Abschnitt 5: Grundlagen der funktionalen und imperativen Programmierung

5. Grundlagen der funktionalen und imperativen Programmierung
5.1 Sorten und abstrakte Datentypen
5.2 Ausdrücke
5.3 Ausdrücke in Java
5.4 EXKURS: Funktionale Algorithmen
5.5 Anweisungen
5.6 Imperative Algorithmen
5.7 Reihungen und Zeichenketten
5.8 Zusammenfassung: Imperative und funktionale Algorithmen in Java

Überblick

- Das Verhalten von Objekten in der oo Programmierung wird meist durch imperative und/oder funktionale Algorithmen spezifiziert (in Java: hauptsächlich imperativ)
- Imperative und funktionale Konzepte sind daher eine wesentliche Basis für die oo Programmierung.
- In diesem Kapitel behandeln wir zunächst die wichtigsten Konzepte der imperativen Programmierung mit einem kurzen Exkurs in die funktionale Programmierung.
- Wir folgen dabei (soweit möglich) der Java-Syntax
- In Java sind imperative Programme meist *statische* Elemente (mit dem Schlüsselwort `static` versehen).

Überblick

5. Grundlagen der funktionalen und imperativen Programmierung
5.1 Sorten und abstrakte Datentypen
5.2 Ausdrücke
5.3 Ausdrücke in Java
5.4 EXKURS: Funktionale Algorithmen
5.5 Anweisungen
5.6 Imperative Algorithmen
5.7 Reihungen und Zeichenketten
5.8 Zusammenfassung: Imperative und funktionale Algorithmen in Java

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*[4]
- Beispiele für Sorten:
  - 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 $CHAR$
  - ...

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

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

  (dabei sind $m = 0$-stellige Operationen explizit zugelassen)
- 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. orientiert sich an der konkreten Programmiersprache):
    - $0 : \emptyset \to \mathbb{N}_0$
    - $1 : \emptyset \to \mathbb{N}_0$
    - $\ldots$ (alle Werte aus $\mathbb{N}_0$, oder besser: $successor : \mathbb{N}_0 \to \mathbb{N}_0$)
    - $+ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    - $- : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    - $\cdot : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    - $DIV : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    - $MOD : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$
    - $=: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$
    - $\neq: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$
    - $\leq: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$
    - $<: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$
    - $\geq: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$
    - $>: \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{B}$

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

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

  (das einstellige Vorzeichen Minus) hinzufügen.
- Entsprechendes gilt für die Grundoperationen aus $\mathbb{R}$ und $CHAR$.

Sorten und Module

- In der bisherigen Disskusion haben wir 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 genau Definition, was diese Operationen tun, wurde bisher nicht angegeben.
- Es gibt verschiedene Möglichkeit, dies zu tun, z.B.
  - könnte man diese Definitionen explizit angeben (was vermutlich einem Algorithmus sehr nahe kommen wird!!!)
  - könnte man die Operationen axiomatisch charakterisieren, die (Realisierungs-)Details der Operationsdefinitionen bleibt damit vor uns verborgen

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.
- Module, die die Realisierungsdetails der Operationen vor dem Benutzer verbergen, heißen auch *abstrakter Datentyp*.
- 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.

Roadmap

- Mit dem Modulkonzept haben wir also zunächst eine Menge von Sorten mit Operationen auf diesen Sorten als gegeben definiert.
- Wir werden in diesem Kapitel 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.
- Wir konzentrieren uns dabei zunächst auf funktionale Konzepte (insbesondere soweit sie für Java relevant sind).

Überblick

5. Grundlagen der funktionalen und imperativen Programmierung
5.1 Sorten und abstrakte Datentypen
5.2 Ausdrücke
5.3 Ausdrücke in Java
5.4 EXKURS: Funktionale Algorithmen
5.5 Anweisungen
5.6 Imperative Algorithmen
5.7 Reihungen und Zeichenketten
5.8 Zusammenfassung: Imperative und funktionale Algorithmen in Java

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 Zeichenketten, die Funktionssymbole[5] und Variablen (z.B. die Eingabevariablen eines Algorithmus) enthalten (also so etwas wie $DIV(100 - r, 5)$).

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

Einführung

- Betrachten wir folgende Zeichenketten:
  - $DIV(100 - r, 5)$
  - $a + 5$
  - 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.

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, die Bedeutung die Semantik.
- Wir können eine korrekte Struktur beschreiben oder überprüfen, ohne etwas über die 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*.

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.
- Wie bei Funktionen stehen 0-stellige Operatoren für Konstanten (also Elemente) der entsprechenden Sorte (des Bildbereichs des Operators), z.B. 5 für die konstante Funktion mit der Signatur $\emptyset \to \mathbb{N}_0$)
- ACHTUNG: wir nennen diese Konstanten ab jetzt *Literale*, da der Begriff *Konstante* später in einer anderen Bedeutung verwendet wird.
- Variablen sind Namen für Ausdrücke. Jede Variable hat eine Sorte.

Syntax von Ausdrücken

- Ein 1-stelliger Operator mit Definitionsbereich D kann offenbar 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 $D_1 \times \ldots \times D_n$ von Ausdrücken 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.
- 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 $\mathbb{B}$

Syntax von Ausdrücken

- Eine Operatorbeschreibung enthält also 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.
- Der Operator $<$ verknüpft z.B. zwei Ausdrücke der Sorte $\mathbb{N}_0$ (seine Operanden) und bildet einen Ausdruck der Sorte $\mathbb{B}$.
- Ausdrücke gehören damit immer einer Sorte an (haben einen Typ).

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 $\emptyset \rightarrow S_i$, 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 $S_1 \times \ldots \times S_n \rightarrow S_{n+1}$ (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

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

$$F = \{ \quad + : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$
$$/ : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$
$$pow : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \quad \}$$

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 Audruck, $/(7, y)$ ist ein Ausdruck
  (jeweils ein zweistelliger Operator aus F angewendet auf zwei Ausdrücke der entsprechenden 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 $op(a_1, \ldots, a_n)$, 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

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. $x \wedge \underbrace{z \vee y}_{\underbrace{}} \wedge z$, 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, jeweils aber mit einer unterschiedlichen Signatur. Dadurch beschreibt op 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

$$
\begin{aligned}
F = \{ \quad & + : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 && (1)\\
& 0 : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
& 1 : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\
& + : \mathbb{R} \times \mathbb{R} \to \mathbb{R} && (2)\\
& 0.0 : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
& 1.0 : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \quad \ldots\}
\end{aligned}
$$

- $0 + 1$ 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

Semantik von Ausdrücken

- 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 (zur Erinnerung: Variablen sind letztlich Namen für Ausdrücke)
- Dann können alle im Ausdruck vorkommenden Operationen ausgeführt werden
- Schließlich kann man für einen Ausdruck einen Wert einsetzen

Semantik von Ausdrücken

- Beispiel: Um die Bedeutung des Ausdrucks $2 + 5$ zu definieren, müssen die Operationen 2, $+$ und 5 definiert sein
- Wir können z.B. definieren:
  - der Operator 2 steht für die natürliche Zahl *zwei*
  - der Operator 5 steht für die natürliche Zahl *fünf*
  - der Operator $+$ steht für die *arithmetische Addition* zweier natürlicher Zahlen
- Dadurch erhalten wir durch Anwendung der vereinbarten Funktionen als Wert (Bedeutung) des Ausdrucks die Zahl *sieben*
- Wir sagen zur Bedeutung eines Ausdrucks auch *Interpretation*; wir interpretieren z.B. den Ausdruck $2 + 5$ unter oben stehenden Vereinbarungen mit dem Wert *sieben*

Semantik von Ausdrücken

- 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 σ aus einer Variablen v und einem Ausdruck a, geschrieben $\sigma = [v/a]$, ansehen
- Man sagt auch: v wird durch a *substituiert* oder *ersetzt*
- Wir fordern dabei, dass v und a zur selben Sorte gehören müssen
- Festzustellen, ob zwei Ausdrücke zur selben Sorte gehören, ist i.Ü. nicht trivial und wird unter dem Thema *Unifikation* studiert (nicht in dieser VL)

Semantik von Ausdrücken

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

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

- Letztlich ist die Substitution σ eine Abbildung, die rekursiv (über den induktiven Aufbau von u) definiert werden kann:
  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. $u[v/t] = op(u_1[v/t], u_2[v/t], \ldots, u_n[v/t]))$, falls u die Anwendung eines n-stelligen Operators op auf die Ausdrücke $u_1, u_2, \ldots, u_n$ ist
- Bemerkung: das Ergebnis einer Substitution ist i.A. wieder ein Ausdruck, typischerweise aber nicht notwendigerweise ein Literal aus der entsprechenden Sorte, das entsprechend interpretiert werden kann.

Semantik von Ausdrücken

Beispiel:

Sei u der Ausdruck $x + x - 1$ in Infixschreibweise (linksassoziativ bindende Operationen) bzw. $-(+(x,x),1)$ in Funktionsschreibweise

Wir wollen die Variable x durch $2 \cdot b$ ersetzen, d.h. $\sigma = [x/2 \cdot b]$

Wir erhalten (in Funktionsschreibweise notiert)

$$\begin{aligned}
\underbrace{-(+(x,x),1)}_{u}\ \underbrace{[x/2 \cdot b]}_{\sigma} &= -(+(x,x)[x/2 \cdot b], 1[x/2 \cdot b]) \\
&= -(+(x,x)[x/2 \cdot b], 1) \\
&= -(+(x[x/2 \cdot b], x[x/2 \cdot b]), 1) \\
&= -(+(2 \cdot b), (2 \cdot b)), 1)
\end{aligned}$$

d.h., in Infixform notiert wird aus $x + x - 1$ durch die Substitution

$$2 \cdot b + (2 \cdot b) - 1$$

Semantik von Ausdrücken

Noch ein Beispiel:

Sei u wieder der Ausdruck $(x + x) - 1$ in Infixschreibweise

Wir wollen die Variable x durch 5 ersetzen, d.h. $\sigma = [x/5]$

Wir erhalten (in Infixschreibweise notiert)

$$\begin{aligned}((x + x) - 1)[x/5] &= (x + x)[x/5] - 1[x/5] \\ &= (x[x/5] + x[x/5]) - 1 \\ &= 5 + 5 - 1\end{aligned}$$

Wenn wir die Operationssymbole entsprechend als Literale aus $\mathbb{N}_0$ bzw. Addition und Subtraktion über $\mathbb{N}_0$ interpretieren, erhalten wir als Wert für $u[x/5]$ die Zahl $9 \in \mathbb{N}$