# Kapitel 4

# Grundlagen der funktionale und imperativen Programmierung

Skript zur Vorlesung

Einführung in die Programmierung

im Wintersemester 2012/13

Ludwig-Maximilians-Universität München



# Übersicht

# Übersicht

# 4.1 Ausdrücke



- Ein grundlegendes (funktionales) Konzept sind die sog. „Ausdrücke“
- Sie stellen ein wichtiges Mittel dar um Algorithmen als funktionalen Zusammenhang zwischen Eingabe und Ausgabe (d.h. als Funktionsdefinition) darzustellen
- Intuitiv sind Ausdrücke nichts anderes als Zeichenketten, die Funktionssymbole und Variablen (z.B. die Eingabevariablen eines Algorithmus) enthalten

# 4.1 Ausdrücke

## 4.1.1 Einführung



- Betrachten wir folgende Zeichenketten:
  - *DIV*(100 – *r*, 5)  wie z.B.in unserm Wechselgeld-Alg. 4
  - *a* + 5
  - blau & gelb
- Diese Zeichenketten können wir informell als Folgen von Symbolen begreifen, die u.a. Funktionssymbole darstellen

  (z.B. die Funktion „*DIV*“ in Funktionsschreibweise, die Funktion „+“ in Infixschreibweise oder die Konstante „5“).
- Eine solche Folge von Symbolen heißt auch *Ausdruck* oder *Term*.
- Das Konzept der Ausdrücke bildet einen Grundbaustein der meisten höheren Programmiersprachen.

# 4.1 Ausdrücke

## 4.1.1 Einführung



- Für einen Ausdruck können wir Regeln aufstellen:

  Intuitiv ist klar, dass etwa die Symbolfolge “5+” nicht korrekt ist, “5 + 2” oder “*a* + 5” aber schon. Es gibt also offenbar eine *Struktur,* die den Aufbau von korrekten Symbolfolgen (Ausdrücken) beschreibt.
- Außerdem hat eine Folge von Symbolen (ein Ausdruck) eine *Bedeutung* (vorausgesetzt, die verwendeten Symbole haben ihrerseits auch eine Bedeutung). Andernfalls könnte man sagen, “den Ausdruck verstehe ich nicht”.
- Die Struktur von korrekten Ausdrücken ist natürlich wieder die “Syntax” (auch “Grammatik”), die Bedeutung “Semantik”.
- Wir können eine korrekte Struktur beschreiben oder überprüfen, ohne etwas über die Bedeutung zu wissen.

- Informell beschreibt der Ausdruck "$a + 5$" die Addition einer Variablen mit einer Konstanten, der Ausdruck "blau & gelb" eine Farbmischung.
- Die beiden Ausdrücke gehören damit zu unterschiedlichen *Sorten* bzw. (*Daten-*) *Typen*.
- Wir könnten z.B. festlegen:
  - "$a + 5$" ist ein Ausdruck der Sorte $\mathbb{N}_0$ .
  - "blau & gelb" ist ein Ausdruck der Sorte "Farbe".
- Die Bezeichnung "Sorte" anstelle von "Wertebereich" macht deutlich, dass wir zunächst nicht an den konkreten Werten interessiert sind.
- Die Werte sind erst interessant, wenn es um die Bedeutung (Semantik) der Ausdrücke geht.

# 4.1 Ausdrücke

## 4.1. 2 Syntax

- Ausdrücke werden zusammengesetzt aus *Operatoren* und *Variablen.*
- Operatoren bezeichnen Funktionen und haben daher wie diese eine Stelligkeit.
- Wie bei Funktionen stehen 0-stellige Operatoren für Konstanten (also Elemente) der entsprechenden Sorte (des Bildbereichs des Operators).
- **ACHTUNG: wir nennen diese Konstanten ab jetzt *Literale*, da der Terminus „Konstante“ später in einer anderen Bedeutung verwendet wird.**
- Variablen sind Namen für Ausdrücke. Jede Variable hat eine Sorte.

- Ein 1-stelliger Operator kann auf einen Ausdruck angewendet werden.
- Ein *n*-stelliger Operator kann auf ein *n*-Tupel von Ausdrücken angewendet werden.
- Die *n* Komponenten des *n*-Tupels heißen *Operanden* des Operators, der auf das *n*-Tupel angewendet wird.
- Ein Operator bildet einen Ausdruck einer Sorte, die dem Bildbereich der vom ihm bezeichneten Funktion entspricht.

- Beispiele für Operatoren

$$
\begin{array}{rcl}
+ & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
- & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
= & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B} \\
> & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B} \\
< & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B} \\
\wedge & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\vee & : & \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\neg & : & \mathbb{B} \to \mathbb{B} \\
\mathit{TRUE} & : & \emptyset \to \mathbb{B} \\
\mathit{FALSE} & : & \emptyset \to \mathbb{B} \\
0 & : & \emptyset \to \mathbb{N}_0 \\
1 & : & \emptyset \to \mathbb{N}_0
\end{array}
$$

- Eine Operatorbeschreibung enthält ein Operatorsymbol und gibt dazu die Signatur der vom Operatorsymbol bezeichneten Funktion an, d.h. die Sorten der Operanden und die Sorte des vom Operator gebildeten Ausdrucks.
- In den Operatorbeschreibungen auf der vorherigen Folie kommen die Sorten $\mathbb{N}_0$ und $\mathbb{B}$ vor.
- Der Operator < verknüpft zwei Ausdrücke der Sorte $\mathbb{N}_0$ (seine Operanden) und bildet einen Ausdruck der Sorte $\mathbb{B}$ .
- Der 0-stellige Operator 0 ist ein Ausdruck der Sorte $\mathbb{N}_0$ und hat keine Operanden (=Literal).
- Ausdrücke gehören damit immer einer Sorte an („haben immer einen Typ“)

Die bisher informell diskutierten Ausdrücke werden nun formal definiert

Sei gegeben:

- Eine Menge von Sorten *S*,
- eine Menge von Operator-Beschreibungen *F* (Funktionssymbolen mit Bildbereich aus *S*) und
- eine Menge von Variablen *V*, die verschieden sind von allen Operator-Symbolen in *F*.
  Es gilt: $V = \bigcup_{S_i \in S} V_{S_i}$ , wobei $V_{S_i}$ die Menge aller

  Variablen der Sorte $S_i \in S$ bezeichnet (d.h. all Variablen aus *V* sind von einer Sorte aus *S*).

Die Menge der Ausdrücke ist dann wie folgt induktiv definiert:

- Eine Variable $v \in V_{S_i}$ ist ein Ausdruck der Sorte $S_i \in S$.
- Ein Literal *op* der Sorte $S_i$, also das Symbol $op \in F$ eines 0-stelligen Operators mit der Signatur $\emptyset \rightarrow S_i$ ($S_i \in S$), ist ein Ausdruck der Sorte $S_i$.
- Sind $a_1, \ldots, a_n$ Ausdrücke der Sorten $S_1, \ldots, S_n$ und $op \in F$ ein Operator mit der Signatur $S_1 \times \ldots \times S_n \rightarrow S_0$ (wobei $S_0, \ldots, S_n \in S$), dann ist $op(a_1, \ldots, a_n)$ ein Ausdruck der Sorte $S_0$.

# 4.1 Ausdrücke

## 4.1.2 Syntax



- Beispiele: Sei
  - $S = \{\mathbb{N}_0\}$
  - $F = \{ \quad +: \ \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    $/ \ : \ \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    $pow: \ \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    $\}$
  - $V = V_{\mathbb{N}_0} = \{x, y, z\}$
  - Dann gilt:
    - *x*, 3, 7 sind jeweils Ausdrücke (*x* ist eine Variable, 3 und 7 sind Literale)
    - dann sind auch +(*x*, 3) und / (18, 7) Ausdrücke
      (jeweils ein 2-stelliger Operator „+“ bzw. „/“ angewendet auf zwei Ausdrücke)
    - und ebenso ist *pow*(+(*x*, 3), / (18,7)) ein Ausdruck
      (2-stelliger Operator *pow* angewendet auf zwei Ausdrücke)

    - Dagegen ist +(3, *pow*(2,3,x)) KEIN Ausdruck (*Warum*?)

Als Schreibweisen für die Anwendung eines mehrstelligen Operators *op* gibt es wie für Funktionen folgendes:

- Funktionsform als $op(a_1, \ldots, a_n)$, wobei auch die $a_i$ in Funktionsform geschrieben sind.
- Präfixform als $op\ a_1 \ldots a_n$, wobei auch die $a_i$ in Präfixform geschrieben sind.
- Postfixform als $a_1 \ldots a_n op$, wobei auch die $a_i$ in Postfixform geschrieben sind.
- Infixform als $(op\ a_1)$ für einstellige Operatoren, $(a_1\ op\ a_2)$ für zweistellige Operatoren, wobei auch die $a_i$ in Infixform geschrieben sind. Bei höherer Stelligkeit als 2 kann der Operator *op* auch aus mehreren Teilen $op_1 \ldots op_{n-1}$ bestehen: $(a_1\ op_1\ a_2 \ldots op_{n-1} a_n)$. Wenn die Zuordnung der Operanden zu den Operatoren eindeutig ist, kann die Klammerung entfallen. Die hier definierte Form heißt *vollständig geklammert*.

# 4.1 Ausdrücke

## 4.1.2 Syntax



- Beispiel:
  - Der Ausdruck +(*x*, 3) ist in Funktionsform notiert
    - Präfixform: + *x* 3
    - Postfixform: *x* 3 +
    - Infixform: *x* + 3
  - Der Ausdruck *pow*(+(*x*, 3), / (18,7)) ist ebenfalls in Funktionsform
    - Präfixform:
    - Postfixform:
    - Infixform:

- Beispiel

  Wir betrachten den folgenden Ausdruck in Infixform:

  $$(x \wedge z) \vee (y \wedge z)$$

  Hierbei sind $x$, $y$, $z$ Variablen, $\vee$ und $\wedge$ Operatoren (aus der obigen Menge von Operator-Beschreibungen). Die äußeren Klammern sind weggelassen, da die Zuordnung der Teilausdrücke zu den Operatoren eindeutig ist. Übersetzt in die anderen Schreibweisen lautet der Ausdruck:

  - Präfixform: $\vee \wedge x\ z \wedge y\ z$
  - Postfixform: $x\ z \wedge y\ z \wedge \vee$
  - Funktionsform: $\vee(\wedge(x, z), \wedge(y, z))$

- Bindung:
  - Während man in der Präfixform und in der Postfixform auf Klammern verzichten kann (**Warum eigentlich?**), ist Klammerung in der Infixform grundsätzlich nötig, um Operanden ihren Operatoren eindeutig zuzuordnen.
  - Diese Zuordnung nennt man *Bindung*.
  - Lassen wir die Klammern im Ausdruck "$(x \wedge z) \vee (y \wedge z)$" weg, so erhalten wir $x \wedge z \vee y \wedge z$
  - Nun wäre auch eine völlig andere Bindung möglich, z.B. $x \wedge ((z \vee y) \wedge z)$

- Eindeutigkeit ohne Klammerung kann man in der Infixform erreichen, indem man den Operatoren unterschiedliche *Bindungsstärken* (*Präzedenzen*) zuweist.
- Beispielsweise hat unter den logischen Operatoren die Negation ¬ höhere Präzedenz als die Konjunktion ∧ und die Konjunktion ∧ hat wiederum höhere Präzedenz als die Disjunktion ∨.
- Damit erhalten wir auch für den Ausdruck

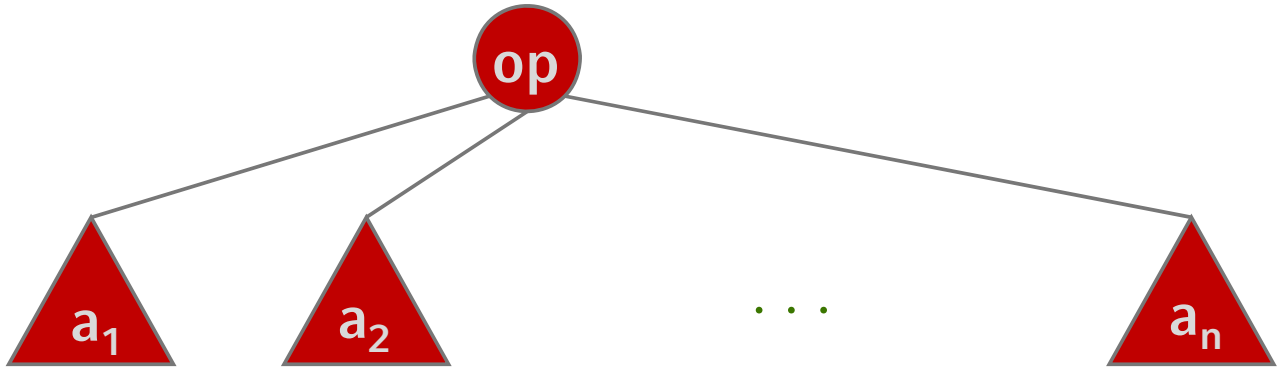
  $$x \wedge z \vee y \wedge z$$

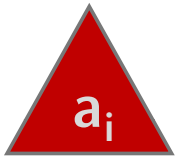
  die ursprünglich gewünschte, nun *implizite* Klammerung:

  $$(x \wedge z) \vee (y \wedge z)$$

- Wenn Operatoren gleicher Bindungsstärke konkurrieren, muss außerdem entschieden werden, ob der linke oder der rechte "gewinnt".
- Man legt hierzu fest, ob die Operanden eines Operators *linksassoziativ* oder *rechtsassoziativ* binden.
- Beispiel: Der Ausdruck "$x \vee y \vee z$" bedeutet
  - linksassoziativ:
    $$(x \vee y) \vee z$$
  - rechtsassoziativ:
    $$x \vee (y \vee z)$$
- Die linksassoziative Bindung ist gebräuchlicher.
- Natürlich ist das Setzen von Klammern dennoch erlaubt und sogar sinnvoll, um anzuzeigen, dass die implizite Klammerung der erwünschten Klammerung entspricht.

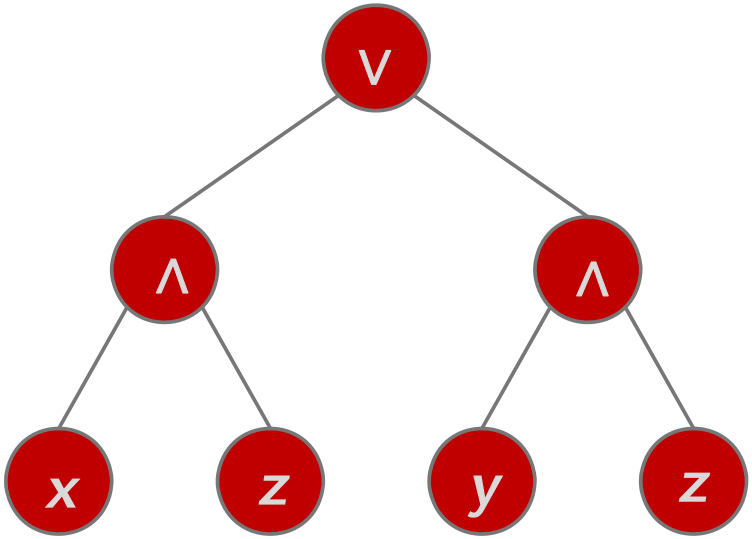
- Eine wichtige Art der Darstellung von Ausdrücken mit eindeutiger Zuordnung von Operanden zu Operatoren (durch implizite Klammerung) ist die *Baumdarstellung von Ausdrücken* (auch *Operatorbaum*) die induktiv definiert werden kann:
  - Eine Variable *v* wird durch den Knoten *v* dargestellt.
  - Ein *n*-stelliger Operator *op* wird durch den Knoten *op* mit den *n* Operanden als Teilbäumen dargestellt.



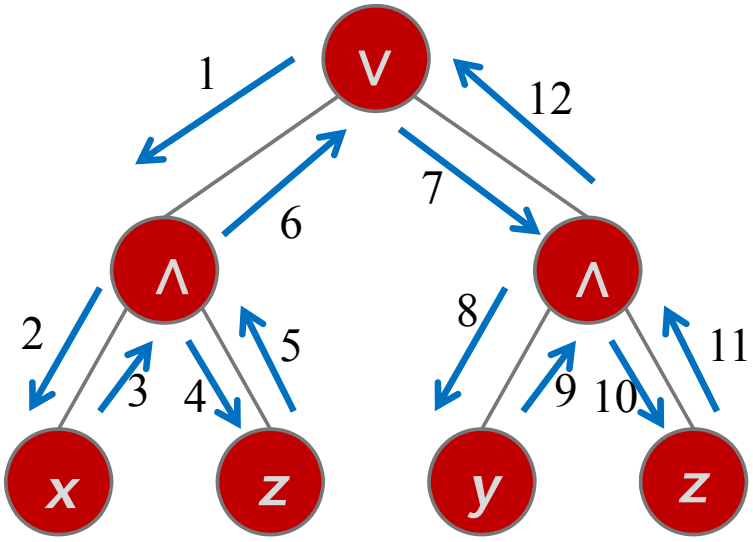

- Hierbei ist 

die Baumdarstellung des Ausdrucks $a_i$.

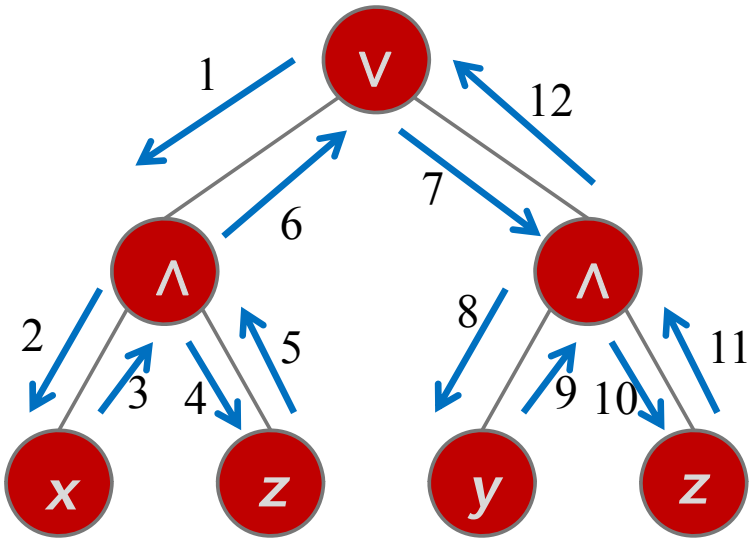
- Beispiel:
  Die Baumdarstellung des Ausdrucks "$(x \wedge z) \vee (y \wedge z)$" ist



- Da die Baumdarstellung (wie ja auch die Ausdrücke selbst) induktiv definiert ist, lassen sich auch naheliegende Funktionen leicht rekursiv definieren, die die Baumdarstellung auf die Präfixform, Postfixform bzw. Infixform abbilden.
- Man durchläuft dazu den Baum links-abwärts:
  1. Besuche den Wurzelknoten.
  2. Wenn der Wurzelknoten kein Blatt ist, führe für jeden Unterbaum *U* (links zuerst!!!) folgende Aktionen durch:
     a) Durchlaufe *U* links-abwärts
     b) Gehe dann wieder zurück zum Wurzelknoten.



- Man erhält die Präfixform, wenn man den Knotennamen beim ersten Besuch eines Knotens ausgibt.
- Man erhält die Postfixform, wenn man den Knotennamen beim letzten Besuch eines Knotens ausgibt.
- Man erhält die Infixform, wenn man beim ersten Besuch die öffnende, beim letzten die schließende Klammer ausgibt und bei den dazwischenliegenden Besuchen die Operatorteile.
- Bemerkung: Dieses Verfahren verdeutlicht, dass die verschiedenen Schreibweisen äquivalent sind.



- Betrachten Sie nun folgende Menge *F* von Operatorbeschreibungen:

$$
\begin{array}{rcl}
+ & : & \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
0 & : & \emptyset \to \mathbb{N}_0 \\
1 & : & \emptyset \to \mathbb{N}_0 \\
+ & : & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
0.0 & : & \emptyset \to \mathbb{R} \\
1.0 & : & \emptyset \to \mathbb{R}
\end{array}
$$

  - "0 + 1" ist ein gültiger Ausdruck (der Sorte $\mathbb{N}_0$ ).
  - "0.0 + 1.0" ist ein gültiger Ausdruck (der Sorte $\mathbb{R}$ ).
  - "0.0 + 1" ist **kein** gültiger Ausdruck.

- Das Operatorsymbol “+” kommt in *F* zweimal vor, erhält aber jeweils eine unterschiedliche Beschreibung (Signatur).
- Das Operatorsymbol “+” beschreibt also in *F* zwei unterschiedliche Funktionen.
- Man sagt: Das Operatorsymbol “+” ist *überladen.*
- Durch die Erfüllbarkeit einer Signatur kann man entscheiden, welcher Operator tatsächlich zu verwenden ist.
- Der Ausdruck “0.0 + 1” erfüllt in *F* keine Signatur.

- Um die *Bedeutung* (Semantik) eines Ausdrucks festzulegen, müssen einerseits die Funktionen, die von den Operatoren bezeichnet werden, definiert sein.
- Andererseits müssen alle Variablen ersetzt werden können durch den von ihnen bezeichneten Ausdruck.
- Dann können alle im Ausdruck vorkommenden Operationen ausgeführt werden.
- Schließlich kann man für einen Ausdruck einen Wert einsetzen.

Beispiel:
Um den Ausdruck “2 + 5” zu interpretieren, müssen die Operationen „2“, „+“ und „5“ definiert sein.

Z.B. vereinbaren wir:

- Der Operator 2 steht für die Zahl 2 $\in \mathbb{N}_0$.
- Der Operator 5 steht für die Zahl 5 $\in \mathbb{N}_0$.
- Der Operator + steht für die arithmetische Addition zweier natürlicher Zahlen.

Dadurch erhalten wir durch Anwendung der vereinbarten Funktionen als Wert des Ausdrucks die Zahl 7 $\in \mathbb{N}_0$.

- Um Variablen durch die von ihnen bezeichneten Ausdrücke zu ersetzen, muss wiederum die Bedeutung einer Variablen vereinbart werden.
- Aus einer Liste von vereinbarten Bedeutungen von Variablen kann man dann die Ersetzung (*Substitution*) der Variablen in einem Ausdruck vornehmen.
- Als einfache Substitution kann man ein Paar $\sigma$ aus einer Variablen $v$ und einem Ausdruck $a$, $\sigma = [v/a]$ ansehen, wobei $v$ und $a$ zur selben Sorte gehören müssen.
- Man sagt auch: „$v$ wird durch $a$ substituiert/ersetzt“
- Festzustellen, ob zwei Ausdrücke zur selben Sorte gehören, ist nicht trivial und wird unter dem Thema “*Unifikation*” studiert (nicht in dieser VL).

Die Anwendung einer Substitution $\sigma = [v/t]$ auf einen Ausdruck $u$ wird geschrieben

$$u\sigma \text{ bzw. } u[v/t].$$

Das Ergebnis dieser Anwendung ist ein Ausdruck, der durch einen der folgenden drei Fälle bestimmt ist:

1. $u[v/t] = t$, falls $u$ die zu ersetzende Variable $v$ ist,
2. $u[v/t] = u$, falls $u$ ein 0-stelliger Operator oder eine andere Variable als $v$ ist,
3. $u[v/t] = f\,(u_1[v/t], u_2[v/t], \ldots, u_n[v/t]))$, falls $u = f\,(u_1, u_2, \ldots, u_n)$.

- Beispiel:
  Sei $\sigma = [x/2 * b]$, d.h. Variable $x$ wird durch $2 * b$ ersetzt

  Angewendet auf den Ausdruck "$x + x - 1$" erhalten wir (in Funktionsform notiert):

$$\underbrace{-(+(x,x),1)}_{u}\underbrace{[x/*(2,b)]}_{\sigma} = -(+(x,x)[x/*(2,b)],1[x/*(2,b)])$$
$$= -(+(x,x)[x/*(2,b)],1)$$
$$= -(+(x[x/*(2,b)],x[x/*(2,b)]),1)$$
$$= -(+(*(2,b),*(2,b)),1)$$

d.h. in Infixschreibweise:

aus $x + x - 1$ wird $2 * b + (2 * b) - 1$

- Für eine gegebene Menge an Sorten (mit ihren Literalen) und eine Menge von Operanden über diesen Sorten können wir also nun Ausdrücke bilden.
- Typischerweise stellt jede höhere Programmiersprache gewisse *Grunddatentypen* als Sorten zur Verfügung.
- Zusätzlich werden auch gewisse *Grundoperationen* bereitgestellt, also eine Menge von (teilweise überladenen) Operationssymbolen.
- Die Semantik dieser Operationen ist durch den zugrundeliegenden Algorithmus zur Berechnung der entsprechenden Funktion definiert.
- Aus den Literalen der Grunddatentypen, den zugehörigen Grundoperationen und Variablen können nun, wie im vorherigen Abschnitt definiert, in Java Ausdrücke gebildet werden.

# 4.1 Ausdrücke

## 4.1.4 Einfache Ausdrücke in Java, Typkonversion



- Java stellt Grunddatentypen (auch *atomare* oder *primitive Typen*) für $\mathbb{B}$, *CHAR*, eine Teilmenge von $\mathbb{Z}$ und für eine Teilmenge von $\mathbb{R}$ gibt, aber keinen eigenen Grunddatentyp für $\mathbb{N}$.
- Die Werte (Literale) der primitiven Typen werden intern binär dargestellt.
- Die Datentypen unterscheiden sich u.a. in der Anzahl der Bits, die für ihre Darstellung verwendet werden.
- Die Anzahl der Bits, die für die Darstellung der Werte eines primitiven Datentyps verwendet werden, heißt *Länge* des Typs.
- Die Länge eines Datentyps hat Einfluss auf seinen Wertebereich.
- Als *objektorientierte* Sprache bietet Java zusätzlich die Möglichkeit, benutzerdefinierte (sog. *abstrakte*) Datentypen zu definieren. Diese Möglichkeiten lernen wir im Teil über objektorientierte Modellierung genauer kennen.

- Grunddatentypen (Sorten) in Java: Überblick

| Typname | Länge | Wertebereich |
|---|---|---|
| `boolean` | 1 Byte | Wahrheitswerte {`true`,`false`} |
| `char` | 2 Byte | Alle Unicode-Zeichen |
| `byte` | 1 Byte | Ganze Zahlen von $-2^7$ bis $2^7 - 1$ |
| `short` | 2 Byte | Ganze Zahlen von $-2^{15}$ bis $2^{15} - 1$ |
| `int` | 4 Byte | Ganze Zahlen von $-2^{31}$ bis $2^{31} - 1$ |
| `long` | 8 Byte | Ganze Zahlen von $-2^{63}$ bis $2^{63} - 1$ |
| `float` | 4 Byte | Gleitkommazahlen (einfache Genauigkeit) |
| `double` | 8 Byte | Gleitkommazahlen (doppelte Genauigkeit) |

### Der Typ **`boolean`**

- Java hat einen Typ **`boolean`** für die Sorte $\mathbb{B}$ der Wahrheitswerte, die Literale sind **`true`** für "wahr" und **`false`** für "falsch".

| Operator | Bezeichnung | Bedeutung (Semantik) |
|---|---|---|
| `!` | Negation | `!a` ergibt **`false`** wenn `a` **`true`** ist, sonst **`true`** |
| `&` | Konjunktion | `a&b` ergibt **`true`**, wenn sowohl `a` als auch `b` **`true`** sind, sonst **`false`**. Beide Teilausdrücke (`a` und `b`) werden ausgewertet. |
| `&&` | Seq. Konj. | `a&&b` ergibt **`true`**, wenn sowohl `a` als auch `b` **`true`** sind, sonst **`false`**. Ist `a` bereits **`false`**, wird **`false`** zurückgegeben und `b` nicht mehr ausgewertet. |
| `\|` | Disjunktion | `a\|b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` **`true`** ist, sonst **`false`**. Beide Teilausdrücke (`a` und `b`) werden ausgewertet. |
| `\|\|` | Seq. Disj. | `a\|\|b` ergibt **`true`**, wenn mindestens einer der beiden Ausdrücke `a` oder `b` **`true`** ist, sonst **`false`**. Ist `a` bereits **`true`**, wird **`true`** zurückgegeben und `b` nicht mehr ausgewertet. |
| `^` | XOR („exkl. Oder") | `a^b` ergibt **`true`**, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wert haben. |

### Der Typ **char**

- Java hat einen eigenen Typ **char** für die Sorte CHAR der (Unicode-) Zeichen.
- Werte (Literale) werden in einfachen Hochkommata gesetzt, z.B. `'A'` für das Zeichen "A". Einige Sonderzeichen können mit Hilfe von *Standard-Escape-Sequenzen* dargestellt werden:

| Sequenz | Bedeutung |
|---|---|
| `\b` | Backspace (Rückschritt) |
| `\t` | Tabulator (horizontal) |
| `\n` | Newline (Zeilenumbruch) |
| `\f` | Seitenumbruch (Formfeed) |
| `\r` | Wagenrücklauf (Carriage return) |
| `\"` | doppeltes Anführungszeichen |
| `\'` | einfaches Anführungszeichen |
| `\\` | Backslash |

Die Typen **`byte`**, **`short`**, **`int`**, und **`long`**

- Java hat vier Datentypen für die Sorte $\mathbb{Z}$ der ganzen Zahlen:
  - **`byte`** (Länge: 8 Bit)
  - **`short`** (Länge: 16 Bit)
  - **`int`** (Länge: 32 Bit)
  - **`long`** (Länge: 64 Bit)
- Literale können geschrieben werden in
  - Dezimalform: bestehen aus den Ziffern `0`, . . ., `9`
  - Oktalform: beginnen mit dem Präfix `0` und bestehen aus Ziffern `0`, . . ., `7`
  - Hexadezimalform: beginnen mit dem Präfix `0x` und bestehen aus Ziffern `0`, . . ., `9` und den Buchstaben `a`, . . ., `f` (bzw. `A`, . . ., `F`)
- Negative Zahlen erhalten ein vorangestelltes „-“.
- **Gehört dieses “-” zum Literal?**

- Vorzeichen-Operatoren haben die Signatur

$$S \rightarrow S$$

mit $S \in \{$**`byte`**, **`short`**, **`int`** . . . $\}$.

- Operationen:

| Operator | Bezeichnung | Bedeutung |
|---|---|---|
| + | Positives Vorzeichen | +n ist gleichbedeutend mit n |
| - | Negatives Vorzeichen | -n kehrt das Vorzeichen von n um |

- Offenbar sind diese Operatoren überladen!!!

## Die Typen **`float`** und **`double`**

- Java hat zwei Datentypen für Fließkommazahlen:
  - **`float`** (Länge: 32 Bit)
  - **`double`** (Länge: 64 Bit)
- Literale werden immer in Dezimalnotation geschrieben und bestehen maximal aus
  - Vorkommateil
  - Dezimalpunkt (*)
  - Nachkommateil
  - Exponent `e` oder `E` (Präfix – möglich) (*)
- Negative Zahlen erhalten ein vorangestelltes -.
- Beispiele:
  - double: `6.23, 623E-2, 62.3e-1`
  - float: `6.23f, 623E-2F, 62.3e-1`

(*) mindestens einer dieser Bestandteile muss vorhanden sein.

- Arithmetische Operationen haben die Signatur

$$S \times S \rightarrow S$$

mit $S \in \{$**`byte`**, **`short`**, **`int`** . . . $\}$.

- Operationen:

| Operator | Bezeichnung | Bedeutung |
|---|---|---|
| `+` | Summe | `a+b` ergibt die Summe von `a` und `b` |
| `-` | Differenz | `a-b` ergibt die Differenz von `a` und `b` |
| `*` | Produkt | `a*b` ergibt das Produkt aus `a` und `b` |
| `/` | Quotient | `a/b` ergibt den Quotienten von `a` und `b` |
| `%` | Modulo | `a%b` ergibt den Rest der ganzzahligen Division von a durch b |

- Auch diese Operatoren sind offenbar überladen!!!

- Vergleichsoperatoren (Prädikate) haben die Signatur

$$S \times S \rightarrow \texttt{boolean}$$

mit $S \in \{$**byte**, **short**, **int** . . . $\}$.

- Operationen:

| Operator | Bezeichnung | Bedeutung |
|---|---|---|
| == | Gleich | a==b ergibt **true**, wenn a gleich b ist |
| != | Ungleich | a!=b ergibt **true**, wenn a ungleich b ist |
| < | Kleiner | a<b ergibt **true**, wenn a kleiner b ist |
| <= | Kleiner gleich | a<=b ergibt **true**, wenn a kleiner oder gleich b ist |
| > | Größer | a>b ergibt **true**, wenn a größer b ist |
| >= | Größer gleich | a>=b ergibt **true**, wenn a größer oder gleich b ist |

- Auch diese Operatoren sind offenbar überladen!!!

- Wir können auch in Java aus Operatoren und Variablen Ausdrücke bilden, so wie wir sie vorher formal definiert haben (wir lassen der Einfachheit halber zunächst die Variablen weg).
- Laut induktiver Definition von Ausdrücken ist ein Literal ein Ausdruck.
- Aus Literalen (z.B. den **`int`** Werten `6` und `8`) und mehrstelligen Operatoren (z.B. `+`, `*`, `<`, `&&`) können wir ebenfalls Ausdrücke bilden. Ein gültiger Ausdruck hat selbst wieder einen Wert (der über die Semantik der beteiligten Operationen definiert ist) und einen Typ (der durch die Ergebnissorte des angewendeten Operators definiert ist):
  - `6 + 8 //Wert: 14 vom Typ int`
  - `6 * 8 //Wert: 48 vom Typ int`
  - `6 < 8 //Wert: true vom Typ boolean`
  - `6 && 8 //ungültiger Ausdruck` **Warum?**

- Typkonversion:
  - Was passiert eigentlich, wenn man verschiedene Sorten / Typen miteinander verarbeiten will?
  - Wir hatten oben bei der Syntax von Ausdrücken den Ausdruck „0.0 + 1“ als ungültig betrachtet da er keine Signatur erfüllt.
  - Analog: Ist der Java-Ausdruck `6 + 7.3` erlaubt?
  - Eigentlich nicht: Die Operation + ist zwar überladen, d.h.

$$+ : S \times S \rightarrow S$$

ist definiert für beliebige primitive Datentypen *S*, aber es gibt keine Operation

$$+ : S \times T \rightarrow U$$

für *verschiedene* primitive Datentypen $S \neq T$.

- Aber es gilt $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$, d.h. die Ausdrücke sind eigentlich sinnvoll wenn man die Zahl `6` als `6.0` $\in$ **`double`** interpretieren würde
- Ganz allgemein nennt man das Konzept der Umwandlung eines Ausdrucks mit einem bestimmten Typ in einen Ausdruck mit einem anderen Typ *Typkonversion* (*Typecasting*).
- In vielen Programmiersprachen gibt es eine automatische Typkonversion meist vom spezielleren in den allgemeineren Typ.
- Eine Typkonversion vom allgemeineren in den spezielleren Typ muss (wenn erlaubt) sinnvollerweise immer explizit durch einen *Typecasting*-Operator herbeigeführt werden.
- Es gibt auch Programmiersprachen, in denen man grundsätzlich ein entsprechendes Typecasting explizit durchführen muss.
- In Java ist der Ausdruck `6 + 7.3` tatsächlich erlaubt.
- *Wann* passiert *was* und *wie* bei der Auswertung des Ausdrucks `6 + 7.3`?

- **Wann**:
  Während des Übersetzens des Programmcodes durch den Compiler.
- **Was**:
  Der Compiler kann dem Ausdruck keinen Typ (und damit auch keinen Wert) zuweisen. Solange kein *Informationsverlust* auftritt, versucht der Compiler diese Situation zu retten.
- **Wie**:
  Der Compiler konvertiert automatisch den Ausdruck `6` vom Typ **`int`** in den Ausdruck `6.0` vom Typ **`double`**, so dass die Operation

$$+ : \texttt{double} \times \texttt{double} \rightarrow \texttt{double}$$

  angewendet werden kann.

- Formal gesehen ist diese Konvertierung eine Operation `i->d` mit der Signatur

  $$\texttt{i->d} : \textbf{int} \rightarrow \textbf{double}$$

  d.h. der Compiler wandelt den Ausdruck

  ```
  6 + 7.3
  ```

  in den Ausdruck

  ```
  i->d(6) + 7.3
  ```

- Dieser Ausdruck erfüllt die Signatur von „+“, hat offensichtlich einen eindeutigen Typ und damit auch einen eindeutig definierten Wert.

- Was bedeutet “Informationsverlust”?
- Es gilt offensichtlich folgende “Kleiner-Beziehung” zwischen Datentypen:

  **`byte`** < **`short`** < **`int`** < **`long`** < **`float`** < **`double`**

- Beispiele:
  - `1 + 1.7` ist vom Typ **`double`**
  - `1.0f + 2` ist vom Typ **`float`**
  - `1.0f + 2.0` ist vom Typ **`double`**
- Java konvertiert Ausdrücke automatisch in den allgemeineren (“größeren”) Typ, da dabei kein Informationsverlust auftritt. **Warum?**

- Will man eine Typkonversion zum spezielleren Typ durchführen, so muss man dies in Java explizit angeben.
- In Java erzwingt man die Typkonversion zum spezielleren Typ **`type`** durch Verwendung des 1-stelligen Typecast-Operators (**`type`**). Dieser wird in Infixschreibweise benutzt, die Klammern gehören zum Namen des Operators
- Der Ausdruck `(`**`type`**`) a` wandelt den Ausdruck `a` in einen Ausdruck vom Typ **`type`** um.
- Der Typ des Operators ist z.B.:

```
(int)   : char∪byte∪short∪int∪long∪float∪double → int
(float) : char∪byte∪short∪int∪long∪float∪double → float
```

- Sie können also z.B. auch **`char`** in **`int`** umwandeln.
  **Klingt komisch? Ist aber so! Und was passiert da?**

- Beispiele:
  - `(byte)3` ist vom Typ **`byte`**
  - `(int)(2.0 + 5.0)` ist vom Typ **`int`**
  - `(float)1.3e-7` ist vom Typ **`float`**
- VORSICHT: Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
- Beispiele:
  - `(int)5.2` ergibt `5` [ein späteres `(double) 5` ergibt `5.0`]
  - `(int)-5.2` ergibt `-5`

# 4.1 Ausdrücke



## Abschließende Bemerkungen

- Ausdrücke stellen ein klassisches funktionales Konzept dar: Der Ausdruck 3 + 5 hat offenbar den Wert 8 $\in \mathbb{N}_0$ .
- Die Funktionen, die durch die Operatoren 3, + und 5 bezeichnet werden, sind in einer vollständigen Semantik definiert.
- Wie der Wert dieses Ausdrucks auf einem Rechner konkret berechnet wird, ist typischerweise aber nicht näher spezifiziert.
- Die formalere Beschäftigung mit Ausdrücken sowie die hier nur skizzierten Problembereiche
  - Sorten und Überladung,
  - Definition und Eigenschaften rekursiver Funktionen, z.B. für Baumdurchläufe,
  - Substitution und Unifikation

  bilden einen wichtigen Inhalt der Vorlesung “Programmierung und Modellierung”.