Prof. Dr. Ralf Hinze
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Miniprojekt 2: Programmierpraktikum 2023

Ausgabe: 23.05.2023
Abgabe: 07.06.2023, 23:59 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 Abgabepartner*innen ab, damit Sie unnötige Konflikte vermeiden.

Anmeldung im Prüfungsamt

Bitte denken Sie daran, das Programmierpraktikum rechtzeitig vor Ende des Anmeldezeitraums bei dem für Sie zuständigen Prüfungsamt (für die meisten von Ihnen also im QIS) anzumelden.

Datentypen zur Darstellung von Zahlen

In Java gibt es verschiedene Typen zur Darstellung von Zahlen. Für die ganze Zahlen $\mathbb{Z}$ gibt es die Typen `int`, `long` und `BigInteger`. Der Typ `int` umfasst die ganze Zahlen von -2^{31} bis $2^{31}-1$, der Typ `long` die von -2^{63} bis $2^{63}-1$. Der Typ `BigInteger` kann ganze Zahlen unbeschränkter Größe darstellen. In der Praxis gibt es auch hier natürlich ein Limit durch den zur Verfügung stehenden Speicher, aber das ist weitaus größer als die Grenzen von `int` und `long`. Der Grund für die verschiedenen Optionen ist, dass diese im Hintergrund unterschiedlich umgesetzt sind. Während Rechenoperationen mit `int` ein einzelner Prozessor-Befehl sind, müssen für große `BigInteger` viele Prozessor-Befehle in einer Schleife ausgeführt werden. Daher sind `int` und `long` performanter als `BigInteger` und vorzuziehen, wenn die Grenzen kein Problem sind.

Für rationale Zahlen $\mathbb{Q}$ wäre eine Bruchdarstellung zweier ganzer Zahlen denkbar. Einige Programmiersprachen bieten das an und können so rationale Zahlen beliebiger Größe darstellen. In der Java-Standardbibliothek ist eine solche Darstellung nicht enthalten.

Die reellen Zahlen $\mathbb{R}$ lassen sich nicht so leicht abbilden, da die Menge überabzählbar groß ist. Hier hat sich der Standard IEEE 754[1] etabliert, der auch von den Java-Typen `float` und `double` umgesetzt wird. Nachteil an diesen Typen ist jedoch, dass es sich um keine genaue Darstellung handelt und viele auch rationale Zahlen wie z.B. die Zahl 0.1 nicht präzise darstellbar sind. So ergibt $0.1 + 0.1 + 0.1 = 0.30000000000000004$. Mehr Genauigkeit bietet der Typ `BigDecimal`[2], der Zahlen intern in der Form $a \cdot 10^{-b}$ darstellt, wobei a ein `BigInteger` und b ein `int` ist. Viele Operationen (Addition, Subtraktion, Multiplikation) lassen sich so exakt berechnen. Bei der Division ist das nicht immer möglich, $\frac{1}{3}$ zum Beispiel müsste eine unendliche Folge von dreien in a abspeichern. Hier erlaubt es der Typ, die gewünschte Präzision anzugeben und rundet auf entsprechend viele Nachkommastellen. Die erhöhte Genauigkeit von `BigDecimal` hat gegenüber `float` und `double` natürlich auch wieder Nachteile in der Performance.

Wir verwenden in diesem Miniprojekt den Typ `BigDecimal`. Hier ein paar Tipps zum Umgang mit dem Typ:

- Die Zahlen 0 und 1 stehen als Konstanten `BigDecimal.ZERO` und `BigDecimal.ONE` zur Verfügung.
- Andere ganze Zahlen können Sie mit `new BigDecimal(42)` oder `new BigDecimal(-42)` erzeugen.
- Sonstige Zahlen, insbesondere Kommazahlen, sollten Sie mit `new BigDecimal("123.4567")` erzeugen.
- Für die Rechenoperationen Addition, Subtraktion, Multiplikation, Division sowie die Potenzfunktion haben wir die Klasse `MathUtils` mit den statischen Methoden `add`, `subtract`, `multiply`, `divide` und `pow` bereitgestellt. Die Methoden erwarten jeweils zwei `BigDecimal` Zahlen als Eingabe und geben den berechneten `BigDecimal` zurück. Die Präzision zur Rundung ist dabei von uns vorgegeben.
- Für die Exponentialfunktion e^x, den natürlichen Logarithmus sowie die trigonometrischen Funktionen Sinus und Cosinus bietet `MathUtils` die statischen Methoden `exp`, `log`, `sin` und `cos`. Die Methoden erwarten jeweils eine `BigDecimal` Zahl als Eingabe und geben den berechneten `BigDecimal` zurück. Auch hier ist die Präzision zur Rundung von uns vorgegeben.
- Zum Vergleichen zweier `BigDecimal` können Sie die `equals` und `compareTo` Methoden direkt auf den `BigDecimal` Objekten aufrufen. Dabei ist `a.equals(b) = true` wenn `a` und `b` den gleichen Wert repräsentieren. `a.compareTo(b)` ist < 0, $= 0$, bzw. > 0, wenn der von `a` repräsentierte Wert kleiner, gleich bzw. größer als der von `b` repräsentierte Wert ist.

[1] https://de.wikipedia.org/wiki/IEEE_754
[2] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/math/BigDecimal.html

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();
    BigDecimal apply(BigDecimal 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.rptu.programmierpraktikum2023.mp2.functions`.

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

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(new BigDecimal("3.14")); // f(x) = 3.14
f.apply(new BigDecimal("-42.123")); // => 3.14
f.apply(BigDecimal.ZERO);           // => 3.14
f.toString();                       // => "3.14"
f.derive().toString();              // => "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(new BigDecimal("-42.123")); // => -42.123
f.apply(BigDecimal.ZERO);           // => 0
f.toString();                       // => "x"
f.derive().toString();              // => "1"
```

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(BigDecimal.ZERO); // => 1
f.apply(BigDecimal.ONE);  // => 2.718...
f.toString();             // => "exp(x)"
f.derive().toString();    // => "exp(x) * 1"
```

d) Implementieren Sie die Sinusfunktion als Klasse `Sin`. Der einstellige Konstruktor soll als Argument eine Funktion erwarten.

```
Function f = new Sin(new X()); // f(x) = sin(x)
f.apply(BigDecimal.ZERO);           // => 0
f.apply(new BigDecimal(Math.PI/2)); // => ≈1, aber minimale Rundungsdifferenz
f.derive().toString();              // => cos(x) * 1
```

e) Implementieren Sie die Cosinusfunktion als Klasse `Cos`. Der einstellige Konstruktor soll als Argument eine Funktion erwarten.

f) Die Additionsfunktion soll als Klasse `Add` mit einem zweistelligen Konstruktor implementiert werden, der die beiden übergebenen Funktionen addiert. Beispiel:

```
Function f = new Add(new Const(new BigDecimal(42)), new X()); // f(x) = 42 + x
f.apply(BigDecimal.ZERO);    // => 42
f.apply(new BigDecimal(10)); // => 52
f.toString();                // => "(42 + x)"
f.derive().toString();       // => "(0 + 1)"
```

g) Die Multiplikationsfunktion soll als Klasse `Mult` 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.

k) In der Klasse `de.rptu.programmierpraktikum2023.mp2.Util` gibt es eine statische Methode `plotFunction`, die als Argumente eine `Function`, eine linke Grenze `xmin` sowie eine rechte Grenze `xmax` erwartet. Rufen Sie diese in der `main`-Methode der `Calculus`-Klasse auf, um eine Funktion und deren Ableitung zu plotten. Überprüfen Sie damit Ihre Implementierung grafisch. Es gibt ansonsten für Aufgabe 1 keine Testfälle! Sobald Aufgabe 3 implementiert ist, wird Aufgabe 1 implizit mitgetestet.

   Bereiten Sie für die Projektabnahme Codezeilen vor, um alle im Anhang gezeigten Beispiele zu plotten. Binden Sie zudem Screenshots der generierten Grafiken in die Präsentation für die Projektabnahme ein. Die Grafiken sollten genauso aussehen wie die Beispiele im Anhang.

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*[3]. 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[4]. Die vom Parser aus der Texteingabe generierte Baumstruktur[5] 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.rptu.programmierpraktikum2023.mp2`. Importieren Sie dort die Klassen `FunctionBaseVisitor` und `FunctionParser` aus dem generierten Paket `de.rptu.programmierpraktikum2023.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 Regelelemente 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[6].

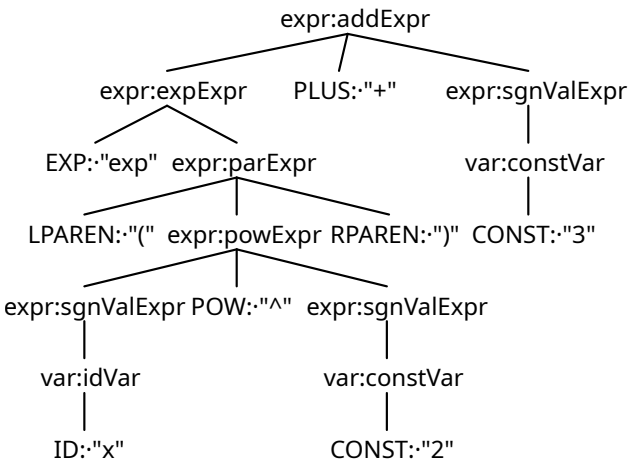
Hinweis: Funktionen mit Vorzeichen werden vom Parser nicht unterstützt. Für die Funktion `f(x) = -sin(x)` müssen Sie also beispielsweise "`-1 * sin(x)`" eingeben.

[3] https://www.antlr.org/
[4] https://de.wikipedia.org/wiki/Besucher_(Entwurfsmuster)
[5] s. Beispiel in der Abbildung in Aufgabe 4
[6] z. B. https://www.wolframalpha.com/

Aufgabe 4. Erweiterungen (freiwillige Zusatzaufgabe)

Machen Sie sich etwas mit der Antlr Dokumentation vertraut[7], 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[8] 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.

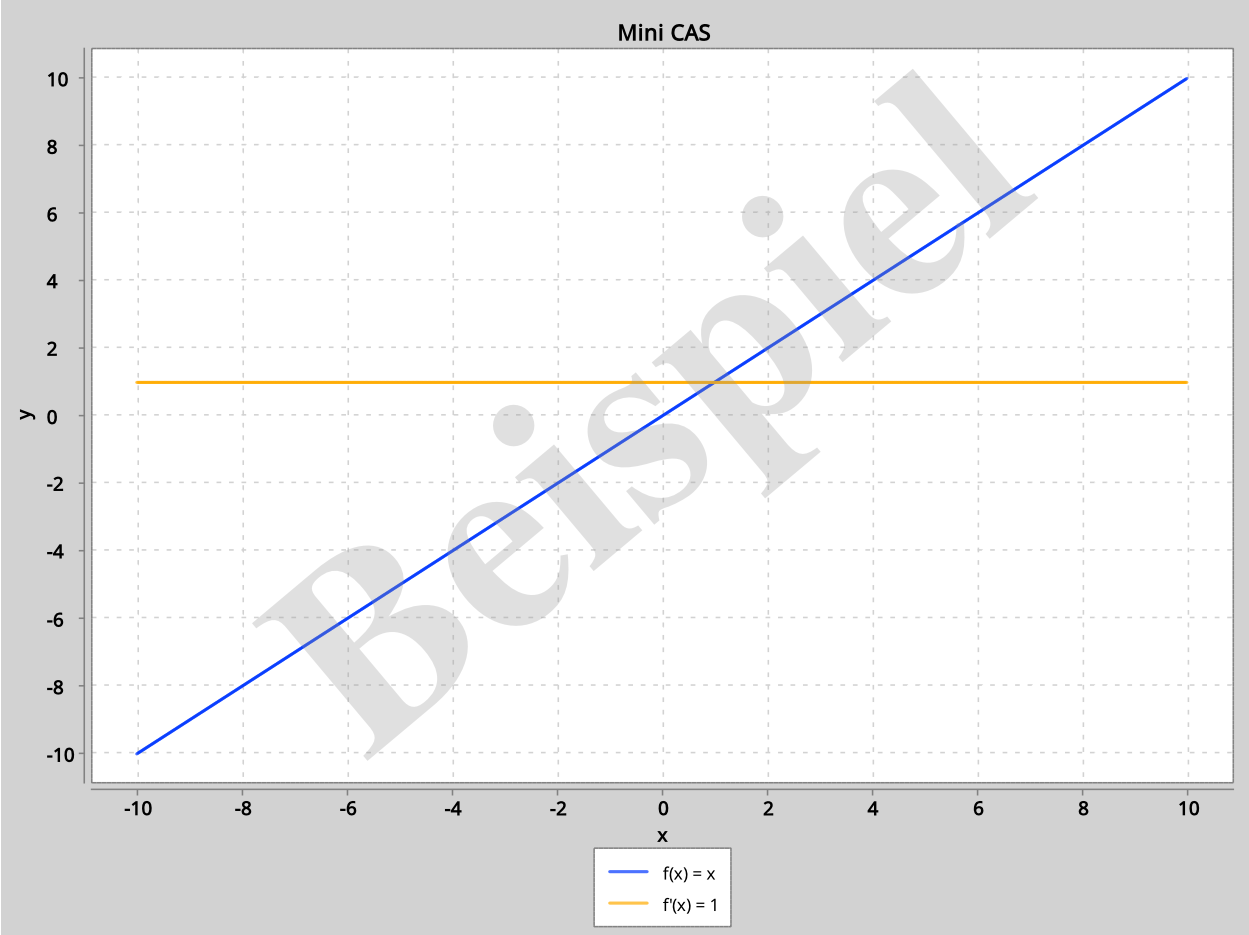


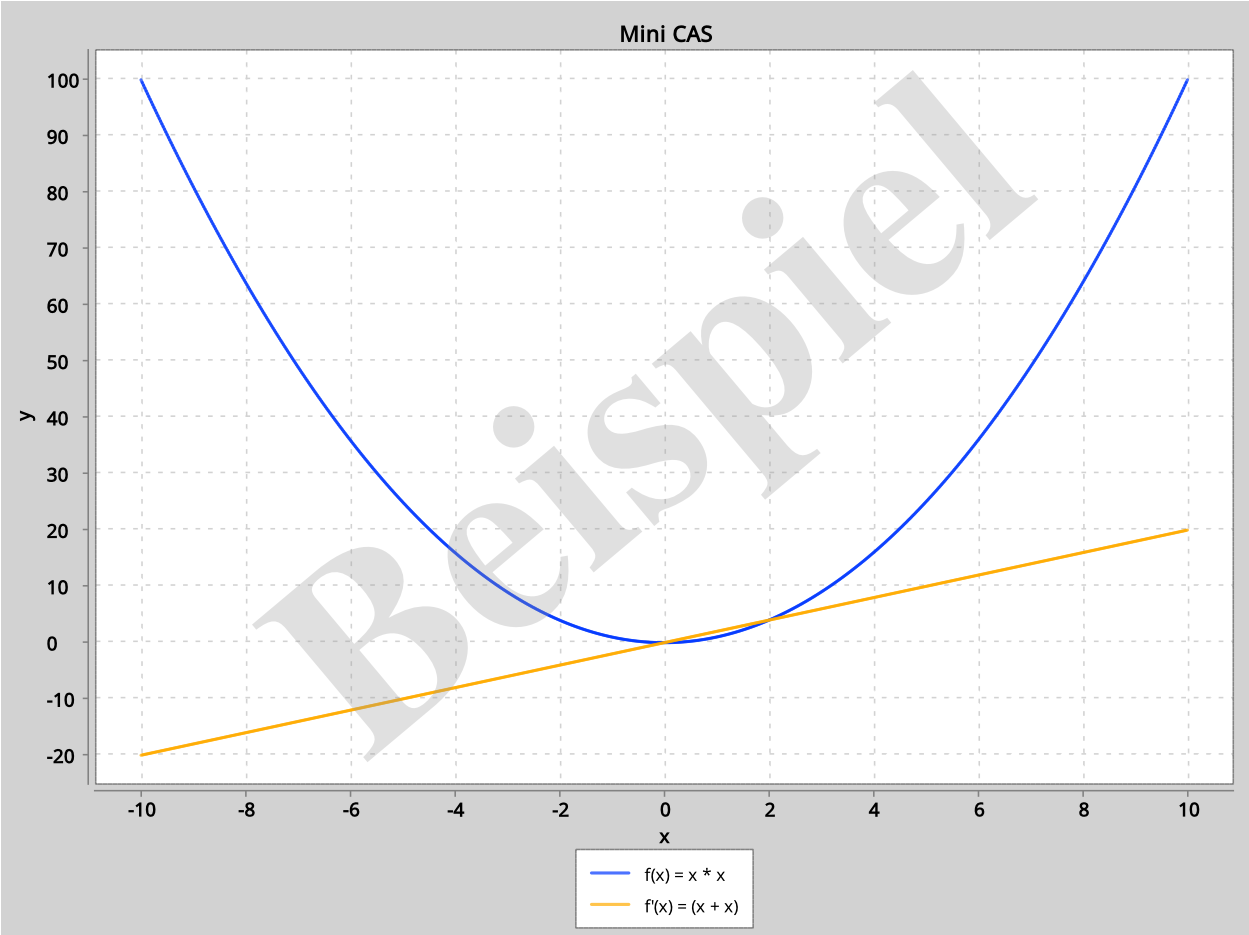
Aufgabe 5. Präsentation für Projektabnahme

Bereiten Sie die Präsentation für die Projektabnahme vor (siehe Deckblatt von Miniprojekt 1).

[7] https://github.com/antlr/antlr4/blob/master/doc/index.md
[8] https://github.com/antlr/intellij-plugin-v4

A. Beispiele


Mini CAS
10
8
6
4
2
0
-2
-4
-6
-8
-10
y
-10
-8
-6
-4
-2
0
2
4
6
8
10
x
f(x) = x
f'(x) = 1
Beispiel


Mini CAS
100
90
80
70
60
50
40
30
20
10
0
-10
-20
y
-10
-8
-6
-4
-2
0
2
4
6
8
10
x
f(x) = x * x
f'(x) = (x + x)
Beispiel