Einführung in die Programmierung

Teil 4: Grundlagen der Programmierung

Prof. Dr. Peer Kröger,
Florian Richter, Michael Fromm

Wintersemester 2018/2019

Übersicht i

1. Sorten und abstrakte Datentypen
2. Ausdrücke
2.1 Syntax
2.2 Semantik
2.3 Ausdrücke in Java
3. Funktionale Algorithmen
4. Variablen, Anweisungen, Prozeduren
5. Prozeduraufrufe
6. Variablen, Anweisungen und Prozeduren in Java
7. Bedingte Anweisungen und Iteration
8. Verzweigung/Iteration in Java
9. Strukturierung von Programmen

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), dessen Syntax wir allerdings meist nicht vollständig 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.

1. Sorten und abstrakte Datentypen

2. Ausdrücke

3. Funktionale Algorithmen

4. Variablen, Anweisungen, Prozeduren

5. Prozeduraufrufe

6. Variablen, Anweisungen und Prozeduren in Java

Kapitel 1: Sorten und abstrakte Datentypen ii

7. Bedingte Anweisungen und Iteration

8. Verzweigung/Iteration in Java

9. Strukturierung von Programmen

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.

Wir formalisieren diese Unterscheidung wie folgt:

Definition (Sorte/(Daten-)Typ)

Eine nicht-leere Menge von Objekten heißt Sorte oder auch (Daten-) Typ.

Bemerkung:
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.

Beispiele

- Die Menge der natürlichen Zahlen $\mathbb{N}_0$
- Die Menge der boolschen Werte $\mathbb{B}$
- Die Menge der reelen Zahlen $\mathbb{R}$
- Die Menge der druckbaren Zeichen *CHAR*
- Die Menge der Farben *Farbe = {rot, grün, blau, gelb, . . .}*
- Die Menge der Texte *STRING* als die Menge aller Folgen über *CHAR*, d.h. $CHAR^*$
- . . .

Auf Basis der Sorten formalisieren wir nun die Idee der Grundoperationen, die wir als gegeben voraussetzen.

Definition (Modul)

Ein Modul ist eine nicht-leere, endliche Menge $\{S_1, \ldots, S_n\}$ von Sorten mit einer nicht-leeren, endlichen Menge von Operationen über diesen Sorten, d.h. Operation mit Funktionalitäten

$$S_{i_1} \times S_{i_2} \times \ldots \times S_{i_m} \rightarrow S_{i_{m-1}}$$

mit $i_1, \ldots, i_{m+1} \in \{1, \ldots n\}$.

Module

- Bemerkungen:
  - 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.
  - Das Modul-Konzept ist in vielen 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.

Beispiele

- Für die Sorte $\mathbb{B}$ liegt es nahe, die Operationen $\neg$, $\vee$ und $\wedge$ als Grundoperationen vorauszusetzen.
- Wenn wir die beiden Elemente der Menge noch als 0-stellige Operationen auffassen, besteht das Modul (das wir z.B. BOOLEAN nennen könnten) also aus:
  - der Sorte $\mathbb{B}$
  - den Operationen:

$$\begin{aligned}
&\mathit{TRUE} : \emptyset \to \mathbb{B} \\
&\mathit{FALSE} : \emptyset \to \mathbb{B} \\
&\neg : \mathbb{B} \to \mathbb{B} \\
&\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
&\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}
\end{aligned}$$

Module: Beispiele

- Wir notieren dies (hoffentlich möglichst intuitiv) in einer *Modulvereinbarung*:

MODULE BOOLEAN
  SORTS $\mathbb{B}$
  OPS $TRUE : \emptyset \to \mathbb{B}$,
  $FALSE : \emptyset \to \mathbb{B}$,
  $\neg : \mathbb{B} \to \mathbb{B}$,
  $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$,
  $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- Damit machen wir diese Operationen (mit entsprechenden Signaturen) bekannt, d.h. wir dürfen sie ab jetzt in unseren Algorithmen verwenden.

Module: Beispiele

Ganz formal können wir die Syntax einer Modulvereinbarung natürlich entsprechend definieren:

Syntaxregel (Modulvereinbarung)

$$\langle \textit{Modulvereinbarung} \rangle ::= \texttt{MODULE} \langle \textit{ModulName} \rangle \; \texttt{SORTS} \; \langle \textit{Sorte} \rangle \; \{, \langle \textit{Sorte} \rangle\}^* \; \texttt{OPS} \; \langle \textit{OpVereinbarung} \rangle$$

$$\langle \textit{ModulName} \rangle ::= \{\mathtt{A}|\mathtt{B}|\ldots|\mathtt{Z}\}^+$$

$$\langle \textit{Sorte} \rangle ::= \mathbb{B}|\mathbb{N}|\mathbb{N}_0|\mathbb{Z}|\mathbb{R}|\textit{CHAR}|\ldots$$

$$\langle \textit{OpVereinbarung} \rangle ::= \langle \textit{Operator} \rangle : \langle \textit{Signatur} \rangle \{, \langle \textit{Operator} \rangle : \langle \textit{Signatur} \rangle\}^*$$

Die Vereinbarung von $\langle \textit{Operator} \rangle$ und Signatur $\langle \textit{Signatur} \rangle$ sei dem Leser überlassen.

Module: Beispiele

- Ein ähnliches Modul kann man für die Grundoperationen über $\mathbb{N}_0$ definieren:

MODULE NAT

SORTS $\mathbb{N}_0, \mathbb{B}$

OPS
$0 : \emptyset \to \mathbb{N}_0$, $1 : \emptyset \to \mathbb{N}_0$, $\ldots$ (alle Elemente aus $\mathbb{N}_0$)

$+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, $- : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$,

$\cdot : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$,

$\mathit{DIV} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, $\mathit{MOD} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$,

$= : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$, $\neq : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$,

$\leq : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$, $< : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$,

$\geq : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$, $> : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$

Module: Beispiele

- 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

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

  (das einstellige Vorzeichen Minus) hinzufügen.
- Entsprechendes gilt für die Grundoperationen aus $\mathbb{R}$ und *CHAR*, die wir hier nicht genauer ausführen.

Module: Beispiele

- Entsprechendes gilt für die Grundoperationen aus $\mathbb{R}$ und *CHAR*, die wir hier nicht genauer ausführen sondern nur andeuten:

MODULE INTEGER
  SORTS $\mathbb{Z}, \mathbb{B}, \ldots$
  OPS $\ldots$

MODULE CHARACTER
  SORTS *CHAR*, $\mathbb{B}, \ldots$
  OPS $\ldots$

MODULE REAL
  SORTS $\mathbb{R}, \mathbb{B}, \ldots$
  OPS $\ldots$

Sorten und Module

- Damit haben wir nun also Grundoperationen für unsere Sorten durch entsprechende 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.
- Das Modul-Konzept ist übrigens nicht auf einfach Datentypen beschränkt, wie wir gleich sehen werden, könnten wir theoretisch beliebige Module angeben (und damit Operationen über beliebige Sorten).

- Es gibt verschiedene Möglichkeit, 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 bleiben 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.

Sorten und Module

- In Java (und vielen anderen Sprachen) gibt es quasi beides:
  - Viele “Module” (und damit Operationen) sind bereits implementiert und stehen bereit (als automatisch in die Sprache “eingebaut”, oder innerhalb der Programmbibliothek, die bei Java SEHR umfangreich ist) — deren Semantik ist allerdings meist nicht sauber axiomatisch definiert, hoffentlich immerhin ausführlich kommentiert[1]
  - Es besteht aber auch die Möglichkeit, eigene Module zu schreiben und die Semantik Operationen durch einen Algorithmus zu spezifizieren (Zum Glück muss man dabei nicht auf Maschinen-Befehl-Ebene programmieren).

[1] Erinnern Sie sich? Kommentare sind wichtig!!!

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.

Beispiel

Ein Modul FARBE stellt Grundoperationen über der Sorte $\mathit{Farbe} = \{\mathit{blau}, \mathit{gruen}, \mathit{rot}, \mathit{gelb}, \ldots\}$ zur Verfügung:

MODULE FARBE

SORTS $\mathit{Farbe}, \mathbb{R}, \ldots$

OPS $\mathit{blau} : \emptyset \to \mathit{Farbe}$,

$\mathit{gruen} : \emptyset \to \mathit{Farbe}$,

$\mathit{rot} : \emptyset \to \mathit{Farbe}$,

$\mathit{gelb} : \emptyset \to \mathit{Farbe}$,

$\ldots$ (weitere Farben)

$\mathit{mix} : \mathit{Farbe} \times \mathit{Farbe} \to \mathit{Farbe}$,

$\mathit{helligkeit} : \mathit{Farbe} \to \mathbb{R}$,

$\ldots$

Abstrakte Datentypen

- Wie gesagt, “irgendjemand” muss diese Operationen auch realisieren.
- Gehen wir davon aus, dass dies der Fall ist, brauchen wir die Realisierungsdetails nicht mehr verstehen.
- Wir müssen nur verstehen, was die Operationen tun, aber nicht wie sie es tun.
- Mit dem Modulkonzept haben wir also zunächst die Möglichkeit, eine Menge von Sorten mit Operationen auf diesen Sorten als “gegeben” zu definiert (wer auch immer diese für uns realisiert).

Road Map

- Nochmal der Hinweis: auch die einzelnen Elemente (Objekte) 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 Algorithmen formal definieren.

1. Sorten und abstrakte Datentypen

2. Ausdrücke

2.1 Syntax

2.2 Semantik

2.3 Ausdrücke in Java

3. Funktionale Algorithmen

4. Variablen, Anweisungen, Prozeduren

Kapitel 2: Ausdrücke ii

5. Prozeduraufrufe

6. Variablen, Anweisungen und Prozeduren in Java

7. Bedingte Anweisungen und Iteration

8. Verzweigung/Iteration in Java

9. Strukturierung von Programmen

- 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, aber auch, um Daten durch Verarbeitungsschritte zu verändern.
- Intuitiv sind Ausdrücke nichts anderes als Zeichenreihen, die Operationssymbole[2] und Variablen (z.B. die Eingabevariablen eines Algorithmus) enthalten (also so etwas wie $DIV(100 - r, 5)$).

[2] Zur Erinnerung: 0-stellige Operationen sind Konstanten, also Elemente aus dem Bildbereich

- Wir bleiben zunächst informell und betrachten folgende Zeichenreihen:
  - $DIV(100 - r, 5)$
  - $a + 5$
  - blau $-$ gelb
- Diese Zeichenreihen können wir als Folgen von Symbolen begreifen, die u.a. Operationssymbole darstellen (z.B. die Funktion[3] *DIV* in Funktionsschreibweise, die Funktion + in Infixschreibweise oder die Konstante 5).
- Eine solche Folge von Symbolen heißt *Ausdruck* oder *Term*.

[3] Wir verwenden teilweise die Begriffe „Funktion“ und „Operation“ als Synonyme

- 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 definieren im folgenden zuerst die Syntax und anschließend die Semantik von Ausdrücken. Abschließend betrachten wir Ausdrücke in Java.

1. Sorten und abstrakte Datentypen

2. Ausdrücke

2.1 Syntax

2.2 Semantik

2.3 Ausdrücke in Java

3. Funktionale Algorithmen

4. Variablen, Anweisungen, Prozeduren

5. Prozeduraufrufe

Syntax von Ausdrücken (informell)

- Informell beschreibt der Ausdruck $a + 5$ die Addition einer Variablen mit einer Konstanten, der Ausdruck $mix(blau, gelb)$ (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 hat jeder Ausdruck grundsätzlich eine Sorte.

Syntax von Ausdrücken (informell)

- Ausdrücke werden zusammengesetzt aus *Operatoren* (wie z.B. 1 oder +) 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 $\emptyset \to \mathbb{N}_0$ (die genaue Bedeutung von 5 ist uns übrigens zunächst egal, wir sind nur an Syntax interessiert).

Syntax von Ausdrücken (informell)

- ACHTUNG Terminologie:
  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 [4].

[4] Dies ist nicht in allen Programmiersprachen so. Aber u.a. Java, als vollständig typisierte Sprache, verlangt die Typangabe von Variablen

Syntax von Ausdrücken (informell)

- 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 $D_1 \times \ldots \times D_n$ kann entsprechend auf ein n-Tupel aus Ausdrücken $D_1 \times \ldots \times D_n$ angewendet werden.
  - Die n Komponenten des n-Tupels heißen Operanden des Operators, der auf das n-Tupel angewendet wird.
  - Ein Operator mit Bildbereich T bildet einen Ausdruck einer Sorte T.
  - Analoges gilt für Variablen vom Typ T.

Beispiel

- Der Operator $<: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$ kann auf ein Tupel $\mathbb{N}_0 \times \mathbb{N}_0$ angewendet werden und ergibt ein Literal (und damit wieder einen Ausdruck) der Sorte $\mathbb{B}$ (das 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 (jetzt wird es formal)

- 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/haben einen eindeutigen Typ).

Definition (Menge der Ausdrücke/Terme (Syntax))

Gegeben seien S, F und V wie oben. Die Menge der Ausdrücke/Terme ist wie folgt induktiv definiert:

- *Eine Variable $v \in V_{S_i}$ ist ein Ausdruck der Sorte $S_i \in S$.*
- *Ein Literal $l \in F$ der Sorte $S_i \in S$, also der Operator mit der Signatur $l : \emptyset \to S_i$, 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 $op : S_1 \times \ldots \times S_n \to S_{n+1}$ (wobei $S_1, \ldots, S_n, S_{n+1} \in S$), dann ist $op(a_1, \ldots, a_n)$ ein Ausdruck der Sorte S_{n+1}.*

Syntax von Ausdrücken

- Wir bezeichnen die Menge der Terme der Sorte S_i als $\mathcal{T}_{S_i}$ und die Menge aller Terme über die Sorten $S = \bigcup_{1 \leq i \leq n} S_i$ als $\mathcal{T}_S = \mathcal{T}_{S_1} \cup \ldots \cup \mathcal{T}_{S_n}$
- Ergibt sich S aus dem Kontext, schreiben wir einfach $\mathcal{T}$.
- Die induktive Definition nochmal kürzer aufgeschrieben:
  - Ist $v \in V_{S_i}$, so ist $v \in \mathcal{T}_{S_i}$.
  - Ist $I \in F$ mit $I : \emptyset \to S_i$, so ist $I \in \mathcal{T}_{S_i}$.
  - Sind $a_1 \in \mathcal{T}_{S_i}, \ldots, a_n \in \mathcal{T}_{S_n}$ und $op \in F$ mit $op : S_1 \times \ldots \times S_n \to S_{n+1}$, so ist $op(a_1, \ldots, a_n) \in \mathcal{T}_{S_{n+1}}$.
  - $\mathcal{T}_S = \bigcup_{S_i \in S} S_i$

Beispiel

Sei

- $S = \{\mathbb{N}_0\}$
- $V = V_{\mathbb{N}_0} = \{x, y\}$
- $F = \{1 : \emptyset \to \mathbb{N}_0 \ldots$ (alle Literale aus $\mathbb{N}_0$)
  $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \qquad / : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
  $pow : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \ldots\}$

Dann gilt:

- x ist ein Ausdruck, y ist ein Ausdruck (da $x, y \in V$)
- 3 ist ein Ausdruck, 7 ist ein Ausdruck (3 und 7 sind Literale aus $\mathbb{N}_0$)

Syntax von Ausdrücken

Weiterhin gilt:

- $+(x, 3)$ ist ein Audruck, $/(7, y)$ 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: Schreibweisen

Für die Anwendung eines n-stelligen Operators op ($n \geq 1$) lassen wir wie für Funktionen folgende Schreibweisen zu:

- Funktionsform $op(a_1, \ldots, a_n)$, wobei auch die a_i in Funktionsform notiert sind
- Präfixform $op\, a_1 \ldots a_n$, wobei auch die a_i in Präfixform notiert sind
- Postfixform $a_1 \ldots a_n\, op$, 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

Beispiele

- 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. $x \wedge \underbrace{z \vee y}_{\wedge z}$, also $x \wedge ((z \vee y) \wedge z)$

Syntax von Ausdrücken

- Eindeutigkeit ohne Klammerung kann man in der Infixform auch erreichen, allerdings nur indem man den Operatoren unterschiedliche *Bindungsstärken* (*Präzedenzen*) zuweist.
- Das Prinzip kennen Sie seit der Grundschule, z.B. als “Punkt-vor-Strich”-Regel, die auch in den meisten Programmiersprachen gilt.
- Es gelten häufig auch weitere Präzedenzen bei anderen Operationen.

Syntax von Ausdrücken

- Die meisten Programmiersprachen kennen z.B. auch Präzedenzen bei den Operatoren über $\mathbb{B}$:
  die Negation hat $\neg$ 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 $(x \wedge z) \vee (y \wedge z)$
- Achtung: im Zweifelsfall empfiehlt es sich, trotzdem Klammern zu setzen — das vermeidet dumme Fehler.

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 diese gleichen Symbole jeweils unterschiedliche Operationen.
- Durch die *Erfüllbarkeit* einer Signatur kann man entscheiden, welcher Operator tatsächlich zu verwenden ist.

Beispiele

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

$$F = \{ \quad + : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \quad (1) \quad + : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \quad (2)$$
$$0 : \emptyset \to \mathbb{N}_0 \qquad 0.0 : \emptyset \to \mathbb{R}$$
$$1 : \emptyset \to \mathbb{N}_0 \qquad 1.0 : \emptyset \to \mathbb{R} \quad \ldots\}$$

- 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

Alles gut?

- 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

$$\mathit{NatToReal} : \mathbb{N}_0 \to \mathbb{R}$$

definieren mit der Abbildungsvorschrift

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

Kurzer Exkurs: Typcasting

- Geht man davon aus, dass man ein entsprechendes Operationssymbol *NatToReal* zur Verfügung hat, kann man im obigen Beispiel nun einen gültigen Ausdruck mit 0.0 + *NatToReal*(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

- Wir werden im Folgenden immer (sinnvolle) Typcasting-Operationen als gegeben voraussetzen, und ggf. auch in Ausdrücken weglassen (z.B. schreiben wir statt $0.0 + \mathit{NatToReal}(1)$ 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), da sie auch etwas mit der Semantik zu tun haben.