Ausdrücke

Variable, Typ, Kontext,
Deklaration, Initialisierung,
Ausdruck, Syntax, Semantik,
Seiteneffekt

Variablen als Stellvertreter

- In der Mathematik
  - Variable ist Stellvertreter eines Wertes
  - ändert sich nicht
  - repräsentiert überall den gleichen Wert
  - natürlich nur innerhalb eines Kontextes

Satz: Für alle n $\in$ N gilt $1 + 2 + \ldots + n = n(n+1)/2$

Beweis: Sei x $\in$ N beliebig.
1. Fall: x=0. Dann ist die Formel trivial wahr.
2. Fall: $x \neq 0$.
   Sei $k = x-1$ und die Formel wahr für k,
   also $1+2+\ldots+k = k(k+1)/2$.
   Dann gilt $1+2+\ldots+x = 1+2+\ldots+k+(k+1)$
   $= k(k+1)/2 + (k+1)$
   $= (k+1)(k+2)/2$
   $= x(x+1)/2$
3. Per Induktion ist die Formel wahr für alle n $\in$ N.

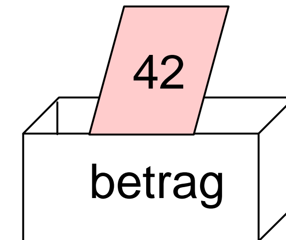
Q.E.D.

Kontext für k

Kontext für x

©H. Peter Gumm, Philipps-Universität Marburg

Variablen als Behälter



Name: betrag
Wert: 42

- In der Informatik
  - Variablen dienen als temporäre *Behälter* für Werte
  - Namen weitgehend frei wählbar
  - Beispiele: **`zins`**, **`betrag`**, **`xPos`**, **`zeichen`**, **`x1`**, **`r2d2`**, ...
- Lexikalische Regeln
  - Nur Buchstaben, Ziffern und einige Sonderzeichen zulässig
  - Schlüsselworte sind zu vermeiden
    - **`class, public, static, while, do, if, int, ...`**
    - Erstes Zeichen darf **keine** Ziffer sein
    - 0achtFuffzehn, 1A, ...
  - Es wird zwischen Groß- und Kleinschreibung unterschieden
- Konvention (in Java üblich)
  - Variablennamen beginnen mit Kleinbuchstaben
    - **`zinsSatz, buchstabe, diplomArbeit`**
  - Zusammengesetzte Namen werden durch Großbuchstaben zerlegt:
    - **`monatsGehalt, untereGrenze, blockEndeMarke, …`**
  - Nicht üblich:
    - **`Monats_gehalt; untere_Grenze; blockendemarke, …`**

©H. Peter Gumm, Philipps-Universität Marburg

Aufgaben von Variablen

- Man kann
  - Eine neue Variable einführen („*deklarieren*“)
    - Platz für zu speichernden Wert wird reserviert
    - Beispiele:
      - `int kontoStand;`
      - `float abstand ;`

  - Einen neuen Wert „in der Variablen“ speichern
    - Der alte Wert wird dabei überschrieben
    - Beispiele:
      - `kontoStand = 235 ;`
      - `abstand = 3.71 ;`

  - Den „in der Variablen“ gespeicherten Wert lesen
    - Implizit, bei der Auswertung von Ausdrücken, in denen Variablen vorkommen
    - Beispiele:
      - `kontoStand + 10`
      - `Math.sqrt(abstand*abstand + 12.5)`

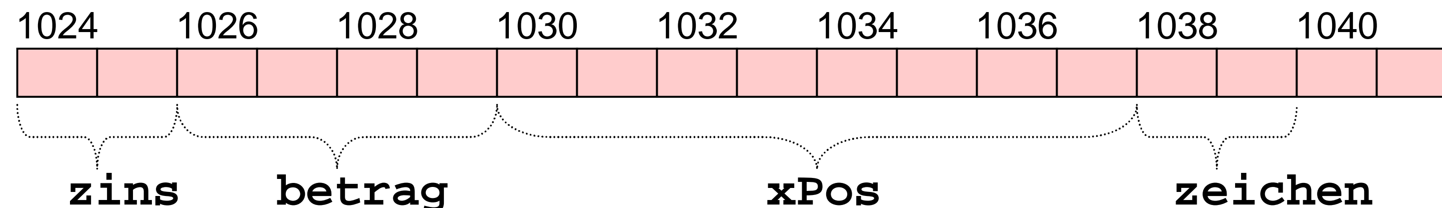
Variablen sind in vieler Weise wie Objektfelder, aber
- Sie erhalten keinen Default-Wert
- Bevor eine Variable gelesen wird, muss ihr ein Wert zugewiesen sein
- ... der Compiler überprüft dies
  `„variable x might not have been initialized“`

©H. Peter Gumm, Philipps-Universität Marburg

Typ = Wertebereich

- Jede Variable hat einen Typ
  - Der Typ von Werten der in ihr gespeichert werden darf
- Gründe
  - Variablen verschiedener Typen benötigen unterschiedlich viel Speicherplatz
  - Ungewollte Programmierfehler werden erkannt
    - `age = thisYear*3.14`
- Variablen müssen vor der Benutzung deklariert werden
  - Dabei wird Speicherplatz *reserviert*
- Beispiel:

```
short zins;       // 2 Byte
int betrag;       // 4 Byte
double xPos;      // 8 Byte
char zeichen;     // 2 Byte
```



©H. Peter Gumm, Philipps-Universität Marburg

Deklaration

- Typname Variablenname ;
- Beispiele
  - `int x ;`
  - `double y, z, eps ;`
  - `boolean ready;`
  - `String name, vorname;`
  - `Konto meins, deins, bills;`
- Formal wie Attribute aber …
  - … werden nicht automatisch initialisiert
    - ```
      { int x;
        x = x+1;
      ```
      Fehler !!

©H. Peter Gumm, Philipps-Universität Marburg
      ```

Deklaration mit Initialisierung

- Typname Variablenname = Wert ;
- Beispiele

```
int     x = 5;
double  y = 3.1, z = 12, eps = 0.001 ;
boolean ready    = true ;
String  name     = "Meier",
        vorname  = "Otto" ;
Konto   meins    = new Konto();
```

- Wert muss zum Typ passen

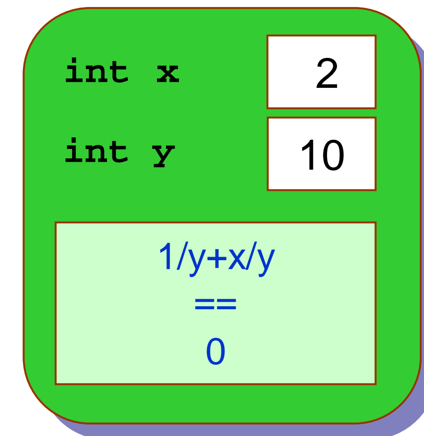
`int x = 2.0 ;`	falsch:	2.0 ist *double*
`float y = 2 ;`	ok	
`int c = 'a' ;`	ok :	char wird zu int konvertiert
`Konto meins = 0 ;`	falsch:	0 ist ein int und kein Konto ist *double*
`String s = null ;`	ok :	die Null-Referenz
`String s = "" ;`	ok :	der leere String. `""` ≠ `null`

©H. Peter Gumm, Philipps-Universität Marburg

Kontext

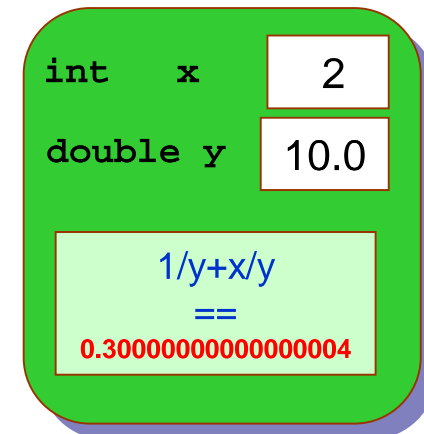
- Ein *Kontext* (Umgebung) wird durch eine aktuelle Belegung von Variablen gegeben
  - Für jede Variable benötigt man
    - Typ
    - Wert

  Beispiele:
  - Kontext_1:
    ```
    int x = 2 ;
    int y = 10 ;
    ```
  - Kontext_2:
    ```
    int    x = 2 ;
    double y = 10.0 ;
    ```



- *Typ* und *Wert* eines Ausdrucks mit Variablen vom gegenwärtigen Kontext abhängig

  Beispielausdruck:
  ```
  1/y + x/y
  ```
  - In Kontext_1: 0 (int)
  - In Kontext_2: `0.30000000000000004 (double)`



©H. Peter Gumm, Philipps-Universität Marburg

Kontexte in Java

- Klammerpaare **{ … }** begrenzen Kontexte
  - Variablen nur im Kontext gültig
    - von der Deklarationsstelle bis zum Ende des Kontextes
  - geschachtelte Kontexte gehören dazu
- im Inneren eines Kontext
  - jede äußere Variable sichtbar
- von außen
  - innere Variable unsichtbar
- Variablen dürfen im geschachtelten Kontext nicht neu deklariert werden
  - anders als in Pascal/Delphi

```
void test(){
  {
    int x = 2;
    x = 5;
  }
  x = 3;
}
```

x

cannot find symbol - variable x

gespeichert

```
void test(){
  {
    int x = 2;
    {
      int y = 3;
      x = x + y;
    }
    x = 3;
    x = x + y;
  }
```

x

y

cannot find symbol - variable y

gespeichert

©H. Peter Gumm, Philipps-Universität Marburg

Ausdrücke

- Syntax
  - Ein *Ausdruck* ist eine korrekt gebildete Formel aus
    - Konstanten,
    - Variablen,
    - Operationszeichen
    - Klammern
  - Beispiel: y/3 + x
- Semantik
  - In einem Kontext, der alle Variablen des Ausdrucks enthält, bezeichnet ein Ausdruck einen eindeutigen
    - Wert eines eindeutigen Typs

`int`	`x`	3
`float`	`y`	6.9

y/3 + x

Wert : 5.3
Typ : float

`int`	`x`	3
`int`	`y`	7

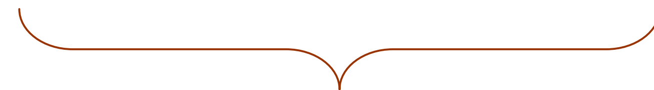
y/3 + x

Wert : 5
Typ : int

©H. Peter Gumm, Philipps-Universität Marburg

int-Ausdrücke

```
int test()
   {
       int betrag   = 123,
           zinssatz = 456 ;
       return -betrag+(betrag*zinssatz)/100 ;
   }
```



int-Ausdruck

- Rekursive Definition
  - int-Ausdrücke sind :

(1) jedes int-Literal

(2) jede *Variable* vom Typ int

(3) Sind E_1, E_2 int -Ausdrücke, dann auch

$E_1 + E_2$, $E_1 - E_2$,

$E_1 * E_2$, E_1 / E_2, $E_1 \% E_2$,

$+ E_2$, $- E_2$

(4) Ist E ein int -Ausdruck, dann auch (E)

©H. Peter Gumm, Philipps-Universität Marburg

Boolesche Ausdrücke

- Rekursive Definition
  - boolesche Ausdrücke sind

1. true, false,
2. jede Variable vom Typ boolean,
3. sind E_1, E_2 int -Ausdrücke, dann sind

   $E_1 == E_2$, $E_1 != E_2$,

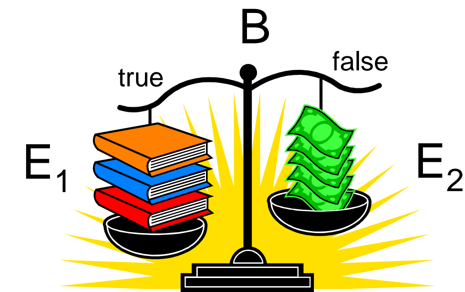
   $E_1 < E_2$, $E_1 <= E_2$, $E_1 > E_2$, $E_1 >= E_2$

   boolesche Ausdrücke,
4. sind B_1, B_2 boolesche Ausdrücke, dann auch

   $! B_1$, $B_1 || B_2$, $B_1 \&\& B_2$ und
5. ist B ein boolescher Ausdruck, dann auch (E)

©H. Peter Gumm, Philipps-Universität Marburg

Bedingte Ausdrücke



- *Falls* B *dann* E_1 *sonst* E_2 :

  B **?** E_1 **:** E_2

- Vorraussetzung:
  - B ein boolescher Ausdruck.
  - E_1 , E_2 Ausdrücke vom gleichen Typ T

- Wert
  - hat Typ T
  - Wert(E_1), falls Wert(B) == true; Wert(E_2), falls Wert(B)=false.

- Beispiele:
  - (2==3) ? 17 : 18 ergibt: 18
  - (x<y) ? y : x ergibt: maximum{x,y}
  - (x <0) ? -x : x ergibt: abs(x)

©H. Peter Gumm, Philipps-Universität Marburg

Semantik boolescher Ausdrücke

- Boolesche Verknüpfungen werden durch Wahrheitstabellen definiert:

∨	false	true
false	false	true
true	true	true

∧	false	true
false	false	false
true	false	true

¬	
false	true
true	false

- boolesche Ausdrücke durch rekursive Definition.

  E_1 und E_2 seien boolesche Ausdrücke,

- Wert(false) := false, Wert(true) := true

  Wert(E_1 || E_2) := Wert(E_1) ∨ Wert(E_2)

  Wert(E_1 && E_2) := Wert(E_1) ∧ Wert(E_2)

  Wert(! E_1) := ¬ Wert(E_1)

  Wert einer Variablen ist gespeicherter Wert

Um den (Wahrheits)Wert eines Ausdrucks der Form E_1|| E_2 zu finden

- berechne den Wert von E_1
- berechne den Wert von E_2

verknüpfe die Ergebnisse anhand der Tabelle für ∨

©H. Peter Gumm, Philipps-Universität Marburg

Semantik kurz evaluierter boolescher Ausdrücke

- E_1 und E_2 seien boolesche Ausdrücke,

$$\text{Wert}(E_1 \,||\, E_2) := (\,\text{Wert}(E_1)\,)\ ?\ \text{true}\ :\ \text{Wert}(E_2)$$

$$\text{Wert}(E_1 \,\&\&\, E_2) := (!\,\text{Wert}(E_1))\ ?\ \text{false}\ :\ \text{Wert}(E_2)$$

```
2 != 1  |  "Russel" == "Papst"
true  (boolean)

2 != 1  |  1/0 == 17
Error: not a statement

2 != 1  ||  1/0 == 17
true  (boolean)
```

```
1 == 2  &  1+1 == 2
false  (boolean)

1 == 2  &  1/0 == 17
Error: not a statement

1 == 2  &&  1/0 == 17
false  (boolean)
```

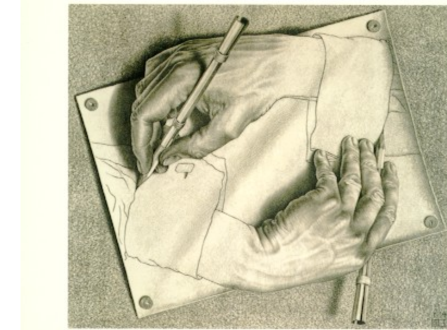
Wie erwartet

Zweiter Ausdruck hätte nicht ausgewertet werden müssen

Kurze Auswertung

©H. Peter Gumm, Philipps-Universität Marburg

Syntaxprüfung: Ausdruck legal ?

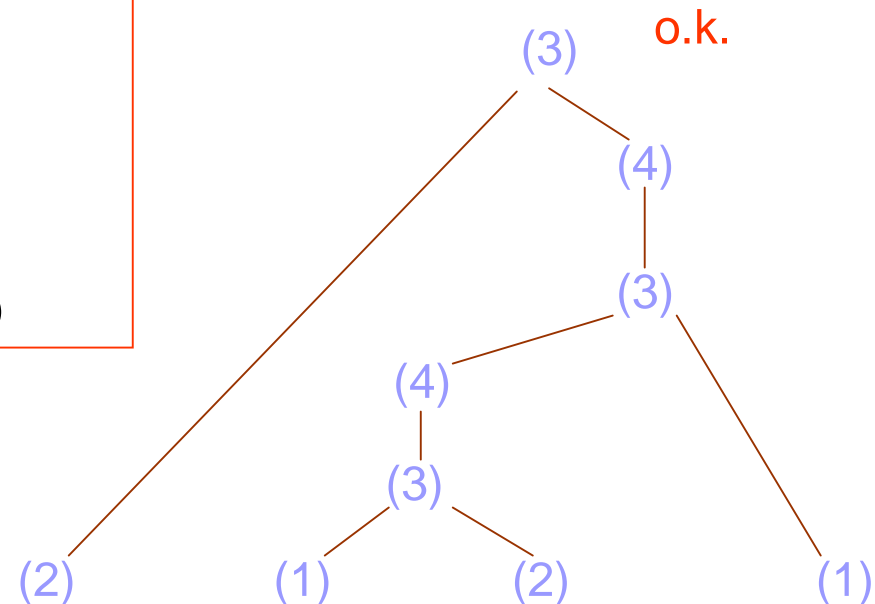


- Rekursive Definition
  1. jedes int-Literal
  2. jede *Variable* vom Typ int
  3. Sind E1, E2 int -Ausdrücke, dann auch

     E1+ E2 , E1- E2,

     E1* E2, E1/ E2, E1% E2,

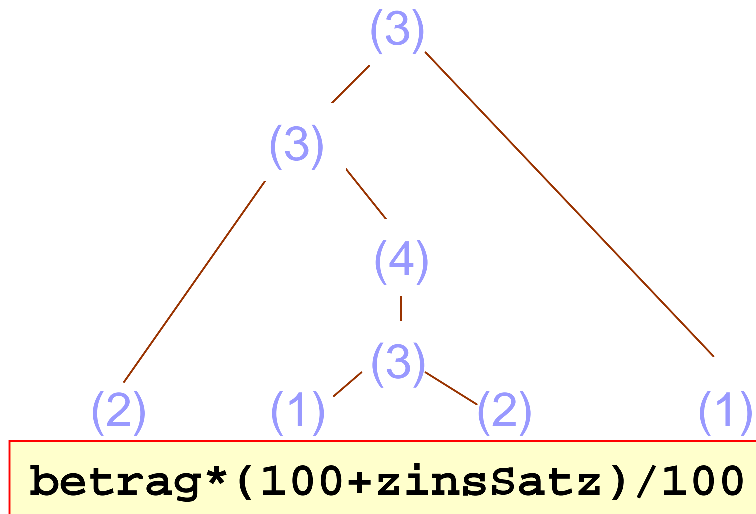
     + E2 , - E2
  4. Ist E ein int -Ausdruck, dann auch (E)

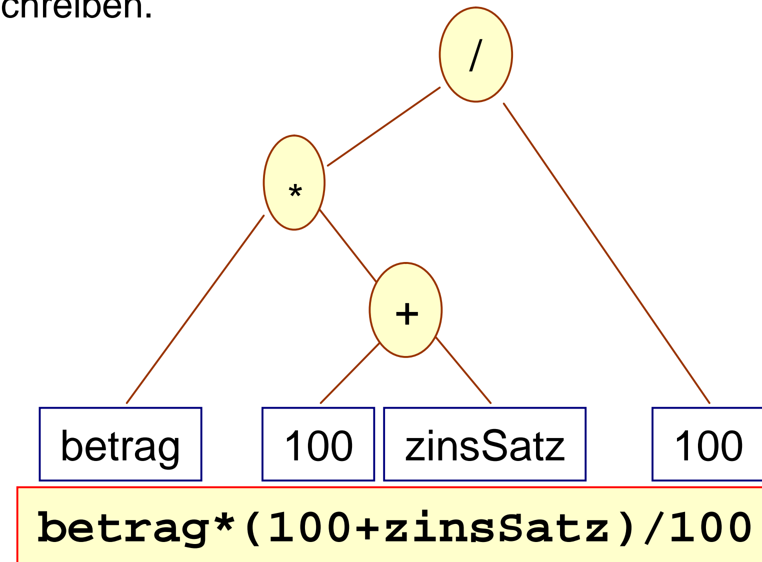


©H. Peter Gumm, Philipps-Universität Marburg

Syntaxbaum

- Korrektheitsanalyse liefert eine Hierarchie, einen „*Baum*“
  - Regelnummern sind Verzweigungspunkte (Knoten)
  - Bestandteile sind Unterbäume
  - Lienien verbinden Knoten mit Unterbäumen
- Ersetzt man die Knoten durch die entsprechende Operatoren, so erkennt man, wie der Ausdruck auszuwerten ist.
  - Berechne den Wert W_1 des linken Teilbaumes
  - Berechne den Wert W_2 des rechten Teilbaumes
  - Verknüpfe W_1 und W_2 mit dem Operator an der Wurzel
- Klammern fallen dabei weg. Sie sind ohnehin nur nötig, um einen Ausdruck 2-dimensional, d.h. in einer Zeile zu schreiben.

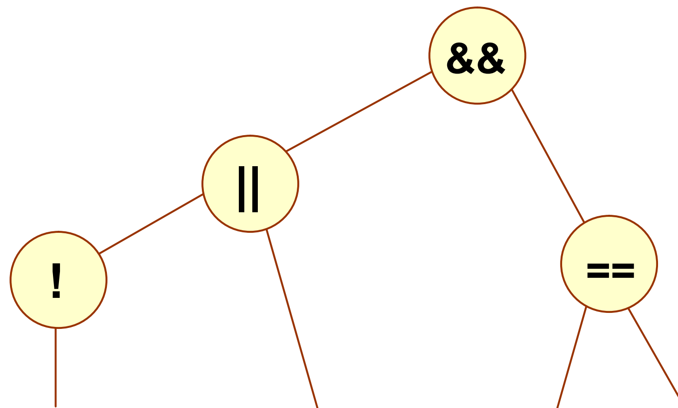


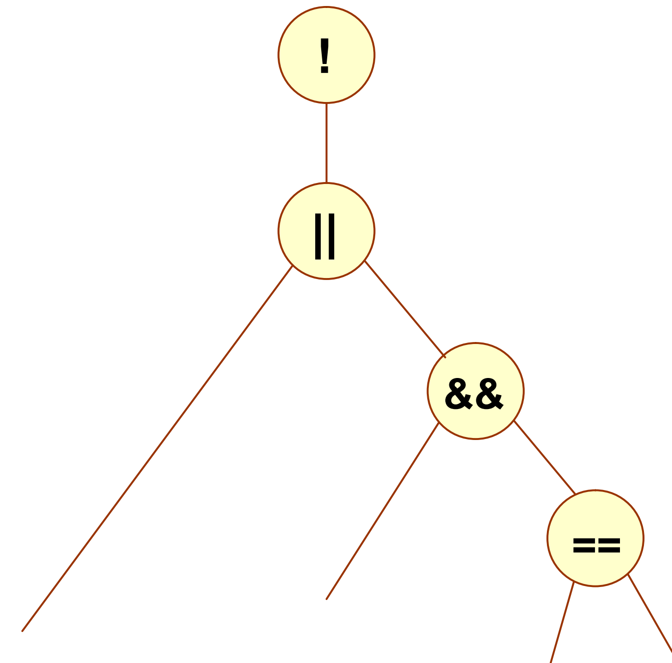


©H. Peter Gumm, Philipps-Universität Marburg

Präzedenzen

- Ausdrücke mit mehreren Operatoren sind mehrdeutig:
  - 2 – 3 – 4 = ?
  - 2 + 3 * 4 = ?
  - ! fertig || fast && 2 == 3 = ?
- Wüsste man, wie der Ausdruck gebildet wurde, so wäre die Sache klar:
  - Ansonsten muss man Klammern verwenden





`! fertig || fast && 2 == 3`

`! fertig || fast && 2 == 3`

Baum entspricht der Klammerung:
`((! fertig)|| fast) && (2 == 3)`

Baum entspricht der Klammerung:
`!(fertig || (fast && (2 ==3)))`

©H. Peter Gumm, Philipps-Universität Marburg

Präzedenzen

- Präzedenzen sind Regeln, um Klammern zu sparen
  - Aus der Schule: Punktrechnung vor Strichrechnung
    - Wir sagen: *, /, % haben Präzedenz über +,-
    - In der Booleschen Algebra: && hat Präzedenz über ||
  - Einstellige Operatoren haben meist Präzedenz über mehrstelligen
    - x && !y || z bedeutet (X && (!y)) || z
    - -x+y bedeutet (-x)+y
  - Für Operatoren gleicher Präzedenzstufe legen wir fest:
    - Linksklammerung (fast immer)
      - x – y – z bedeutet: (x-y)-z
      - x % y / z bedeutet: (x%y)/z
    - Rechtsklammerung bei „Zuweisungsausdrücken“:
      - x = y = x++ bedeutet : x = (y = x++)

Präzedenzen in Java

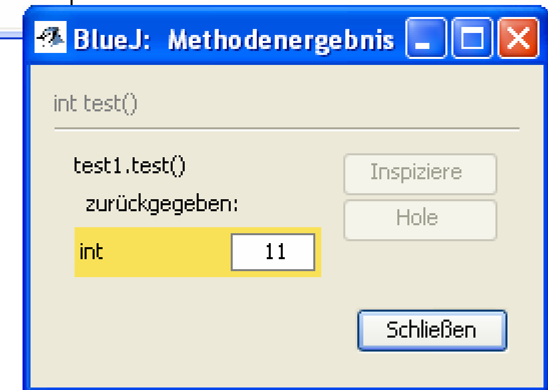
	(1)	postfix	[ ]	.	(params)	expr++	expr –	
hoch	(2)	unary	++expr	--expr	+expr	-expr	!	
	(3)	creation/cast	new	(type)expr				
	(4)	multiplicative	*	/	%			
	(5)	additive	+	-				
	(6)	shift	<<	>>	>>	>>		
	(7)	relational	<	>	>=	<=	instanceof	
	(8)	equality	==	!=				
	(9)	bitwise AND	&					
	(10)	bitwise exclusive OR	^					
	(11)	bitwise inclusive OR	|					
	(12)	logical AND	&&					
	(13)	logical OR	||					
	(14)	conditional	?:					
niedrig	(15)	assignment	=	+=	-=	*=	/=	%=
			>>=	<<=	>>>=	&=	^=	|=

©H. Peter Gumm, Philipps-Universität Marburg

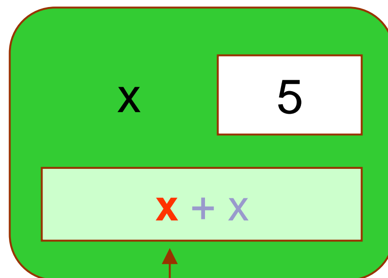
Pragmatik: Empfehlungen

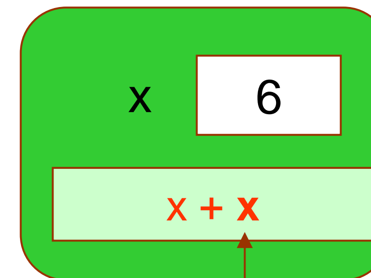
- Ein *Ausdruck* sollte einen *Wert* ausdrücken – sonst nichts
  - Insbesondere sollte die Auswertung keine Seiteneffekte haben
  - Dies bedeutet, dass wir auf einige „Tricks“ in Java bewusst verzichten
    - z.B.: x++, ++x, x--, --x, … als Teil komplexerer Ausdrücke wie 3*x++ , (x++ * y--)/10, …
  - Unkritisch:
    - Als isolierte Anweisung, z.B.: x++ ;
    - In Schleifen for(x=0; x <10; x++) tuwas();

Konstanz des Kontextes



- Mathematische Gleichungen sind nur in einem *festen Kontext* gültig.
  - Ändert sich während der Auswertung von x+x der Kontext,
    so kann eine einfache Gleichung falsch werden
    - x + x == 2*x ???





t_1: x wird gelesen.
Resultat: 5

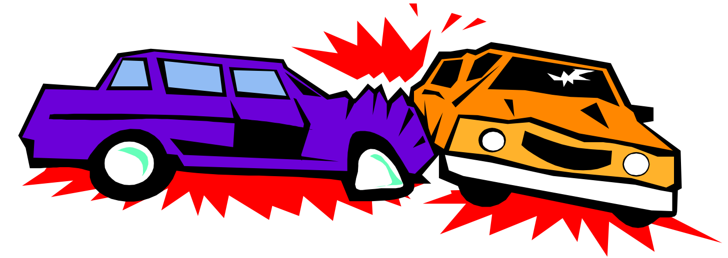
t_2: der Kontext wird verändert

t_3: zweites x wird gelesen
Resultat: 6

t_4: Addition wird ausgeführt:
Ergebnis: 11

©H. Peter Gumm, Philipps-Universität Marburg

Seiteneffekte



- Ausdrucksauswertung verändert Kontext
  - In „sauberen“ Sprachen verboten
    - Haskell, Prolog, Pascal, Oberon, …
  - In „schludrigen Sprachen“ üblich
    - C, C++, Java, …
- Konsequenzen: Mathematische Regeln verletzt:
    - x++ + x++ ≠ 2*(x++)
    - x++ + x ≠ x + x++
  - Ergebnis eines Ausdrucks hängt von Berechnungsreihenfolge ab

©H. Peter Gumm, Philipps-Universität Marburg