Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr.-Ing. Mark Minas

ST 2021

Testat 1: BigIntRationalExpressions

Abgabetermin: Tag und Uhrzeit der Klausur im ST21

In diesem Testat geht es um rationale Zahlen $\mathbb{Q}$ mit beliebiger Genauigkeit und Ausdrücke, die diese rationalen Zahlen verwenden.

Laden Sie das zip-File zu diesem Testat aus Ilias herunter und erstellen Sie daraus ein neues IntelliJ-Projekt.

Aufgabe 1 (Rational)

In dieser Aufgabe soll eine Klasse `Rational` zur Speicherung von Brüchen erstellt werden. Um beliebige rationale Zahlen darstellen zu können, reicht die Repräsentation von Zähler und Nenner mit `int`-Werten nicht aus. Stattdessen verwenden wir hier die Klasse `BigInteger` der Java-API. Wie üblich sollen alle Objekte der Klasse `Rational` *immutable* sein.

a) Erstellen Sie dazu im Package `rational` die Java-Klasse `Rational`. Geben Sie der Klasse passende Konstruktoren, mit denen Brüche definiert werden können. Ist der Nenner 1, soll man ihn weglassen können. Achten Sie darauf, dass Brüche bei ihrer Erzeugung gekürzt und normalisiert werden.

```
// fuer einen Bruch 1/2
Rational r = new Rational(BigInteger.ONE, BigInteger.valueOf(2));

// fuer eine ganze Zahl 2
Rational r = new Rational(BigInteger.TWO);
```

Zusätzlich werden auch Konstruktoren mit `long`-Parametern benötigt. Intern werden diese natürlich in `BigInteger`-Attributen vorgehalten. Zum Beispiel:

```
// fuer einen Bruch 1/2
Rational r = new Rational(1L, 2L);

// fuer eine ganze Zahl 2
Rational r = new Rational(2L);
```

b) Stellen Sie die beiden Getter-Methoden `getNum()` und `getDenom()` für den Zähler (*numerator*) und den Nenner (*denominator*) zur Verfügung.

c) Realisieren Sie zum Rechnen mit rationalen Zahlen die Methoden

```
Rational abs()               // Absolutbetrag
Rational neg()               // Negation
Rational add(Rational that)  // Addition
Rational sub(Rational that)  // Subtraktion
Rational mult(Rational that) // Multiplikation
Rational div(Rational that)  // Division
```

`Rational pow(int exponent)  // ganzzahlige Potenz`

Überladen Sie die Methoden außerdem, so dass man beispielsweise zu einer rationalen Zahl eine ganze Zahl addieren kann, die als `long`-Wert übergeben wird.

Lösen Sie eine `IllegalArgumentException` aus, falls bei der Division der Wert 0 übergeben wird.

d) Überschreiben Sie die `toString`-, `equals`- und `hashCode`-Methoden geeignet. Achten Sie darauf, dass Brüche sinnvoll ausgegeben werden, also z.B. `1/2`, aber nicht `2/1`, sondern `2`.

e) Stellen Sie sicher, dass die Klasse `Rational` das Interface `Comparable` implementiert, und überschreiben Sie die Methode `compareTo(Rational r)` in geeigneter Weise.

f) Implementieren Sie außerdem die Methoden

```
boolean lessThan(Rational that)
boolean lessThanOrEquals(Rational that)
boolean greaterThan(Rational that)
boolean greaterThanOrEquals(Rational that)
```

mit ihrer naheliegenden Bedeutung. Stützen Sie sich bei der Implementierung auf der `compareTo`-Methode der vorherigen Teilaufgabe ab. Überladen Sie diese Methoden außerdem durch Methoden mit `long`-Parametern.

g) Implementieren Sie die Methode `Rational sqrt(Rational eps)`, die einen rationalen Näherungswert derjenigen Zahl zurückgibt, für die man `sqrt` aufruft. Verwenden Sie dazu das *Heron-Verfahren*; die Iteration soll enden und den Näherungswert s_i für $\sqrt{a}$ zurückgeben, sobald $|s_i^2/a - 1| \leq \epsilon$, wobei ϵ für den Wert des Parameters `eps` steht.

Lösen Sie eine `ArithmeticException` aus, wenn die Wurzel eines negativen Wertes berechnet werden soll, und eine `IllegalArgumentException`, wenn ein negativer Wert für `eps` übergeben wird.

Aufgabe 2 (Termbäume)

Ausdrücke werden üblicherweise mittels Termbäumen repräsentiert. Das Klassendiagramm in Abbildung 1 beschreibt die aus der Vorlesung bekannte Klassenstruktur für einfache Ausdrücke, die aus konstanten Werten (Klasse `Const`), Negativen (Klasse `Neg`) sowie der binären Addition, Subtraktion (Klasse `Plus` und `Minus`) und Multiplikation (Klasse `Mult`) bestehen können. Um die Implementierung von Algorithmen auf solchen Ausdrücken von ihrer Repräsentation zu trennen, wollen wir das **Visitor Pattern** (vgl. Vorlesung Folie 232ff.) nutzen.

Die in Abbildung 1 dargestellten Klassen befinden sich bereits im Package `rational` in Ihrem Projekt.

a) Überarbeiten Sie zuerst alle gegebenen Klassen aus dem Package `rational` dahingehend, dass statt `int`-Werten nun Objekte der Klasse `Rational` aus Aufgabe 1 verwendet werden.

b) Erweitern Sie die zur Verfügung stehenden Termbäume um den binären Ausdruck `Div` für die Division. Stellen Sie sicher, dass alle vorhandenen `Visitor`-Implementierungen mit diesem Ausdruck umgehen können.

c) Um Ausdrücke potenzieren zu können, implementieren Sie die Klasse `Power`, deren Konstruktor `Power(Expr base, int exponent)` einen Ausdruck als *Basis* und einen ganzzahligen Exponenten vom Typ *int* erwartet. Stellen Sie auch hier die Verwendung durch die `Visitor`-Klassen sicher. Die textuelle Darstellung folgt dem Muster `(<basis>)^<exponent>`, z.B. `(3/4)^2`.

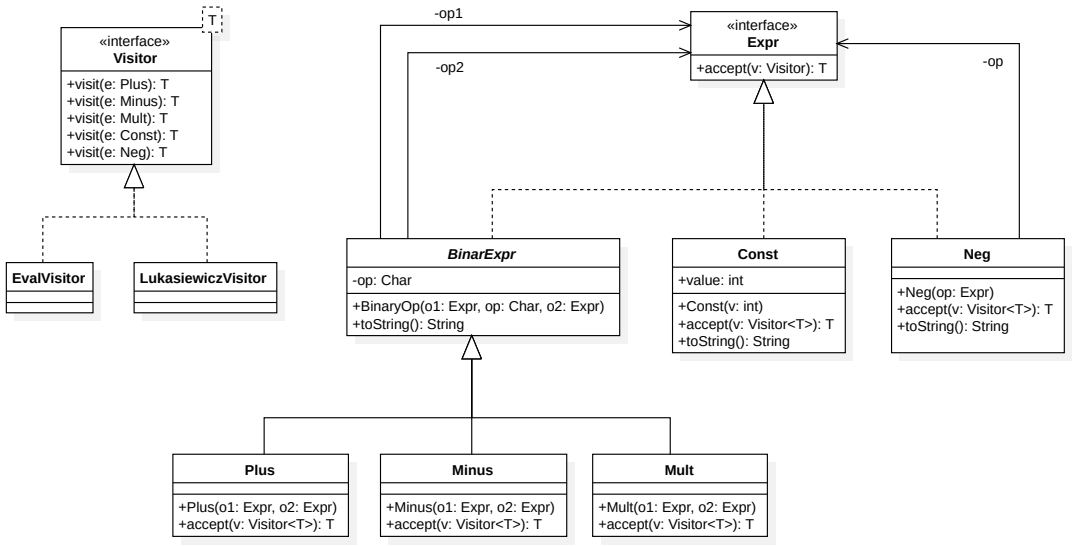

Abbildung 1: Klassendiagramm

d) Für die Darstellung von Quadratwurzeln innerhalb eines Terms implementieren Sie die Klasse `Sqrt`. Die textuelle Darstellung der Operation soll `sqrt(...)` entsprechen. Erweitern Sie den `EvalVisitor` um den Konstruktor `EvalVisitor(Rational eps)`, der den ϵ-Wert für die Berechnung der Quadratwurzel entgegennimmt, und passen Sie die anderen Visitor ebenfalls für die Verwendung von Quadratwurzel-Ausdrücken an.

e) Erweitern Sie die Implementierung dahingehend, dass nun auch Variablen (Klasse `Var`) in Ihren Ausdrücken dargestellt werden können.

f) Überschreiben sie für alle `rational.Expr` implementierenden Klassen die `equals`-Methode. Die `equals`-Methoden sollen sich dabei auf die `equals`-Methoden der enthaltenden Ausdrücke abstützen. Stellen Sie dabei sicher, dass das Kommutativgesetz für die kommutativen Operationen + und · gilt.

g) Erweitern Sie den Konstruktor der Klasse `EvalVisitor` um den Parameter `Map<Var, Rational> vars`. Der Konstruktor bekommt entsprechend eine Map übergeben, die den verschiedenen Variablen ihre Werte zuordnet. Erweitern Sie die `visit`-Methoden entsprechend, dass auch ein Term mit Variablen ausgewertet werden kann.

   Sollte der Wert einer Variablen zum Zeitpunkt der Auswertung nicht bekannt sein, lösen Sie eine Exception vom Typ `rational.UnknownVariableException` aus. Die entsprechende Klasse muss ebenfalls von Ihnen implementiert werden.

h) Erstellen Sie nun die weitere Visitor-Klasse `SimplifyVisitor`. Wie der Name es bereits verrät, soll dieser Visitor den Term nach Möglichkeit vereinfachen. Setzen Sie dabei die folgenden Vereinfachungen um (x, y: beliebige Terme; c: Konstante; d, e: ganzzahlige Exponenten):

   $x + 0 = x$, $x \cdot 1 = x$, $x \cdot 0 = 0$ und die entsprechenden kommutativen Regeln sowie
   $x - 0 = x$, $0 - x = -x$, $-(c) = -c$, $-(-(x)) = x$, $y - (-(x)) = y + x$, $\frac{x}{1} = x$, $\frac{0}{x} = 0$,
   $x^1 = x$, $x^0 = 1$, $1^d = 1$, $0^d = 0$, $(x^d)^e = x^{d \cdot e}$, $\sqrt{0} = 0$, $\sqrt{1} = 1$.

   Hinweise:

   1. Beachten Sie, dass das Resultat des `SimplifyVisitors` wieder ein Term ist. Bedenken Sie zudem, dass sich beispielsweise $x \cdot (y \cdot 0 + 1)$ mit den genannten Regeln zu x

vereinfachen lässt.

2. Gehen Sie beim Vereinfachen von unten nach oben (Bottom-Up) vor, ansonsten müssten Sie das Verfahren mehrfach anwenden.

i) Aus der Analysis kennen Sie die Ableitung von Funktionen. Ist f eine Funktion, bezeichnet

$$f' = \frac{\mathrm{d}f}{\mathrm{d}x}$$

ihre Ableitung nach x. Es gelten die üblichen Rechenregeln für Ableitungen. So hat die Funktionen f mit $f(x) = 2 \cdot x + 3$ die Ableitung f' mit $f'(x) = 2$.

Im Folgenden seien Funktionen durch Ausdrücke wie in den beiden vorherigen Aufgaben definiert. Es gelten die folgenden, üblichen Regeln, um Ableitungen nach x zu berechnen. Dabei seien f und g beliebige solche Ausdrücke, c eine beliebige rationale Zahl, n eine beliebige ganze Zahl und y eine beliebige Variable:

$$\begin{aligned}
c' &= 0 \\
y' &= \begin{cases} 1 & \text{wenn } x = y \\ 0 & \text{sonst} \end{cases} \\
(f \pm g)' &= f' \pm g' \\
(f \cdot g)' &= f' \cdot g + f \cdot g' \\
\left(\frac{f}{g}\right)' &= \frac{f' \cdot g - f \cdot g'}{g^2} \\
\Big(f(g)\Big)' &= f'(g) \cdot g' \\
(f^n)' &= (n \cdot f^{n-1}) \cdot f' \\
\left(\sqrt{f}\right)' &= \frac{f'}{2 \cdot \sqrt{f}}
\end{aligned}$$

Wie üblich, steht $f(g)$ für die Funktion f, in der jedes Vorkommen von x durch g ersetzt wurde. Und f^n is definiert durch $f^0 = 1$ und $f^n = f \cdot f^{n-1}$.

Zur Realisierung von Ausdrücken nutzen wir wieder die bisherige Modellierung und Implementierung. Die Erzeugung der Ableitung eines Ausdrucks soll nach obigen Regeln mit Hilfe des Visitor-Patterns erfolgen, d.h. durch Erstellung einer konkreten Besucherklasse, die das `Visitor`-Interface implementiert.

Vervollständigen Sie die Implementierung durch die Klasse `DerivativeVisitor`. Angewandt auf einen Ausdruck, erzeugt dieser Besucher die Ableitung nach x als neuen Ausdruck.

Hinweis: Die Implementierung jeder `visit`-Methode wird durch genau eine der obigen Regeln festgelegt. Nutzen Sie den `SimplifyVisitor`, um Ableitungen zu vereinfachen.

j) Ändern Sie den `DerivativeVisitor` so, dass Ableitungen nicht nur nach x, sondern nach beliebigen Variablen berechnet werden können.

k) Ein Ausdruck f hat eine Nullstelle $q \in \mathbb{Q}$ in x, wenn die Auswertung des Terms f den Wert 0 ergibt, wenn x den Wert q hat. Ein bekanntes Verfahren zur näherungsweisen Bestimmung von Nullstellen ist das *Newton-Verfahren*. Erstellen Sie zu seiner Implementierung die Klasse `rational.Newton`. Sie muss den Konstruktor

```
Newton(Expr f, Var x)
```

und die Methode

```
Rational approximate(Rational start, Rational eps)
```

zur Verfügung stellen. Um eine Nullstelle q von f in x zu bestimmen, wird

```
new Newton(f, x).approximate(start, eps)
```

aufgerufen. Die Zahl `start` ist der Startwert q_0 des Näherungsverfahrens und `eps` ein Schwellwert ϵ. Das Verfahren endet, sobald sich die Näherungswerte zweier aufeinanderfolgender Iterationen um weniger als `eps` unterscheiden, d.h. $|q_i - q_{i+1}| < \epsilon$. Als Ergebnis wird dann q_{i+1} zurückgegeben.

Beachten Sie, dass nicht jeder Ausdruck eine Nullstelle hat und das Newton-Verfahren in einem Durchlauf nicht mehr als eine bestimmen kann. In bestimmten Fällen konvergiert das Newton-Verfahren nicht. Ihre Implementierung soll daher nach 15 Iterationen eine `IllegalArgumentException` werfen, wenn das o.g. Abbruchkriterium zuvor nicht erfüllt wurde.