

Semantik von Ausdrücken

- ▶ Da nun die Syntax von Ausdrücken eindeutig definiert ist, widmen wir uns nun (zunächst wieder informell) deren Semantik.
- ▶ Um die *Bedeutung* (*Semantik*) eines Ausdrucks festzulegen, müssen einerseits die Funktionen, die von den Operatoren bezeichnet werden (einschl. die Literale), definiert (d.h. interpretierbar) sein.
- ▶ Andererseits müssen alle Variablen ersetzt werden können durch den von ihnen bezeichneten Ausdruck (zur Erinnerung: Variablen sind letztlich Namen für Ausdrücke), der selbst eine Bedeutung hat.
- ▶ Dann können alle im Ausdruck vorkommenden Operationen ausgeführt werden.
- ▶ Schließlich kann man für einen Ausdruck einen Wert einsetzen (bzw. man erhält als Ergebnis ein Literal, das entsprechend interpretiert wird).

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

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

d.h. im Ausdruck u wird die Variable v durch t ersetzt

- ▶ 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. $u[v/t] = op(u_1[v/t], u_2[v/t], \dots, u_n[v/t])$, falls u die Anwendung eines n -stelligeren Operators op auf die Ausdrücke u_1, u_2, \dots, u_n ist
- ▶ Bemerkung: das Ergebnis einer Substitution ist i.A. wieder ein Ausdruck, aber nicht notwendigerweise ein Literal aus der entsprechenden Sorte.

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' der Ausdruck $u[x/2 \cdot b]$, d.h. $(2 \cdot b + 2 \cdot b) - 1$ in Infixschreibweise

Wir wollen die Variable b durch 3 ersetzen, d.h. $\sigma = [b/3]$

Wir erhalten (in Infixschreibweise notiert)

$$\begin{aligned}((2 \cdot b + 2 \cdot b) - 1)[b/3] &= (2 \cdot b + 2 \cdot b)[b/3] - 1[b/3] \\ &= ((2 \cdot b)[b/3] + (2 \cdot b)[b/3]) - 1 \\ &= (2[b/3] \cdot b[b/3] + 2[b/3] \cdot b[b/3]) - 1 \\ &= (2 \cdot 3 + 2 \cdot 3) - 1\end{aligned}$$

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 $u'[b/3]$ die Zahl $11 \in \mathbb{N}_0$

Überblick

5. Grundlagen der funktionalen Programmierung

5.1 Sorten und abstrakte Datentypen

5.2 Ausdrücke

5.3 Ausdrücke in Java

5.4 Funktionale Algorithmen



Allgemeines

- ▶ Für eine Menge an Sorten S , eine Menge von Variablen V dieser Sorten und eine Menge von Operationen F über diesen Sorten können wir nun Ausdrücke bilden (syntaktisch) und deren Semantik bestimmen (wenn die Semantik der Operationssymbole klar ist)
- ▶ Typischerweise stellt jede höhere Programmiersprache gewisse *Grunddatentypen* als Sorten S zur Verfügung (über die Ausdrücke gebildet werden können)
- ▶ Zusätzlich werden (über entsprechende Module) auch gewisse *Grundoperationen* F bereitgestellt, also eine Menge von (teilweise überladenen) Operationssymbolen inklusive der Literalen der Sorten



Allgemeines

- ▶ Die Semantik dieser Operationen ist durch den zugrundeliegenden Algorithmus zur Berechnung der entsprechenden Funktion definiert
- ▶ In den meisten Programmiersprachen ist dies aber vor dem Benutzer verborgen, und es erfolgt auch keine axiomatische Spezifikation, sondern (wenn überhaupt) eine textuelle Beschreibung

Sorten in Java

- ▶ Java stellt grundlegende Sorten (Datentypen) (auch *atomare* oder *primitive* Typen genannt) bereit: für \mathbb{B} , *CHAR*, eine Teilmenge von \mathbb{Z} und für eine Teilmenge von \mathbb{R} (aber keinen eigenen Grunddatentyp für \mathbb{N})
- ▶ Die Werte der primitiven Typen werden intern binär dargestellt.
- ▶ Wie bereits diskutiert unterscheiden sich die Datentypen u.a. in der Anzahl der Bits (auch Länge), die für ihre Darstellung verwendet werden und die Einfluss auf den Wertebereich des Typs haben
- ▶ die Datentypen für eine Teilmenge von \mathbb{R} basieren auf der Menge der Gleitpunktzahlen



Sorten in Java

► Bemerkungen:

- Als objektorientierte Sprache bietet Java zusätzlich die Möglichkeit, benutzerdefinierte Datentypen zu definieren
- Diese Möglichkeiten lernen wir im Teil über objektorientierte Modellierung genauer kennen
- die Operationen dieser Datentypen können explizit durch einen entsprechenden Algorithmus angegeben werden und dann „beliebig“ als *Black Box* verwendet werden

Sorten in Java

Überblick: Atomare Datentypen (Sorten) in Java:

Name	Länge	Wertebereich
boolean	1 Byte	Wahrheitswerte, Literale: true und false
char	2 Byte	Literale: alle Unicodezeichen in Hochkommata, z.B. 'A'
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)



Sorten in Java

- ▶ Wie erwähnt, stellt Java zu diesen atomaren Datentypen auch entsprechende abstrakte Module mit Grundoperationen bereit
- ▶ Die folgenden Folien stellen einige Besonderheiten dieser Operationen vor, bietet aber **keinen** vollständigen Überblick!
- ▶ Machen Sie sich selbstständig mit allen zur Verfügung stehenden Grundoperationen vertraut, diese finden Sie in den meisten einschlägigen Java-Büchern

Grundtypen und -operationen in Java

Für den Typ `boolean` gelten u.a. folgende Besonderheiten:

- ▶ Neben dem logischen UND (\wedge), das mit `&` dargestellt wird, gibt es das *sequentielle* UND:

`a && b` ergibt zunächst wie gewohnt `true`, wenn sowohl `a` als auch `b` wahr ist.

Achtung: Ist `a` bereits falsch, wird `false` zurückgegeben und `b` **nicht** ausgewertet!

- ▶ Analog: Neben dem logischen ODER (\vee), das mit `|` dargestellt wird, gibt es das *sequentielle* ODER:

`a || b` ergibt `true`, wenn mindestens einer der beiden Ausdrücke `a` oder `b` wahr ist.

Ist bereits `a` wahr, so wird `true` zurückgegeben und `b` nicht ausgewertet.

Grundtypen und -operationen in Java

- ▶ Zusätzlich zur logischen Negation `!` gibt es für den Typ `boolean` noch das exklusive ODER (XOR) `^`:
`a ^ b` ergibt `true`, wenn beide Ausdrücke `a` und `b` einen unterschiedlichen Wahrheitswert haben.

Ganze Zahlen in Java

- ▶ Java hat vier Datentypen für ganze (vorzeichenbehaftete) Zahlen: `byte` (Länge: 8 Bit), `short` (Länge: 16 Bit), `int` (Länge: 32 Bit) und `long` (Länge: 64 Bit).
- ▶ Werte (*Konstanten*) können geschrieben werden in
 - ▶ Dezimalform: bestehen aus den Ziffern 0, ..., 9
 - ▶ Binärform: beginnen mit dem Präfix `0b` und bestehen aus Ziffern 0 und 1
 - ▶ 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 `-`.



Inkrement-Operationen in Java

- ▶ Inkrement-Operationen haben die Signatur

$$S \rightarrow S$$

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

- ▶ Operationen:

Operator	Bezeichnung	Bedeutung
++	Präinkrement	++a ergibt a+1 und erhöht a um 1
++	Postinkrement	a++ ergibt a und erhöht a um 1
--	Prädecrement	--a ergibt a-1 und verringert a um 1
--	Postdecrement	a-- ergibt a und verringert a um 1

- ▶ **Achtung:**

- ▶ Diese Operatoren haben offenbar *Nebeneffekte*
- ▶ Der Ausdruck `a++` hat einen Wert (in Abhängigkeit des Wertes von `a` so, wie wir es gewohnt sind), verändert aber auch die Größe `a`
- ▶ Wie besprochen: dieser Effekt ist im (rein) funktionalen Paradigma eigentlich unerwünscht

Vergleichsoperationen in Java

- ▶ Der Gleichheitsoperator ist eine beliebte Fehlerquelle!!!
- ▶ Hier nochmal alle Vergleichsoperationen
(Signatur: $S \rightarrow S$ mit $S \in \{\text{byte}, \text{short}, \text{int}, \dots\}$):

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

Ausdrücke in Java

- ▶ Wir können auch in Java aus Operatoren Ausdrücke (zunächst ohne Variablen) bilden, so wie im vorherigen Kapitel besprochen.
- ▶ Laut induktiver Definition von Ausdrücken ist ein Literal ein Ausdruck.
- ▶ Interessanter ist, aus Literalen (z.B. den `int` Werten `6` und `8`) und mehrstelligen Operatoren (z.B. `+`, `*`, `<`, `&&`) Ausdrücke zu bilden.
- ▶ Ein gültiger Ausdruck hat, wie wir wissen, 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, keine passende Signatur`

Typkonversion

- ▶ Wie bereits erwähnt, gibt es auch in Java das Konzept der Sortenanpassung (Typkonversion, Typecasting) um Daten aus unterschiedlichen Sorten miteinander zu verarbeiten.
- ▶ Offenbar gilt z.B. $\mathbb{Z} \subseteq \mathbb{R}$, d.h. ein Ausdruck $6 + 1.3$ ist eigentlich sinnvoll wenn man die Zahl 6 als $6.0 \in \text{double}$ interpretieren würde.
- ▶ Wir können auch sagen: \mathbb{R} ist der *allgemeinere* Typ, \mathbb{Z} der *speziellere*.

Typkonversion

- ▶ In vielen Programmiersprachen gibt es eine automatische Typkonversion meist vom spezielleren in den allgemeineren Typ (dazu gleich mehr).
- ▶ Eine Typkonversion vom allgemeineren in den spezielleren Typ muss (wenn erlaubt) sinnvollerweise immer explizit durch einen *Typecasting*-Operator herbeigeführt werden (auch dazu gleich mehr).
- ▶ Es gibt aber auch Programmiersprachen, in denen man grundsätzlich zur Typkonversion ein entsprechendes Typecasting explizit durchführen muss.

Automatische Typkonversion in Java

- ▶ Unterschätzen Sie Typecasting nicht als Fehlerquelle bei der Entwicklung von Algorithmen bzw. der Erstellung von Programmen!
- ▶ Der Ausdruck `6 + 7.3` ist in Java tatsächlich auch erlaubt, d.h. hier findet offenbar eine implizite Typkonversion statt.
- ▶ *Wann* passiert *was* und *wie* bei der Auswertung des Ausdrucks `6 + 7.3`?

Automatische Typkonversion in Java

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

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

angewendet werden kann.



Automatische Typkonversion in Java

- ▶ wie wir bereits besprochen haben, ist formal gesehen diese Konvertierung eine Operation (nennen wir sie hier einfach mal $i \rightarrow d$) mit der Signatur

$i \rightarrow d : \mathbf{int} \rightarrow \mathbf{double}$

d.h. der Compiler wandelt den Ausdruck

`6 + 7.3`

um in den Ausdruck

`i->d(6) + 7.3`

- ▶ Dieser Ausdruck hat offensichtlich einen eindeutigen Typ und damit auch einen eindeutig definierten Wert.

Typkonversion in Java

- ▶ Was bedeutet “Informationsverlust”?
- ▶ Es gilt folgende “Kleiner-Beziehung” (auch “spezieller/allgemeiner-Beziehung”) zwischen Datentypen:

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

- ▶ Beispiele:
 - ▶ `1 + 1.7` ist vom Typ `double`
 - ▶ `1.0f + 2` ist vom Typ `float`
- ▶ Java konvertiert Ausdrücke automatisch in den allgemeineren (“größeren”) Typ, da dabei kein Informationsverlust auftritt.
- ▶ Was kann aber passieren, wenn man einen Ausdruck in einen spezielleren Typ umwandeln will?



Explizite Typkonversion in Java: Type-Casting

- ▶ Daher: 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 Voranstellen von `(type)`.
- ▶ Der Ausdruck
`(type) a`
wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um.
- ▶ Beispiele:
 - ▶ `(byte) 3` ist vom Typ `byte`
 - ▶ `(int) (2.0 + 5.0)` ist vom Typ `int`
 - ▶ `(float) 1.3e-7` ist vom Typ `float`

Explizite Typkonversion in Java: Type-Casting

- ▶ Bei der Typkonversion in einen spezielleren Typ kann Information verloren gehen.
- ▶ Beispiele:
 - ▶ `(int) 5.2` ergibt 5
 - ▶ `(int) -5.2` ergibt -5



Der Type-Cast-Operator in Java

Im Ausdruck

```
(type) a
```

ist `(type)` ein Operator. Type-Cast-Operatoren bilden zusammen mit einem Ausdruck wieder einen Ausdruck.

Der Typ des Operators ist z.B.:

```
(int) : charUbyteUshortUintUlongUfloatUdouble → int
```

```
(float) : charUbyteUshortUintUlongUfloatUdouble → float
```

Sie können also z.B. auch `char` in `int` umwandeln.

Klingt komisch? Ist aber so! Und was passiert da?

