Algorithmen und Datenstrukturen
Übungsblatt 05

Prof. Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

SoSe 2022	v1.5
Themen:	Vielwegbäume
Relevante Foliensätze:	Foliensatz/Video/Zusammenfassungen zu (Binary) Search Trees sowie Foliensatz MultiWayTrees zu B-Bäumen
Abgabe der Hausübung:	bis 17.06.2022, 23:50 Uhr

Hausübung 05 — Gesamt: 32 Punkte

Arithmetische Ausdrücke in Racket

Verbindliche Anforderungen für alle Hausübungen:

Das Dokumentieren und Kommentieren Ihres Quelltextes ist nicht verbindlich, wird zum besseren Verständnis Ihrer Lösung jedoch empfohlen. Alle zur Bewertung dieser Hausübung relevanten Deklarationen von Klassen, Methoden (hierzu zählen auch Konstruktoren) und Attributen sind bereits in der Quelltext-Vorlage enthalten und dürfen nicht modifiziert oder entfernt werden. Ihnen steht aber frei, Hilfskonstrukte in Form von weiteren Klassen, Methoden und Attributen zu erstellen, sofern dies nicht explizit auf dem Übungsblatt verboten wurde und Ihre Hilfskonstrukte nicht gegen verbindliche Anforderungen verstoßen. Datenstrukturen und Hilfsmethoden aus der Java-Standardbibliothek sowie Arrays sind nicht erlaubt, sofern dies nicht explizit auf dem Übungsblatt gefordert oder erlaubt wurde. Ihre Methoden müssen auch dann funktionieren, wenn Aufrufe von in der Vorlage deklarierten Methoden (auch von solchen, welche von Ihnen implementiert werden) durch andere, korrekte Implementationen ersetzt werden.

Der Verstoß gegen verbindliche Anforderungen führt zu Punktabzügen und kann die korrekte Bewertung Ihrer Abgabe unter Umständen beeinflussen. Die Implementation einer in der Quelltext-Vorlage deklarierten Methode wird nur bewertet, wenn der mit `TODO` markierte Exception-Wurf entfernt wird.

Hinweise für alle Hausübungen:

Die zu verwendenen Zugriffsmodifizierer sind in der Vorlage bereits gegeben und werden auf dem Übungsblatt nicht immer angegeben. Beachten Sie die Informationen im Moodle-Abschnitt *Technisches und Probe-Übungsblatt*.

Bei Fragen stehen wir Ihnen vorzugsweise im Moodle-Kurs und in den Sprechstunden zur Verfügung.

Die für diese Hausübung in der Vorlage relevanten Verzeichnisse sind `src/main/h05` und `src/test/h05`.

Hinweise (für dieses Übungsblatt zum Nachschlagen nützliche Quellen und erlaubte Konstrukte**):**

(1) Java-Dokumentation: `BigInteger`, `BigDecimal`, `Math`, `List`, `Map`, `Iterator`, `String`, `Enum`, `Pattern` und `Matcher`

(2) Racket-Dokumentation: Zahlen

(3) Wikipedia: Zahlbereiche

Einleitung

In diesem Übungsblatt beschäftigen wir uns mit der Auswertung und Operationen auf einem arithmetischen Ausdruck (stark vereinfacht) in der funktionalen Programmiersprache Racket.

Es stellt sich die Frage, wie man die Auswertungsreihenfolge von Operatoren in einer geeigneten Datenstruktur darstellen kann. Wir verwenden hierzu die Datenstruktur „Baum“, da man mit Bäumen leicht hierarchische Strukturen abbilden kann. Also können wir einen Binärbaum verwenden, um binäre Operationen auswerten zu können. Das schränkt uns allerdings auf binäre Operationen ein, Racket hingegen unterstützt auch die unäre als auch n-äre Operationen mit $n > 0$. Dafür bieten sich Vielwegbäume mit beliebig vielen Kinderknoten an.

Dazu definieren wir folgende Eigenschaften für einen arithmetischen Vielwegbaum:

(1) Ein Knoten ist entweder ein Operationsknoten oder ein Operandenknoten.

(2) Bei Operationsknoten sind die Operanden die unmittelbaren Nachfolgerknoten im Baum. Diese können Operandenknoten sein oder lassen sich aus Operationsknoten rekursiv zu einem Operanden auswerten.

(3) Operandenknoten sind Blätter im Baum.

(4) Ein Operandenknoten ist entweder ein Literal oder ein Identifier (als `String`).

Ein Literal kann sich – wie in Racket – in der Menge der ganzen Zahlen $\mathbb{Z}$, der rationalen Zahlen $\mathbb{Q}$ oder der reellen Zahlen $\mathbb{R}$ befinden, um möglichst exakt rechnen zu können (ausgenommen komplexe Zahlen $\mathbb{C}$).

Außerdem werden folgende arithmetische Operationen unterstützt:

Operation	Anzahl an Operanden	Beschreibung
`+`	0 bis n	Addition
`-`	1 bis n	Subtraktion
`*`	0 bis n	Multiplikation
`/`	1 bis n	Division (Teiler $\neq 0$)
`exp`	1	Potenz von e hoch x ($\exp(x), x > 0$)
`expt`	2	Potenz von x hoch y ($x^y, x > 0, y > 0$)
`ln`	1	Natürlicher Algorithmus ($\ln(x), x > 0$)
`log`	2	Logarithmus von x in y ($log_y(x), x > 0, y > 0$)
`sqrt`	1	Quadratwurzel von x ($\sqrt{(x)}, x \geq 0$)

Tabelle 1: Übersicht der Operationen

Mit dieser Grundlage können wir arithmetische Ausdrücke in Racket darstellen und erhalten wie in der Abbildung 1 einen arithmetischen Vielwegbaum für einen beispielhaften arithmetischen Ausdruck.

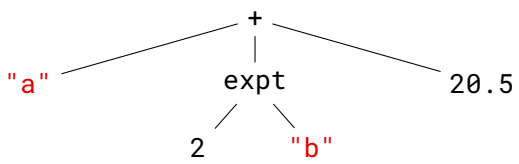


Abbildung 1: Beispiel arithmetischer Ausdruck `(+ a (expt 2 b) 20.5)` als Baum

H1: Racket-Zahlen (stark vereinfacht) 4 Punkte

In dieser Aufgabe beschäftigen wir uns mit der Darstellung von Zahlen in der Menge der ganzen Zahlen $\mathbb{Z}$, der rationalen Zahlen $\mathbb{Q}$ und reellen Zahlen $\mathbb{R}$.

Um die jeweiligen Zahlenmengen zu repräsentieren, definieren wir im Package `h05.math` eigene Klassen.

Die Klasse `MyNumber` ist das Grundgerüst zur Darstellung von Racket-Zahlen und spezifiziert nützliche Operationen, um die Zahlen später verarbeiten zu können.

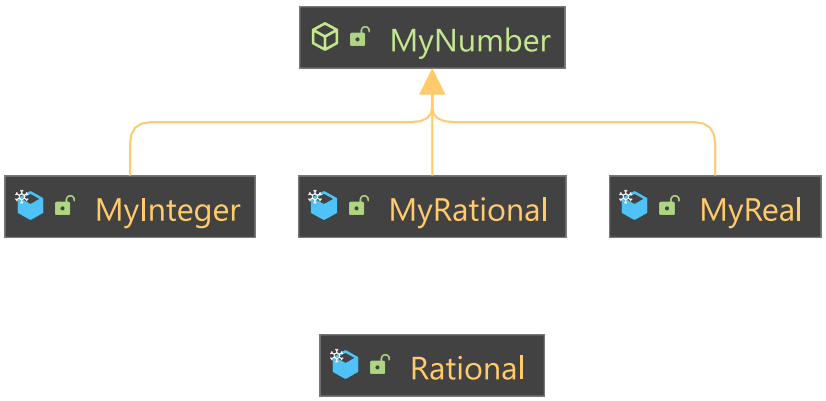


Abbildung 2: Übersicht der Racket-Zahlen Klassen

H1.1: Rationale Zahlen 2 Punkte

Um Zahlen aus der Zahlenmenge $\mathbb{Q}$ darstellen zu können, definieren wir die Klasse `Rational`.

Die Klasse hat folgende Attribute:

Formaler Typ	Bezeichner	Beschreibung
`BigInteger`	`numerator`	Beschreibt den Zähler der rationalen Zahl und enthält das Vorzeichen.
`BigInteger`	`denominator`	Beschreibt den Nenner der rationalen Zahl und ist immer positiv.

Tabelle 2: Attribute von `Rational`

Ihre Aufgabe besteht es darin die Attribute zu initialisieren. Der Konstruktor soll den Bruch weitestmöglich kürzen d.h. `numerator` und `denominator` sind betragsmäßig möglichst klein. Beachten Sie ebenfalls die Anforderungen aus Tabelle 2.

H1.2: Konvertierung einer Zahlenmenge in eine andere Zahlenmenge 2 Punkte

Mit dem Grundgerüst `MyNumber` können wir die Racket-Zahlen darstellen. Dazu finden Sie die folgenden drei Klassen, welche die ganzen, rationalen und reellen Zahlen darstellen und diese jeweils im Attribut `value` speichern:

Implementieren Sie die zwei Methoden `toRational` und `toReal`, die in der Oberklasse spezifiziert sind. Die Rückgabewerte der drei Methoden sollen möglichst nah am jeweils eingekapselten Zahlenwert sein.

Klassenname	Attributtyp	Beschreibung
`MyInteger`	`BigInteger`	Eine Klasse, die eine ganze Zahl darstellt.
`MyRational`	`Rational`	Eine Klasse, die eine rationale Zahl darstellt.
`MyReal`	`BigDecimal`	Eine Klasse, die eine reelle Zahl darstellt.

Tabelle 3: Subklassen von `MyNumber`

- `toRational`: Der eingekapselte Wert wird ggf. betragsmäßig abgerundet (für negative Werte bedeutet das Aufrundung).
- `toReal`: Für alle drei Klassen ist dies offensichtlich exakt möglich.

Verbindliche Anforderung:

Beachten Sie, dass bei allen Operationen, die `BigDecimal` verwenden, die Skalierung und Rundungsart anzugeben ist. (Siehe vordefinierte Konstanten in der Klasse `MyReal`)

Hinweise:
Die Implementierung von der Konvertierung von reellen Zahlen in rationale Zahlen ist leider ungenau. Wir erhalten bspw. bei der Konvertierung 0.25 in einer rationale Zahl folgendes Ergebnis raus:

$$\frac{\lfloor 0.25 \rfloor}{1} = \frac{0}{1}$$

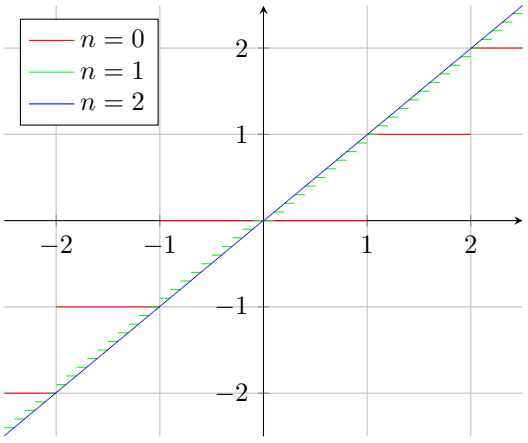
Um die Genauigkeit zu verbessern, können Sie folgende Formel verwenden:

$$\frac{\lfloor x \cdot 10^n \rfloor}{10^n},$$

wobei x die Zahl ist, die in eine rationale Zahl konvertiert werden soll und n die Anzahl der Stellen nach dem Komma. Beachten Sie, dass die oben genannte Formel nur für positive Werte für x korrekt ist.

Rechts sehen Sie die jeweilige Funktion für verschiedene Werte von n.

Den Wert für n können Sie aus der Konstante `MyReal.SCALE` entnehmen.



Unbewertete Verständnisfrage:

Sehen Sie Ähnlichkeiten zu Klasse `Number` und ihren Subklassen (wenn auch verkürzt und vereinfacht)?

H2: Arithmetische Operationen 7 Punkte

Bevor wir mit der Implementierung eines arithmetischen Vielwegbaums anfangen, müssen wir noch arithmetische Operationen definieren, die dann mithilfe eines arithmetischen Vielwegbaums ausgewertet werden können.

Verbindliche Anforderung:

Beachten Sie, dass bei allen Operationen, die `BigDecimal` verwenden, die Skalierung und Rundungsart anzugeben ist. (Siehe vordefinierte Konstanten in der Klasse `MyReal`)

Hinweise:
Sofern sich die Implementierung in den Subklassen nicht ausdrücklich unterscheiden lässt, können Sie die Methoden in der Klasse `MyNumber` implementieren. Sie dürfen dann ausnahmsweise die Methoden aus den Subklassen entfernen, die Sie in der Oberklasse implementiert haben.

H2.1: Arithmetische Grundoperationen 2 Punkte

Wir können nun die arithmetischen Operationen auf den Zahlen anhand der folgenden Tabelle implementieren:

Rückgabetyp	Bezeichner	Parameter	Beschreibung
`MyNumber`	`plus`	–	$0 + x$
`MyNumber`	`plus`	`MyNumber`	$x + y$
`MyNumber`	`minus`	–	$0 - x$
`MyNumber`	`minus`	`MyNumber`	$x - y$
`MyNumber`	`times`	–	$1 \times x$
`MyNumber`	`times`	`MyNumber`	$x \times y$
`MyNumber`	`divide`	–	$1 \div x$
`MyNumber`	`divide`	`MyNumber`	$x \div y$

Tabelle 4: Übersicht der arithmetischen Grundoperationen von `MyNumber`

Implementieren Sie nun also die arithmetischen Grundoperationen für die Subtraktion und Division in den Subklassen von `MyNumber`.

Falls einer der Operanden die Anforderungen aus der Tabelle 1 (siehe Beschreibung) nicht einhält, so wird eine `WrongOperandException` geworfen mit der entsprechenden Botschaft (siehe JavaDoc).

Der dynamische Typ des Ergebnisses einer Operation sollte möglichst in der kleinsten Zahlenmenge sein. ($\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$) Für die vier arithmetischen Grundrechenoperationen gilt folgendes (unäre Operationen ausgenommen):

(1) Falls beide Zahlen ganze Zahlen sind, so ist das Ergebnis eine ganze Zahl (ausgenommen Division). Das Ergebnis bei einer Division von zwei ganzen Zahlen ist eine rationale Zahl.

(2) Falls eine Zahl eine reelle Zahl ist, so ist das Ergebnis entweder eine ganze oder eine reelle Zahl.

(3) Ansonsten ist das Ergebnis entweder eine ganze oder eine rationale Zahl.

(4) Falls das Ergebnis einer der Grundrechenoperationen als ganze Zahl dargestellt werden kann, so ist das Ergebnis eine ganze Zahl.

H2.2: Wurzel, Exponenzieren und Logarithmus — 5 Punkte

Analog zu den arithmetischen Grundoperationen, implementieren Sie die folgenden Methoden:

Rückgabetyp	Bezeichner	Parameter	Beschreibung
`MyNumber`	`sqrt`	–	$\sqrt{x}$
`MyNumber`	`exp`	–	$\exp(x) = e^x$
`MyNumber`	`expt`	`MyNumber`	x^y
`MyNumber`	`ln`	–	$\ln(x)$
`MyNumber`	`log`	`MyNumber`	$\log_x(y)$

Tabelle 5: Übersicht weitere arithmetische Operationen von `MyNumber`

Das Ergebnis der Operationen `sqrt`, `exp`, `expt`, `ln` und `log` ist entweder im Zahlenbereich der ganzen Zahlen oder der reellen Zahlen.

Hinweise:

Auf der letzten Seite finden Sie im Abschnitt „Zusatz: BigDecimal verträgliche Exponentialrechnung“ nützliche Informationen bezüglich der Implementierung des Logarithmus und dem Exponenzieren.
Für die Implementierung von `sqrt` könnte die Klasse `BigDecimal`, vor allem `MathContext#DECIMAL128`, hilfreich sein.

H3: Arithmetischer Vielwegbaum — 8 Punkte

Nun kommen wir zu der eigentlichen Aufgabe – die Repräsentation eines arithmetischen Ausdrucks in Racket als arithmetischen Vielwegbaum. Dazu modellieren wir folgende Struktur im Package `h05.tree` (dazu später mehr):

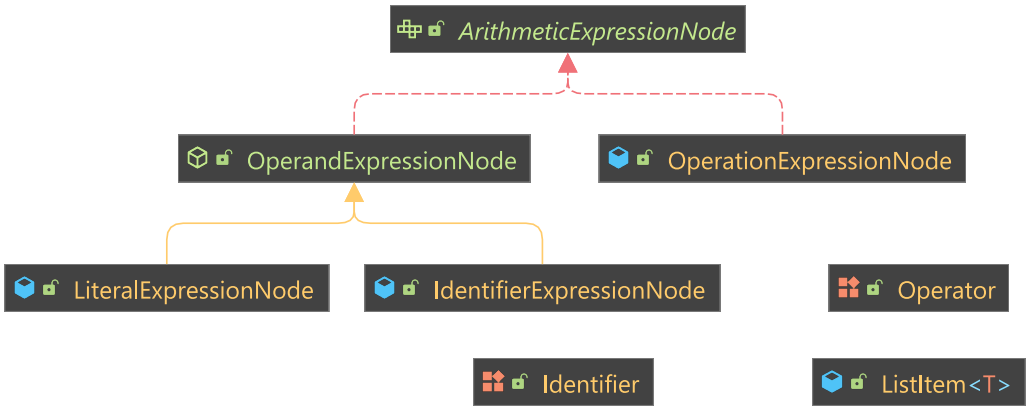


Abbildung 3: Übersicht des arithmetischen Baums

Das Interface `ArithmeticExpressionNode` ist das Grundgerüst für alle Knoten im arithmetischen Vielwegbaum und besitzt die folgenden Methoden:

Die Methode `evaluate` evaluiert einen Knoten mit den Variablenwerten in der `java.util.Map`, die Methode `isOperand` bzw. `isOperation` gibt `true` zurück, wenn der Knoten ein Operand bzw. eine Operation ist und die Methode `clone` gibt eine Kopie des Knotens zurück.

Rückgabetyp	Bezeichner	Parameter
`MyNumber`	`evaluate`	`Map<String, MyNumber>`
`boolean`	`isOperand`	–
`boolean`	`isOperation`	–
`ArithmeticExpressionNode`	`clone`	–

Tabelle 6: Übersicht der Methoden von von `ArithmeticExpressionNode`

Neben den Methoden besitzt das Interface `ArithmeticExpressionNode` zwei Attribute `LEFT_BRACKET` und `RIGHT_BRACKET` vom formalen Typ `String`. Wie die Namen schon verraten, repräsentiert `LEFT_BRACKET` ein „öffnende Klammer“-Symbol `"("` und `RIGHT_BRACKET` ein „schließende Klammer“-Symbol `")"`, um die Ausführungsreihenfolge der Operationen zu verdeutlichen.

Verbindliche Anforderung:

Die Kopie soll tief sein im Sinne von Kapitel 03b, ab Folie 57 der FOP. Das heißt, alle Objekte im zurückgelieferten Baum (alle Knoten, Zahlen und Strings) werden mit Operator `new` neu erzeugt.

Unbewertete Verständnisfrage:

Um welchen Typ von Attributen handelt es sich hier im Interface bzw. welche Arten von Attributen können wir im Interface überhaupt definieren?

Da es zwei verschiedene Arten von Operandenknoten (Literal und Identifier) gibt, definieren wir eine abstrakte Klasse `OperandExpressionNode`, die allgemein einen Operandenknoten darstellt. Sie implementiert die zwei Methoden `isOperand` und `isOperator` aus dem Interface `ArithmeticExpressionNode`, d.h. `isOperand` liefert `true` zurück und `isOperator` liefert `false` zurück.

Erinnerung:

Beachten Sie, dass die Sichtbarkeit einer ererbten Methode nicht eingeschränkt werden, sondern nur vergrößert werden kann!

H3.1: Literale — 1 Punkt

Wir implementieren nun den ersten Operandenknoten, und zwar ein Literal. Die Klasse `LiteralExpressionNode` stellt einen Knoten dar, der wiederum ein Literal darstellt.

Ein Literal wird durch ein `MyNumber`-Objekt dargestellt, welches durch den Konstruktor mit einem entsprechenden Parameter initialisiert wird.

Implementieren Sie die Methoden `evaluate` und `clone` aus dem Interface `ArithmeticExpressionNode`. Die Methode `evaluate` liefert immer den Wert des Literals zurück und die Methode `clone` immer eine neue Instanz der

Klasse mit dem kopierten Literal.

H3.2: Identifier — 3 Punkte

Wir können nun den zweiten Operandenknoten implementieren. Analog zu der Klasse für Literale, gibt es die Klasse `IdentifierExpressionNode`, die ebenfalls `OperandExpressionNode` erweitert und einen Identifier darstellt.

Ein Identifier wird durch einen `String` dargestellt, welcher durch einen entprechenden Parameter im Konstruktor initialisiert wird. Hierbei sind folgende Regeln zu beachten:

(1) Falls der Parameterwert `null` ist, so wird eine `NullPointerException` geworfen.

(2) Falls der Parameterwert leer ist, so wird eine `IllegalArgumentException` mit der Botschaft `"empty string"` geworfen (Leerzeichen zwischen beiden Wörtern).

(3) Falls der Parameter nicht nach folgenden Schema definiert ist, so wird eine `IllegalIdentifierExceptions` mit dem Parameter `value` geworfen.

Ein Identifier ist eine Sequenz von Klein- und Großbuchstaben. Ebenfalls sind Bindestriche erlaubt.

Beispiel: `"-a-b-cDeFG"`

Implementieren Sie außerdem die Methoden `evaluate` und `clone` aus dem Interface `ArithmeticExpressionNode`. Die Methode `evaluate` liefert immer den Wert des Identifiers aus der `Map` zurück. Methode `clone` liefert immer eine neue Instanz der Klasse mit dem kopierten Identifier zurück.

Beachten Sie, dass die Methode `evaluate` ggf. eine Exception werfen kann.

(1) Falls der Identifier bereits vordefiniert ist, so wird eine `IllegalIdentifierExceptions` mit dem Identifier als Botschaft geworfen. Schauen Sie sich hierzu die Enumklasse `Identifier` an.

(2) Falls der Identifier nicht definiert ist, so wird eine `UndefinedIdentifierException` mit dem Identifier als Botschaft geworfen.

H3.3: Operationknoten — 4 Punkte

Es fehlt nur noch ein Operationsknoten und der arithmetische Vielwegbaum für Racket-Ausdrücke wäre fertig. Die Klasse `OperationExpressionNode` repräsentiert einen Operationsknoten und implementiert ebenfalls das Interface `ArithmeticExpressionNode`.

Sie besitzt folgende Attribute:

Formaler Typ	Bezeichner	Beschreibung
`Operator`	`operator`	Operator des Knotens
`ListItem<ArithmeticExpressionNode>`	`operands`	Liste der Operanden

Tabelle 7: Übersicht der Attribute von `OperationExpressionNode`

Initialisieren Sie die Attribute mit den Parametern `operator` und `operands` des Konstruktors. Dieser überprüft, ob die Anzahl an Operanden zu dem Operator passen. (Siehe Tabelle 1 - Anzahl an Operanden) Falls nicht, so wird eine `WrongNumberOfOperandsException` mit dem aktualen Anzahl an Operanden und dem Intervall für die erwartete Anzahl an Operanden als Parameter geworfen.

Implementieren Sie zum Schluss die Methoden `evaluate` und `clone`.

Die Methode `evaluate` liefert das Ergebnis der Berechnung mit `operator` zurück. Der Rückgabewert ergibt sich aus der rekursiven Evaluation aller Operanden. Beachten Sie, dass die Addition und Multiplikation 0 Operanden besitzen können, d.h.

(1) die Addition gibt das neutrale Element der Addition zurück. (Ganze Zahl 0)

(2) die Multiplikation gibt das neutrale Element der Multiplikation zurück. (Ganze Zahl 1)

Methode `clone` liefert immer eine neue Instanz der Klasse mit den kopierten Attributen zurück.

> **Unbewertete Verständnisfrage:**
>
> Warum brauchen wir keinen zusätzlichen Code, um einen Ausdruck als Ganzes zu evaluieren? Warum reicht eine Baumknotenklasse, warum definieren wir nicht zusätzlich eine Baumklasse in Anlehnung bspw. an `ListItem<T>` und `MyLinkedList<T>`? Ist das nur falsche Bequemlichkeit oder gerechtfertigt?

H4: (Re-)Konstruktion eines aritmetischen Ausdrucks — 10 Punkte

In dieser Aufgabe wollen wir einen arithmetischen Vielwegbaum aus einer geordneten Sequenz von Tokens (re-) konstruieren und vice versa. Dies geschieht in der Klasse `ExpressionTreeHandler`.

Sei M die Menge der darstellbaren Zahlen in Racket inklusive Identifier, O die Menge der Operatoren, P die Menge der Klammer-Symbole und $T = M \cup O \cup P$ die Menge der Tokens, um einen arithmetischen Ausdruck darzustellen.

Eine geordnete Sequenz von Tokens S ist folgendermassen definiert:

$$S = \{s_1, \ldots, s_n\}, s_i \in T, n, i \in \mathbb{N}$$

Konkret repräsentiert ein Token entweder den Namen einer Operation, ein Identifier, ein Zahlenliteral oder ein einzelnes Klammer-Symbol.

H4.1: Rekursiver Aufbau eines arithmetischen Vielwegbaums — 4 Punkte

Die Methode `buildRecursively` hat einen Parameter vom formalen Typ `Iterator<String>` und Rückgabetyp `ArithmeticExpressionNode`.

Sie soll einen arithmetischen Baum aus einer gegebenen Liste von Tokens erstellen. Für jede öffnende Klammer in einem Ausdruck gibt es genau einen rekursiven Aufruf von `buildRecursively`, der alle Tokens von dieser öffnenden Klammer bis zur zugehörigen schließenden Klammer abarbeitet (und für jedes darin eingebettete Klammerpaar natürlich auch wieder sich selbst rekursiv aufruft).

Ein Fehler wird geworfen, falls einer dieser Fälle eintritt:

(i) Falls der Iterator leer ist, so wird eine `BadOperationException` mit der Botschaft `"No expression"` geworfen (Leerzeichen zwischen beiden Wörtern).

(ii) Falls eine öffnende oder schließende Klammer erwartet wird, aber keine vorkommt wird, so wird eine `ParenthesesMismatchException` geworfen.

(iii) Falls eine ungültige Operation gelesen wird, so wird eine `UndefinedOperatorException` mit dem aktuellen Token als Botschaft geworfen.

(iv) Falls die Anzahl der Operanden bis zur nächsten schließenden Klammer (unter Berücksichtigung der rekursiven Aufrufe für eingebettete Klammerpaare) nicht für diese Operation zulässig ist, wird die Exception `WrongNumberOfOperandsException` mit entsprechender Botschaft geworfen.

(v) Falls ein ungültiger Operand gelesen wird, so wird eine `IllegalIdentifierExceptions` mit dem aktuellen Token als Botschaft geworfen.

(vi) Falls der Ausdruck mehrere Fehler enthält, ist nicht vorgegeben, welcher davon zum Wurf einer `Exception` führt. Sie können also einfach den ersten identifizierten Fehler hernehmen.

Verbindliche Anforderung:

Die Methode `buildRecursively` ist rein rekursiv implementiert, d.h. Schleifen sind nicht erlaubt und die Sequenz darf nur einmal durchlaufen werden. Falls Sie Funktionalität von `buildRecursively` in andere von Ihnen geschriebene Methoden auslagern (werden Sie wohl müssen) oder Konstrukte verwenden, dann gilt das auch für alle diese.

Hinweise:

1. Natürlich darf nur beim Gesamtausdruck nach der schließenden Klammer nichts mehr folgen; bei Teilausdrücken folgt hingegen noch Weiteres. Falls dies doch der Fall ist, wird eine `BadOperationException` mit dem aktuell eingelesen Token als Botschaft geworfen.
2. Der Fall, dass die Sequenz, die vom Iterator zurückgeliefert wird, zu Ende ist, bevor der Ausdruck vollständig ist, wird oben nur implizit adressiert. Sie sollten diesen Fall besser explizit an mehreren Stellen mitbedenken.
3. Eine reelle Zahl hat folgende Form:

   $$\text{Reelle Zahl} = \text{Zahl.Zahl}$$

   Falls die Zahl negative ist, so muss das Vorzeichen mit einem Minuszeichen vor der Zahl stehen.
4. Eine rationale Zahl hat folgende Form:

   $$\text{Rationale Zahl} = \text{Zahl}/\text{Zahl}.$$

   Falls die Zahl negative ist, so muss das Vorzeichen mit einem Minuszeichen vor der Zahl stehen.

H4.2: Iterativer Aufbau eines arithmetischen Vielwegbaums — 4 Punkte

Schreiben Sie nun eine Methode `buildIteratively`, die völlig analog zu `buildRecursively` ist, nur dass die verbindliche Anforderung eine andere ist.

Verbindliche Anforderung:

Die Methode `buildIteratively` ist rein iterativ implementiert, d.h. Rekursion ist nicht erlaubt und die Sequenz darf nur einmal durchlaufen werden. Falls Sie Funktionalität von `buildRecursively` in andere von Ihnen geschriebene Methoden auslagern oder Konstrukte verwenden, dann gilt das auch für alle diese.

Hinweis:

In FOP Kapitel 04a finden Sie eine Darstellung des Call-Stacks bei Rekursion. Simulieren Sie diesen Call-Stack mit einem Objekt der Klasse `Stack<ListItem<ArithmeticExpressionNode>>` aus der Java-Standardbibliothek.
Die Schleifeninvariante wäre dann wie folgt – nach h Iterationen (=Durchläufen durch die Schleife) gilt:

1. Die ersten h Strings sind aus dem Iterator ausgelesen, aber noch keine weiteren.
2. Seien $A_1, \ldots, A_k$ diejenigen Teilausdrücke, deren öffnende Klammer schon eingelesen wurde, aber die schließende Klammer noch nicht, und zwar in der Reihenfolge, in der die öffnenden Klammern dieser Intervalle eingelesen wurden (d.h. A_{i+1} ist in A_i eingebettet für $i \in \{1, \ldots, k-1\}$). Dann enthält das `Stack`-Objekt genau k Elemente, und gemäß der Logik der Methode `search` von `java.util.Stack` korrespondiert das Element mit Distanz j zu A_{k-j+1}.

Unbewertete Verständnisfrage:

Ist Ihnen die iterative oder die rekursive Umsetzung leichter gefallen? Warum?

H4.3: Rekonstruktion eines arithmetischen Ausdruck — 2 Punkte

Zur Rekonstruktion einer geordneten Sequenz von Tokens aus einem arithmetischen Vielwegbaum, schreiben Sie nun eine Methode `reconstruct` mit einem Parameter vom formalen Typ `ArithmeticExpressionNode` und Rückgabetyp `List<String>`.

Ihre Aufgabe besteht darin einen arithmetischen Baum als Sequenz von Tokens (Strings) zurückzugeben. Genauer gesagt: Die Methode `reconstruct` ist die Umkehroperation zu `buildRecursively` bzw. `buildIteratively`.

Das heißt,

- `reconstruct(buildRecursively(list.iterator()))` bzw.
- `reconstruct(buildIteratively(list.iterator()))`

liefern eine Kopie von `list`.

Es ist Ihnen freigestellt, ob Sie die Methode `reconstruct` iterativ oder rekursiv implementieren. Bedenken Sie aber die Verständnisfrage am Ende von H4.2!

H5: Schrittweise Auswertung eines Ausdrucks — 3 Punkte

In dieser Aufgabe wollen wir es ermöglichen einen arithmetischen Vielwegbaum schrittweise auszuwerten. Dies geschieht durch Methode `nextStep` der Klasse `ArithmeticExpressionEvaluator`.

Die Klasse besitzt zwei Attribute:
Die Werte der Attribute werden durch entsprechende Parameter des Konstruktors initialisiert, wobei der Baum immer kopiert wird. Das ist notwendig, weil der Baum modifiziert werden muss, die aktualen Parameter aber nicht modifiziert werden sollen.

Des Weiteren besitzt die Klasse eine `public`-Methode `nextStep` mit Rückgabetyp `List<String>`, die Sie implemen-

Formaler Typ	Bezeichner	Beschreibung
`ArithmeticExpressionNode`	`root`	Baum zur Auswertung des Ausdrucks
`Map<String, MyNumber>`	`identifiers`	Zuordnung von Identifiern zu Zahlen

Tabelle 8: Übersicht der Attribute von `ArithmeticExpressionEvaluator`

tieren müssen. Bei jedem Aufruf von `nextStep` wird als erstes `reconstruct` mit dem intern gespeicherten Baum aufgerufen und das Ergebnis zurückgeliefert. Falls der Baum nicht aus einem einzigen Zahlenliteral besteht, macht `nextStep` einen elementaren Auswertungsschritt auf dem Baum, d.h. der abgespeicherte Baum und die geordnete Sequenz von Tokens werden ggf. modifiziert!

Es gibt zwei Arten von elementaren Auswertungsschritten:

(1) Ersetzung der Identifier durch das Zahlenliteral, das zum entsprechenden Identifier in der Map gespeichert ist. Falls zum Identifier kein Eintrag in der Map ist, wird der Identifier durch den String `"<unknown!>"` ersetzt.

(2) Ersetzung der Klammerausdrücke, in dem keine anderen Klammerausdrücke eingebettet sind, durch den Wert dieses Klammerausdrucks. (Evaluation der innersten Klammerausdrücke)

Andernfalls (der Baum nur noch aus einem einzigen Zahlenliteral) macht `nextStep` nichts weiter.

Auch hier sind Ihnen die Details der Implementation freigestellt.

Beispiel:

Gegeben sei folgender arithmetischer Ausdruck `(+ a (/ (expt 2 b) (* (ln e) c)))` mit den Identifierwerten `a = 2/3, b = 3, c = 2.5, e = 2.7182818284590452354`.

n	Ergebnis
0	`(+ a (/ (expt 2 b) (* (ln e) c)))`
1	`(+ 2/3 (/ 8 (* 1 2.5)))`
2	`(+ 2/3 (/ 8 2.5))`
3	`(+ 2/3 3.2)`
4	`3.866666666666667`

Tabelle 9: Auswertung von `nextStep`

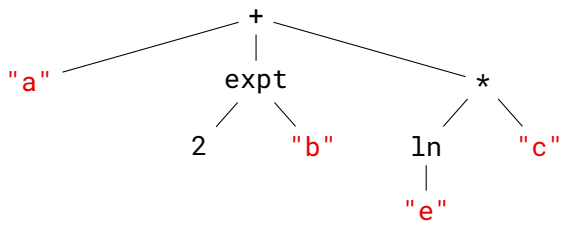


Abbildung 4: Visualisierung

Unbewertete Verständnisfrage:

In den vorangehenden Abschnitten haben wir uns mit mehreren Fehlerarten befassen müssen, warum hier nur mit unbekannten Identifiern?

H6: Test

Verbindliche Anforderungen für alle Hausübungen:

Auch wenn Sie für das Erstellen von Tests keine Punkte erhalten: Das Teilen jeglicher Tests zu Hausübungen ist *nicht* erlaubt. Unser Ziel ist, die Verbreitung fehlerhafter Tests in Ihrem Sinne zu verhindern. Alle auf diesem Übungsblatt genannten verbindlichen Anforderungen gelten für diese Aufgabe *nicht*. Insbesondere empfiehlt es sich sogar, zum Testen Datenstrukturen und Hilfsmethoden aus der Java-Standardbibliothek zu verwenden.

Hinweise für alle Hausübungen:

Die von uns bereitgestellten Public Tests überprüfen nur einen kleinen Teil Ihrer Implementation. Erfüllt Ihre Lösung nicht alle Public Tests, erhalten Sie auf keinen Fall die volle Punktzahl. Im Umkehrschluss bedeutet dies aber nicht, dass Sie die volle Punktzahl erhalten, wenn Ihre Lösung alle Public Tests besteht.

Der folgende Leitfaden dient als Unterstützung zum Aufbau Ihrer eigenen Tests. Sie können vom Leitfaden abweichen und dabei mindestens genauso aussagekräftige Testergebnisse erzeugen.

Es bietet sich an, Testbeispiele in Form von `List<String>` zu generieren. Testen Sie `buildRecursively` und `buildIteratively`, indem Sie auf jedem Testbeispiel beide Methoden aufrufen. Rufen Sie mit beiden Ergebnissen jeweils die Methode `reconstruct` auf und vergleichen Sie dann die beiden Ergebnislisten mit der Ausgangsliste.

Sie können die Methoden `evaluate` und `ArithmeticExpressionEvaluator#nextStep()` gegeneinander testen.

Durch Ausgabe der `String`-Listen aus `reconstruct` und `ExpressionTreeHandler.nextStep` können Sie auch per Augenschein prüfen, ob die Strukturen der Ausdrücke stimmen.

Vergessen Sie nicht, Randfälle und Fehlerfälle zu testen.

Zusatz: BigDecimal verträgliche Exponentialrechnung

Wollen wir in Java den Logarithmus einer Zahl bestimmen, reicht uns meist `Math#log(double)` aus. Dies ist allerdings aufgrund der internen Ungenauigkeit von Gleitkommazahlen für sehr große, sowie sehr kleine, Argumente in unserem Fall zu ungenau. Weiter wäre schon die Konvertierung von `BigDecimal` zu `double` sehr verlustreich. Daher beschränken wir uns auf das Intervall $[1, 10]$ und gehen davon aus, dass wir in diesem eine gute Approximation des Logarithmus berechnen können (Namentlich `Math#log(double)`; Basis 10 deshalb, weil `BigDecimal` mit dem „Verschieben des Kommas“, wie sie es aus der Schule kennen verlustfrei verträglich sind).

Betrachten Sie zunächst folgende Beispielrechnung, wie wir den Logarithmus zur Basis 10 einer sehr großen Zahl annähern können:

$$\begin{aligned}
\log_{10}(12345678909876543210123456789) &= \log_{10}(1234567890987654321012345678, 9 \cdot 10) \\
&= \log_{10}(1234567890987654321012345678, 9) + \log_{10}(10) \\
&= \log_{10}(1234567890987654321012345678, 9) + 1 \\
&= \log_{10}(123456789098765432101234567, 89 \cdot 10) + 1 \\
&= \log_{10}(123456789098765432101234567, 89) + \log_{10}(10) + 1 \\
&= \log_{10}(123456789098765432101234567, 89) + 2 \\
&\vdots \\
&= \log_{10}(1, 23456789098765432101234567 89) + 28
\end{aligned}$$

Mit `Math#log10(double)` erhalten wir ab hier die sehr gute Abschätzung $28,091514977516706$. Ist eine Zahl zu klein, können wir ähnlich verfahren:

$$\begin{aligned}
\log_{10}(0,00000000000000000000123456789) &= \log_{10}(00,00000000000000000000123456789 \cdot 10^{-1}) \\
&= \log_{10}(00,00000000000000000000123456789) + \log_{10}(10^{-1}) \\
&= \log_{10}(00,00000000000000000000123456789) - 1 \\
&= \log_{10}(000,00000000000000000000123456789 \cdot 10^{-1}) - 1 \\
&= \log_{10}(000,00000000000000000000123456789) + \log_{10}(10^{-1}) - 1 \\
&= \log_{10}(000,00000000000000000000123456789) - 2 \\
&\vdots \\
&= \log_{10}(00000000000000000001,23456789) - 21
\end{aligned}$$

Und erneut können wir schätzen: $-20,908485022830728$.

In Java ist dies leicht nachzustellen: Sei $x \in \mathbb{R}$ die Zahl, dessen Logarithmus wir approximieren wollen. Der Logarithmus beginnt bei 0. Dann müssen wir zwei Fälle betrachten:

- Solange unsere Zahl zu groß ($x > 10$) ist, dividieren wir x durch 10 und erhöhen den Logarithmus um 1.
- Solange unsere Zahl zu klein ($x < 1$) ist, multiplizieren wir x mit 10 und verringern den Logarithmus um 1.

Nun ist x auf jeden Fall im richtigen Intervall. Schließlich wandeln wir x in einen `double`-Wert um und addieren den $\log_{10}(x)$ auf unseren Logarithmus.

Exponenzieren ist noch leichter: Als Basis fixieren wir wieder 10 und gehen davon aus, dass wir nur Exponenten im Intervall [0, 1] verarbeiten können (In Java: `x -> Math.pow(10, x)`).

Sei $x \in [0, \infty)$ unser Exponent. Wir teilen x wie folgt auf: $x = n + r$, mit $n \in \mathbb{N}, r \in [0, 1]$. Dann ist

$$10^x = 10^{n+r} = 10^n \cdot 10^r$$

Denn Faktor 10^n kann `BigDecimal` verlustfrei für uns berechnen. Für 10^r nutzen wir `double`, wie oben.