

## Miniprojekt 2: Programmierpraktikum 2020

Ausgabe: 19. Mai 2020  
Abgabe: 02. Juni 2020, 15 Uhr

### GitLab Team Repositories

Wir verwenden auch für das zweite Miniprojekt wieder die GitLab Repositories. Führen Sie den Befehl `git pull` aus, damit die Vorlagen heruntergeladen werden. Sie sollten nun einen neuen Ordner MP2 sehen, den Sie wie gewohnt in IntelliJ IDEA öffnen können. Es handelt sich wieder um ein Gradle Projekt, sodass Abhängigkeiten automatisch heruntergeladen werden.

Ihre Lösungen müssen Sie in das lokale Repository committen (`git commit`) und dann auf den GitLab Server übertragen (`git push`). Sprechen Sie sich zur Bearbeitung der Aufgaben mit Ihren Abgabepartnern\*innen ab, damit Sie unnötige Konflikte vermeiden.

### Vorlesungsumfrage

Wir sind an Ihrem Feedback interessiert. Bitte nehmen Sie sich die Zeit, die anonyme Vorlesungsumfrage auszufüllen. Studierende mit @cs.uni-kl.de Mailadresse haben dazu bereits eine E-Mail erhalten, alle anderen können sich auf <https://vlu.cs.uni-kl.de/teilnahme/> z. B. mit ihrer RHRK Mailadresse einen Link anfordern. Ihr Feedback hilft uns, die Vorlesung und die Übungen weiter zu verbessern.

# Mini CAS

Bei Ihrem Job als Mathe-Nachhilfelehrer\*in müssen Sie viele Aufgaben korrigieren. Um sich die Arbeit etwas zu erleichtern, beschließen Sie ein kleines Computer Algebra System (CAS) zu entwickeln, mit dem Sie Ableitungen von gängigen eindimensionalen Funktionen symbolisch bestimmen und plotten können.

## Aufgabe 1 Funktionen

Wir beginnen damit, einige elementare Funktionen zu definieren, mit denen später komplexere Funktionen aufgebaut werden können. Unsere Anforderungen sind, dass

1. aus einer Funktion eine textuelle Repräsentation generiert werden kann,
2. eine Funktion an einer Stelle  $x$  ausgewertet werden kann
3. und dass die Ableitung der Funktion berechnet werden kann.

Dies manifestiert sich in der folgenden Schnittstelle:

```
public interface Function {
    String toString();
    double apply(double x);
    Function derive();
    default Function simplify() {
        return this;
    }
}
```

In dieser ersten Aufgabe müssen nur die Methoden `toString`, `apply` und `derive` implementiert werden. Die Methode `simplify` ist als default-Methode hinterlegt und kann in der freiwilligen Aufgabe 2 implementiert werden.

**Implementieren Sie** folgende Funktionen im Paket `de.tukl.programmierpraktikum2020.mp2.functions`.

- a) Die Klasse `public class Const implements Function` soll eine konstante Zahl vom Typ `double` repräsentieren. Diese Zahl soll dem einstelligen Konstruktor übergeben werden. Der Aufruf von `apply(double x)` soll für jeden Wert  $x$  die bei der Initialisierung vordefinierte Zahl zurückgeben. Die Ableitung einer konstanten Zahl ist die konstante Zahl `0`.

Beispiel:

```
Function f = new Const(3.14); // f(x) = 3.14
f.apply(-42.123);           // => 3.14
f.apply(0.0);               // => 3.14
f.toString();               // => "3.14"
f.derive().toString();     // => "0.0"
```

- b) Da wir nur eindimensionale Funktionen  $f(x)$  betrachten möchten, brauchen wir eine Klasse, welche die Variable  $x$  repräsentiert. Implementieren Sie dazu die Klasse `X` (groß geschrieben, mit nullstelligem Konstruktor). Beispiel:

```
Function f = new X(); // f(x) = x
f.apply(-42.123);    // => -42.123
f.apply(0.0);        // => 0.0
f.toString();        // => "x"
f.derive().toString(); // => "1.0"
```

- c) Implementieren Sie die Exponentialfunktion als Klasse `Exp`. Der einstellige Konstruktor soll als Argument eine Funktion erwarten. Beachten Sie beim Ableiten die Kettenregel! Beispiel:

```
Function f = new Exp(new X()); // f(x) = Eulersche Zahl e hoch x
f.apply(0.0); // => 1.0
f.apply(1.0); // => 2.718...
f.toString(); // => "exp(x)"
f.derive().toString(); // => "exp(x) * 1.0"
```

- d) Implementieren Sie die Sinusfunktion als Klasse `Sin`. Der einstellige Konstruktor soll als Argument eine Funktion erwarten.
- e) Implementieren Sie die Cosinusfunktion als Klasse `Cos`. Der einstellige Konstruktor soll als Argument eine Funktion erwarten.
- f) Die Additionsfunktion soll als Klasse `Plus` mit einem zweistelligen Konstruktor implementiert werden, der die beiden übergebenen Funktionen addiert. Beispiel:

```
Function f = new Plus(new Const(42.0), new X()); // f(x) = 42.0 + x
f.apply(0.0); // => 42.0
f.apply(10.0); // => 52.0
f.toString(); // => "(42 + x)"
f.derive().toString(); // => "(0.0 + 1.0)"
```

- g) Die Multiplikationsfunktion soll als Klasse `Multi` implementiert werden, welche die beiden im Konstruktor übergebenen Funktionen multipliziert.
- h) Entsprechend soll die Division als Klasse `Div` implementiert werden, welche die erste im Konstruktor übergebene Funktion durch die zweite teilt.
- i) *Freiwillige Zusatzaufgabe:* Implementieren Sie den natürlichen Logarithmus (Basis Eulersche Zahl  $e$ ) in der Klasse `Log` (einstelliger Konstruktor).
- j) *Freiwillige Zusatzaufgabe:* Implementieren Sie die Potenzfunktion `Pow`, deren Konstruktor zwei Funktionen  $g$  und  $h$  erwartet und die Potenz  $f(x) = g(x)^{h(x)}$  repräsentiert.

*Hinweis: Um die Ableitung der meisten Funktionen oben angeben zu können, brauchen Sie bereits eine oder mehrere der anderen Funktionen. Dieses Problem können Sie umgehen, indem Sie zuerst einmal die Klassen aus a) bis h) anlegen und die Methoden `toString` und `apply` implementieren. In den `derive` Methoden können Sie zunächst einfach nur `null` zurückgeben. Sind dann alle Funktionen soweit implementiert, lassen sich die Ableitungen mit Hilfe der anderen Funktionen ausdrücken.*

*Hinweis: Für die Berechnungen in `apply` dürfen Sie `java.lang.Math` aus der Standardbibliothek verwenden. Alle Klassen aus `java.lang` müssen übrigens nicht extra importiert werden, sie können also z. B. direkt `Math.exp` in Ihrem Code verwenden.*

*Hinweis: In der Klasse `de.tu1.programmierpraktikum2020.mp2.Util` gibt es eine Funktion `plotFunction`, die als Argumente eine Funktion, eine linke Grenze  $x_{min}$  sowie eine rechte Grenze  $x_{max}$  erwartet. Sie können diese in der `main`-Methode der `Calculus`-Klasse aufrufen, um eine Funktion und deren Ableitung zu plotten. Überprüfen Sie damit Ihre Implementierung grafisch. **Es gibt ansonsten für diesen Aufgabenteil keine Testfälle!** Sobald Aufgabe 3 gemacht ist, wird Aufgabe 1 implizit mitgetestet.*

## Aufgabe 2 Vereinfachung (freiwillige Zusatzaufgabe)

Wie Ihnen vielleicht bereits aufgefallen ist, sind die abgeleiteten Funktionen teilweise in einer nicht sehr leserlichen Form, insbesondere durch 0- und 1-Faktoren aus der Kettenregel. Implementieren Sie die Methode `simplify` in den einzelnen Funktionsklassen, um entsprechende Vereinfachungen durchzuführen.

Beginnen Sie evtl. mit simpleren Vereinfachungen, z. B.  $f(x) \cdot 0 = 0$ ,  $f(x) \cdot 1 = f(x)$ ,  $f(x) + 0 = f(x)$ , und fahren Sie dann mit Vereinfachungsregeln für Brüche, Logarithmen usw. fort.

*Hinweis: Mit `instanceof` können Sie prüfen, ob ein Objekt Instanz einer bestimmten Klasse ist.*

## Aufgabe 3 Parser

Die Eingabe von Funktionen ist bisher noch etwas umständlich: Wir müssen die entsprechenden Java-Objekte vor dem Kompilieren erstellen und können zur Laufzeit keine Änderungen mehr vornehmen. Eigentlich möchten wir Formeln als `String` zur Laufzeit eingeben und daraus ein Objekt vom Typ `Function` erhalten. Um den Implementierungsaufwand dafür in Grenzen zu halten, müssen Sie keinen kompletten Parser schreiben. Stattdessen verwenden wir den Parser Generator *Antlr*<sup>1</sup>. Dabei handelt es sich um ein Programm, das aus einer Grammatikspezifikation automatisch Java-Code für einen Parser generiert. Wir haben Ihnen die Grammatik in `MP2/src/main/antlr/Function.g4` bereits vorgegeben.

In IntelliJ IDEA kann nun der Parser-Code generiert werden, indem Sie auf das grüne Build Icon klicken. An dieser Stelle ist sehr schön der Vorteil eines Build-Tools sichtbar: Gradle erkennt automatisch, dass Quelldateien generiert werden müssen, da das `build` Target vom `generateGrammarSource` Target abhängt. Alternativ können Sie auch direkt das Gradle Target `generateGrammarSource` ausführen. Die generierten Dateien werden im Verzeichnis `MP2/build/generated-src` abgelegt und sollten nicht manuell verändert werden!

Der generierte Parser folgt dem Visitor-Entwurfsmuster<sup>2</sup>. Die vom Parser aus der Texteingabe generierte Baumstruktur<sup>3</sup> wird vom allgemeinen Besucher `FunctionBaseVisitor` durchlaufen. Ihre Aufgabe ist es nun, diese Besucherklasse so zu erweitern, dass zu jedem besuchten Element des Syntaxbaums eine entsprechende `Function` zurückgegeben wird. Daraus ergibt sich dann insgesamt der abstrakte Syntaxbaum mit Elementen vom Typ `Function`.

Implementieren Sie dazu die Klasse `public class Parser extends FunctionBaseVisitor<Function>` im Paket `de.tukl.programmierpraktikum2020.mp2`. Importieren Sie dort die Klassen `FunctionBaseVisitor` und `FunctionParser` aus dem generierten Paket `de.tukl.programmierpraktikum2020.mp2.antlr`.

In `FunctionBaseVisitor` können Sie nachschauen, welche Methoden überschrieben werden müssen. Beim Vergleich mit der Grammatik `Function.g4` fällt auf, dass Antlr für die Regeln `expr` und `var` für jede mit einem Label (`#`-Zeichen) gekennzeichneten Alternative eine entsprechende Methode generiert hat. In einigen Regelalternativen sind zudem Regelemente mit dem `"="`-Zeichen gelabelt, z. B. ein Operator `op` oder ein linker Teilausdruck `lexpr`. Damit können Sie z. B. in der Methode `visitMultExpr` mit `ctx.lexpr` auf den linken Faktor der Multiplikation zugreifen und diesen mit `this.visit(ctx.lexpr)` "besuchen". Ist nur ein Unterausdruck ohne Label vorhanden, erfolgt der Zugriff über `ctx.expr()` bzw. `ctx.var()`. Mit `ctx.op.getType()` können Sie den Typ des mit `op` gelabelten Operators identifizieren und mit den entsprechenden Attributen in `FunctionParser` vergleichen (z. B. `FunctionParser.MULT`).

Die Benutzer\*in Ihres Programms soll auf der Kommandozeile eine Funktion sowie die Grenzen `xmin` und `xmax` eingeben können. Rufen Sie dazu die Methoden `Util.promptFunction`, `Util.promptXMin` und `Util.promptXMax` in der `main` Methode der Klasse `Calculus` auf und plotten Sie die eingegebene Funktion im ausgewählten Intervall mit Hilfe der Methode `Util.plotFunction`. Der Methode `Util.promptFunction` müssen Sie als Argument eine Instanz Ihrer Parser Klasse übergeben.

Testen Sie Ihr Mini CAS mit eigenen Eingaben und überprüfen Sie die Ableitungsergebnisse und die Plots ggf. mit einem anderen CAS<sup>4</sup>.

Beispiele:

- Gauß-Glocke (nicht normiert):  $\exp(-x \cdot x)$
- Sinc-Funktion:  $\sin(x)/x$
- Fourierreihe der Rechteckfunktion (bis 6. Ordnung):  
 $\sin(x) + \sin(3x)/3 + \sin(5x)/5 + \sin(7x)/7 + \sin(9x)/9 + \sin(11x)/11$

---

<sup>1</sup><https://www.antlr.org/>

<sup>2</sup>[https://de.wikipedia.org/wiki/Besucher\\_\(Entwurfsmuster\)](https://de.wikipedia.org/wiki/Besucher_(Entwurfsmuster))

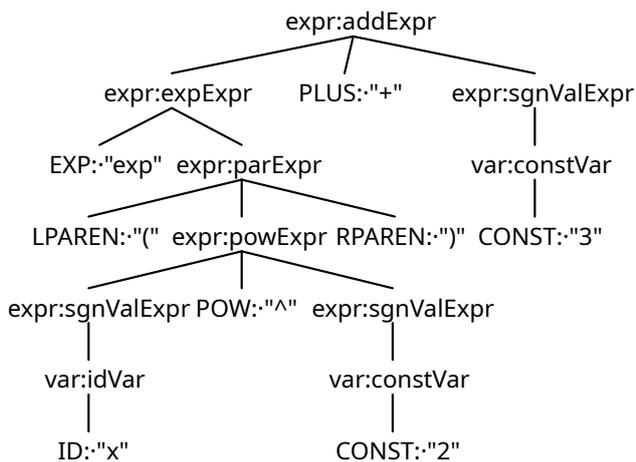
<sup>3</sup>s. Beispiel in der Abbildung in Aufgabe 4

<sup>4</sup>z. B. <https://www.wolframalpha.com/>

## Aufgabe 4 Erweiterungen (freiwillige Zusatzaufgabe)

Machen Sie sich etwas mit der Antlr Dokumentation vertraut<sup>5</sup>, insbesondere mit dem Abschnitt "Parser Rules" und erweitern Sie anschließend die Grammatik um eigene Funktionen, zum Beispiel `tan`, `acos`, `asin`, `abs`, usw. Erweitern Sie entsprechend die Parser Klasse und implementieren Sie zugehörige Function Klassen.

Beim Erweitern der Grammatik ist evtl. das Antlr Plugin<sup>6</sup> für IntelliJ IDEA hilfreich, da Eingaben damit direkt im Panel "ANTLR Preview" getestet und in einem Syntaxbaum grafisch dargestellt werden können. Dazu muss in `Function.g4` mit der rechten Maustaste auf die Parserregel `expr` geklickt und der Menüpunkt "Test rule expr" ausgewählt werden. So wird z. B. aus der Eingabe `exp(x^2)+3` der folgende Syntaxbaum generiert.



<sup>5</sup><https://github.com/antlr/antlr4/blob/master/doc/index.md>

<sup>6</sup><https://github.com/antlr/intellij-plugin-v4>