1.2 Typsystem (I)

- Einfache Typen, Ausdrücke, Variablenvereinbarungen -

Getypte Sprache *(strongly typed language, strong typing):*

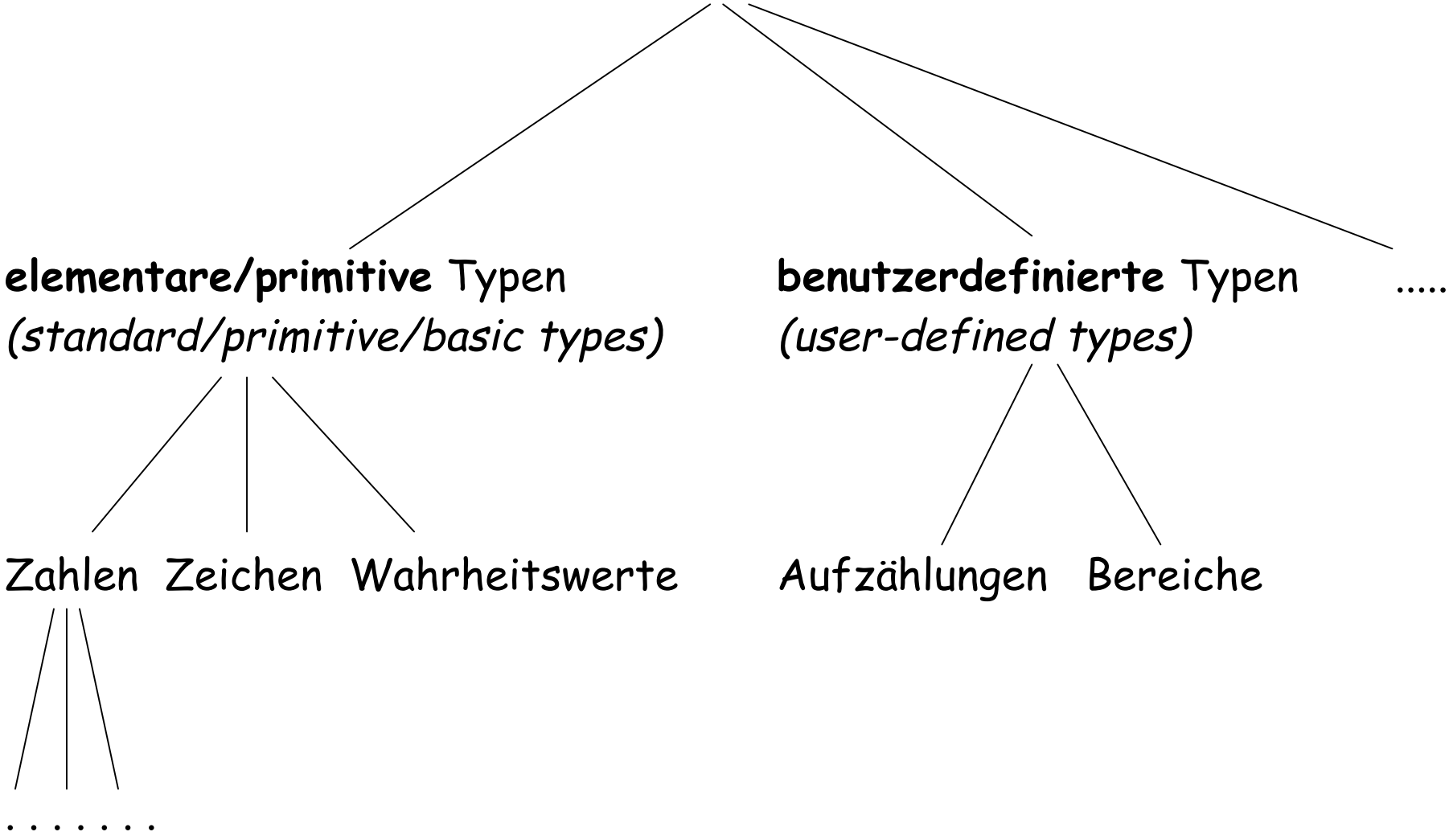
- Jeder Wertbezeichner hat einen Typ
- Jede Variable/Konstante hat einen Typ
- Jeder Ausdruck hat einen Typ

Zu jedem Typ gibt es passende **Operatoren**,
die auf den Werten des Typs definiert sind.

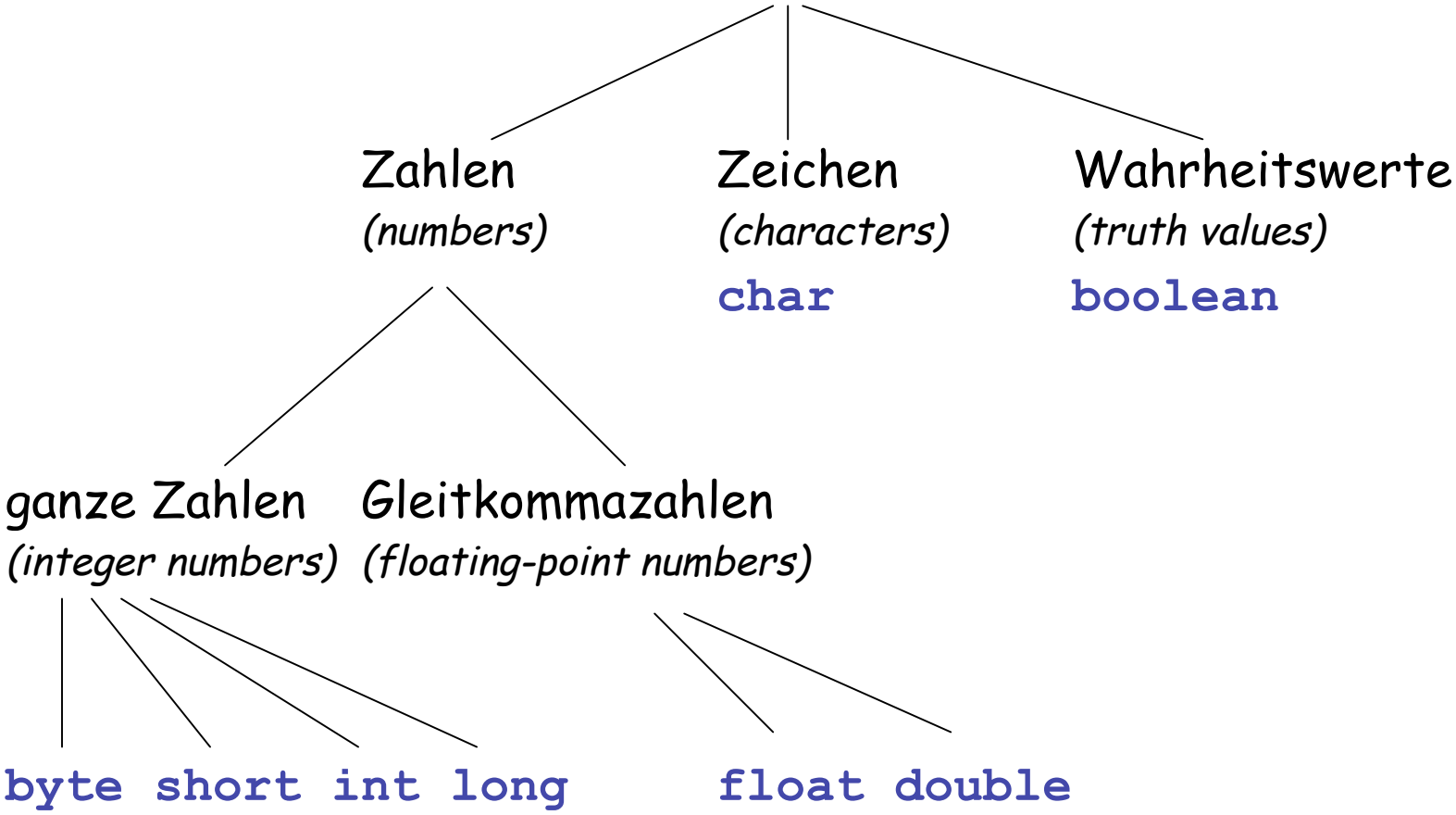
Statisch getypte Sprache *(static typing):*

Typkorrektheit - d.h. „passen die Operatoren zu den Operanden?" - wird - möglichst weitgehend - vom Übersetzer überprüft.

1.2.1 Einfache Typen



Primitive Typen z.B. in **Java:**



1.2.1.1 Universelle Operatoren

sind *allen* einfachen Typen gemeinsam:

❶ **Vergleichsoperatoren** *(comparison operators)*

= überprüft die Werte zweier Ausdrücke auf *Gleichheit*
= in Modula, ...
== in Java, ...

≠ überprüft die Werte zweier Ausdrücke auf *Ungleichheit*
oder <> in Modula, ...
!= in Java, ...

(beide sind *dyadische Operatoren*, d.h. mit zwei Operanden, auch *„binary operators"*)

Das Ergebnis ist Boole'sch.

❷ **Bedingungsoperator** *(conditional operator)*

WENN . DANN . SONST .

ist *triadischer (ternary)* Operator
mit Boole'schem 1. Operanden

WENN bedingung DANN wert1 SONST wert2

heißt **bedingter Ausdruck** *(conditional expression)*

`? :` in Java, ... z.B. `x<0 ? -x : x`

❸ Zuweisungsoperator *(assignment operator)*

Ist die **Zuweisung** eine Anweisung oder ein Ausdruck ?

Modula, ...: *Variable* := *Expression* ist einfache Anweisung

Java, ...: *Variable* = *Expression* ist zusammengesetzter Ausdruck -
Auswertung liefert den Wert des *Expression*
und hat den Zuweisungs-Effekt !

Variable = *Expression*; ist einfache Anweisung !

1.2.1.2 Java `boolean`

Werte: Wahrheitswerte, Boole'sche Werte *(Boolean values,* nach G. Boole, engl. Mathematiker, 19. Jhdt.)

Bezeichner: `true` `false`

Repräsentation: 1 Bit

Voreinstellung: `false`

Operatoren:
- `!` Negation (monadischer Präfix-Operator)
- `&` Konjunktion, `&=` mit Zuweisung an linken Operanden
- `|` Disjunktion, `|=` mit Zuweisung an linken Operanden
- `&&` Konjunktion, nichtstrikt im rechten Operanden
- `||` Disjunktion, nichtstrikt im rechten Operanden

  („McCarthy-And/Or", nach J. McCarthy, amerik. Informatiker)

1.2.1.3 Java `char`

Werte: *Unicode*-Zeichen *(characters)*
(Buchstaben, Ziffern, Sonderzeichen)

Bezeichner: `'a'` `'b'` `'c'` ...
`'\n'` `'\''` ...
`'\u0000'` ...

Repräsentation: 2 Bytes

Voreinstellung: `'\u0000'`

Operatoren: `++` `--` monadische *(unary)* Operatoren,
Präfix oder Postfix. Nur auf Variable anwendbar:
Nachfolger bzw. Vorgänger wird *zugewiesen;*
geliefert wird der neue bzw. alte Wert (Prä/Postfix).

1.2.1.3 Java `int`

Werte: ganze Zahlen *(integer numbers)* aus $[-2^{31}, 2^{31}-1]$

Bezeichner: Dezimalzahl, gegebenenfalls mit Vorzeichen
Oktalzahl, beginnend mit 0 (Null)
Sedezimalzahl, beginnend mit 0X (Null, großes X)

Repräsentation: 32 Bits (Zweierkomplement)

Voreinstellung: `0`

Operatoren:

`++ --`	Nachfolger bzw. Vorgänger wie bei `char`
`+ -`	monadisch (Präfix)
`+ - *`	dyadisch, *mit Ignorierung von Über/Unterlauf(!)*
`/`	mit Ignorierung des gebrochenen Anteils
`%`	Restbildung

Weitere Operatoren:

`&` `|` `^` bitweises Und, Oder, exklusives Oder

`+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=`
mit Zuweisung an linken Operanden

`<` `<=` `>` `>=` Vergleichsoperatoren,
liefern Boole'schen Wert

Weitere Ganzzahltypen:

`long` 64 Bits
(*Bezeichner:* angehängtes `L` oder `l`)

`short` 16 Bits

`byte` 8 Bits

mit gleichen Operatoren wie `int`

1.2.1.4 Java `float/double`

Werte: Gleitkommazahlen *(floating-point numbers)*,
d.i. Teilmenge der rationalen Zahlen

Bezeichner: Dezimalzahl, bei Bedarf mit Dezimalpunkt und/oder angehängtem `e`/`E` mit 10er-Exponent für `double`
(und zusätzlich angehängtem `f`/`F` für `float`),
z.B. `31.415926e-1` (π). Außerdem:

`POSITIVE_INFINITY`
`NEGATIVE_INFINITY`
`NaN` („not a number", „undefiniert")

Repräsentation: 32 Bits (`float`), 64 Bits (`double`) gemäß IEEE 754

Voreinstellung: `0.0` bzw. `0.0F`

Operatoren: wie bei ganzen Zahlen, aber ohne bitweise Operatoren

Achtung: Wegen der Rundungsfehler ist Gleitkomma-Arithmetik mit äußerster Vorsicht zu betreiben.

Abfragen auf Gleichheit oder Ungleichheit sind grundsätzlich fragwürdig! Z.B.

nach `pi = 3.1415926;`
gilt `(pi*47)/47 - pi == 0` ,
aber `pi + 7 - 7 - pi != 0` !!

Ferner droht Auslöschung kleiner durch große Summanden:

```
0.01f + 12345678 - 12345678 == 0.0
```

d.h. hier gilt **nicht das Assoziativgesetz** !

1.2.1.5 Aufzählungstypen

Aufzählungstyp *(enumeration type)* hat endlich viele Werte,
deren Bezeichner Namen sind und *vom Programmierer*
explizit angegeben werden.

Ein solcher Typ wird in einer **Typdefinition** vereinbart,
z.B. in Java (1.5) mit

```
enum Colour {red, green, blue };
```

Voreinstellung:

Operatoren: `==` `!=` (und weitere Manipulationsmöglichkeiten)

1.2.1.6 Bereichstypen

Bereichstyp *(subrange type)* umfasst einen zusammenhängenden Teilbereich der Werte eines primitivenTyps (typischerweise Ganzzahlen oder Zeichen), z.B.

(Modula)

```
TYPE digit = ["0".."9"];
```

[Solche Typen sind aus verschiedenen Gründen problematisch (Eingliederung ins Typsystem, Überprüfungen zur Laufzeit, ...) und sind daher aus der Mode gekommen.]

1.2.2 Ausdrücke

(Java, vereinfachte Syntax)

Elementare Ausdrücke:

Primary:

Literal

Identifier [Postfix]

[Prefix] Identifier

(Expression)

Allgemeine Ausdrücke:

Expression:

UnaryExpression

BinaryExpression

ConditionalExpression

AsssignmentExpression

UnaryExpression:

Primary

UnaryOperator UnaryExpression

BinaryExpression:

Expression BinaryOperator Expression

ConditionalExpression:

Expression ? *Expression* : *Expression*

AsssignmentExpression:

Variable = *Expression*

+ typbezogene *Kontextbedingung:*

Die Typen der Operanden müssen zu den Operatoren *passen* (s.u.)

Die **Auswertung** eines **Ausdrucks** liefert einen **Wert**,
dessen Typ sich aus den Typen seiner Bestandteile ergibt.

Regel 1 - Links-Rechts-Auswertung *(left-to-right evaluation):*
Bei dyadischen Operatoren wird erst der linke,
dann der rechte Operand ausgewertet.

Regel 2 - Wertübergabe *(eager evaluation):*
Jeder Operand wird vollständig ausgewertet,
bevor die Operation ausgeführt wird.
(Ausnahmen: nichtstrikte Operatoren wie `&&` `||` `? :`)

Regel 3 - Bindungsstärke (Operatorvorrang , *operator precedence)*
der Operatoren wird berücksichtigt,
ebenso Klammerung mit `( )` .

Anordnung der Operatoren nach *abfallenden* **Bindungsstärken:**

`++  --`	Inkrementierung/Dekrementierung
`+  -  .....`	weitere monadische Operatoren
`+  - *  /  %`	dyadische arithmetische Operatoren
`==  !=  <  <=  >  >=`	dyadische Vergleichsoperatoren
`&	^ && ||`	dyadische Boole'sche Operatoren
`?  :`	Bedingter Ausdruck
`=  +=  .......`	Zuweisungen

Rechts- oder Linksassoziativität ?

(Die Frage *ist relevant*, wie man z.B. auf S. 12 sehen konnte!)

Alle *dyadischen* Operatoren *außer der Zuweisung*
werden *linksassoziativ* angewendet:
z.B. `a + b + c` wird wie `(a + b) + c` berechnet.

Aber: `a = b = c = 0` ≡ `a = (b = (c = 0))`

`a ? b : c ? d : e` ≡ `a ? b : (c ? d : e)`

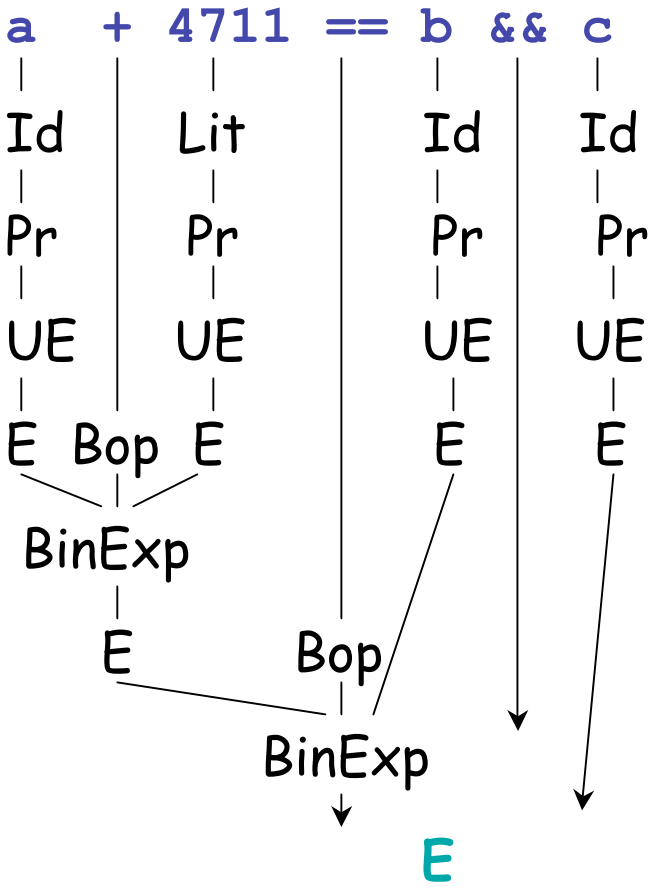
Syntaxanalyse des Übersetzers

prüft **syntaktische Korrektheit** eines Ausdrucks:

Versuch, einen gültigen **Syntaxbaum** zu finden

(unter Berücksichtigung von Syntax/Vorrang-Regeln)

Beispiel:



Semantische Analyse des Übersetzers

prüft **Typkorrektheit** eines syntaktisch korrekten Ausdrucks:
passen die Typen der Operanden zu den Operatoren?

Beispiel: `a + 4711 == b && c`

ist z.B. dann typkorrekt, wenn

1. `a` den Typ `int` hat,
2. `b` den Typ `int` hat,
3. `c` den Typ `boolean` hat.

! Diese Bedingungen können aber abgeschwächt werden: → 1.2.3

1.2.3 Typanpassung

(Typumwandlung, *type conversion)*

Beachte: `int1 = int2` ist typkorrekt

`long1 = int1` ist *auch* typkorrekt, denn `int` ist mit `long` **verträglich** *(compatible):*

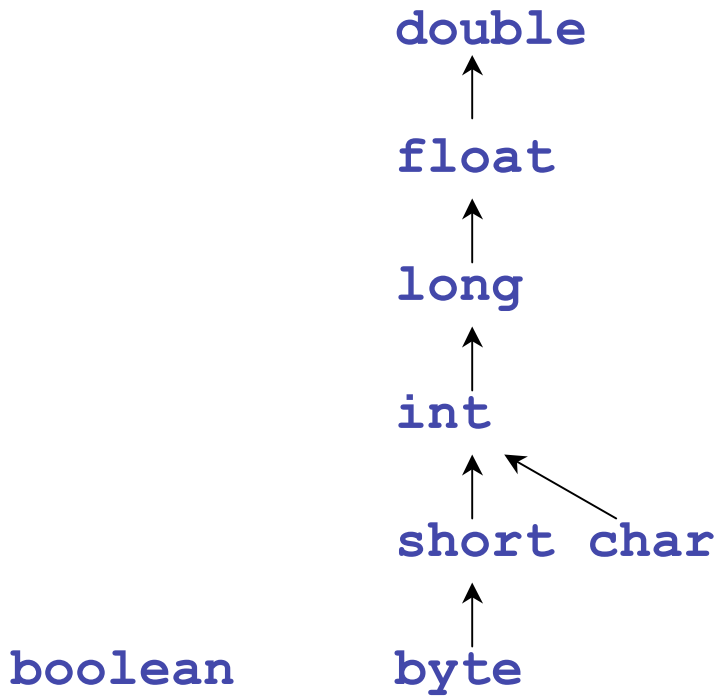
zur Laufzeit wird aus dem Wert von `int1` ein `long`-Wert gewonnen, und dieser wird an `long1` zugewiesen.

`int1 = long1` ist *nicht* typkorrekt, da das Gelingen der Umwandlung nicht statisch garantiert werden kann!

Typverträglichkeit *(type compatibility)*

ist eine *Halbordnung* auf den Typen,

für Javas primitive Typen wie folgt festgelegt:



1.2.3.1 Implizite Umwandlung bei primitiven Typen

Statische Typprüfung:

Wo ein Ausdruck vom Typ A erwartet wird,
darf ein Ausdruck mit anderem Typ B genau dann stehen,
wenn B mit A verträglich ist.

Umwandlung zur Laufzeit (implizit, automatisch):

Wo ein A-Wert erwartet wird, aber ein B-Wert
vorliegt, wird dieser in einen A-Wert umgewandelt.
(widening conversion)

Achtung: `int1 = char1` ist erlaubt !

`float1 = long1` bewirkt evtl. Genauigkeitsverlust !

`float1 = 0.0` *ist verboten*, weil `0.0 double` ist !

`float1 = 3.14f` ist erlaubt !

Außerdem gilt für `byte, short, char, int` :

Zuweisung eines konstanten Ausdrucks ist trotz Nichtverträglichkeit erlaubt, wenn die Repräsentation des Werts „in die Variable passt", z.B.

```
short1 = '*'
short1 = 4711          (narrowing conversion)
```

Bei Operatoren mit numerischen Operanden:

Symmetrische Anpassung *(binary numeric promotion)*

Anpassung eines Operanden an den anderen,
damit der Operator angewendet werden kann
(widening conversion):

☛ „Kleinerer" Typ wird an „größeren" Typ angepasst

☛ gemeinsamer Typ ist *mindestens* `int`,
d.h. evtl. werden *beide* Typen an `int` angepasst

(Es gibt auch *unary numeric promotion;*
z.B. liefert `-short1` einen `int`-Wert.)

1.2.3.2 Explizite Umwandlung

Beachte: `int1 = float1` ist nicht typkorrekt

`int1 = (int)float1` *ist* typkorrekt

Java, vereinfachte Syntax:

CastExpression: `(` *Type* `)` *Primary*

Umwandlungsoperator *(cast operator)*

Typkorrekt, wenn *Type* und *Primary*-Typ beide `boolean` oder beide nicht `boolean` sind.

Der *CastExpression* und sein Wert haben den explizit angegeben Typ.

Wert des CastExpression:

- ergibt sich ohne Aktion aus „umgedeuteter" Repräsentation

  z.B. `(short)char1` *(narrowing conversion)*

- ergibt sich wie bei impliziter Umwandlung der Repräsentation

  z.B. `(float)int1` *(widening conversion)*

- ergibt sich durch Umwandlung der Repräsentation,

  z.B. `(short)long1` *(narrowing conversion)*

  Wegwerfen der höherwertigen Bits (!)

  `(int)float1` *(narrowing conversion)*

  Wegwerfen des gebrochenen Anteils („Runden")