

# Abschnitt 5: Grundlagen der funktionalen Programmierung

## 5. Grundlagen der funktionalen Programmierung

5.1 Sorten und abstrakte Datentypen

5.2 Ausdrücke

5.3 Ausdrücke in Java

5.4 Funktionale Algorithmen



# Road Map

- ▶ Wir formalisieren zunächst die Idee, dass wir eine Menge an Basisoperationen über einer Menge an Datentypen als gegeben annehmen.
- ▶ Dazu führen wir den Begriff des Moduls und des abstrakten Datentyps ein.
- ▶ Wir leiten dann her, wie funktionale Algorithmen syntaktisch aufgebaut sind und formalisieren deren Semantik.
- ▶ Wir erweitern das Modulkonzept als „Container“, in dem wir eigene (funktionale und später auch imperative) Algorithmen aufschreiben können.



# RoadMap

- ▶ Wir verwenden teilweise eine eigene Schreibweise (Pseudo-Code), um die Konzepte in ihrer Reinheit vorzustellen.
- ▶ Wir werden die Syntax dieser „Pseudo-Sprache“ allerdings nicht formal definieren (und damit auch nicht deren Semantik).
- ▶ Die Pseudo-Sprache ist v.a. dazu da um:
  - ▶ die Konzepte darzustellen
  - ▶ möglichst intuitiv und Menschen-lesbar zu sein
- ▶ Daneben schauen wir uns auch immer an, wie die entsprechenden Konzepte in einer konkreten Programmiersprache (Java) umgesetzt sind.

# Überblick

## 5. Grundlagen der funktionalen Programmierung

### 5.1 Sorten und abstrakte Datentypen

#### 5.2 Ausdrücke

#### 5.3 Ausdrücke in Java

#### 5.4 Funktionale Algorithmen



# Sorten

- ▶ Wir hatten gesehen, dass z.B. die Zeichenkette 123 sowohl eine natürliche Zahl als auch einen Text (String) darstellen kann, d.h. unterschiedliche Daten bezeichnet.
- ▶ Basis unserer bisherigen Algorithmen waren *elementar ausführbare* (*Grund-*) *Operationen* (wie z.B. *DIV*,  $-$ ,  $+$ , etc.), die wir (implizit) als gegeben vorausgesetzt haben.
- ▶ Solche Grundoperationen gelten offenbar für verschiedene Daten (*DIV* macht z.B. für natürliche Zahlen aber nicht für Zeichen oder Texte Sinn).
- ▶ Wir sollten also Daten anhand ihres *Typs* unterscheiden.



# Sorten und Module

- ▶ Wir formalisieren diese Unterscheidung wie folgt:
- ▶ Eine nicht-leere Menge von Objekten heißt *Sorte* oder auch (*Daten-*) *Typ*<sup>6</sup>
- ▶ Beispiele für Sorten:
  - ▶ Die Menge der natürlichen Zahlen  $\mathbb{N}_0$
  - ▶ Die Menge der booleschen Werte  $\mathbb{B}$
  - ▶ Die Menge der reellen Zahlen  $\mathbb{R}$
  - ▶ Die Menge der druckbaren Zeichen *CHAR*
  - ▶ Die Menge der Farben *Farbe* = {rot, grün, blau, gelb, ...}
  - ▶ ...

---

<sup>6</sup>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) geht.

# Sorten und Module

- ▶ Auf Basis der Sorten formalisieren wir nun die Idee der Grundoperationen, die wir als gegeben voraussetzen.
- ▶ Ein *Modul* ist eine nicht-leere, endliche Menge  $\{S_1, \dots, S_n\}$  von Sorten mit einer nicht-leeren, endlichen Menge von Operationen über diese Sorten, d.h. Operation mit Funktionalitäten

$$S_{i_1} \times S_{i_2} \times \dots \times S_{i_m} \rightarrow S_{i_{m-1}}, \quad \text{mit } i_1, \dots, i_{m+1} \in \{1, \dots, n\}.$$

- ▶ Dabei sind insbesondere  $m = 0$ -stellige Operationen explizit zugelassen: Sie formalisieren die Syntax der Elemente („Werte“) der einzelnen Sorten.
- ▶ Später werden wir diesen Operationen Bedeutungen (Semantiken) geben, und damit auch den einzelnen Elementen der Sorten.

# Sorten und Module

## ► Bemerkungen:

- Das Modul-Konzept ist in vielen imperativen Sprachen vorhanden. Wir werden es im Folgenden auch entsprechend erweitern und Module als Strukturierungsmittel von Programmen kennenlernen.
- Zunächst verwenden wir aber das Modul-Konzept, um Grundoperationen über Sorten zu definieren.

# Module: Beispiele

- ▶ Für die Sorte  $\mathbb{B}$  liegt es nahe, die Operationen  $\neg$ ,  $\vee$  und  $\wedge$  als Grundoperationen vorauszusetzen.
- ▶ Die aus einer einzigen Sorte bestehende Menge  $\{\mathbb{B}\}$  zusammen mit diesen Operationen ist ein erstes Beispiel für ein Modul.
- ▶ Wenn wir die beiden Elemente der Menge noch als 0-stellige Operationen auffassen, besteht das Modul also aus
  - ▶ der Sorte  $\mathbb{B}$
  - ▶ den Operationen:

$TRUE : \emptyset \rightarrow \mathbb{B}$

$FALSE : \emptyset \rightarrow \mathbb{B}$

$\neg : \mathbb{B} \rightarrow \mathbb{B}$

$\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

$\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$



# Module: Beispiele

- ▶ Ein ähnliches Modul kann man für die Grundoperationen über  $\mathbb{N}_0$  definieren, bestehend aus
  - ▶ den Sorten  $\mathbb{N}_0$  und  $\mathbb{B}$  (da  $\mathbb{B}$  Bildbereich einiger Operationen ist)
  - ▶ den Operationen (die Auswahl ist zunächst willkürlich bzw. sollte sich an der konkreten Programmiersprache orientieren):

$$0 : \emptyset \rightarrow \mathbb{N}_0$$

$$1 : \emptyset \rightarrow \mathbb{N}_0$$

... (alle Elemente aus  $\mathbb{N}_0$ )

$$+ : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$- : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$\cdot : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$DIV : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$MOD : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$=: \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

$$\neq : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

$$\leq : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

$$< : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

$$\geq : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$

$$> : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$$



# Sorten und Module

- ▶ Analog kann man ein Modul für die Grundoperationen aus  $\mathbb{Z}$  definieren. Die beteiligten Sorten sind dann  $\mathbb{Z}$  und  $\mathbb{B}$ , die Operationen des vorherigen Moduls kann man übernehmen (allerdings überall die Sorte  $\mathbb{N}_0$  durch  $\mathbb{Z}$  in den Signaturen ersetzen) und z.B. eine weitere Grundoperation

$$- : \mathbb{Z} \rightarrow \mathbb{Z} \quad \text{mit} \quad - : x \mapsto -x$$

(das einstellige Vorzeichen Minus) hinzufügen.

- ▶ Entsprechendes gilt für die Grundoperationen aus  $\mathbb{R}$  und *CHAR*.



# Sorten und Module

- ▶ Damit haben wir nun also Grundoperationen für unsere Sorten durch Module definiert.
- ▶ Wir haben dabei aber genau genommen nur die Signatur und damit die Syntax der Operationen spezifiziert.
- ▶ Die Semantik, also letztlich die genaue Definition, was diese Operationen tun, wurde bisher nicht angegeben.

# Sorten und Module

- ▶ Es gibt verschiedene Möglichkeiten, dies zu tun, z.B.
  - ▶ Man könnte diese Definitionen explizit angeben, indem man z.B. einen Algorithmus angibt, der dann vermutlich aber nur Maschinenbefehle verwenden sollte.
  - ▶ Man könnte die Operationen axiomatisch charakterisieren, die (Realisierungs-)Details der Operationsdefinitionen bleibt damit vor uns verborgen (und wir legen uns zunächst auch nicht auf eine Plattform fest). Natürlich setzt dies voraus, dass „irgendjemand“ diese Operationen dann tatsächlich bereitstellt.



# Abstrakte Datentypen

- ▶ Wir verlassen uns auf eine axiomatische Charakterisierung (auch wenn wir sie in der Vorlesung bzw. Übung nicht angeben; gängige Bücher über algebraische Spezifikationen bieten hier Abhilfe)
- ▶ Der Vorteil dieser Variante ist, dass wir die Operationen verwenden können, ohne uns um die Komplexität der Realisierungsdetails kümmern zu müssen.

# Abstrakte Datentypen: Beispiel

- ▶ Ein Beispiel, bei dem dieser Vorteil sichtbar wird, ist ein Modul, das Grundoperationen über der Sorte *Farbe* (und hier im Beispiel  $\mathbb{R}$  als zusätzlich beteiligte Sorte) definiert.
- ▶ Als Operationen könnten wir z.B. verwenden:

*blau* :  $\emptyset \rightarrow \text{Farbe}$

*gruen* :  $\emptyset \rightarrow \text{Farbe}$

*rot* :  $\emptyset \rightarrow \text{Farbe}$

*gelb* :  $\emptyset \rightarrow \text{Farbe}$

...

*-* :  $\text{Farbe} \times \text{Farbe} \rightarrow \text{Farbe}$

*+* :  $\text{Farbe} \times \text{Farbe} \rightarrow \text{Farbe}$

*helligkeit* :  $\text{Farbe} \rightarrow \mathbb{R}$

*==* :  $\text{Farbe} \times \text{Farbe} \rightarrow \mathbb{R}$

...

- ▶ Eine Erweiterung dieser Menge sowie eine (sinnvolle) axiomatische Spezifikation überlassen wir den Eifrigen unter Ihnen.



# Abstrakte Datentypen

- ▶ Module, die die Realisierungsdetails der Operationen vor dem Benutzer verbergen, heißen auch *abstrakte Datentypen*.
- ▶ Das ist ein wichtiges Konzept in der Informatik, daher nochmals aus anderer Perspektive:  
Ein abstrakter Datentyp stellt ganz allgemein Operationen über einer Menge von Sorten zur Verfügung, deren Realisierung vom Benutzer verborgen wird.
- ▶ Dieses Verbergen von Detailinformationen ist offensichtlich ein großer Vorteil beim Erstellen komplexerer Algorithmen (siehe das Modul für die Operationen über *Farbe*).



# Road Map

- ▶ Mit dem Modulkonzept haben wir also zunächst eine Menge von Sorten mit Operationen auf diesen Sorten als gegeben definiert (wer auch immer diese für uns realisiert).
- ▶ Nochmal der Hinweis: auch die einzelnen Elemente der Sorten haben wir damit syntaktisch „eingeführt“.
- ▶ Wir werden das Modulkonzept nun Schritt für Schritt erweitern: wir werden die Möglichkeit einführen, die Operationen, die ein Modul bereitstellt, selbst zu implementieren.
- ▶ Damit erhalten wir die Möglichkeit, einen Algorithmus als Operation eines Moduls zu definieren.
- ▶ Außerdem werden wir die Syntax und anschließend die Semantik von funktionalen Algorithmen formal definieren.

# Überblick

## 5. Grundlagen der funktionalen Programmierung

5.1 Sorten und abstrakte Datentypen

5.2 Ausdrücke

5.3 Ausdrücke in Java

5.4 Funktionale Algorithmen

**LMU**

# Einführung

- ▶ Ein grundlegendes (funktionales) Konzept praktisch aller Programmiersprachen sind die sog. *Ausdrücke*.
- ▶ Sie stellen u.a. ein wichtiges Mittel dar um Algorithmen als funktionalen Zusammenhang zwischen Eingabe und Ausgabe (d.h. als Funktionsdefinition) darzustellen.
- ▶ Intuitiv sind Ausdrücke nichts anderes als Zeichenreihen, die Operationssymbole<sup>7</sup> und Variablen (z.B. die Eingabevariablen eines Algorithmus) enthalten (also so etwas wie  $DIV(100 - r, 5)$ ).



---

<sup>7</sup>Zur Erinnerung: 0-stellige Operationen sind Konstanten, also Elemente aus dem Bildbereich

# Einführung

- ▶ Betrachten wir folgende Zeichenreihen:
  - ▶  $DIV(100 - r, 5)$
  - ▶  $a + 5$
  - ▶ blau – gelb
- ▶ Diese Zeichenreihen können wir informell als Folgen von Symbolen begreifen, die u.a. Operationssymbole darstellen (z.B. die Funktion<sup>8</sup>  $DIV$  in Funktionsschreibweise, die Funktion  $+$  in Infixschreibweise oder die Konstante 5).
- ▶ Eine solche Folge von Symbolen heißt auch *Ausdruck* oder *Term*.

<sup>8</sup>Wir verwenden teilweise die Begriffe „Funktion“ und „Operation“ als Synonyme

# Einführung

- ▶ Für einen Ausdruck können wir Regeln aufstellen:
  - ▶ Intuitiv ist klar, dass etwa die Zeichenreihe  $5+$  nicht korrekt ist,  $5 + 2$  oder  $a + 5$  aber schon.
  - ▶ Es gibt also offenbar eine Struktur, die den Aufbau von korrekten Zeichenreihen für Ausdrücke beschreibt (die Syntax von Ausdrücken).
- ▶ Außerdem hat eine Folge von Symbolen (ein Ausdruck) eine Bedeutung, vorausgesetzt, die verwendeten Symbole haben ihrerseits auch eine Bedeutung (die Semantik von Ausdrücken).
- ▶ Wir können wiederum eine korrekte Struktur von Ausdrücken beschreiben oder überprüfen, ohne etwas über deren Bedeutung zu wissen.



# Syntax von Ausdrücken

- ▶ Informell beschreibt der Ausdruck  $a + 5$  die Addition einer Variablen mit einer Konstanten, der Ausdruck  $blau - gelb$  (möglicherweise) eine Farbmischung.
- ▶ Die beiden Ausdrücke gehören damit zu unterschiedlichen Sorten.
- ▶ Wir könnten z.B. festlegen:
  - ▶  $a + 5$  ist ein Ausdruck der Sorte  $\mathbb{N}_0$ .
  - ▶  $blau - gelb$  ist ein Ausdruck der Sorte *Farbe*.
- ▶ Also grundsätzlich hat jeder Ausdruck eine Sorte.



# Syntax von Ausdrücken

- ▶ Ausdrücke werden zusammengesetzt aus *Operatoren* (wie z.B.  $+$ ) und *Variablen* (wie z.B.  $a$ ).
- ▶ Operatoren bezeichnen (zunächst) Funktionen und haben daher wie diese eine Stelligkeit.
- ▶ 0-stellige Operatoren stehen für Konstanten (also Elemente) der entsprechenden Sorte (des Bildbereichs des Operators), z.B.  $5$  für die konstante Funktion mit der Signatur  $\emptyset \rightarrow \mathbb{N}_0$  (die genaue Bedeutung von  $5$  ist uns übrigens zunächst egal. **WARUM?**)
- ▶ **ACHTUNG:** wir nennen diese Konstanten ab jetzt *Literale*, da der Begriff *Konstante* später in einer anderen Bedeutung verwendet wird.
- ▶ Variablen sind Namen für Ausdrücke und wir fordern, dass jede Variable ebenfalls eine Sorte hat.

# Syntax von Ausdrücken

- ▶ Analog dem Konzept der Komposition von Funktionen gilt offenbar:
  - ▶ Ein 1-stelliger Operator mit Definitionsbereich  $D$  kann auf einen Ausdruck der Sorte  $D$  angewendet werden.
  - ▶ Ein  $n$ -stelliger Operator mit den Definitionsbereich  $D_1 \times \dots \times D_n$  kann entsprechend auf ein  $n$ -Tupel aus Ausdrücken  $D_1 \times \dots \times D_n$  angewendet werden.
  - ▶ Die  $n$  Komponenten des  $n$ -Tupels heißen Operanden des Operators, der auf das  $n$ -Tupel angewendet wird.
  - ▶ Ein Operator mit Bildbereich  $B$  bildet einen Ausdruck einer Sorte  $B$ .
  - ▶ Analoges gilt für Variablen vom Typ  $T$ .



# Syntax von Ausdrücken

## ▶ Beispiel:

- ▶ Der Operator  $<: \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{B}$  kann auf ein Tupel  $\mathbb{N}_0 \times \mathbb{N}_0$  angewendet werden und ergibt ein Literal (und damit wieder einen Ausdruck) der Sorte  $\mathbb{B}$  (der wiederum überall dort in Operationen eingesetzt werden könnte, wo ein Argument der Sorte  $\mathbb{B}$  verlangt wird).
- ▶ Konkret könnte  $<$  z.B. auf  $5 \in \mathbb{N}_0$  und  $7 \in \mathbb{N}_0$  angewendet werden und  $5 < 7$  (in Infixschreibweise) ergibt natürlich ein Element aus  $\mathbb{B}$  (welches ist uns aktuell egal, wie gesagt, wir sind nur an der Syntax interessiert, nicht an der Semantik!!!)



# Syntax von Ausdrücken

- ▶ Die bisher informell diskutierten Ausdrücke werden nun formal definiert
- ▶ Sei gegeben:
  - ▶ eine Menge von Sorten  $S$ ,
  - ▶ eine Menge von Operator-Beschreibungen  $F$  (Funktionssymbolen mit Definitionsbereich und 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. alle Variablen aus  $V$  sind von einer Sorte aus  $S$ ).



# Syntax von Ausdrücken

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  $S_i \in S$ , also das Symbol  $op \in F$  eines 0-stelligen Operators mit der Signatur  $op : \emptyset \rightarrow S_i$ , ist ein Ausdruck der Sorte  $S_i$ .
- ▶ Sind  $a_1, \dots, a_n$  Ausdrücke der Sorten  $S_1, \dots, S_n$  und  $op \in F$  ein Operator mit der Signatur  $op : S_1 \times \dots \times S_n \rightarrow S_{n+1}$  (wobei  $S_1, \dots, S_n, S_{n+1} \in S$ ), dann ist  $op(a_1, \dots, a_n)$  ein Ausdruck der Sorte  $S_{n+1}$ .



# Syntax von Ausdrücken

Beispiel: Sei  $S = \{\mathbb{N}_0\}$ ,  $V = V_{\mathbb{N}_0} = \{x, y\}$  und

$$F = \left\{ \begin{array}{ll} 1 : \emptyset \rightarrow \mathbb{N}_0 & \dots \text{ (alle Literale aus) } \mathbb{N}_0 \\ + : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 & / : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 \\ pow : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 & \dots \end{array} \right\}$$

Dann gilt

- ▶  $x$  ist ein Ausdruck,  $y$  ist ein Ausdruck ( $x, y \in V$ )
- ▶ 3 ist ein Ausdruck, 7 ist ein Ausdruck (3 und 7 sind Literale aus  $\mathbb{N}_0$ )
- ▶  $+(x, 3)$  ist ein Ausdruck,  $/(7, y)$  ist ein Ausdruck (jeweils ein 2-stelliger Operator aus  $F$  angewendet auf zwei Ausdrücke der entspr. Sorte)
- ▶  $pow(+(x, 3), /(7, y))$  ist ein Ausdruck.
- ▶  $+(3, pow(3, 7, x))$  ist dagegen KEIN Ausdruck (Warum?)

# Syntax von Ausdrücken

Als Schreibweisen für die Anwendung eines  $n$ -stelligen Operators  $op$  ( $n \geq 1$ ) gibt es wie für Funktionen

- ▶ Funktionsform  $op(a_1, \dots, a_n)$ , wobei auch die  $a_i$  in Funktionsform notiert sind
- ▶ Präfixform  $op a_1 \dots a_n$ , wobei auch die  $a_i$  in Präfixform notiert sind
- ▶ Postfixform  $a_1 \dots a_n op$ , wobei auch die  $a_i$  in Postfixform notiert sind
- ▶ Infixform  $(op a_1)$  für 1-stellige Operationen und  $(a_1 op a_2)$  für 2-stellige Operationen, wobei auch die  $a_i$  in Infixform notiert sind



# Syntax von Ausdrücken

## Beispiel

- ▶ Der Ausdruck  $+(x, 3)$  ist in Funktionsform notiert

Präfixform:  $+ x 3$

Postfixform:  $x 3 +$

Infixform:  $x + 3$

- ▶ Der Ausdruck  $pow(+ (x, 3), / (7, y))$  ist ebenfalls in Funktionsform notiert

Präfixform:  $pow + x 3 / 7 y$

Postfixform:  $x 3 + 7 y / pow$

Infixform:  $(x + 3) pow (7 / y)$

# Syntax von Ausdrücken

- ▶ 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*
- ▶ Beispiel:
  - ▶ Betrachten wir den Ausdruck  $(x \wedge z) \vee (y \wedge z)$  mit Variablen  $x, y, z \in V$  und Operationssymbolen  $\wedge, \vee \in F$  in Infixform
  - ▶ Wenn wir die Klammern weglassen, erhalten wir  $x \wedge z \vee y \wedge z$
  - ▶ Nun wäre auch eine andere Bindung möglich, z.B.  $x \wedge z \vee y \wedge z$ , also  $x \wedge ((z \vee y) \wedge z)$



# Syntax von Ausdrücken

- ▶ Eindeutigkeit ohne Klammerung kann man in der Infixform erreichen, indem man den Operatoren unterschiedliche *Bindungsstärken* (*Präzedenzen*) zuweist.
- ▶ Beispielsweise hat unter den Operatoren über  $\mathbb{B}$  die Negation  $\neg$  eine höhere Präzedenz als die Konjunktion  $\wedge$  und die Konjunktion hat wiederum eine höhere Präzedenz als die Disjunktion  $\vee$ .
- ▶ Damit erhalten wir auch für den Ausdruck  $x \wedge z \vee y \wedge z$  die ursprünglich gewünschte, nun *implizite* Klammerung  $(x \wedge z) \vee (y \wedge z)$



# Syntax von Ausdrücken

- ▶ Wenn Operatoren gleicher Bindungsstärke konkurrieren, muss 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

# Syntax von Ausdrücken

- ▶ Wir erlauben nun noch das *Überladen* von Operationssymbolen.
- ▶ Dazu erlauben wir, dass ein Operationssymbol  $op$  in der Menge  $F$  mehrmals vorkommen darf, fordern aber, dass jedes  $op$  jeweils eine *unterschiedliche* Signatur hat. Nur dadurch beschreiben die verschiedenen  $op$  Symbole jeweils unterschiedliche Operationen.
- ▶ Durch die Erfüllbarkeit einer Signatur kann man entscheiden, welcher Operator tatsächlich zu verwenden ist.

# Syntax von Ausdrücken

Beispiel: Sei  $S = \{\mathbb{N}_0, \mathbb{R}\}$  und

$$F = \left\{ \begin{array}{ll} + : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 & (1) \\ 0 : \emptyset \rightarrow \mathbb{N}_0 & \\ 1 : \emptyset \rightarrow \mathbb{N}_0 & \\ \dots & \end{array} \right. \quad \left\{ \begin{array}{l} + : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} & (2) \\ 0.0 : \emptyset \rightarrow \mathbb{R} & \\ 1.0 : \emptyset \rightarrow \mathbb{R} & \end{array} \right.$$

- ▶  $0 + 1$  erfüllt die Signatur von (1) und ist daher ein gültiger Ausdruck der Sorte  $\mathbb{N}_0$
- ▶  $0.0 + 1.0$  erfüllt die Signatur von (2) und ist daher ein gültiger Ausdruck der Sorte  $\mathbb{R}$
- ▶  $0.0 + 1$  erfüllt keine Signatur und ist daher kein gültiger Ausdruck

# Recap

- ▶ Die Syntax von Ausdrücken ist nun definiert, aber nach dem letzten Beispiel bleibt ein blöder Nachgeschmack:
  - ▶  $0.0 + 1$  erfüllt keine Signatur und ist daher kein gültiger Ausdruck
  - ▶ Ist das Verarbeiten von Daten unterschiedlicher Sorten damit unmöglich?
- ▶ Antwort: natürlich nicht, das wäre eine sehr herbe Einschränkung.
- ▶ Schließlich sind Sorten wie  $\mathbb{N}$ ,  $\mathbb{N}_0$ ,  $\mathbb{Z}$ ,  $\mathbb{R}$  miteinander verknüpft: Die Elemente in  $\mathbb{N}_0$  sind z.B. in  $\mathbb{R}$  enthalten, z.B. kann die Zahl  $2 \in \mathbb{N}_0$  auch als  $2.0 \in \mathbb{R}$  aufgefasst werden (und umgekehrt)).



## Kurzer Exkurs: Typcasting

- ▶ Die Einschränkung kann sehr leicht mit einem kleinen Trick beseitigt werden, der diese „Inklusion“ von Sorten durch eine entsprechende Abbildung formalisiert.
- ▶ Diese Abbildungen werden *Typcast* (deutsch *Sortenanpassung*) genannt.
- ▶ Für das (Typ-)Casting von  $\mathbb{N}_0$  nach  $\mathbb{R}$  kann man eine Abbildung

$$\text{NatNReal} : \mathbb{N}_0 \rightarrow \mathbb{R}$$

definieren mit der Abbildungsvorschrift

$$\text{NatNReal} : \text{natürliche Zahl } x \in \mathbb{N}_0 \mapsto \text{reele Zahl } x \in \mathbb{R}.$$

## Kurzer Exkurs: Typcasting

- ▶ Geht man davon aus, dass man ein entsprechendes Operationssymbol  $NatNReal$  zur Verfügung hat, kann man im obigen Beispiel nun einen gültigen Ausdruck mit  $0.0 + NatNReal(1)$  bilden.
- ▶ Analoge Typcasting-Operationen kann man entsprechend für die Anpassungen zwischen den anderen Mengen bilden (soweit sinnvoll).
- ▶ Typcasting-Operationen lösen also das (zunächst rein syntaktische) Problem der Erfüllbarkeit von Signaturen, wenn man verschiedene Sorten gleichzeitig verarbeiten will.



# Kurzer Exkurs: Typcasting

## Achtung:

- ▶ Wir werden im Folgenden immer implizit (sinnvolle) Typcasting-Operationen als gegeben voraussetzen, und ggf. auch in Ausdrücken weglassen (z.B. schreiben wir statt  $0.0 + \text{NatNReal}(1)$  wieder  $0.0 + 1$  und betrachten dies als gültigen Ausdruck mit *implizitem* Typcast).
- ▶ Typcasting ist in unterschiedlichen Programmiersprachen unterschiedlich gelöst (auch wenn das Prinzip dasselbe ist), denn es gibt ein paar Feinheiten und Stolperfallen, die wir uns später genauer anschauen werden (wenn wir Typcasting in Java besprechen).

