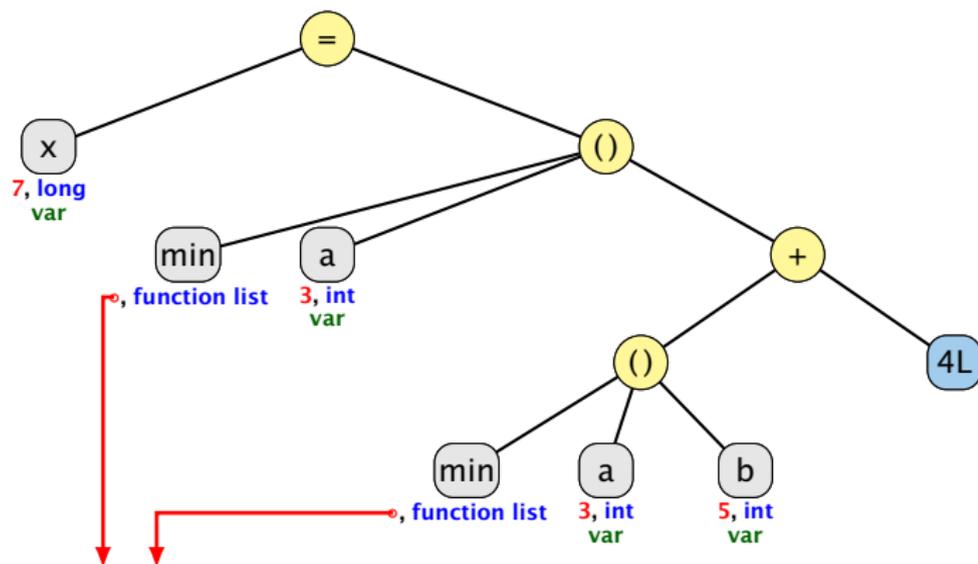
Beispiel: $x = \min(a, \min(a,b) + 4L)$



int min(int,int)
float min(float,float)
double min(double,double)

long x [7] int a [3] int b [5]

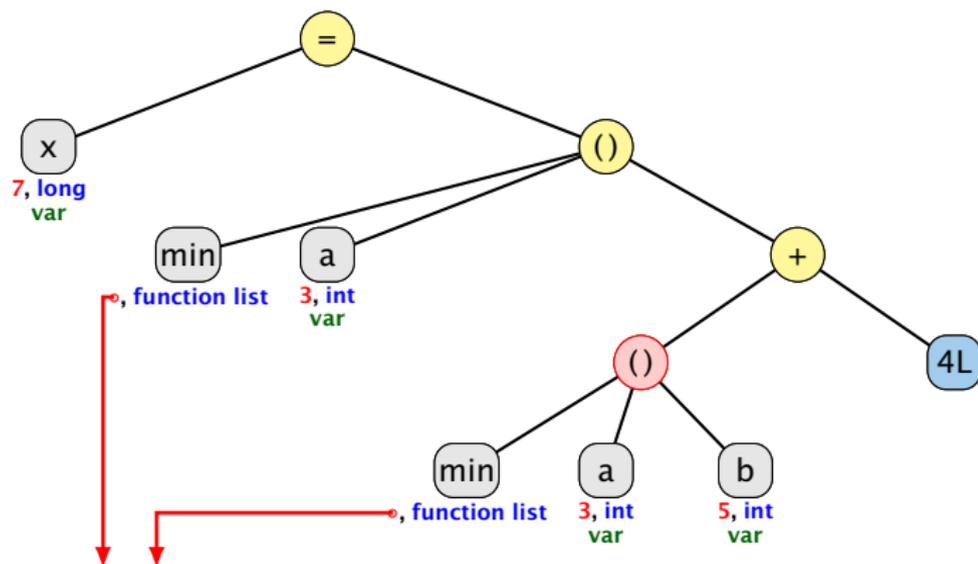
Beispiel: $x = \min(a, \min(a,b) + 4L)$



int min(int,int)
float min(float,float)
double min(double,double)

long x int a int b

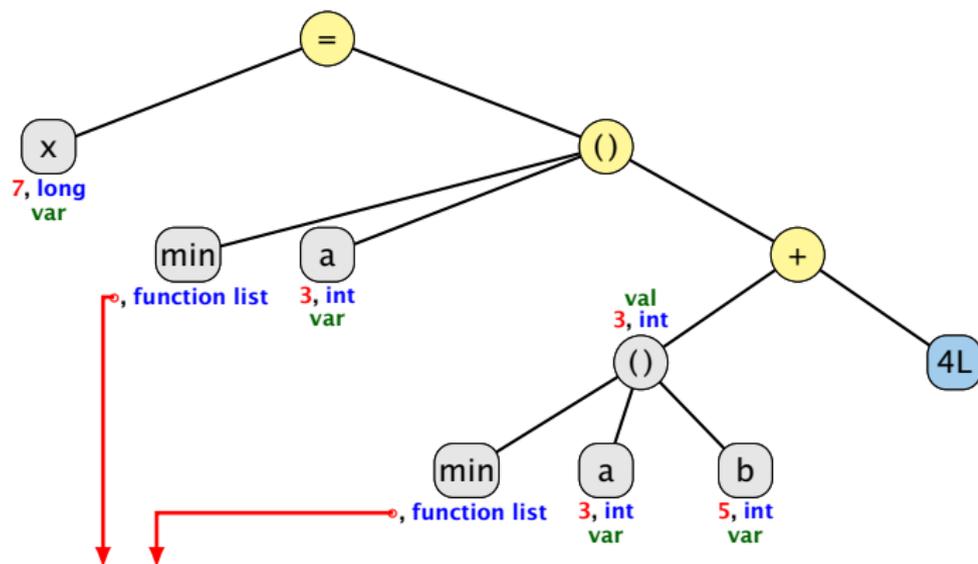
Beispiel: $x = \min(a, \min(a,b) + 4L)$



`int min(int,int)`
`float min(float,float)`
`double min(double,double)`

`long x` `7` `int a` `3` `int b` `5`

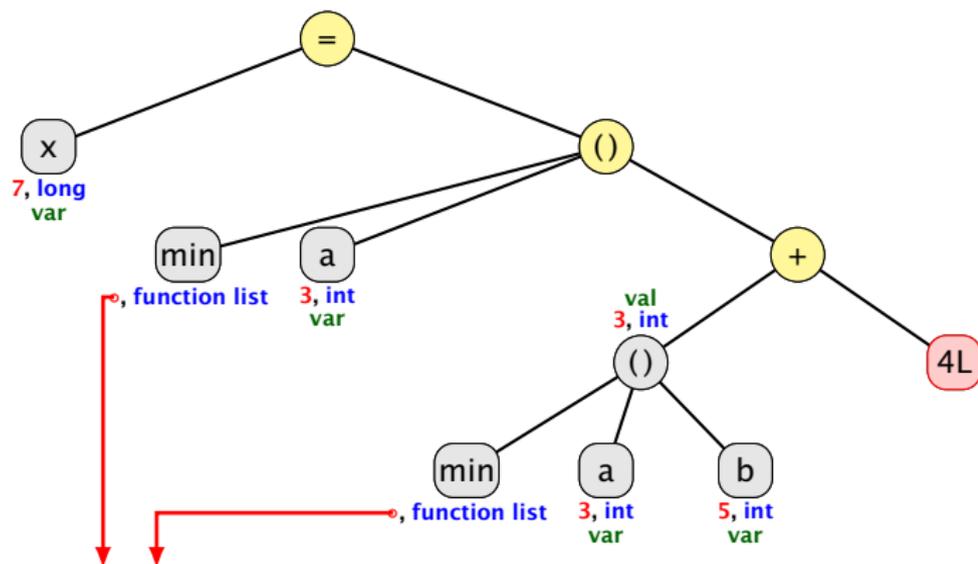
Beispiel: $x = \min(a, \min(a,b) + 4L)$



int min(int,int)
float min(float,float)
double min(double,double)

long x int a int b

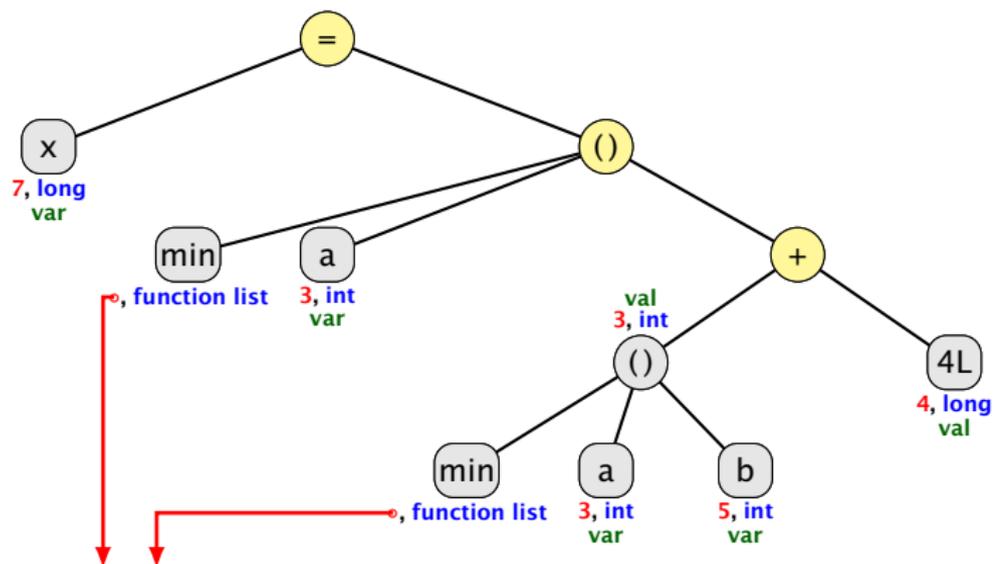
Beispiel: $x = \min(a, \min(a,b) + 4L)$



`int min(int,int)`
`float min(float,float)`
`double min(double,double)`

`long x` `int a` `int b`

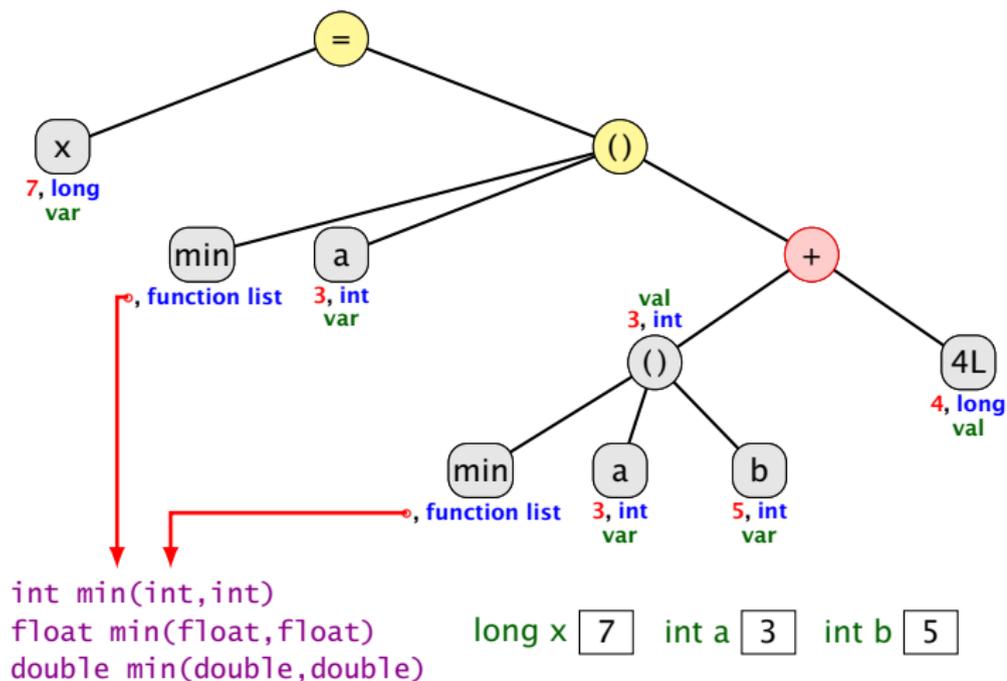
Beispiel: $x = \min(a, \min(a,b) + 4L)$



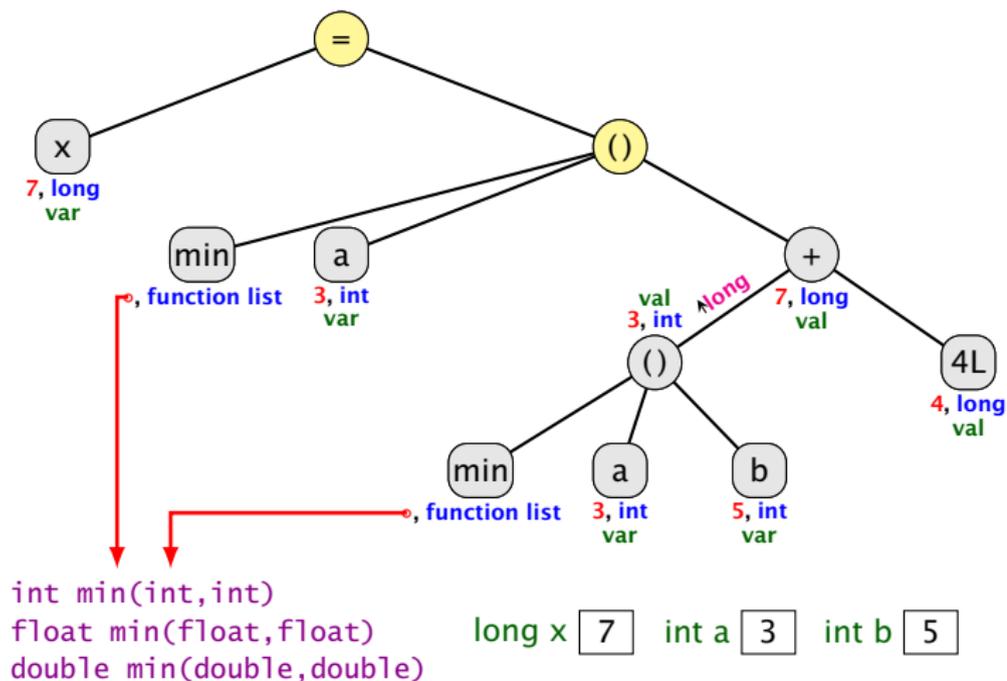
int min(int,int)
float min(float,float)
double min(double,double)

long x int a int b

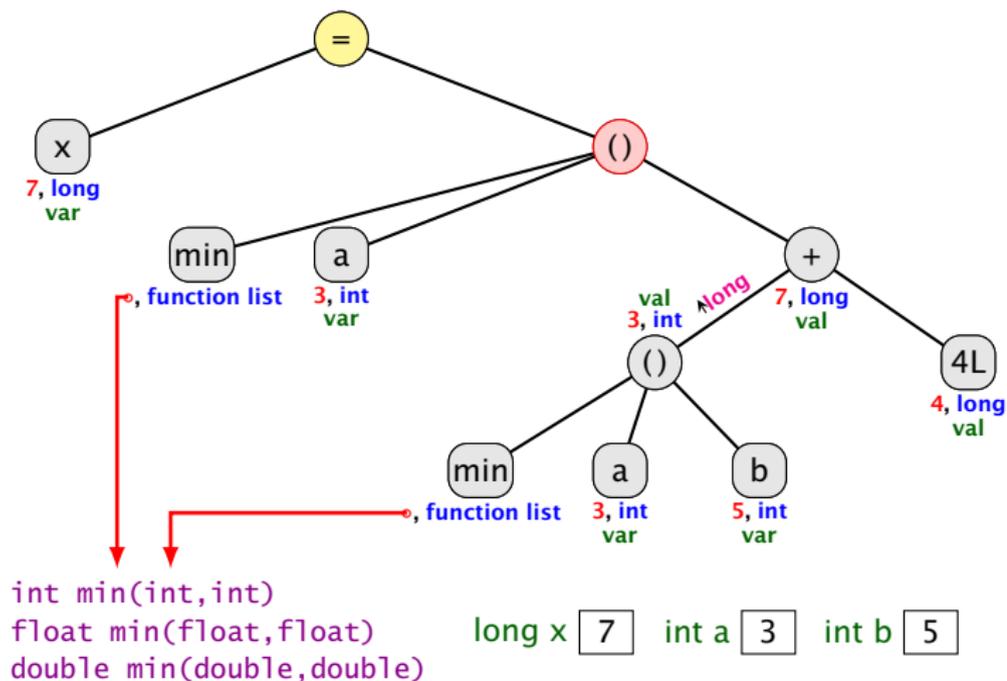
Beispiel: $x = \min(a, \min(a,b) + 4L)$



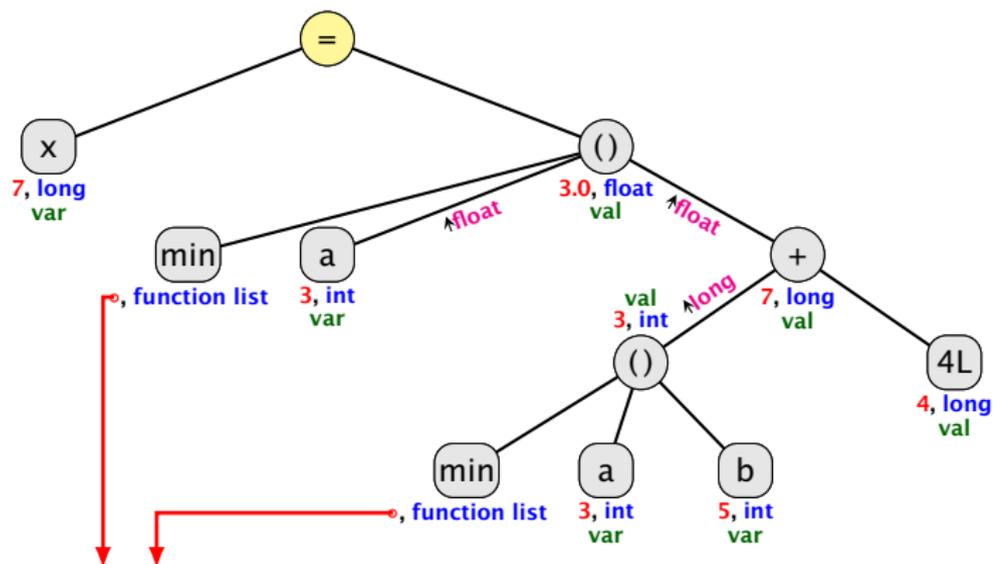
Beispiel: $x = \min(a, \min(a,b) + 4L)$



Beispiel: $x = \min(a, \min(a,b) + 4L)$



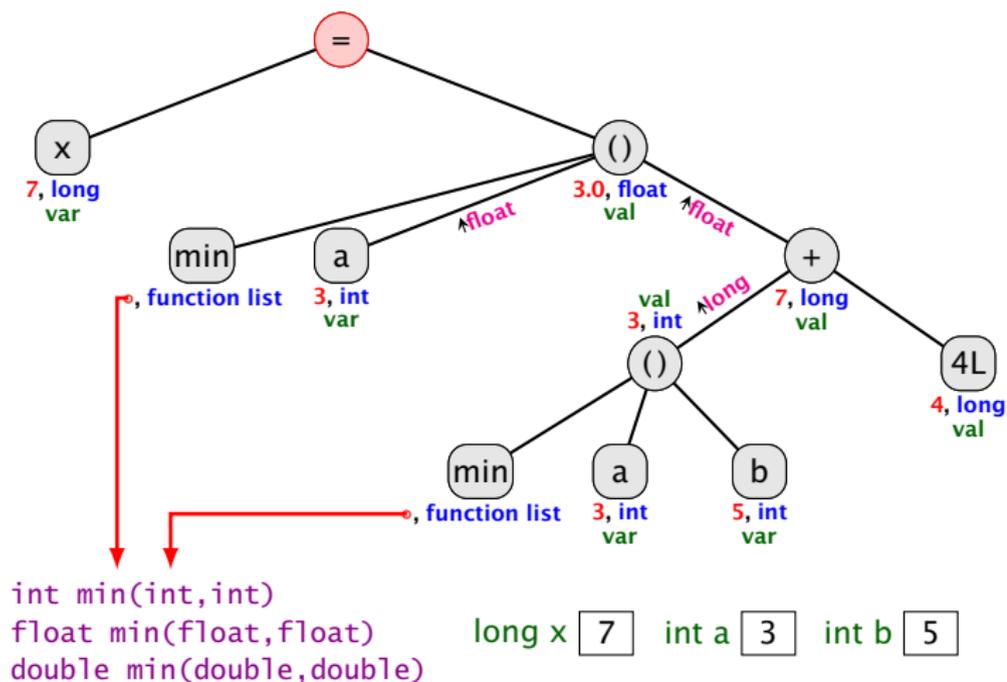
Beispiel: $x = \min(a, \min(a,b) + 4L)$



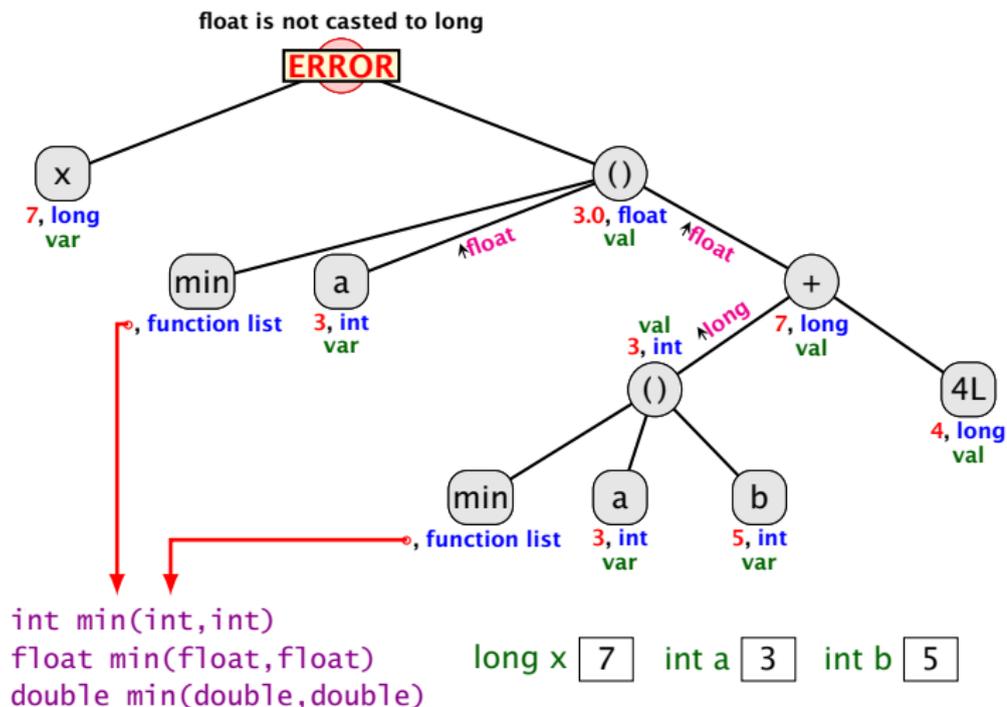
int min(int,int)
float min(float,float)
double min(double,double)

long x [7] int a [3] int b [5]

Beispiel: $x = \min(a, \min(a,b) + 4L)$



Beispiel: $x = \min(a, \min(a,b) + 4L)$



Impliziter Typecast – Strings

Spezialfall

- ▶ Falls beim Operator `+` ein Typ vom Typ `String` ist, wird der andere auch in einen `String` umgewandelt.
⇒ Stringkonkatenation.
- ▶ Jeder Typ in `Java` besitzt eine Stringrepräsentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: $s = a + b$

s = a + b

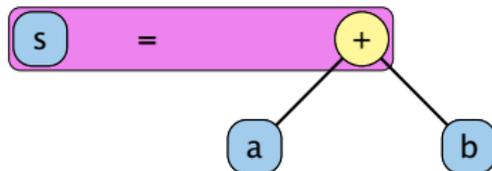
Beispiel: $s = a + b$



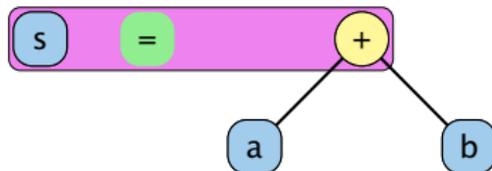
Beispiel: $s = a + b$



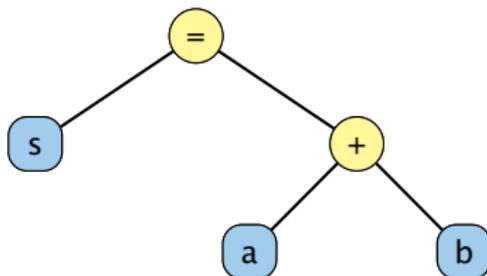
Beispiel: $s = a + b$



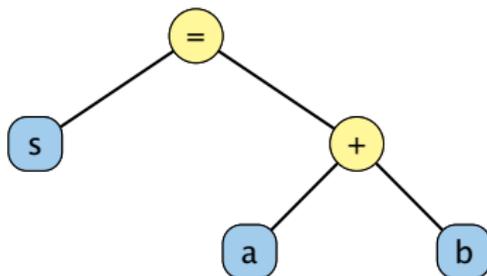
Beispiel: $s = a + b$



Beispiel: $s = a + b$

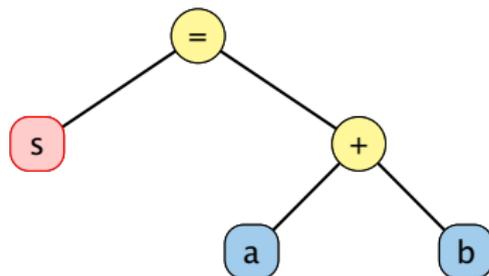


Beispiel: $s = a + b$



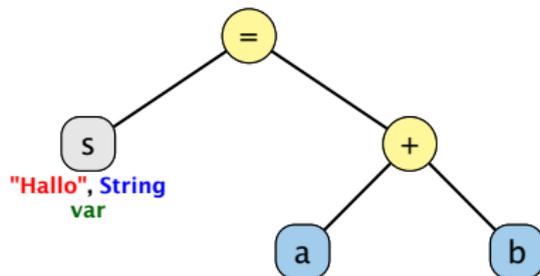
String s \rightarrow "Hallo" a 2 b 6

Beispiel: $s = a + b$



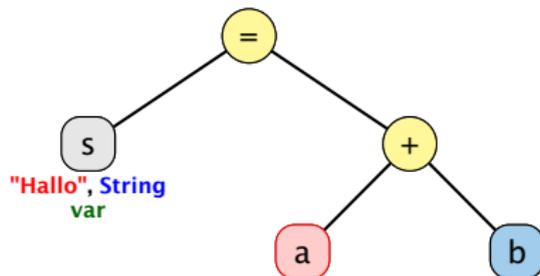
String s \rightarrow a b

Beispiel: $s = a + b$



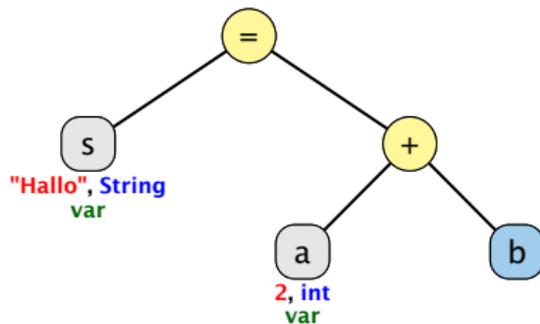
String s → "Hallo" a b

Beispiel: $s = a + b$



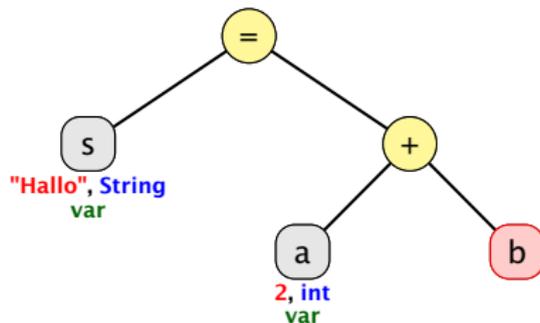
String s → "Hallo" a b

Beispiel: $s = a + b$



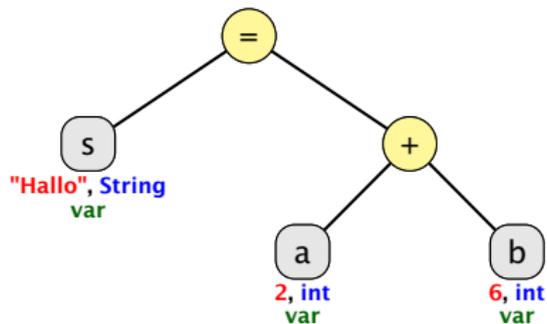
String s **→** "Hallo" a 2 b 6

Beispiel: $s = a + b$



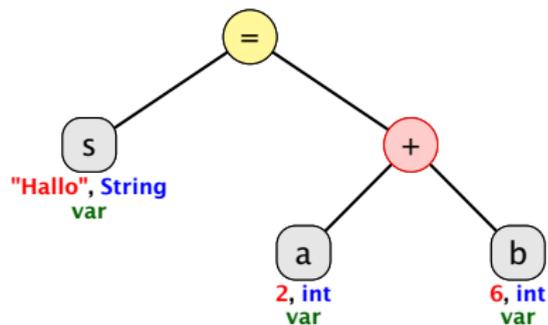
String s → "Hallo" a 2 b 6

Beispiel: $s = a + b$



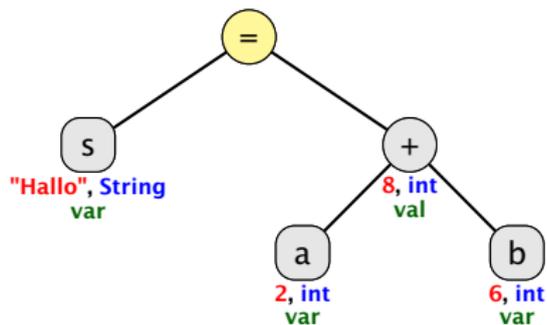
String s \rightarrow "Hallo" a 2 b 6

Beispiel: $s = a + b$



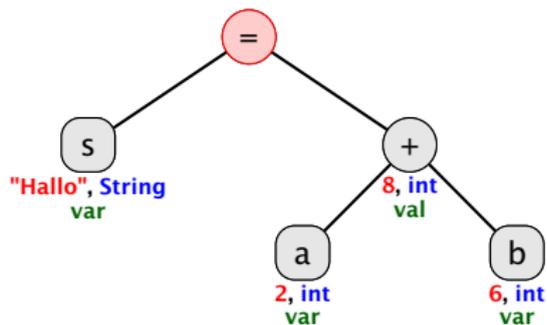
String s → "Hallo" a 2 b 6

Beispiel: $s = a + b$



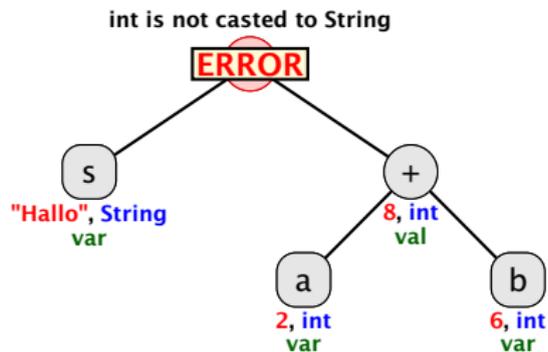
String s → "Hallo" a 2 b 6

Beispiel: $s = a + b$



String s → "Hallo" a 2 b 6

Beispiel: $s = a + b$



String s → "Hallo" a b

Beispiel: $s = "" + a + b$

$s = "" + a + b$

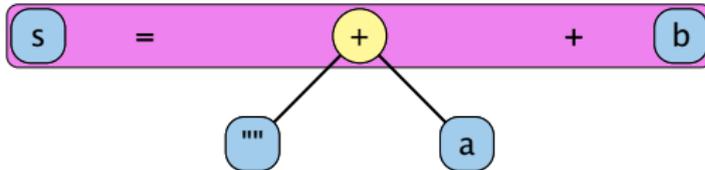
Beispiel: $s = "" + a + b$



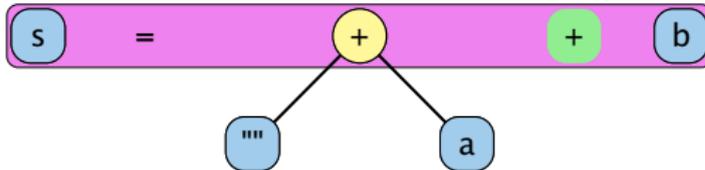
Beispiel: $s = "" + a + b$



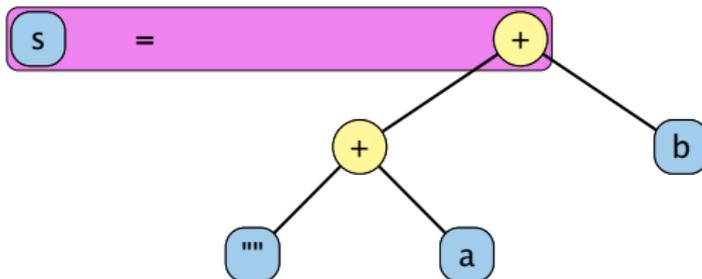
Beispiel: $s = "" + a + b$



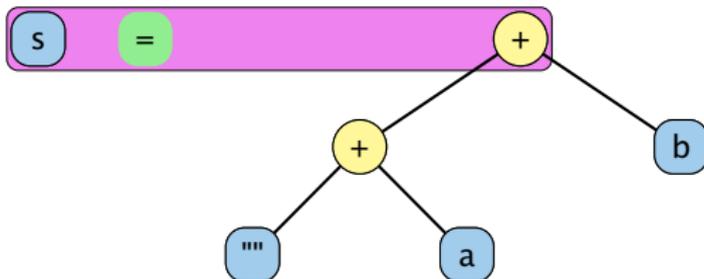
Beispiel: $s = "" + a + b$



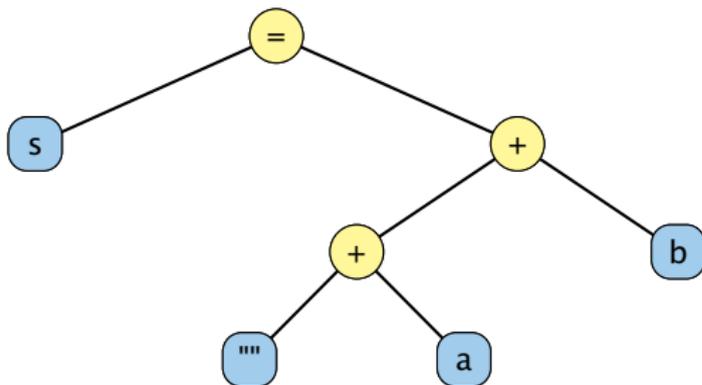
Beispiel: $s = "" + a + b$



Beispiel: $s = "" + a + b$

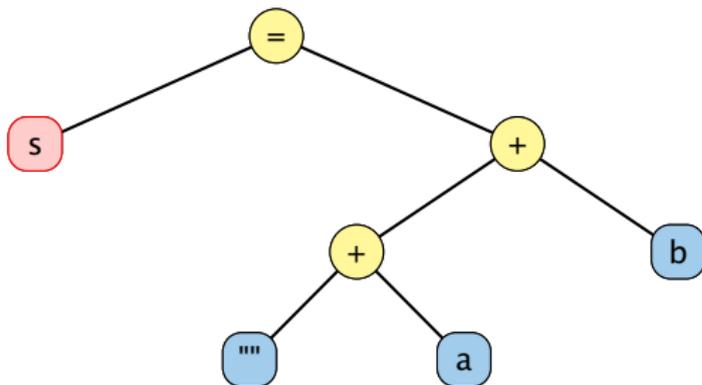


Beispiel: $s = "" + a + b$



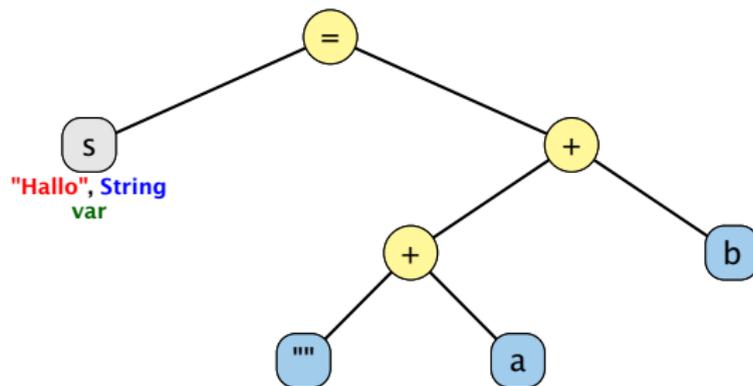
String s \rightarrow "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



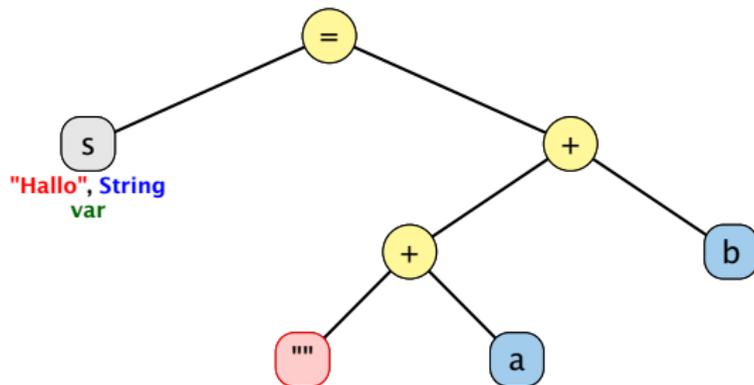
String s \rightarrow a b

Beispiel: $s = "" + a + b$



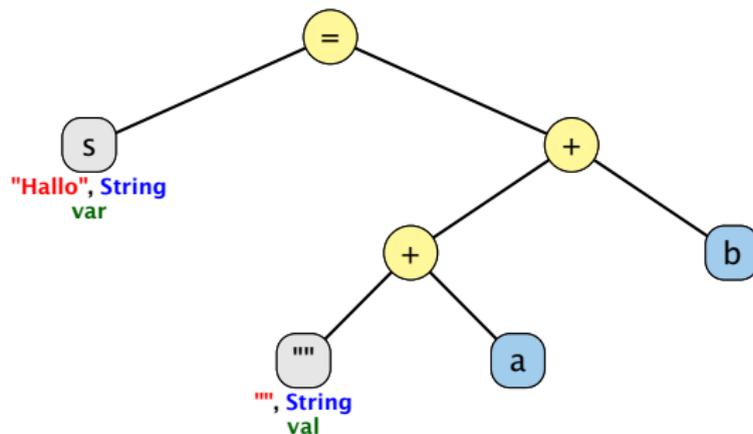
String `s` \rightarrow `"Hallo"` `a` `b`

Beispiel: $s = "" + a + b$



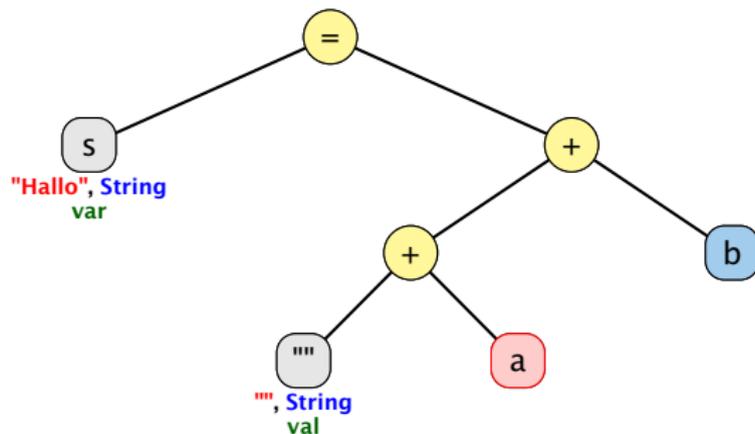
String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



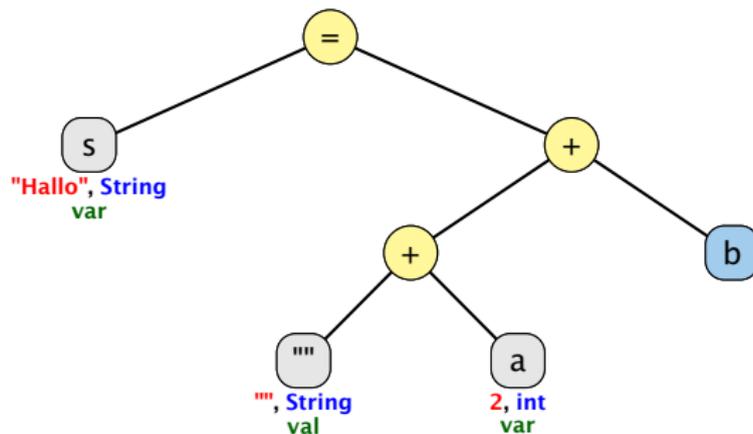
String s → a b

Beispiel: $s = "" + a + b$



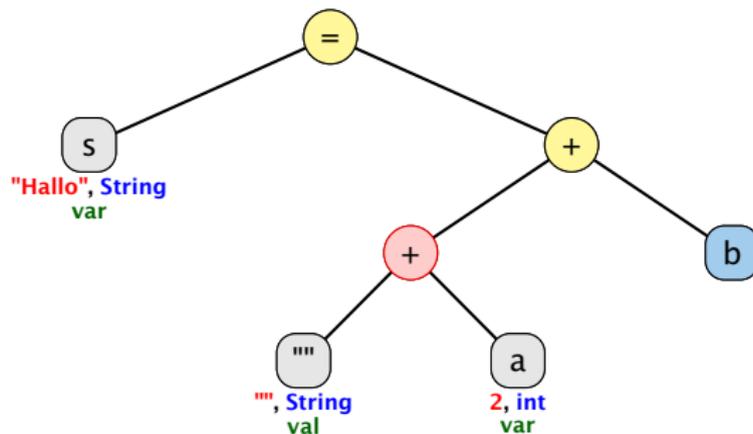
String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



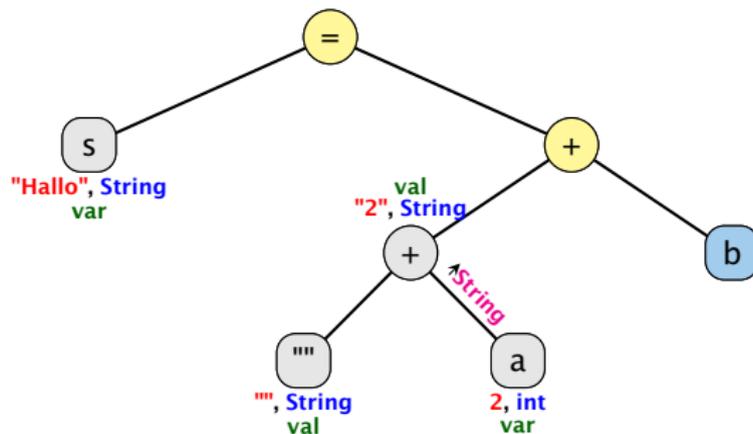
String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



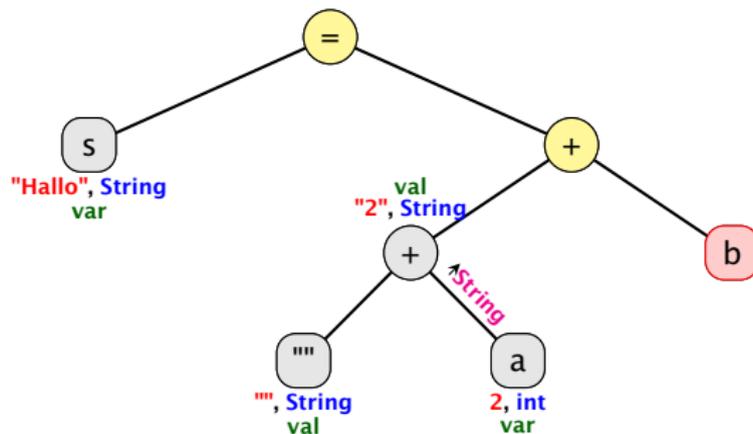
String s → a b

Beispiel: $s = "" + a + b$



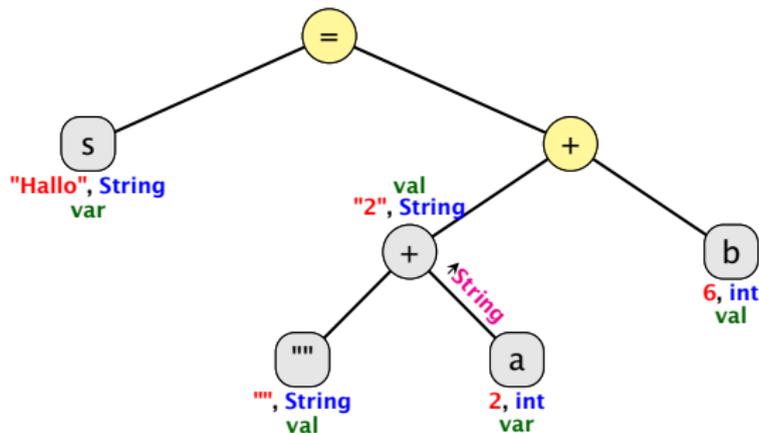
String s → a b

Beispiel: $s = "" + a + b$



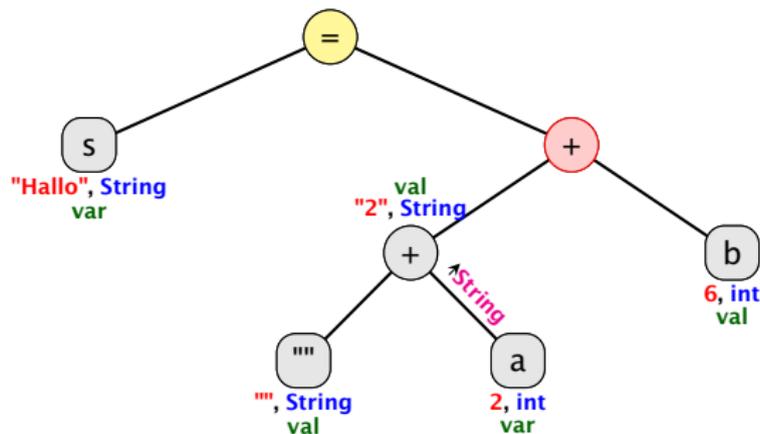
String s → a b

Beispiel: $s = "" + a + b$



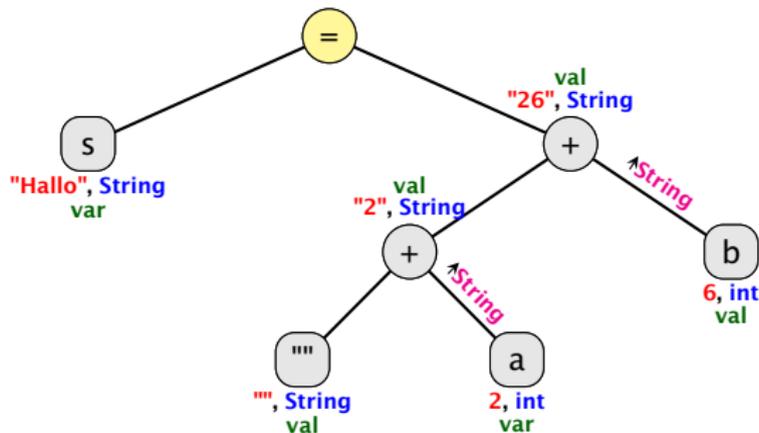
String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



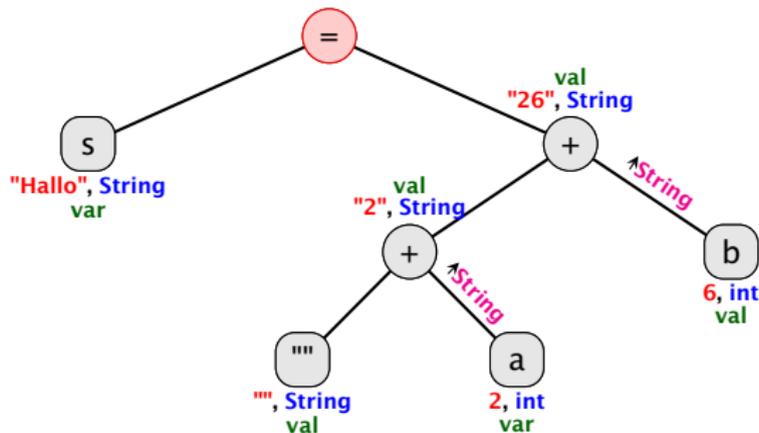
String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



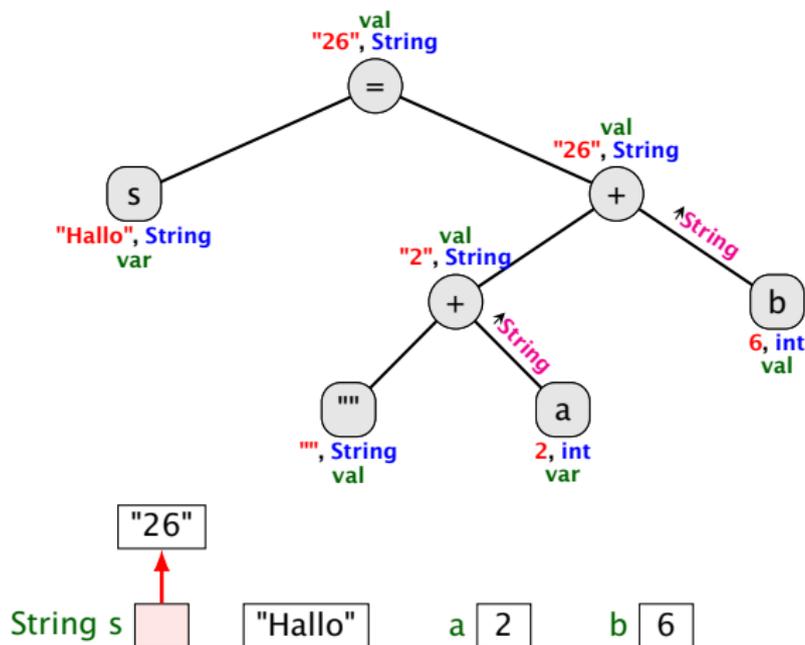
String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



String s → "Hallo" a 2 b 6

Beispiel: $s = "" + a + b$



Beispiel: $s = s + 1$

$$s = s + 1$$

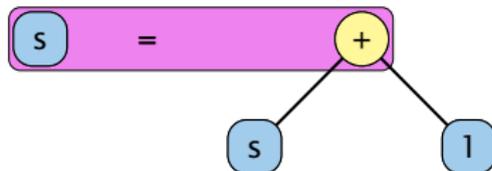
Beispiel: $s = s + 1$



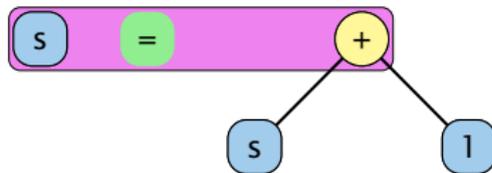
Beispiel: $s = s + 1$



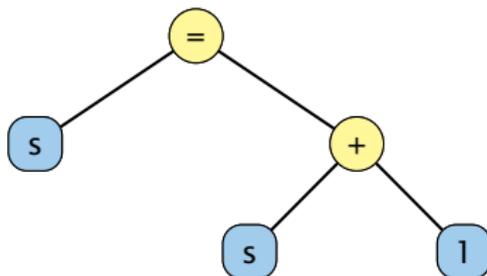
Beispiel: $s = s + 1$



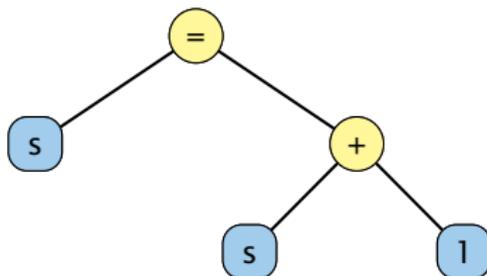
Beispiel: $s = s + 1$



Beispiel: $s = s + 1$

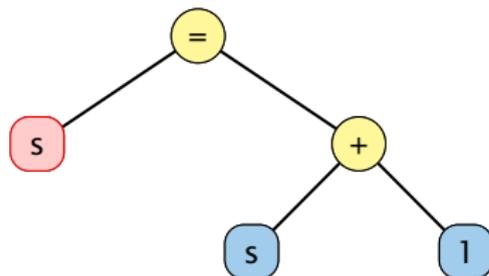


Beispiel: $s = s + 1$



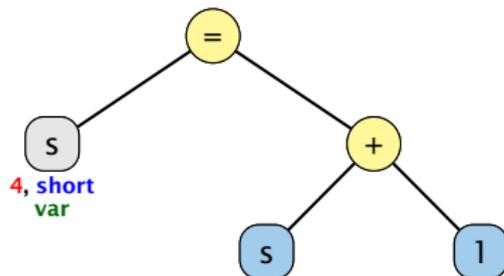
short s 4

Beispiel: $s = s + 1$



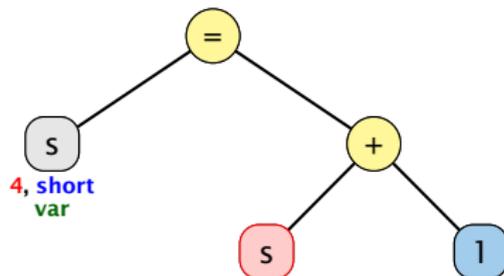
short s 4

Beispiel: $s = s + 1$



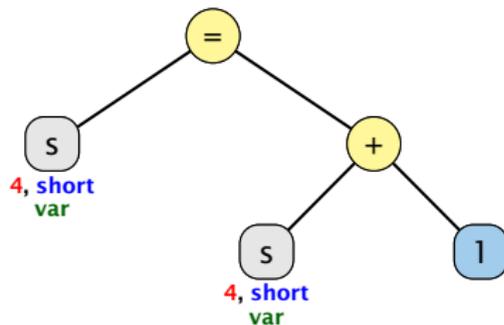
short s 4

Beispiel: $s = s + 1$



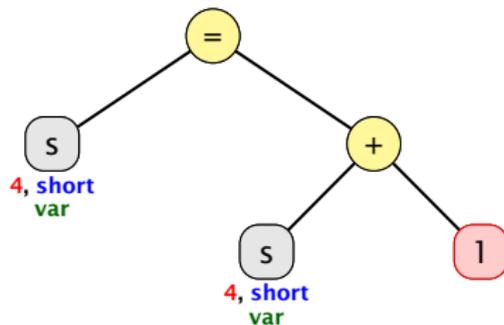
short s 4

Beispiel: $s = s + 1$



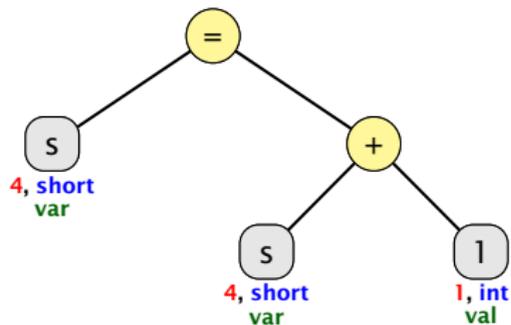
short s 4

Beispiel: $s = s + 1$



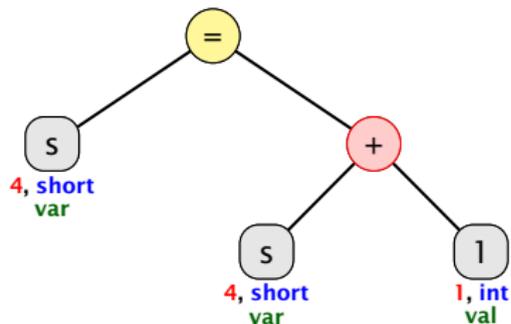
short s 4

Beispiel: $s = s + 1$



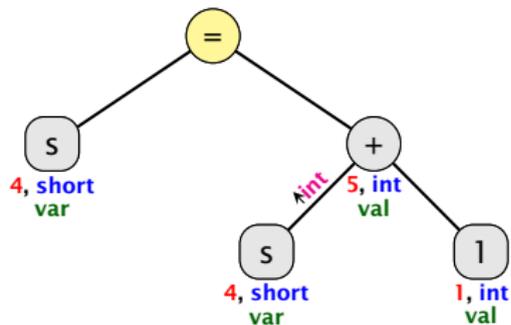
short s 4

Beispiel: $s = s + 1$



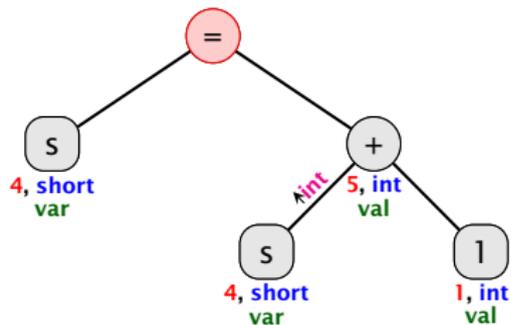
short s 4

Beispiel: $s = s + 1$



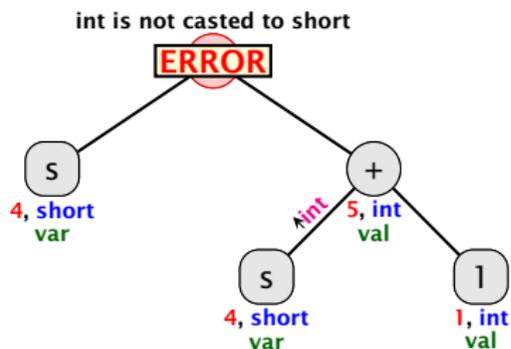
short s 4

Beispiel: $s = s + 1$



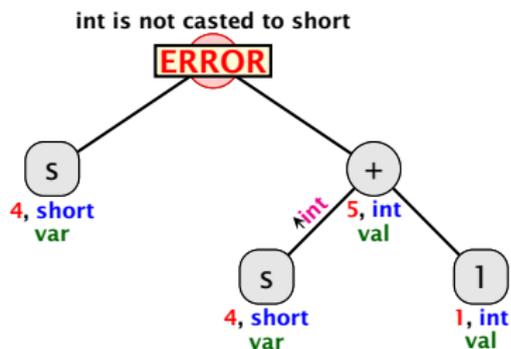
short s 4

Beispiel: $s = s + 1$



short s 4

Beispiel: $s = s + 1$



short s 4

Beispiel: $s = 7 + 1$

$$s = 7 + 1$$

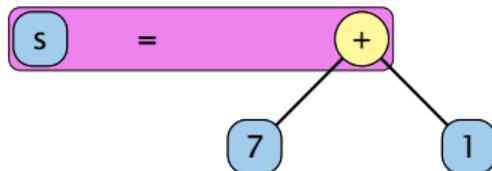
Beispiel: $s = 7 + 1$



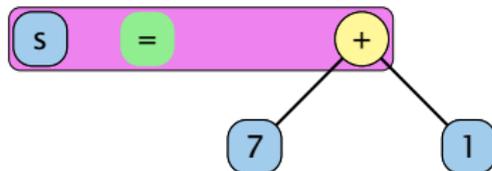
Beispiel: $s = 7 + 1$



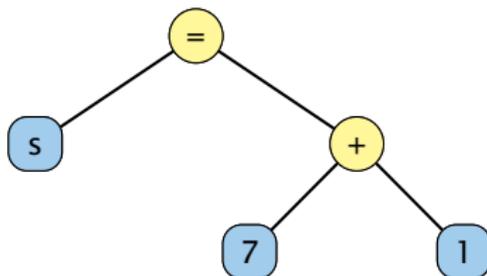
Beispiel: $s = 7 + 1$



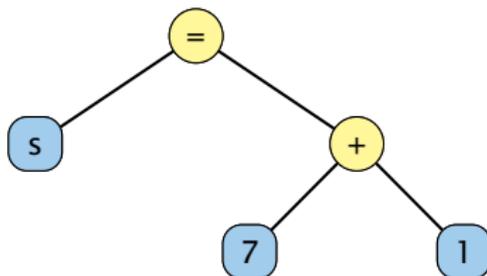
Beispiel: $s = 7 + 1$



Beispiel: $s = 7 + 1$

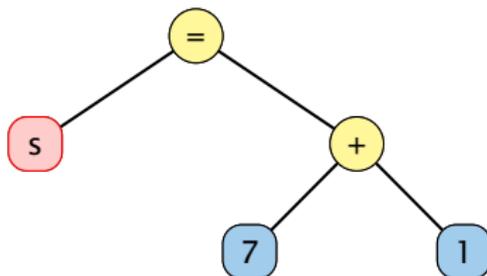


Beispiel: $s = 7 + 1$



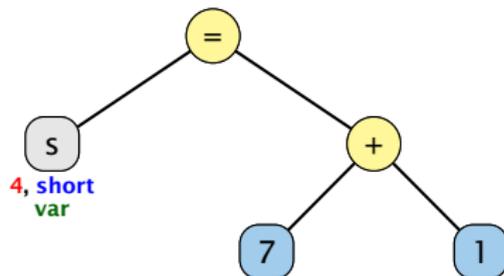
short s 4

Beispiel: $s = 7 + 1$



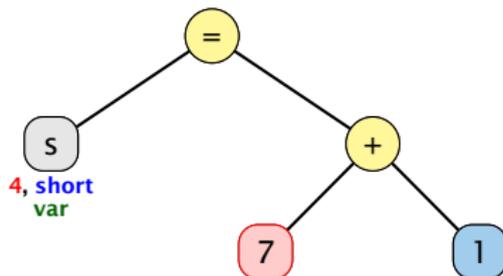
short s 4

Beispiel: $s = 7 + 1$



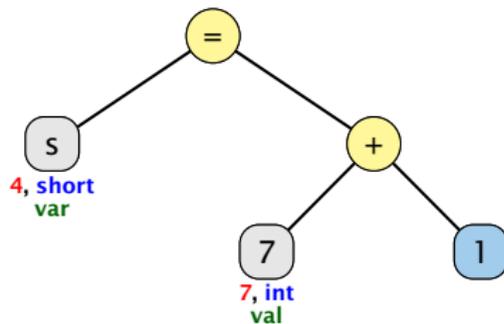
short s 4

Beispiel: $s = 7 + 1$



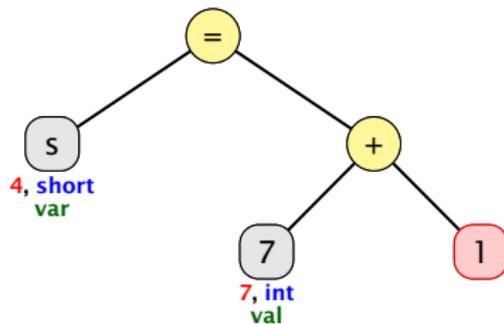
short s 4

Beispiel: $s = 7 + 1$



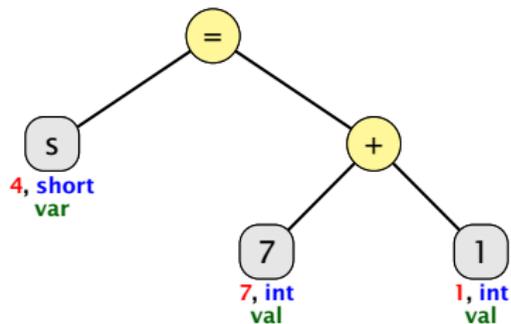
short s 4

Beispiel: $s = 7 + 1$



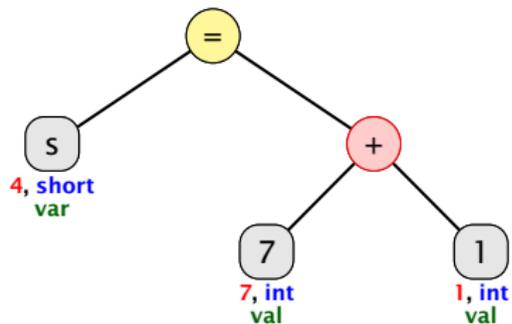
short s 4

Beispiel: $s = 7 + 1$



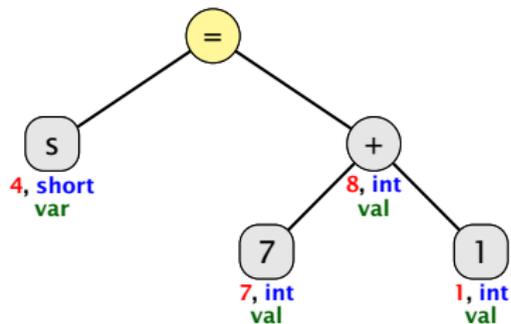
short s 4

Beispiel: $s = 7 + 1$



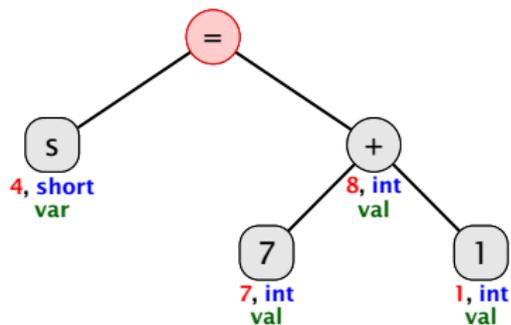
short s 4

Beispiel: $s = 7 + 1$



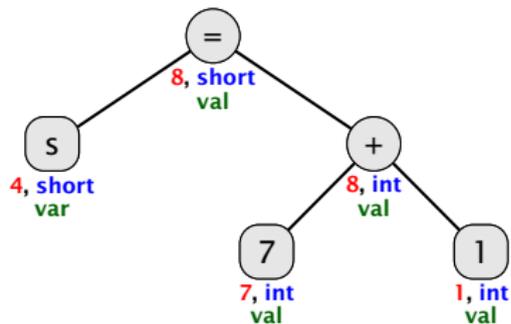
short s 4

Beispiel: $s = 7 + 1$



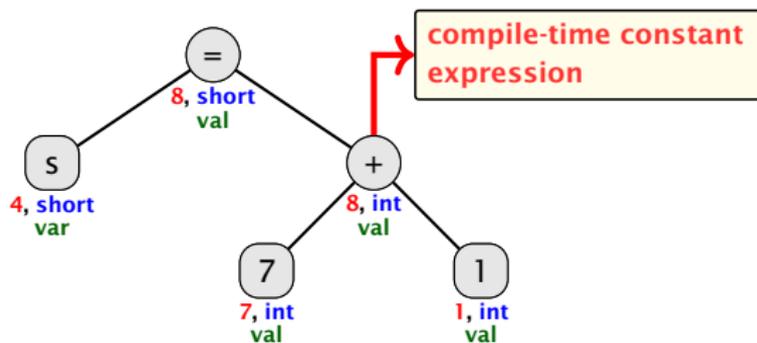
short s 4

Beispiel: $s = 7 + 1$



short s 8

Beispiel: $s = 7 + 1$



short s 8

Expliziter Typecast

<i>symbol</i>	<i>name</i>	<i>type</i>	<i>L/R</i>	<i>level</i>
(type)	typecast	Zahl, char	rechts	3

Beispiele mit Datenverlust

- ▶ `short s = (short) 23343445;`

Die obersten bits werden einfach weggeworfen...

- ▶ `double d = 1.5;`
`short s = (short) d;`
s hat danach den Wert 1.

...ohne Datenverlust:

- ▶ `int x = 5;`
`short s = (short) x;`

5.4 Arrays

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- ▶ Lege sie konsekutiv ab!
- ▶ Greife auf einzelne Werte über ihren Index zu!

Feld:	17	3	-2	9	0	1
Index:	0	1	2	3	4	5

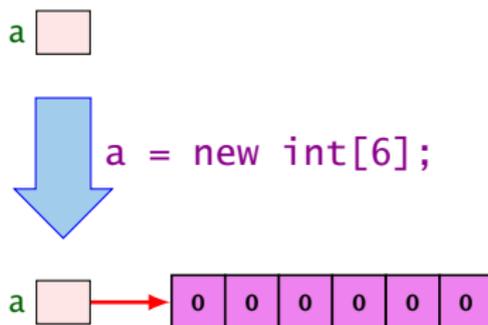
Beispiel

```
1 int[] a; // Deklaration
2 int n = read();
3
4 a = new int[n]; // Anlegen des Felds
5 int i = 0;
6 while (i < n) {
7     a[i] = read();
8     i = i + 1;
9 }
```

Einlesen eines Feldes

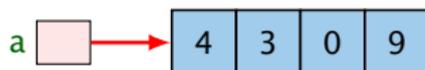
Beispiel

- ▶ `type[] name;` deklariert eine Variable für ein Feld (`array`), dessen Elemente vom Typ `type` sind.
- ▶ Alternative Schreibweise:
`type name[];`
- ▶ Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen `Verweis` darauf zurück:



Was ist eine Referenz?

Eine Referenzvariable speichert eine Adresse; an dieser Adresse liegt der eigentliche Inhalt der Variablen.



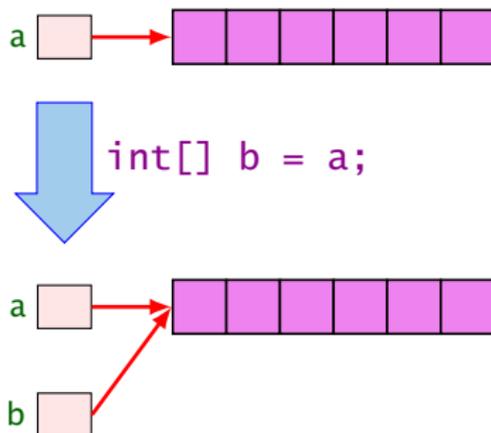
Wir können die Referenz nicht direkt manipulieren (nur über den **new**-Operator, oder indem wir eine andere Referenz zuweisen).

Adresse	Inhalt
⋮	⋮
0000 0127	
a: 0000 0128	0000 012C
0000 0129	
0000 012A	
0000 012B	
0000 012C	0000 0004
0000 012D	0000 0003
0000 012E	0000 0000
0000 012F	0000 0009
0000 0130	
⋮	⋮

A red arrow originates from the 'a:' label in the second column of the table and points to the memory address '0000 012C' in the first column.

5.4 Arrays

- ▶ Der Wert einer Feld-Variable ist also ein Verweis!!!
- ▶ `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



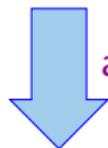
- ▶ **Alle nichtprimitive Datentypen sind Referenztypen, d.h., die zugehörige Variable speichert einen Verweis!!!**

5.4 Arrays

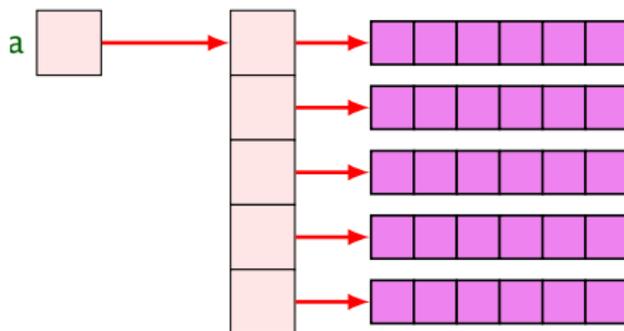
- ▶ Die Elemente eines Feldes sind von `0` an durchnummeriert.
- ▶ Die Anzahl der Elemente des Feldes `name` ist `name.length`.
- ▶ Auf das i -te Element greift man mit `name[i]` zu.
- ▶ Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall `{0, ..., name.length-1}` liegt.
- ▶ Liegt der Index außerhalb des Intervalls, wird eine `ArrayIndexOutOfBoundsException` ausgelöst (↑`Exceptions`).

Mehrdimensionale Felder

- ▶ **Java** unterstützt direkt nur eindimensionale Felder.
- ▶ ein zweidimensionales Feld ist ein Feld von Feldern...



`a = new int[5][6];`



Der new-Operator

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
new	new	Typ, Konstruktor	links	1

Erzeugt ein Objekt/Array und liefert eine Referenz darauf zurück.

1. Version: Erzeugung eines Arrays (Typ ist Arraytyp)

- ▶ `new int[3][7];` oder auch
- ▶ `new int[3][];` (ein Array, das 3 Verweise auf `int` enthält)
- ▶ `new String[10];`
- ▶ `new int[]{1,2,3};` (ein Array mit den `ints` 1, 2, 3)

2. Version: Erzeugung eines Objekts durch Aufruf eines Konstruktors

- ▶ `String s = new String("Hello World!");`

Der Index-Operator

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
<code>[]</code>	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: $x = a[3][2]$

$x = a[3][2]$

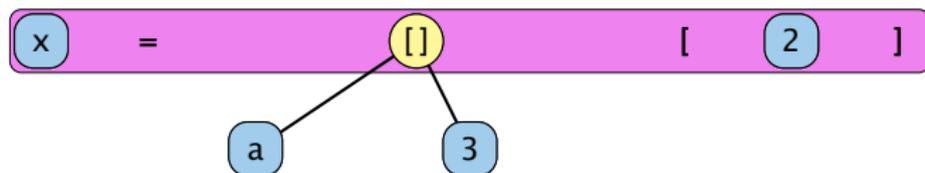
Beispiel: $x = a[3][2]$



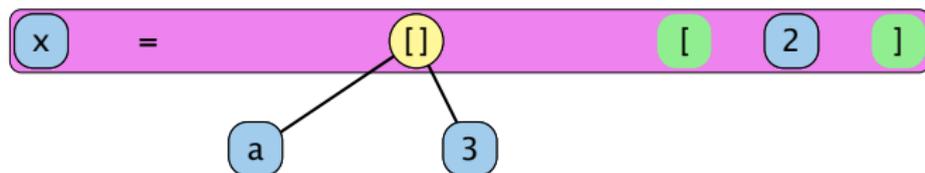
Beispiel: $x = a[3][2]$



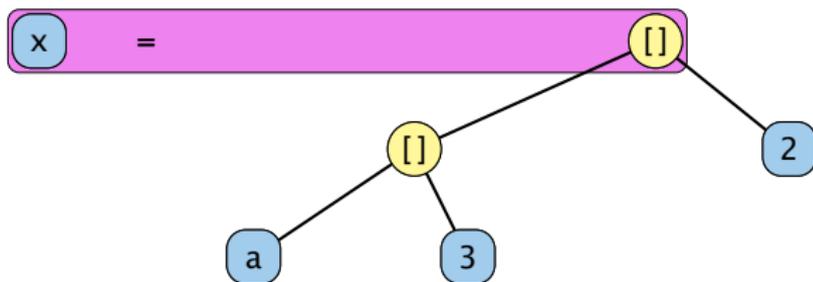
Beispiel: $x = a[3][2]$



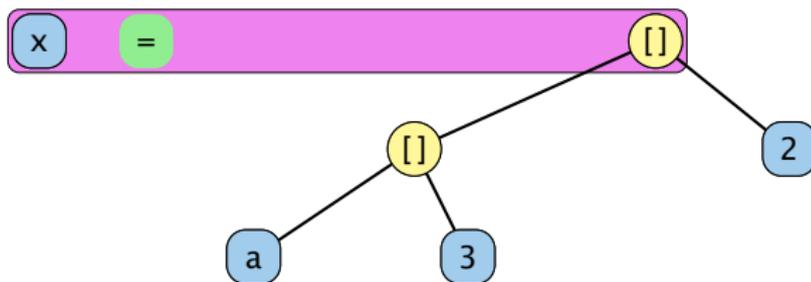
Beispiel: $x = a[3][2]$



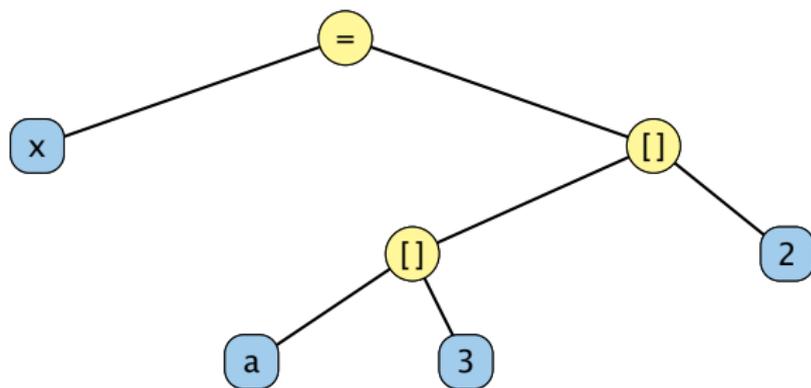
Beispiel: $x = a[3][2]$



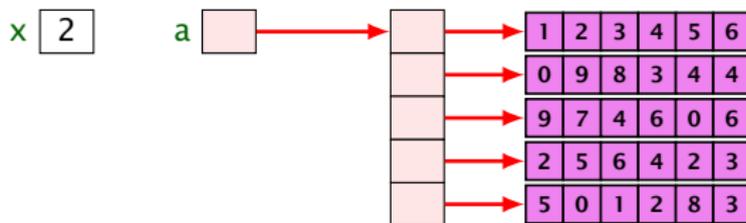
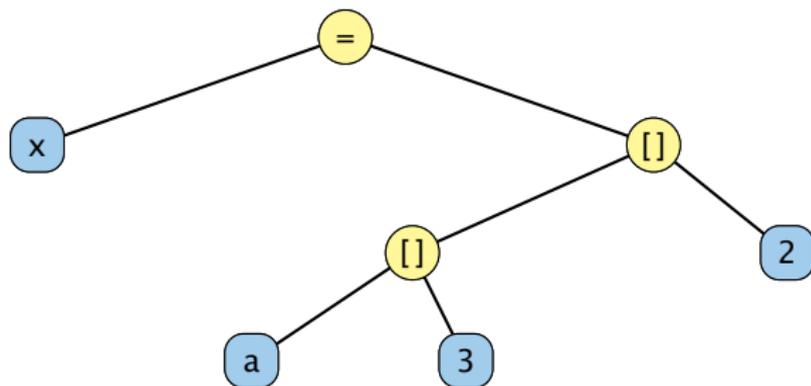
Beispiel: $x = a[3][2]$



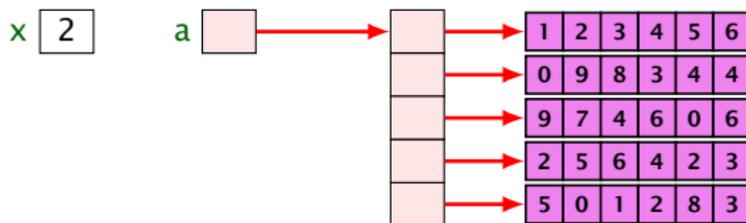
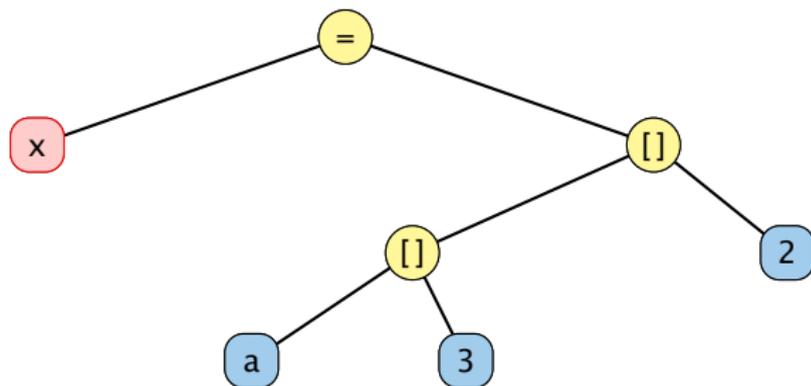
Beispiel: $x = a[3][2]$



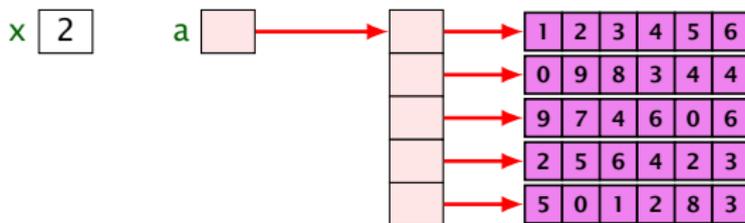
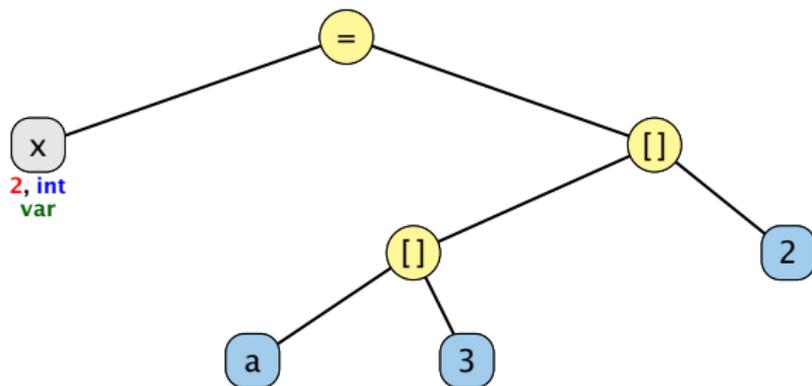
Beispiel: $x = a[3][2]$



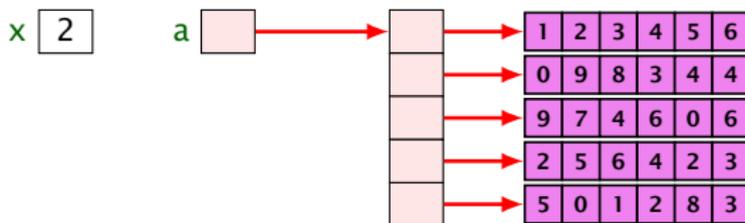
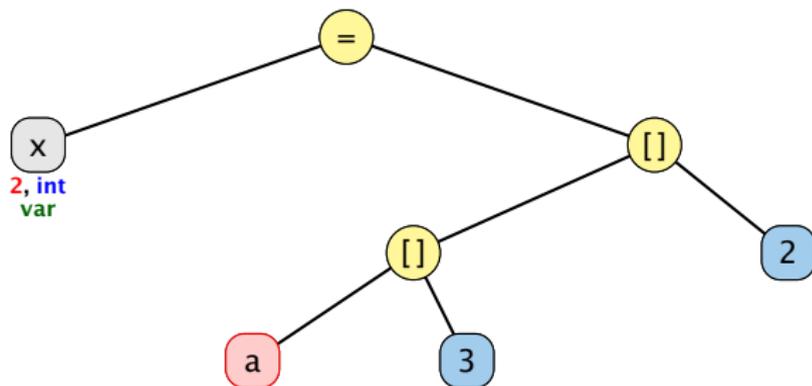
Beispiel: $x = a[3][2]$



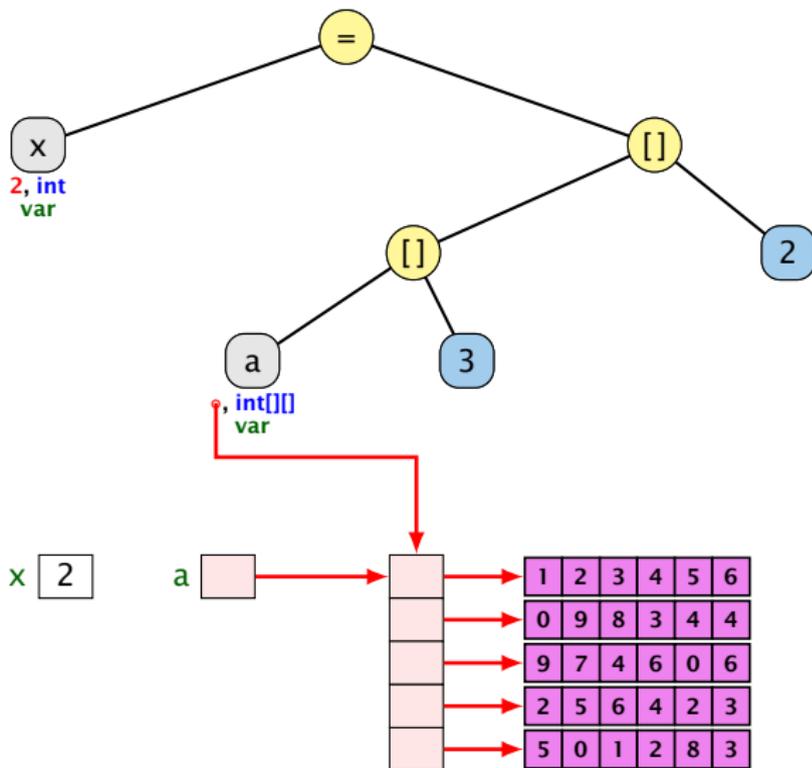
Beispiel: $x = a[3][2]$



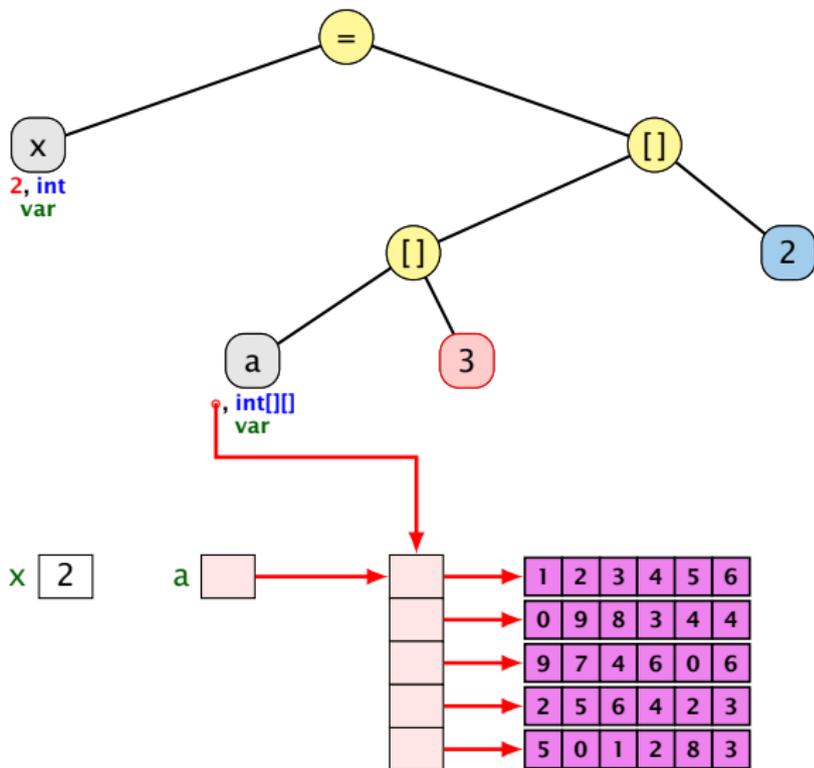
Beispiel: $x = a[3][2]$



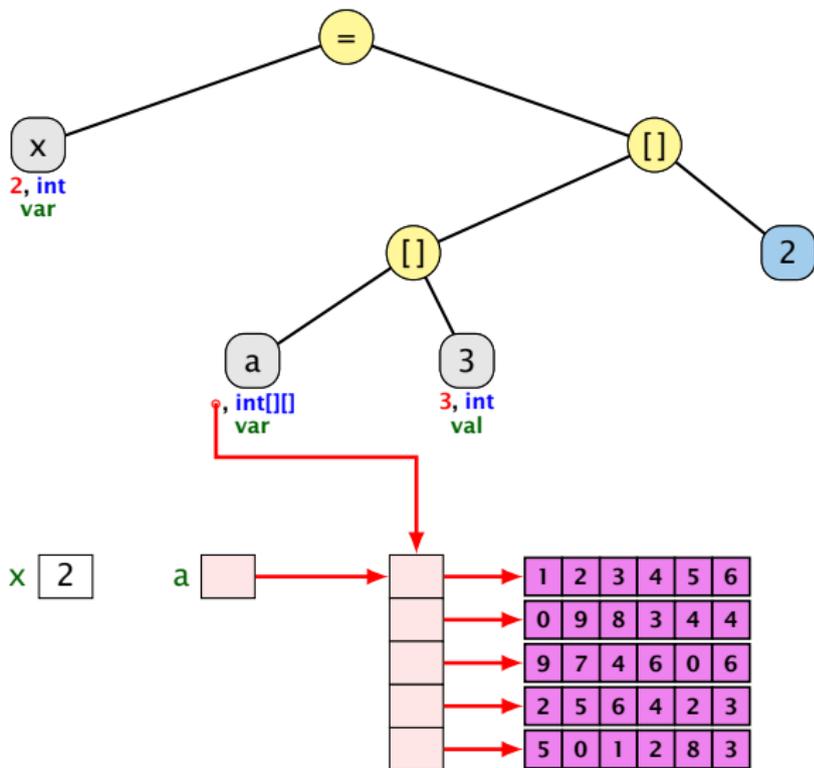
Beispiel: $x = a[3][2]$



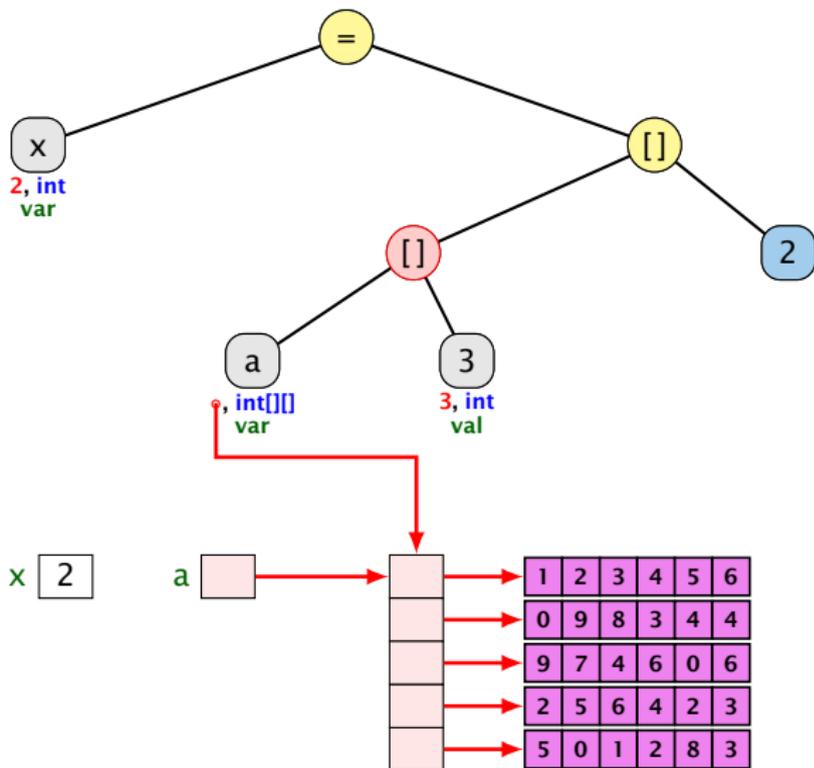
Beispiel: $x = a[3][2]$



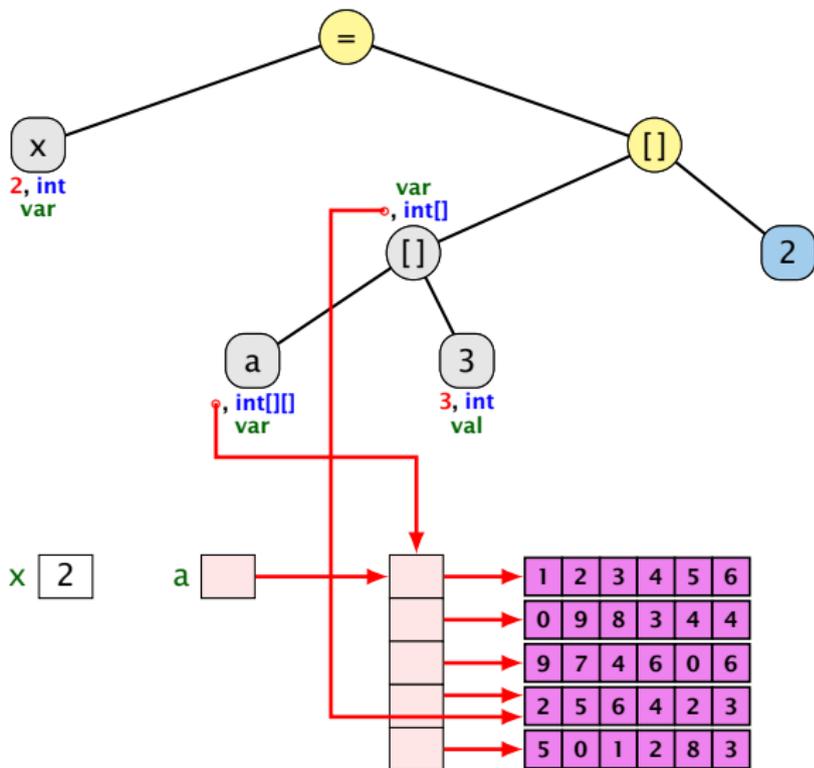
Beispiel: $x = a[3][2]$



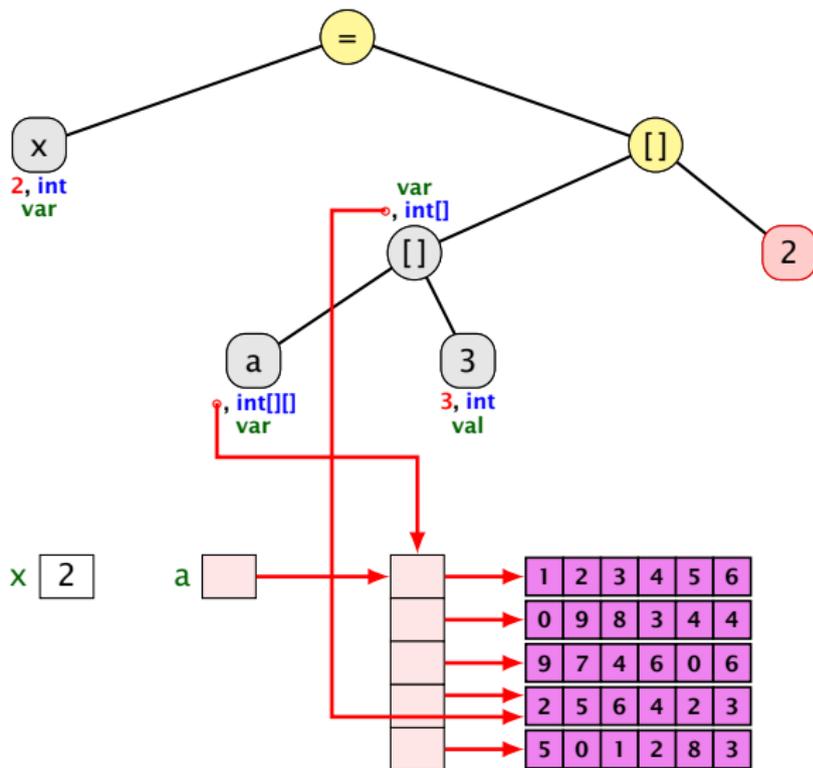
Beispiel: $x = a[3][2]$



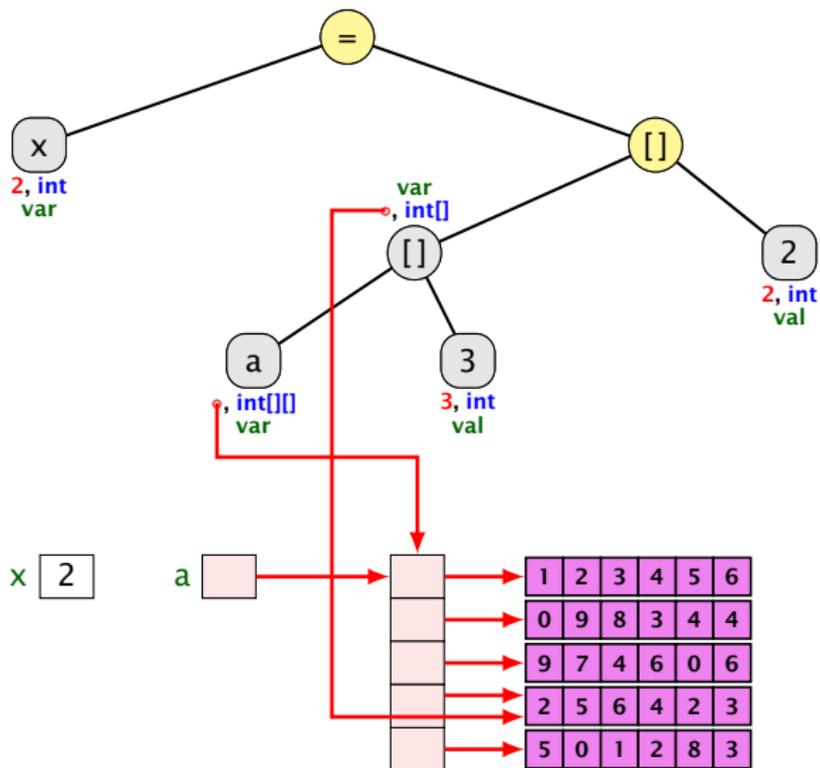
Beispiel: $x = a[3][2]$



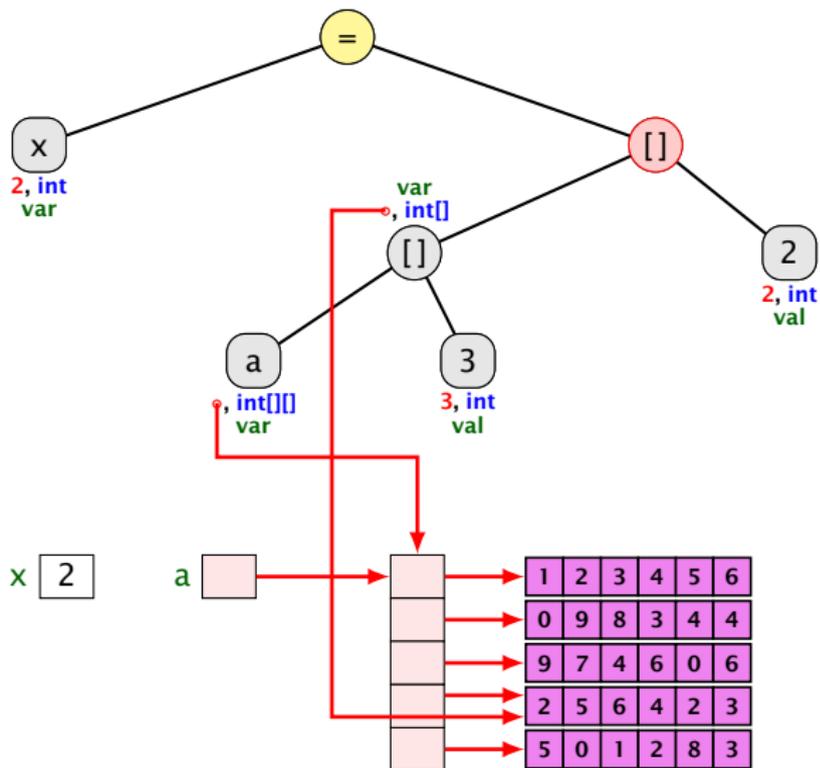
Beispiel: $x = a[3][2]$



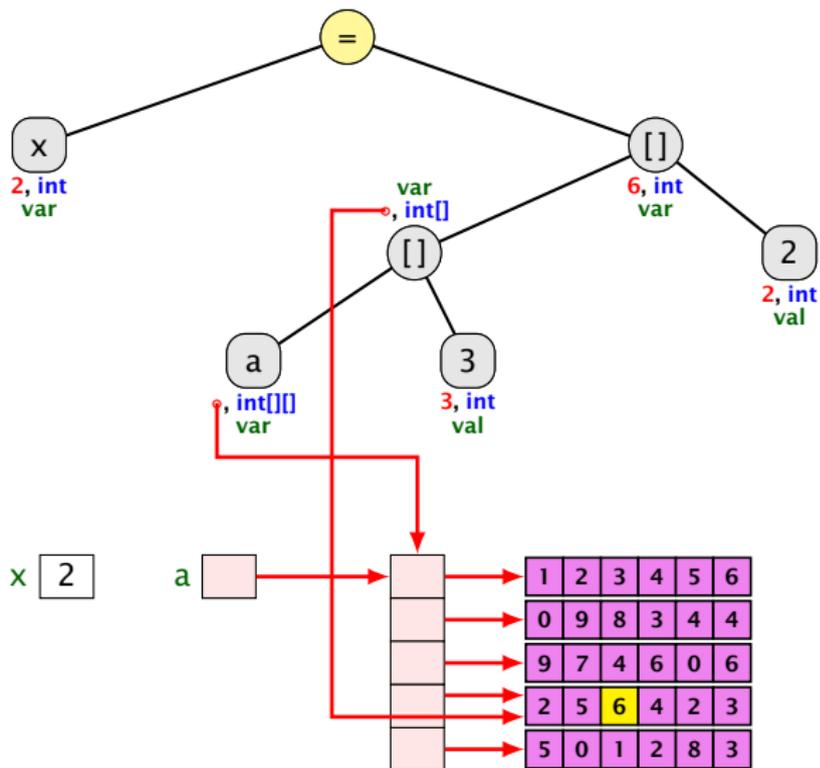
Beispiel: $x = a[3][2]$



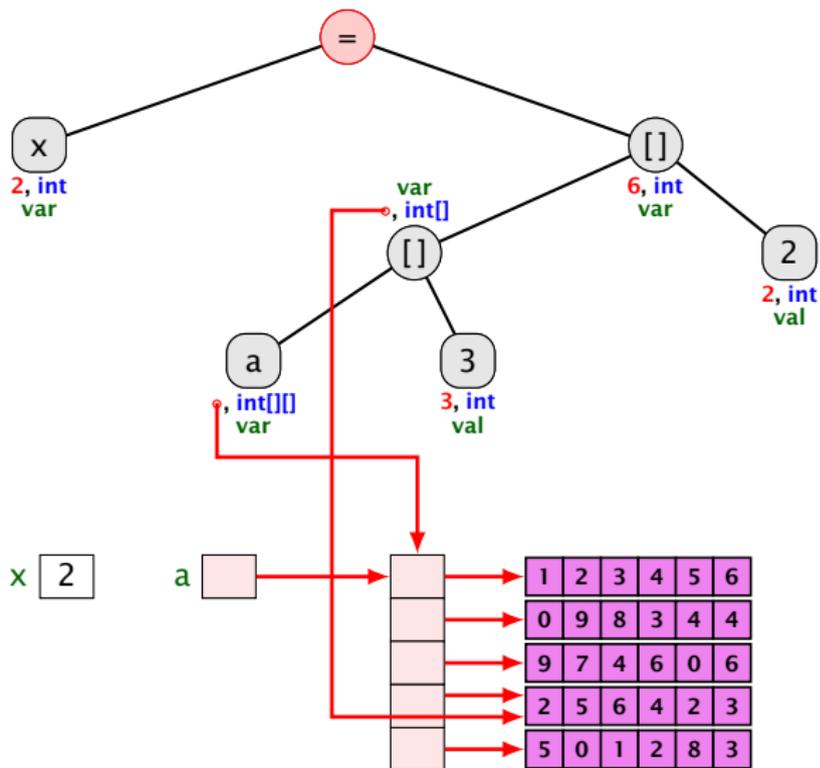
Beispiel: $x = a[3][2]$



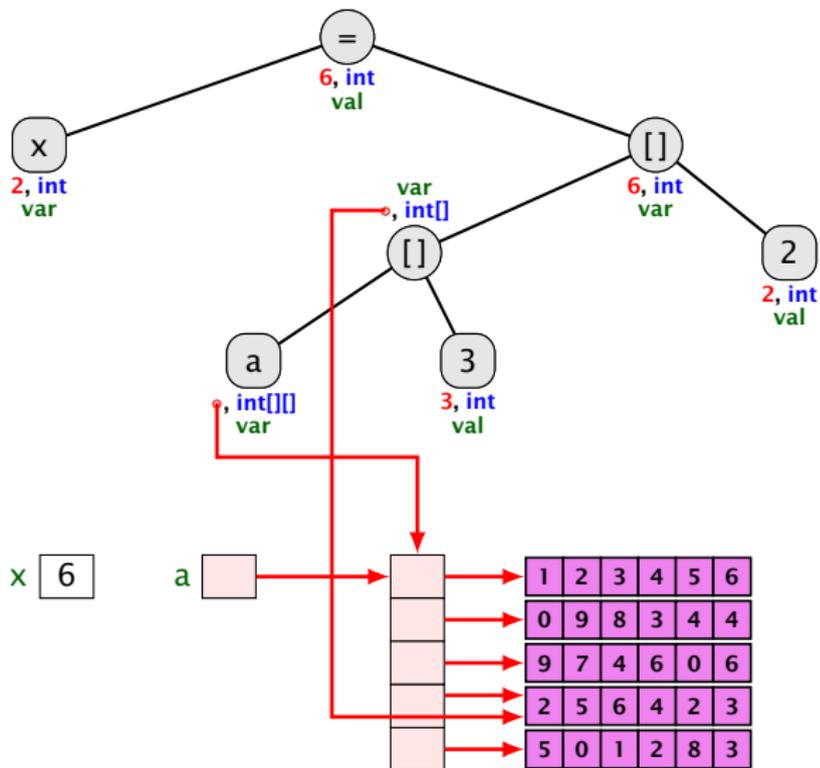
Beispiel: $x = a[3][2]$



Beispiel: $x = a[3][2]$



Beispiel: $x = a[3][2]$



Der .-Operator

<i>symbol</i>	<i>name</i>	<i>types</i>	<i>L/R</i>	<i>level</i>
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- ▶ `x = new int[2][4].length`
x hat dann den Wert 2.

Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:

```
new int [ 2 ] [ 4 ] . length
```

Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:



new int [2] [4] . length

Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:



Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:



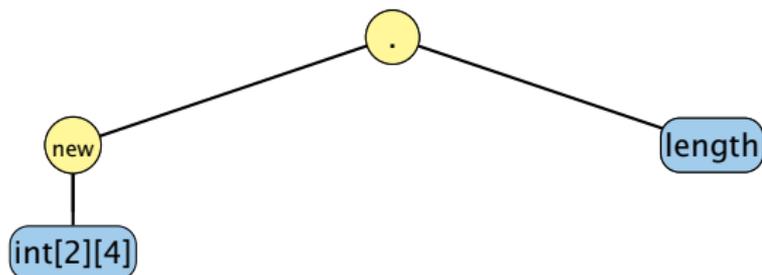
Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:



Beispiel: `new int[2][4].length`

Das Parsing für den `new`-Operator passt nicht in das Schema:



Arrayinitialisierung

1. `int[] a = new int[3];`
`a[0] = 1; a[1] = 2; a[2] = 3;`
2. `int[] a = new int[]{ 1, 2, 3};`
3. `int[] a = new int[3]{ 1, 2, 3};`
4. `int[] a = { 1, 2, 3};`
5. `char[][] b = { { 'a', 'b' }, new char[3], {} };`
6. `char[][] b;`
`b = new char[][]{ { 'a', 'b' }, new char[3], {} };`
7. `char[][] b;`
`b = { { 'a', 'b' }, new char[3], {} };`

5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- ▶ Initialisierung des Laufindex;
- ▶ `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- ▶ Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel

```
1 int result = a[0];
2 int i = 1;      // Initialisierung
3 while (i < a.length) {
4     if (a[i] < result)
5         result = a[i];
6     i = i + 1;  // Modifizierung
7 }
8 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
1 int result = a[0];
2 for (int i = 1; i < a.length; ++i)
3     if (a[i] < result)
4         result = a[i];
5 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
for (init; cond; modify) stmt
```

entspricht:

```
{ init; while (cond) { stmt modify; } }
```

Erläuterungen:

- ▶ `++i;` ist äquivalent zu `i = i + 1;`
- ▶ die `while`-Schleife steht innerhalb eines **Blocks** (`{...}`)

die Variable `i` ist außerhalb dieses Blocks nicht sichtbar/zugreifbar

5.6 Funktionen und Prozeduren

Oft möchte man:

- ▶ Teilprobleme **separat** lösen; und dann
- ▶ die Lösung **mehrfach** verwenden.

Beispiel

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Beispiel

Typ des Rückgabewertes

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Beispiel

Funktionsname

Typ des Rückgabewertes

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Beispiel

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Beispiel

Funktionsname

Typ des Rückgabewertes

Liste der formalen Parameter

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Funktionsrumpf

Beispiel

Funktionsname

Typ des Rückgabewertes

Liste der formalen Parameter

```
public static int[] readArray(int number) {  
    // number = Anzahl zu lesender Elemente  
    int[] result = new int[number]; // Feld anlegen  
    for (int i = 0; i < number; ++i) {  
        result[i] = read();  
    }  
    return result;  
}
```

Einlesen eines Feldes

Rückgabe des Ergebnisses + Beenden der Funktion

Funktionsrumpf

5.6 Funktionen und Prozeduren

Erläuterungen:

- ▶ Die erste Zeile ist der **Header** der Funktion.
- ▶ **public** und **static** kommen später
- ▶ **int[]** gibt den Typ des Rückgabe-Werts an.
- ▶ **readArray** ist der Name, mit dem die Funktion aufgerufen wird.
- ▶ Dann folgt (in runden Klammern und komma-separiert) die Liste der **formalen Parameter**, hier: **(int number)**.
- ▶ Der Rumpf der Funktion steht in geschweiften Klammern.
- ▶ **return expr;** beendet die Ausführung der Funktion und liefert den Wert von **expr** zurück.

5.6 Funktionen und Prozeduren

Erläuterungen:

- ▶ Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von '{' und '}', sind nur innerhalb dieses Blocks **sichtbar** d.h. benutzbar.
- ▶ Der Rumpf einer Funktion ist ein Block. Dort deklarierte Variablen nennt man **lokale Variablen**.
- ▶ Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- ▶ Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert `7` (**aktueller Parameter**).

Beispiel

```
public static int min(int[] b) {  
    int result = b[0];  
    for (int i = 1; i < b.length; ++i) {  
        if (b[i] < result)  
            result = b[i];  
    }  
    return result;  
}
```

Bestimmung des Minimums

Beispiel

```
public class Min extends MiniJava {
    public static int[] readArray(int number) { ... }
    public static int min(int[] b) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main(String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

Programm zur Minimumsberechnung

Beispiel

Erläuterungen:

- ▶ Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen **Prozeduren**.
- ▶ Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- ▶ In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {  
    public static void main (String[] args) {  
        write(args[0]+args[1]);  
    }  
} // end of class Test
```

Beispiel

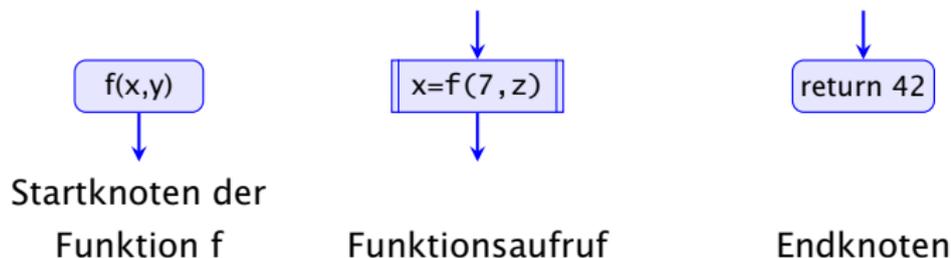
Der Aufruf

```
java Test "He1" "1o Wor1d!"
```

liefert: He1lo Wor1d!

5.6 Funktionen und Prozeduren

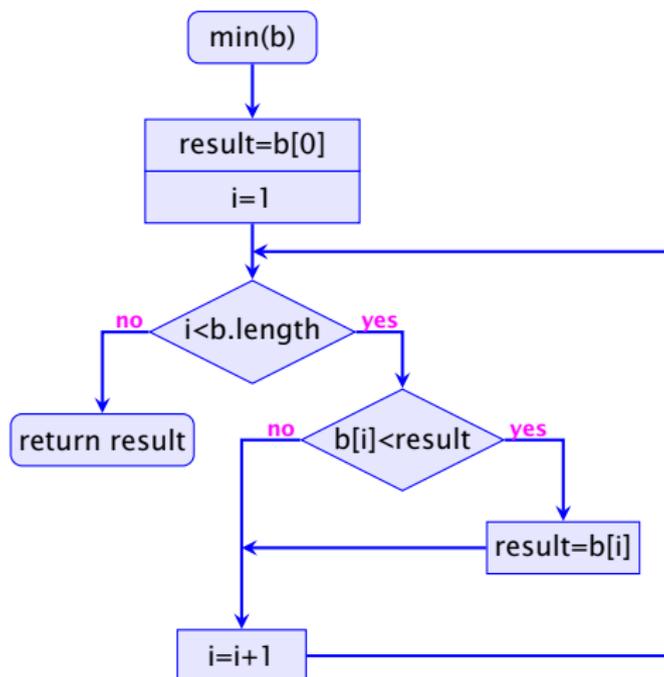
Um die Arbeitsweise von Funktionen zu veranschaulichen erweitern/modifizieren wir die Kontrollflussdiagramme



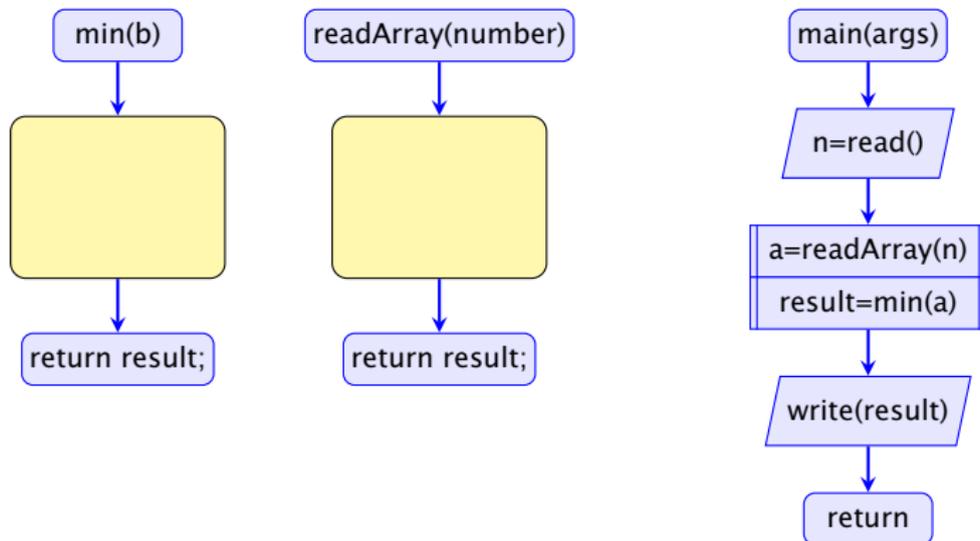
- ▶ Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- ▶ Ein Aufrufknoten repräsentiert eine Teilberechnung der aufgerufenen Funktion.

5.6 Funktionen und Prozeduren

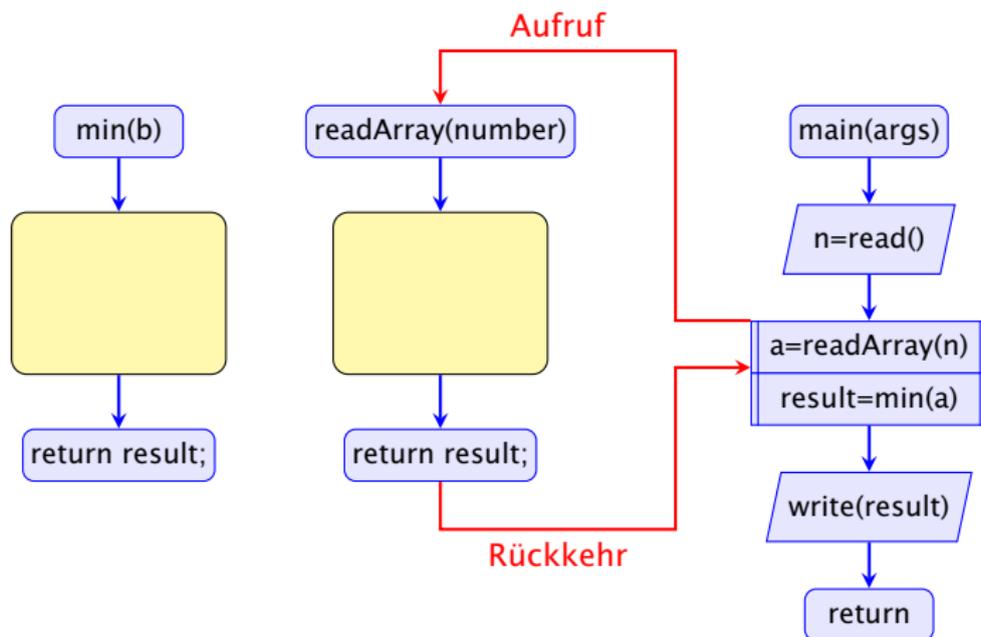
Teildiagramm der Funktion `min()`:



5.6 Funktionen und Prozeduren



5.6 Funktionen und Prozeduren



5.6 Funktionen und Prozeduren

