Prof. Dr. Stephan Kleuker	Theoretische Informatik theoriesammlung Version 1.01

Dieses Dokument fasst verschiedene Möglichkeiten zusammen, wie mit der Bibliothek theoriesammlung verschiedene Kernthemen der theoretischen Informatik bearbeitet werden können. Die Sammlung liegt als Prototyp in Java vor und basiert auf den elementaren Standard-Algorithmen, die zum Beweis von Zusammenhängen und Eigenschaften genutzt werden. Die Algorithmen sind nicht optimiert und haben deshalb meist eine schwache Performance. Weiterhin ist der Code des Prototypen nicht überarbeitet, es fehlt z. B. eine sinnvolle Refaktorierung, mit der u. a. einige Methoden aufgeteilt werden müssten. Der Prototyp ist im Rahmen der Vorbereitung einer Vorlesung entstanden und ermöglicht es Lösungen zu Praktikumsaufgaben zu überprüfen, aber auch eigene Aufgaben zu erstellen und zu prüfen. Der Prototyp wird nicht wesentlich überarbeitet werden.
Zum Start der Nutzung wird ein Blick auf die ersten beiden Kapitel empfohlen. Es wird erklärt, welche eventuell nicht bekannten Java-Anteile und JUnit relevant sind. Die nachfolgenden Kapitel zu den verschiedenen Maschinen-Modellen sind unabhängig voneinander, also in beliebiger Reihenfolge, lesbar. Ausnahmen für kleine Details sind im Text erwähnt.

Inhalt

1 Java-Spezialitäten

Da in Grundlagen-Vorlesungen zur objektorientierten Programmierung Java als Beispiel-Programmiersprache genutzt wird, wird typischerweise nicht auf spezielle Möglichkeiten der Sprache eingegangen. Hier werden sehr kurz zwei Ideen vorgestellt, die bei der Nutzung der vorgestellten Bibliothek hilfreich sein können.

Im klassischen Java ist es immer etwas nervig, wenn Textblöcke angegeben werden sollen, die gut lesbar sein sollen. Ein typisches Beispiel sind SQL-Anfragen.

```
try (ResultSet rs = stmt.executeQuery(
      "SELECT City.Name,Longitude,Latitude "
      + " FROM City, Country "
      + " WHERE City.Country = Country.Code "
      + "   AND Country.Name = '" + country + "'")) {
```

Dies wird ab Java 15 durch Java Text Blocks etwas vereinfacht. Dazu kann ein String mit drei Anführungszeichen beginnen und danach wird der Text, der in der Folgezeile beginnen muss, mit seiner Formatierung, also auch den Zeilenumbrüchen und Leerzeichen übernommen. Der String endet wieder mit drei Anführungszeichen.

```
try (ResultSet rs = stmt.executeQuery("""
       SELECT City.Name,Longitude,Latitude
         FROM City, Country
         WHERE City.Country = Country.Code
           AND Country.Name = '""" + country + "'")) {
```

Ergänzend werden Leerzeichen, die nach dem ersten Zeilenumbruch direkt folgen in dieser und allen folgenden Zeilen einfach ignoriert. Dadurch wird eine einfach lesbare Einrückung ermöglicht. Die Anzahl der ignorierten Leerzeichen wird auch durch die Position der schließenden Anführungszeichen definiert. Das folgende Beispiel zeigt bei s1, dass immer zwei Leerzeichen vor jeder Zeile stehen, da die schließenden Leerzeichen zwei Zeichen vor dem Text stehen.

Der String s1 enthält am Ende einen Zeilenumbruch, ist das nicht gewünscht, stehen die schließenden Anführungszeichen in der letzten Zeile des Textes, wie es mit String s2 gezeigt wird. Anführungszeichen können direkt geschrieben werden, können aber auch einen Escape-Strich vorgestellt bekommen. Dies ist notwendig, wenn mehr als drei Anführungsstriche direkt hintereinander verarbeitet werden sollen. Ein einfacher Escape-Strich (Backslash) führt dazu, dass ein Zeilenumbruch ignoriert wird.

```
public static void main(String[] args) {
  String s1 = """
       kein
       wirklich
         sinnvoller
       Text
     """;
  System.out.println("String:\n" + s1);
  String s2 = """
      kein \
      wirklich \\
        "sinnvoller\"
      Text """;
```

```
    System.out.println("String:\n" + s2);
    String[] zeilen = s2.split("\\n");
    for (String s:zeilen) {
      System.out.print("(" + s + ") ");
    }
    System.out.println();
  }
```

Die Ausgabe lautet:

```
String:
  kein
  wirklich
    sinnvoller
  Text

String:
kein wirklich \
  "sinnvoller"
Text
(kein wirklich \)  (  "sinnvoller")  (Text)
```

Da es sich um ganz normale Strings handelt, findet eine Bearbeitung mit den üblichen Methoden statt, wie im obigen Beispiel ergänzend angedeutet.

Für die Nutzung der Bibliothek ist nur relevant, dass so Maschinenmodelle entweder in Text-Dateien oder direkt im Programm-Code stehen können.

Die zentrale Erfolgsidee der Objektorientierung ist die Nutzung von Interfaces beziehungsweise abstrakter Klassen. Klassisch werden Interfaces durch neue Klassen realisiert, die die Methoden des Interfaces ausprogrammieren.

```
public interface Beispielinterface {
  public int mach(int x, int y);
}

public class Realisierung implements Beispielinterface {

  @Override
  public int mach(int x, int y) {
    return x + y;
  }
}
```

Java bietet aber auch die Möglichkeit Klassen beziehungsweise Objekte einer sonst anonymen Klasse direkt im Programmcode zu schreiben, was den Code kompakter und meist auch lesbarer macht. Einige Möglichkeiten werden im folgenden Code vorgestellt.

```
  public static void main(String[] args) {
    Beispielinterface b = new Realisierung();
    System.out.println("A: " + b.mach(41, 1));

    // direkte Angabe eines Objekts einer anonymen Klasse
```

```
    // jede Methode muss implementiert sein
    b = new Beispielinterface() {
      @Override
      public int mach(int x, int y) {
        return x - y;
      }
    };
    System.out.println("B: " + b.mach(40, 2));

    // da das Interface nur eine Methode hat (FunctionalInterface)
    // ist auch die Angabe eines Lambda-Ausdrucks moeglich
    b = (int u, int v) -> {
      // weitere Code-Zeilen moeglich
      return v/u;
    };
    System.out.println("C: " + b.mach(40, 2));

    // da Parametertypen eindeutig, geht auch
    b = (int u,int v) -> {
      return u/v;
    };
    System.out.println("D: " + b.mach(40, 2));

    // wenn nur einzeilig, dann noch kuerzer
    b = (u,v) -> u*v;
    System.out.println("E: " + b.mach(40, 2));

    // die Klasse Math hat eine Klassenmethode max, die
    // zwei int uebergeben bekommt und ein int als Ergebnis
    // liefert. Das ist die gleiche Signatur wie mach, also
    // 2 int rein, ein int raus, dann geht noch kuerzer
    b = Math::max;
    System.out.println("F: " + b.mach(40, 2));
  }
```

Die zugehörige Ausgabe lautet:

```
A: 42
B: 38
C: 0
D: 20
E: 80
F: 40
```

Interfaces werden an verschiedenen Stellen der Bibliothek genutzt und können bei eigenen Experimenten hilfreich sein.

2 JUnit - Kurzeinblick

Grundsätzlich existiert eine Software erst, wenn sie systematisch getestet wurde. Diese Tests werden typischerweise programmiert und sind so jederzeit automatisch ausführbar. Tests folgen immer dem Aufbau

> der Vorbereitung, das System wird in den zum Testen benötigten Zustand gebracht,
>
> der Durchführung, die zu testenden Aktionen werden mit dem Testobjekt ausgeführt,
>
> der Analyse, es wird geprüft ob das Testobjekt die gewünschten Eigenschaften hat und die gewünschten Ergebnisse liefert, weiterhin ob das genutzte System sich in einem sinnvollen Zustand befindet.

Die Schritte werden auch AAA für Arrange, Act und Assert genannt.

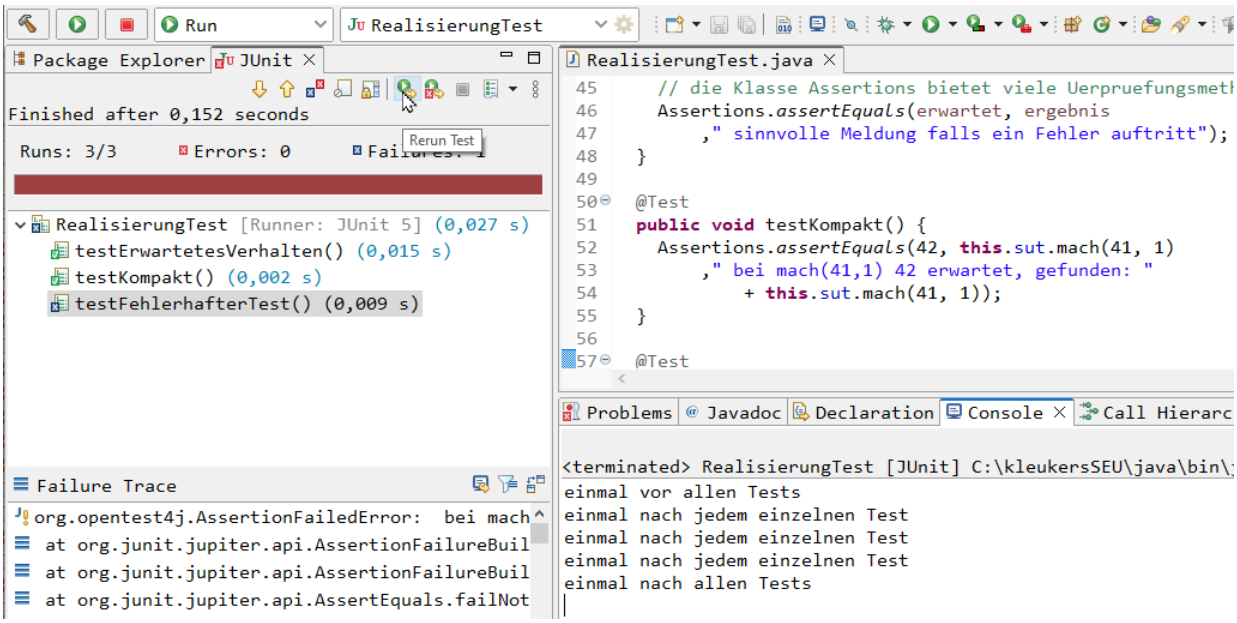
Für Java wird meist JUnit für Tests genutzt. JUnit, wie alle anderen Testframeworks unterstützen den AAA-Ansatz. So gibt es neben den eigentlichen Testmethoden, in Java @Test annotiert, die Möglichkeit eine Ausgangslage für Tests mit einer @BeforeEach annotierten Methode herzustellen, die vor jedem Test ausgeführt wird. Müssen bestimmte Eigenschaften, wie das Setzen von Klassenvariablen nur einmal vor allen Tests ausgeführt werden, gibt es eine @BeforeAll annotierte Methode. Falls es notwendig ist nach jedem Test etwas aufzuräumen ist eine mit @AfterEach annotierte Methode zu nutzen. Um nach der Ausführung aller Tests aufzuräumen, steht eine mit @AfterAll annotierte Klassenmethode zur Verfügung. Die Nutzung wird im folgenden Beispielcode gezeigt, dabei sind Ausgaben in Tests sonst unüblich. Es wird die Klasse Realisierung aus dem vorherigen Kapitel getestet.

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class RealisierungTest {

  private Beispielinterface sut;  // system under test

  @BeforeAll
  public  static void setUpBeforeClass() throws Exception {
    System.out.println("einmal vor allen Tests");
  }

  @AfterAll
  public static void tearDownAfterClass() throws Exception {
    System.out.println("einmal nach allen Tests");
  }

  @BeforeEach
  public void setUp() throws Exception {
    this.sut = new Realisierung();
  }
```

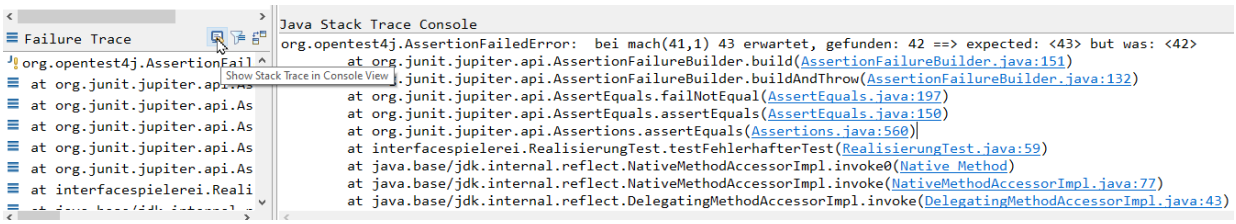
```
  @AfterEach
  public void tearDown() throws Exception {
    System.out.println("einmal nach jedem einzelnen Test");
  }

  @Test
  public void testErwartetesVerhalten() {
    // arrange (dazu gehoert auch setUp())
    int parameter1 = 41;
    int parameter2 = 1;
    int erwartet = 42;
    // act
    int ergebnis = this.sut.mach(parameter1, parameter2);
    // assert
    // die Klasse Assertions bietet viele Ueberpruefungsmethoden
    Assertions.assertEquals(erwartet, ergebnis
        ," sinnvolle Meldung falls ein Fehler auftritt");
  }

  @Test
  public void testKompakt() {
    Assertions.assertEquals(42, this.sut.mach(41, 1)
        ," bei mach(41,1) 42 erwartet, gefunden: "
            + this.sut.mach(41, 1));
  }

  @Test
  public void testFehlerhafterTest() {
    Assertions.assertEquals(43, this.sut.mach(41, 1)
        ," bei mach(41,1) 43 erwartet, gefunden: "
            + this.sut.mach(41, 1));
  }
}
```

Die Tests werden in JUnit z. B. mit einem Rechtsklick auf der Testklasse wie folgt gestartet:

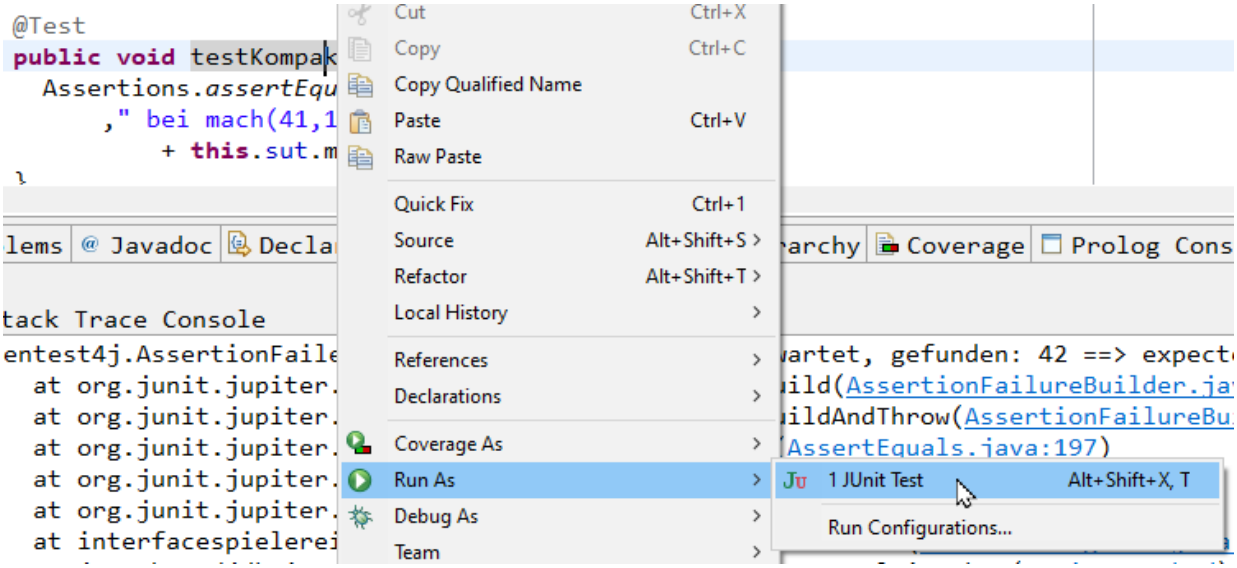
Das Ergebnis in JUnit-sieht wie folgt aus, die resultierende Ausgabe steht rechts-unten.



Um genauere Informationen zum gescheiterten Test zu erhalten, wird auf das dritte Feld von rechts („Show Stack Trace in Console) geklickt. In dem angegebenen Fehlerstack kann dann nach der auslösenden Zeile gesucht werden.

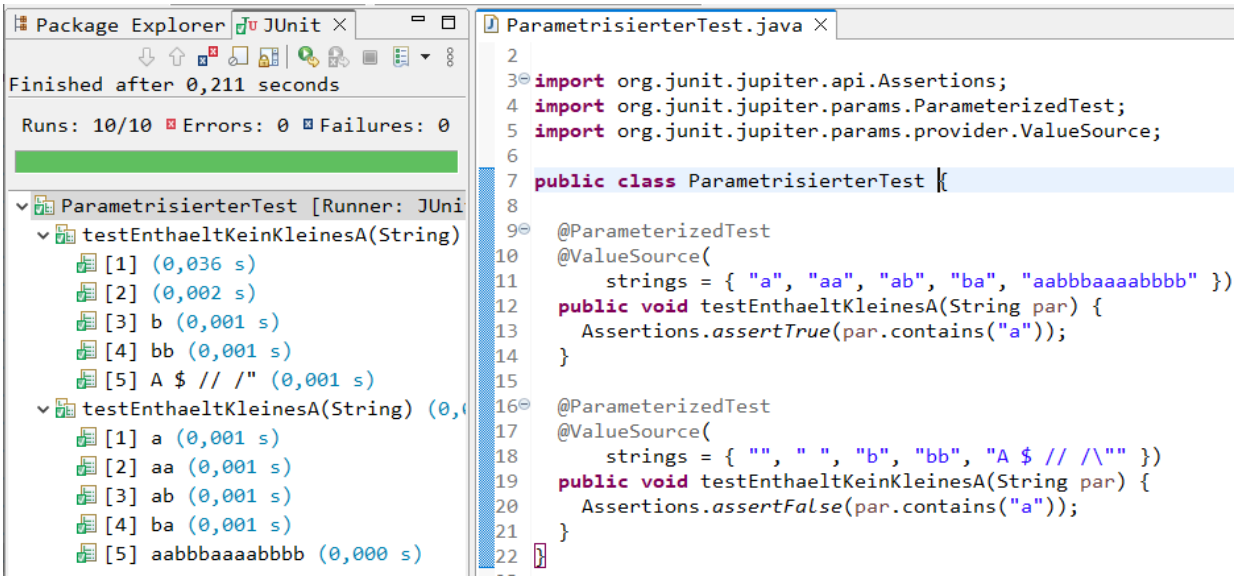


Eclipse bietet einige Möglichkeiten zur Ausführung einzelner Tests, eine zeigt die nachfolgende Abbildung mit einem Rechtsklick auf dem Testmethodennamen.



Generell ist die Auswahl der Tests sehr wichtig für die Qualität des Testens. Es werden typische und Extremwerte berücksichtigt. Dies führt oft dazu, dass vom Ablauf her identische Tests mit verschiedenen

Daten ausgeführt werden müssen. Um die Tests nicht zu kopieren bestehen verschiedene Möglichkeiten sie zu parametrisieren. Eine der Möglichkeiten wird im folgenden Beispiel mit dem Ergebnis auf der linken Seite gezeigt.




```
package interfacespielerei;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

public class ParametrisierterTest {

  @ParameterizedTest
  @ValueSource(
      strings = { "a", "aa", "ab", "ba", "aabbbaaaabbbb" })
  public void testEnthaeltKleinesA(String par) {
    Assertions.assertTrue(par.contains("a"));
  }

  @ParameterizedTest
  @ValueSource(
      strings = { "", " ", "b", "bb", "A $ // /\"" })
  public void testEnthaeltKeinKleinesA(String par) {
    Assertions.assertFalse(par.contains("a"));
  }
}
```

3 Turing-Maschinen

Dieses Kapitel zeigt Möglichkeiten Turing-Maschinen in einfachen Text-Dateien zu beschreiben und diese dann interaktiv oder automatisch zu nutzen. Weiterhin wird gezeigt, wie andere im Internetz verfügbare Werkzeuge eingesetzt werden können.

3.1 Eingeben und Simulieren

Mit der theoriesammlung können Turing-Maschinen simuliert und so auch berechnete Ergebnisse überprüft werden. Zur Beschreibung von Turing-Maschinen wird folgendes Format genutzt, hier erklärt an einer Maschine, die a in b und b in a verwandelt.

```
% Zustaende
Z: Start wandel S
% Alphabet, Leerzeichen # automatisch dabei
A: a b
% Start
S: Start
% Ueberfuehrungsfunktion
% alt  lesen neu    schreiben Richtung
 Start   #   wandel      #        L
 wandel  a   wandel      b        L
 wandel  b   wandel      a        L
 wandel  #   S           #        S // Zustand S auch Endzustand
```

Generell müssen Zustände, das verwendete Alphabet, der Startzustand und die Überführungsfunktion angegeben werden. Beginnt eine Zeile mit „%“ handelt es sich um einen Kommentar, diese Zeilen könnten also weggelassen werden. Beginnt eine Zeile mit „Z:“ folgen dahinter durch Leerzeichen getrennt die Namen der Zustände, die so selbst keine Leerzeichen enthalten dürfen. Beginnt eine Zeile mit „A:“ folgen danach wieder durch Leerzeichen getrennt die Zeichen des verwendeten Alphabets. Diese Zeichen dürfen durchaus selbst aus mehreren Zeichen bestehen, eine Randbedingung ist dabei, dass ein Zeichen nicht der Präfix eines anderen Zeichen sein darf, so darf es nur eines der Zeichen „O“ und „OK“ geben. Das in der Veranstaltung benutzte Leerzeichen (Blank) # ist nicht beim Alphabet anzugeben und wird automatisch intern hinzugefügt. Beginnt eine Zeile mit einem „S:“ folgt danach genau der eine Name des Startzustands, der vorher in einer mit „Z:“ markierten Zeile definiert werden musste. Die Überführungsfunktion wird als 5-Tupel getrennt mit Leerzeichen zeilenweise angegeben. Die generelle Form ist: aktueller Zustand, gerade eingelesenes Zeichen, neuer Zustand, zu schreibendes Zeichen, Bewegung des Schreib-Lese-Kopfs. Die Elemente müssen mit mindestens einem Leerzeichen getrennt sein, es sind auch mehrere Leerzeichen möglich, so dass es einfache Formatierungsmöglichkeiten gibt. Die Richtungen links, rechts und stopp können ausgeschrieben oder mit ihrem ersten Buchstaben dargestellt werden. Am Ende von Zeilen mit einem Schritt der Überführungsfunktion kann hinter „//“ ein Kommentar folgen.

Ein Turing-Maschine kann entweder aus einer Datei oder als String gelesen werden.

```
TuringMaschine tm = new TuringMaschine();
tm.dateiEinlesen("Erste.tm");
```

oder

```
TuringMaschine tm = new TuringMaschine();
tm.stringAlsTuringMaschine("""
```

```
        % Zustaende
        Z: Start wandel S
        % Alphabet, Leerzeichen # automatisch dabei
        A: a b
        % Start
        S: Start
        % Ueberfuehrungsfunktion
        % alt  lesen neu    schreiben Richtung
        Start   #   wandel      #        L
        wandel  a   wandel      b        L
        wandel  b   wandel      a        L
        wandel  #   S           #        S // Ende
        """);
```

Die Klasse TuringMaschine stellt einige meist dokumentierte Methoden zur Verfügung. Die beiden wichtigsten sind:

```
Wort ergebnis = tm.berechnen(eingabe) // eingabe ist Wort oder String

boolean erg = tm. akzeptieren(eingabe); // eingabe ist Wort oder String
```

Soll eine Turing-Maschine schrittweise ausgeführt werden, ist das z. B. wie folgt möglich.

```
  public static void main(String...strings) {
    TuringMaschine tm = new TuringMaschine();
    tm.stringAlsTuringMaschine("""
        % Zustaende
        Z: Start wandel S
        % Alphabet, Leerzeichen # automatisch dabei
        A: a b
        % Start
        S: Start
        % Ueberfuehrungsfunktion
        % alt      lesen  neu    schreiben Richtung
         Start       #   wandel      #        L
         wandel      a   wandel      b        L
         wandel      b   wandel      a        L
         wandel      #   S           #        S
        """);
    tm.bearbeite("abba");
    tm.ausfuehren(true);
  }
```

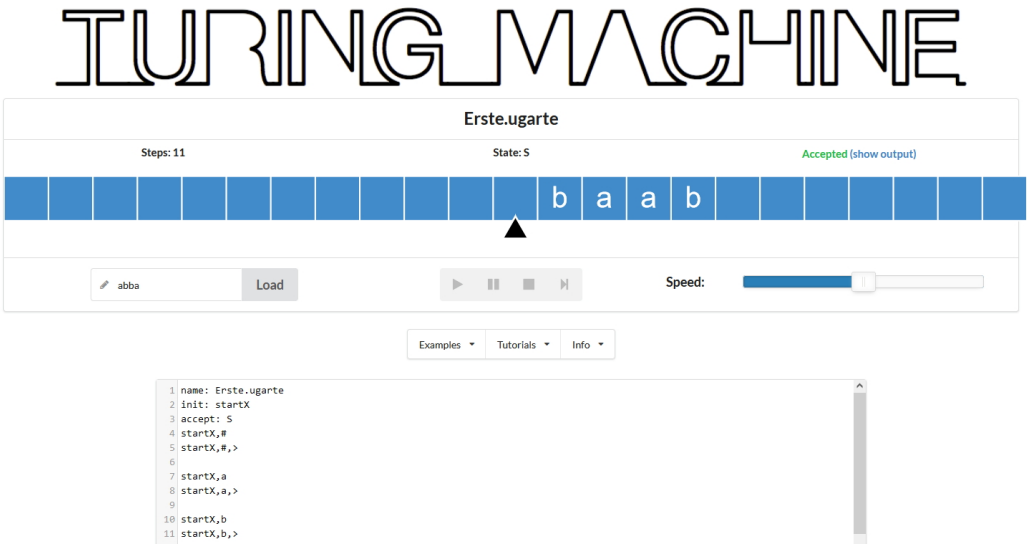
Es erfolgt eine schrittweise Abarbeitung mit Anzeige der aktuellen Konfiguration,

```
Start: (Start, #abba#)
(wandel, #abba#)
Weiter mit <Return>
(wandel, #abbb#)
Weiter mit <Return>
(wandel, #abab#)
Weiter mit <Return>
(wandel, #aaab#)
```

```
Weiter mit <Return>
(wandel, #baab#)
Weiter mit <Return>
(S, #baab#)
Weiter mit <Return>
Ende: (S, #baab#)
```

3.2 Nutzung anderer Simulatoren

Es gibt sehr viele Umsetzungen von Turing-Maschinen mit Simulationsprogrammen, da dies ein zentrales Informatik-Grundlagenthema ist. Generell sind viele Varianten leicht zu ergoogeln, wobei es keine einheitliche Form der syntaktischen Darstellung gibt. Allgemein sind aber praktisch alle Modelle logisch äquivalent, so dass die Übersetzung einer Turing-Maschine leicht in eine andere erfolgen kann. Exemplarisch sollen hier zwei Varianten vorgestellt werden.



Eine graphisch gelungene Simulation mit einer guten Visualisierung des Ablaufs befindet sich auf der Seite https://turingmachinesimulator.com/. Bei einer typischen Nutzung wird zuerst die Turing-Maschine in einem Text-Editor entwickelt und dann in das Eingabefeld kopiert. Die Syntax-Prüfung erfolgt mit dem unteren Knopf „Compile“. Links vom Knopf „Load“ wird das Eingabewort geschrieben, mit dem Knopf „Load“ auf das Band im oberen Bild kopiert und die Ausführung über den „Run-Button >“ oder über den „Schritt-Button >|“ gestartet. Der Ablauf der Schritte kann oben auf dem Band verfolgt werden, etwas tiefer rechts ist die Geschwindigkeit spezifizierbar. Da die Turing-Maschine auf dem ersten (linken) Zeichen der Eingabe beginnt, müssen unsere Turing-Maschinen so ergänzt werden, dass sie am Anfang erst an das Ende des Wortes laufen und dann mit dem eigentlichen Startzustand beginnen. Die oberhalb des Programm-Fensters ausgegebenen Fehlermeldungen sind leider nicht sehr detailliert, Details zur Darstellung können den auf der Web-Seite erhältlichen Beispielen und dem Tutorial entnommen werden. Es ist zu beachten, dass die Aufgabenstellungen in den Beispielen teilweise unpräzise formuliert sind.

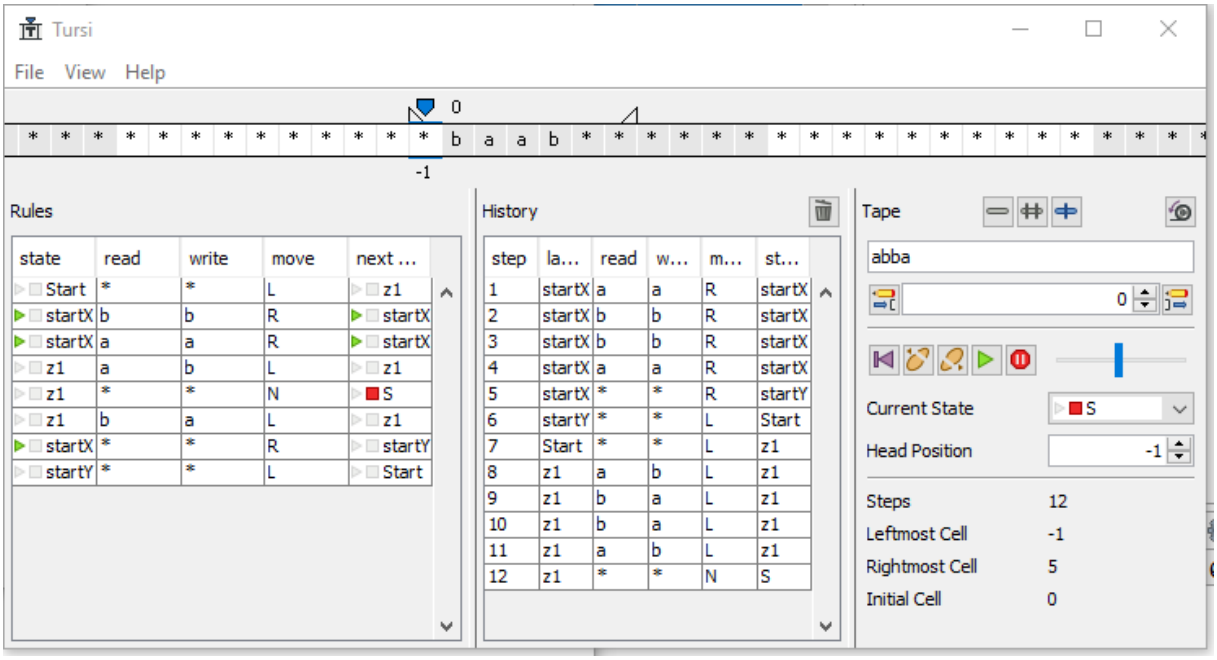
Die gegebene Bibliothek unterstützt die Umwandlung in das hier genutzte Modell, dazu stehen zwei Methoden zur Verfügung. Mit der folgenden Methode werden automatisch Schritte ergänzt, so dass am Anfang hinter die Eingabe gelaufen wird und so die Ausgangssituation der Vorlesung entsteht, dazu wird ein spezieller Zustand „startX“ genutzt.

Weiterhin muss es einen Endzustand mit Namen S geben.

```
    tm.dateiSpeichernUgarte("Erste.ugarte");
```

Soll nicht an das Ende der Eingabe gelaufen werden, ist als zweiter Parameter „false zu übergeben.

```
    tm.dateiSpeichernUgarte("Erste.ugarte", false);
```



Ein zweiter interessanter Turing-Maschinen-Simulator ist Tursi, das als Standalone-Programm von der Webseite https://schaetzc.github.io/tursi/ , genauer https://github.com/schaetzc/tursi/releases geladen werden kann. Die jar-Datei ist z.B. über die Kommandozeile startbar, die z. B. über StartKonsole.bat aus der kleukerSEU ausgeführt werden kann.

```
F:\Lehre\Material_Theorie\TMSimulatoren> java -jar Tursi.jar
```

Turing-Maschinen werden über „File > Open…“ geladen und dann auf der linken Seite angezeigt. Rechts unter „Tape“ wird die Eingabe für das Band eingegeben und durch Drücken der Return-Taste übergeben. Darunter befinden sich dann die Steuerungsmöglichkeiten, u. a. auch um einen Schritt vorwärts und rückwärts zu machen. Interessant ist u. a. der Mittelteil, der die durchgeführten Schritte protokolliert. Der Schreib-Lese-Kopf kann in der oberen graphischen Darstellung auch genutzt werden, um ihn mit gedrückter linker Maustaste auf dem Bildschirm neu zu platzieren, falls interessante Bandinhalte nicht sichtbar sind.

Die gegebene Bibliothek der Veranstaltung unterstützt die Umwandlung in das hier genutzte Modell, dazu stehen zwei Methoden zur Verfügung. Mit der folgenden Methode werden automatisch Schritte ergänzt, so dass am Anfang hinter die Eingabe gelaufen wird und so die Ausgangssituation der Vorlesung entsteht, dazu werden zwei spezielle Zustände „startX“ und „startY“ genutzt.

```
    tm.dateiSpeichernTursi("Erste.tursi");
```

Soll nicht an das Ende der Eingabe gelaufen werden, ist als zweiter Parameter „false“ zu übergeben.

```
    tm.dateiSpeichernTursi("Erste.tursi", false);
```

4 Kontextfreie Grammatiken

Dieses Kapitel zeigt Möglichkeiten kontextfreie Grammatiken mit einfachen Text-Dateien zu beschreiben und für diese dann interaktiv oder automatisch Ableitungen zu erstellen. Weiterhin wird gezeigt, wie kontextfreie Grammatiken als Objekte erzeugt werden können, welche Möglichkeiten zur Lösung des Wortproblems mit zugehörigen Transformationen es gibt und wie zu einer einfachen Programmiersprache eine zugehörige Semantik definiert werden kann.

4.1 Eingeben, Ausgeben und Ableiten

Die theoriesammlung bietet eine schrittweise Ableitung mit einer kontextfreien Grammatik und die Möglichkeit zu überprüfen, ob ein Wort abgeleitet werden kann oder nicht. Zur Beschreibung von kontextfreien Grammatiken wird folgendes Format genutzt, hier erklärt an der Grammatik, die

$\{a^m b^n \mid m \geq n\}$ erzeugt.

```
% Prozentzeichen am Anfang fuer Kommentar
% Nichtterminalzeichen
N: Start A B
% Terminalzeichen
T: a b
% Startsymbol
S: Start
% Regeln
% mehrere in einer Zeile moeglich, muss aber nicht
Start -> AStartB | /eps
A -> Aa | a
B -> b | /eps
```

Generell müssen Nichtterminale, Terminale, das Start-Symbol und die Regeln angegeben werden. Beginnt eine Zeile mit „%“ handelt es sich um einen Kommentar, diese Zeilen könnten also weggelassen werden. Beginnt eine Zeile mit „N:“ folgen dahinter durch Leerzeichen getrennt die Namen der Nichtterminale, die selbst keine Leerzeichen enthalten dürfen. Beginnt eine Zeile mit „T:“ folgen danach wieder durch Leerzeichen getrennt die Terminale. Eine Randbedingung ist dabei, dass Terminale und Nichtterminale nicht der Präfix eines anderen Elements sein dürfen, so darf es nur eines der Terminal- oder Nichtterminal-Zeichen „O“ und „OK“ geben. Beginnt eine Zeile mit einem „S:“ folgt danach genau der eine Name des Startsymbols, der vorher in einer mit „N:“ markierten Zeile definiert werden musste. Die Regeln werden zeilenweise angegeben. Die generelle Form ist: Nichtterminalzeichen, dann ein Pfeil und dann das abgeleitete Wort aus Nichtterminalen und Terminalen. Mehrere Varianten können in einer Zeile stehen und sind mit einem senkrechten Strich getrennt. Das leere Wort wird mit /eps angegeben.

Ein Turing-Maschine kann entweder aus einer Datei oder als String gelesen werden.

```
KontextfreieGrammatik kfg = new KontextfreieGrammatik();
kfg.dateiEinlesen("kontextfreiegrammatiken\\Spachea_nb_m.kfg");
```

oder

```
String grammatik =
    """
    N: AX BX
    T: a b
    S: AX
    r1:: AX -> BX | a
```

```
        r2:: BX -> b
        r3:: BX -> /eps
        """;
    KontextfreieGrammatik g = new KontextfreieGrammatik();
    g.stringAlsGrammatik(grammatik);
```

Die Grammatik im String zeigt eine weitere optionale Möglichkeit, jeder Zeile mit Regeln einen eindeutigen Namen zuzuordnen. Der Name wird mit einem doppelten Doppelpunkt abgeschlossen.

Die Klasse KontextfreieGrammatik stellt einige meist dokumentierte Methoden zur Verfügung. Die am Anfang wichtigste ist:

```
boolean erg = g.ableitbar(eingabe); // eingabe ist Wort oder String
```

Mit der folgenden Methode kann aus einem String ein Wort-Objekt erzeugt werden.

```
Wort erg = g.stringAlsWort(eingabe); // eingabe ist String
```

Soll eine Ableitung schrittweise ausgeführt werden, ist das z. B. wie folgt möglich.

```
  public static void main(String[] args) {
    String grammatik =
        """
        N: S B
        T: a b
        S: S
        r1:: S -> aSbaSb | ab
        """;
    KontextfreieGrammatik g = new KontextfreieGrammatik();
    g.stringAlsGrammatik(grammatik);
    g.ausfuehren();
  }
```

Im Dialog werden dann alle Regeln angeboten. Falls eine Regel an mehreren Stellen anwendbar ist, kann dies im Folgeschritt geklärt werden. Ein Beispieldialog sieht wie folgt aus. Es ist auch erkennbar, dass die Regeln individuelle Namen erhalten haben. In Klammern steht die aktuelle Länge des abgeleiteten Wortes. Eingaben sind umrandet.

```
Terminale =[a, b]
Nichtterminale = [B, S]
Start = S
 (1) r1_0 :: S -> aSbaSb
 (2) r1_1 :: S -> ab
 (0) abbrechen
aktuelles Wort: S (1)

Welche Regel wollen Sie anwenden: 1
Terminale =[a, b]
Nichtterminale = [B, S]
Start = S
 (1) r1_0 :: S -> aSbaSb
 (2) r1_1 :: S -> ab
 (0) abbrechen
```

```
aktuelles Wort: aSbaSb (6)

Welche Regel wollen Sie anwenden: 1
Weches Vorkommen (erstes ist 1): 2
Terminale =[a, b]
Nichtterminale = [B, S]
Start = S
 (1) r1_0 :: S -> aSbaSb
 (2) r1_1 :: S -> ab
 (0) abbrechen
aktuelles Wort: aSbaaSbaSbb (11)

Welche Regel wollen Sie anwenden: 2
Weches Vorkommen (erstes ist 1): 2
Terminale =[a, b]
Nichtterminale = [B, S]
Start = S
 (1) r1_0 :: S -> aSbaSb
 (2) r1_1 :: S -> ab
 (0) abbrechen
aktuelles Wort: aSbaaabbaSbb (12)

Welche Regel wollen Sie anwenden: 2
Weches Vorkommen (erstes ist 1): 1
Terminale =[a, b]
Nichtterminale = [B, S]
Start = S
 (1) r1_0 :: S -> aSbaSb
 (2) r1_1 :: S -> ab
 (0) abbrechen
aktuelles Wort: aabbaaabbaSbb (13)

Welche Regel wollen Sie anwenden: 2
Ergebnis: aabbaaabbaabbb
```

Der spezielle Kommentar %ignorespace führt dazu, dass bei der Angabe der Regeln Leerzeichen zwischen den Zeichen stehen dürfen, was die Lesbarkeit erhöht. Weiterhin werden so Leerzeichen aus Eingabewörtern automatisch entfernt. Sollen trotzdem Leerzeichen genutzt werden, sind das Terminalzeichen /space zu ergänzen und passende Regeln zu definieren. Dies zeigt das folgende Beispiel mit den Nichtterminalzeichen $Optional und $Spaces die dafür sorgen dass beliebig viele bzw. mindestens ein Leerzeichen zu nutzen sind. Das $-Zeichen ist nur ein kleiner Trick auch Großbuchstaben als Terminalzeichen zu ermöglichen, da es so keine Präfix-Problematik gibt. Diese Möglichkeiten werden in der folgenden Grammatik zusammengefasst. Die statisch-polymorphe Methode ableitbar liefert bei Worten automatisch ein Objekt vom Typ Ableitung, das mit isErfolgreich() darauf geprüft werden kann, ob das gesamte Wort abgeleitet werden kann.

```
public static void main(String[] args) {
      String grammatik =
            """
            %ignorespaces
```

```
                N: Start $Optional $Spaces
                T: a b /space
                S: Start
                Start -> a Start b | $Spaces
                $Optional -> /eps | $Spaces
                $Spaces -> /space | /space $Spaces
                """;
        KontextfreieGrammatik g = new KontextfreieGrammatik();
        g.stringAlsGrammatik(grammatik);
        Wort w = g.stringAlsWort("aa bb");
        System.out.println("aa bb: " + g.ableitbar(w).isErfolgreich());
        Wort w2 = new Wort("aa bb");
        System.out.println("aa bb: " + g.ableitbar(w2).isErfolgreich());
    }
```

Die Ausgabe lautet:

```
aa bb: false
aa bb: true
```

Die erste Ausgabe „false“ beruht darauf, dass bei der Umwandlung des Strings „aa bb“ automatisch die Leerzeichen entfernt wurden, was beim Wort-Konstruktor nicht der Fall ist. Kritisch sollte im Beispiel auffallen, dass das Nichtterminal $Optional in diesem Fall überflüssig ist. Weiterhin wird der Kommentar %ignorespaces statt %ignorespace genutzt, was egal ist, da nur der Anfang übereinstimmen muss.

4.2 Kontextfreie Grammatiken als programmierte Objekte

Das nachfolgende Programm zeigt Möglichkeiten Grammatiken aus Java-Objekten verschiedener Domänen-Klassen zusammenzusetzen. Der Ansatz ist besonders dann interessant, falls die Grammatik weiterverarbeitet werden soll, wenn ihr z.B. eine Semantik zugordnet wird. Bei der Methode ignoriereLeerzeichen(true) werden Leerzeichen aus Worten entfernt, deren Syntax überprüft werden soll. Alternativ kann new Wort(String) genutzt werden, wodurch Leerzeichen erhalten bleiben und dann natürlich auch als Terminalzeichen /space existieren sollten. Die umgesetzte Grammatik ist: Nichtterminale = {X, A, B}, Terminale = {a,b,c}, Startsymbol ist X und die Regeln sind:

X -> aaX | A
A -> bbA | B
B -> ccX | ε

```
package main;

import alphabet.Nichtterminal;
import alphabet.Terminal;
import ausfuehrung.Ableitung;
import grammatik.KontextfreieGrammatik;
import grammatik.KontextfreieRegel;
import semantik.Ableitungsbaum;

public class KontextfreieGrammatikProgrammiert {

  public static void main(String[] args) {
```

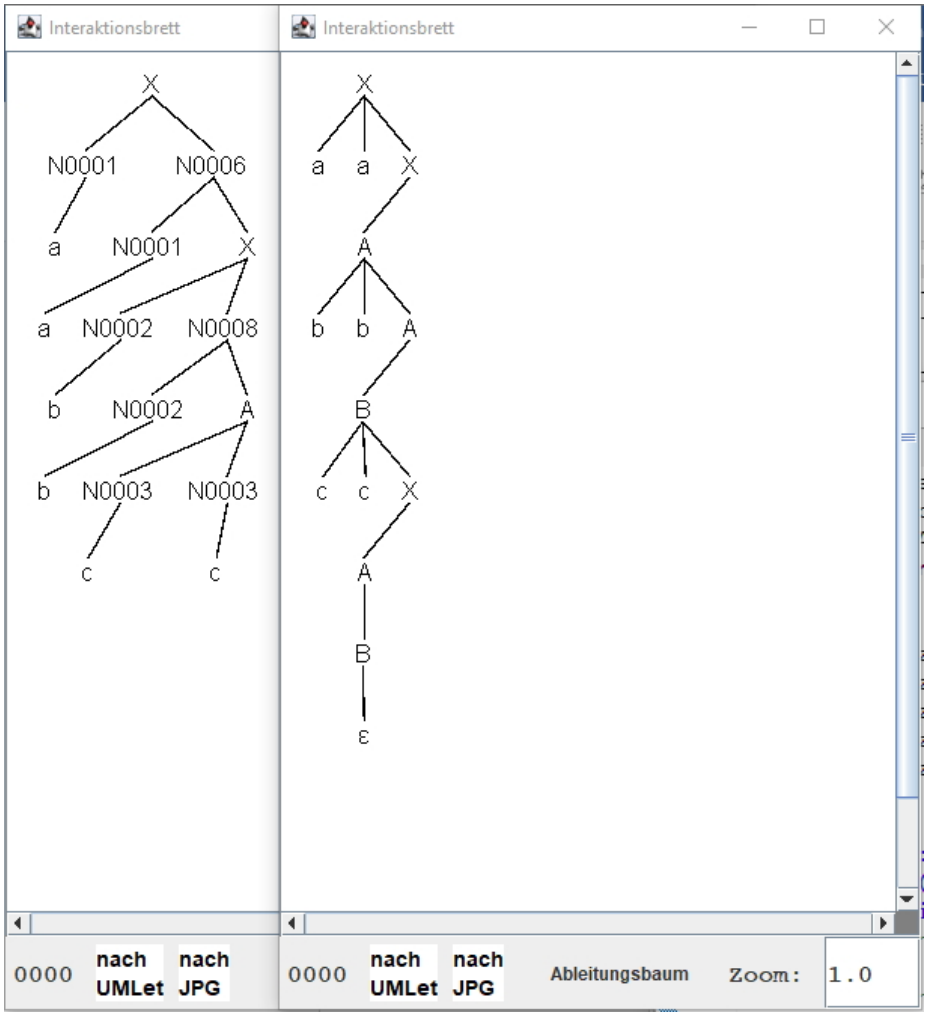
```
    KontextfreieGrammatik g = new KontextfreieGrammatik();
    g.setIgnoriereLeerzeichen(true);
    Nichtterminal nX = Nichtterminal.nichtterminal("X");
    Nichtterminal nA = Nichtterminal.nichtterminal("A");
    Nichtterminal nB = Nichtterminal.nichtterminal("B");
    g.addNichtterminale(nX, nA, nB);
    Terminal ta = Terminal.terminal("a");
    Terminal tb = Terminal.terminal("b");
    Terminal tc = Terminal.terminal("c");
    g.addTerminale(ta, tb, tc);
    g.setStart(nX);
    KontextfreieRegel r1 = new KontextfreieRegel(nX, g.stringAlsWort("aaX"));
    KontextfreieRegel r2 = new KontextfreieRegel(nX, g.stringAlsWort("A"));
    KontextfreieRegel r3 = new KontextfreieRegel(nA, g.stringAlsWort("bbA"));
    KontextfreieRegel r4 = new KontextfreieRegel(nA, g.stringAlsWort("B"));
    KontextfreieRegel r5 = new KontextfreieRegel(nB, g.stringAlsWort("ccX"));
    KontextfreieRegel r6 = new KontextfreieRegel(nB, g.stringAlsWort("/eps"));
    g.addRegeln(r1, r2, r3, r4, r5, r6);
    System.out.println(g);

    Ableitung abl = g.ableitbar(g.stringAlsWort("aabbcc"));
    System.out.println("Ist aabbcc ableitbar: " + abl.isErfolgreich());
    Ableitungsbaum ablb = Ableitungsbaum.ableitungAlsBaum(abl);
    ablb.linksdurchlauf();
    ablb.visualisieren();

    Ableitung abl2 = g.berechneAbleitung(g.stringAlsWort("aabbcc"));
    System.out.println("\nIst aabbcc ableitbar: " + abl2.isErfolgreich());
    Ableitungsbaum ablb2 = Ableitungsbaum.ableitungAlsBaum(abl2);
    ablb2.linksdurchlauf();
    ablb2.visualisieren();
  }
}
```

Die zugehörige Ausgabe sieht wie folgt aus.

```
Terminale =[a, b, c]
Nichtterminale = [A, B, X]
Start = X
 (1) X -> aaX
 (2) X -> A
 (3) A -> bbA
 (4) A -> B
 (5) B -> ccX
 (6) B -> ε

Ist aabbcc ableitbar: true
0: X -> N0001N0006
1: N0001 -> a
2: N0006 -> N0001X
3: N0001 -> a
4: X -> N0002N0008
5: N0002 -> b
6: N0008 -> N0002A
7: N0002 -> b
8: A -> N0003N0003
```

```
9: N0003 -> c
10: N0003 -> c

Ist aabbcc ableitbar: true
0: X -> aaX
1: X -> A
2: A -> bbA
3: A -> B
4: B -> ccX
5: X -> A
6: A -> B
7: B -> ε
```

Die nachfolgende Abbildung zeigt die generierten graphischen Ausgaben. Es wird deutlich, dass bei der Nutzung von ableitbar(.) ein Ableitungsobjekt erzeugt wird, das zur Grammatik in Chomsky-Normalform passt. Wird die aufwändigere Methode `berechneAbleitung()` genutzt, bezieht sich die berechnete Ableitung auf die Ursprungsgrammatik. Das Ergebnis ist auf der rechten Seite dargestellt.



Unter den Ausgaben befinden sich zwei Knöpfe, mit denen das Ergebnis in eine JPG-Datei und in eine Datei des graphischen Werkzeugs UMLet verwandelt werden kann. Letzte ist zwar bearbeitbar, da aber keine pixelgenaue Platzierung unterstützt wird, wohl von eher geringem Nutzen.

Besteht ein reines Interesse an einer nicht-graphischen Ableitung ist diese mit der Methode `zeigeLinksableitung()` berechenbar, wie folgendes Beispiel zeigt.

```
public class KFGAbleitungAngebenLassen {

  public static void main(String[] args) {
    String grammatik =
        """
        N: S T
        T: a b
        S: S
        r0:: S -> TT
        r1:: T -> aTb | ab
        """;
    KontextfreieGrammatik g = new KontextfreieGrammatik();
    g.stringAlsGrammatik(grammatik);
    g.zeigeLinksableitung("aaabbbaabb");
    g.zeigeLinksableitung("abaab");
  }
}
```

Die Ausgabe sieht wie folgt aus. Ist das Wort nicht ableitbar, erfolgt ein Hinweis, gegebenenfalls mit einem ableitbaren Teilwort (hier nicht).

```
S -> TT -> aTbT -> aaTbbT -> aaabbbT -> aaabbbaTb -> aaabbbaabb
abaab ist nicht ableitbar, nur:
```

4.3 Lösung des Wortproblems für kontextfreier Grammatiken

Für Grammatiken in Chomsky-Normalform kann, wie für andere Grammatiken auch, mit der Methode `ableitbar()` geprüft werden, ob ein Wort abgeleitet werden kann. Nach der Nutzung dieser Methode kann mit der Methode `cykTabelleAusgeben()` die verwendete Grammatik ausgegeben werden, wie folgendes Beispiel für den positiven und den negativen Fall zeigt.

```
public static void main(String[] args) {
  String grammatik =
      """
      %ignorespaces
      N: X Y A B
      T: a b
      S: X
      X -> YA | YB
      Y -> XA | XB | AX | BX
      A -> a
      B -> b
      X -> a
      """;
  KontextfreieGrammatik g = new KontextfreieGrammatik();
  g.stringAlsGrammatik(grammatik);
  System.out.println("abbab: " + g.ableitbar("abbab"));
  g.cykTabelleAusgeben();
  g.zeigeLinksableitung("abbab");
```

```
    System.out.println("\nbbbaa: " + g.ableitbar("bbbaa"));
    g.cykTabelleAusgeben();
  }
```

Die zugehörige Ausgabe lautet:

```
abbab: true
[A, X] [B]    [B]    [A, X] [B]
[Y]    []     [Y]    [Y]    []
[X]    []     [X]    []     []
[Y]    [Y]    []     []     []
[X]    []     []     []     []
X -> YB -> XAB -> YBAB -> XBBAB -> aBBAB -> abBAB -> abbAB -> abbaB -> abbab

bbbaa: false
[B]    [B]    [B]    [A, X] [A, X]
[]     []     [Y]    [Y]    []
[]     []     [X]    []     []
[]     [Y]    []     []     []
[]     []     []     []     []
```

Die Ausgabe bezieht sich dabei immer auf die letzte erfolgreich berechnete Ableitung.

4.4 Ausführung der Basisprogrammiersprache

Programme der nachfolgenden Basisgrammatik können mit der Bibliothek ausgeführt werden. Es ist zu beachten, dass in den Regeln hier Zeilenumbrüche stehen, die es in der Originalgrammatik nicht gibt.

```
%ignorespace
N: Zahl Variable BTerm Ausdruck Sequenz ITerm Bedingung Befehl
T: or ( ) false + - 0 1 2 3 4 5 u 6 v 7 8 x 9 y z ; { true < } > :=
T: while not and else if _ ==
S: Sequenz
Zahl -> Zahl 0 | Zahl 1 | Zahl 2 | Zahl 3 | Zahl 4 | Zahl 5 | Zahl 6 |  Zahl
7 | Zahl 8 | Zahl 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Variable -> u Zahl | v Zahl | _ Zahl | x Zahl | y Zahl | z Zahl |u | v | _ |
x | y | z
Ausdruck -> ITerm+Ausdruck | ITerm-Ausdruck | (Ausdruck) |  ITerm
ITerm -> Zahl
ITerm -> Variable
Bedingung -> BTerm and Bedingung
Bedingung -> BTerm or Bedingung
Bedingung -> (Bedingung)
Bedingung -> BTerm
BTerm -> Ausdruck < Ausdruck
BTerm -> Ausdruck > Ausdruck
BTerm -> Ausdruck == Ausdruck
BTerm -> not(Bedingung)
BTerm -> true
BTerm -> false
Sequenz -> Befehl Sequenz
Sequenz -> Befehl
Befehl -> Variable := Ausdruck;
Befehl -> if (Bedingung) {Sequenz} else {Sequenz}
```

```
Befehl -> while (Bedingung) {Sequenz}
```

Zur Ausführung wird die Klasse SemantikBasissprache genutzt. Zustände können nicht explizit übergeben werden, sie sind von Hand als erste Anweisungen vor dem eigentlichen Programm zu ergänzen.

```
import semantik.SemantikBasissprache;

public class SemantikBeispiel {

  public static void main(String[] args) {
    SemantikBasissprache.ausfuehren("""
        x := 6;
        u := 0;
        y := 0;
        z := 0;
        while (not(z == x)) {
          if (u == 1) {
            y := y + z;
          } else {
            y := y;
          }
          u := 1 - u;
          z := z + 1;
        }
        """);
  }
}
```

Die Ausgabe sieht wie folgt aus, dabei gibt der Rand links an, ob es sich um eine Anweisung, ein nach true ausgewertetes if, ein nach false ausgewertetes if, ein nach true ausgewertetes while und ein nach false ausgewertetes while zusammen mit dem Zustand angegeben.

```
:= :     {x=6}
:= :     {u=0, x=6}
:= :     {u=0, x=6, y=0}
:= :     {u=0, x=6, y=0, z=0}
while t: {u=0, x=6, y=0, z=0}
if f:    {u=0, x=6, y=0, z=0}
:= :     {u=0, x=6, y=0, z=0}
:= :     {u=1, x=6, y=0, z=0}
:= :     {u=1, x=6, y=0, z=1}
while t: {u=1, x=6, y=0, z=1}
if t:    {u=1, x=6, y=0, z=1}
:= :     {u=1, x=6, y=1, z=1}
:= :     {u=0, x=6, y=1, z=1}
:= :     {u=0, x=6, y=1, z=2}
while t: {u=0, x=6, y=1, z=2}
if f:    {u=0, x=6, y=1, z=2}
:= :     {u=0, x=6, y=1, z=2}
:= :     {u=1, x=6, y=1, z=2}
:= :     {u=1, x=6, y=1, z=3}
while t: {u=1, x=6, y=1, z=3}
```

```
if t:    {u=1, x=6, y=1, z=3}
:= :     {u=1, x=6, y=4, z=3}
:= :     {u=0, x=6, y=4, z=3}
:= :     {u=0, x=6, y=4, z=4}
while t: {u=0, x=6, y=4, z=4}
if f:    {u=0, x=6, y=4, z=4}
:= :     {u=0, x=6, y=4, z=4}
:= :     {u=1, x=6, y=4, z=4}
:= :     {u=1, x=6, y=4, z=5}
while t: {u=1, x=6, y=4, z=5}
if t:    {u=1, x=6, y=4, z=5}
:= :     {u=1, x=6, y=9, z=5}
:= :     {u=0, x=6, y=9, z=5}
:= :     {u=0, x=6, y=9, z=6}
while f: {u=0, x=6, y=9, z=6}
```

4.5 Umsetzung der Semantik im Framework

Generell kann die Semantik der Programmiersprache erweitert bzw. für beliebige Grammatiken eine neue Semantik geschrieben werden. Da es allerdings keine festen Regeln zur Umsetzung geben kann, muss hier gegebenenfalls eine aufwändige Umprogrammierung erfolgen. Der durchaus erweiterbare Ansatz für die Basisprogrammiersprache wird hier vorgestellt.

Generell wird jeder Regel eine Berechnung zugeordnet. Dabei haben Berechnungen den folgenden vorgegebenen Aufbau.

```
@FunctionalInterface
public interface Berechnung {
  /** Die uebergebene Semantik wird auf den uebergebenen Ableitungsbaum
   * angewandt, wobei der aktuelle Zustand uebergeben wird; dieser
   * Zustand kann in der Methode veraendert werden, das Ergebnis ist der
   * im naechsten Schritt zu bearbeitende Ableitungsbaum
   * @param sem anzuwendende Semantik
   * @param baum Baum, auf den die Semantik angewandt werden soll
   * @param z aktueller Zustand, der veraendert werden kann
   * @return im naechsten Schritt zu bearbeitender Ableitungsbaum
   */
  public Ableitungsbaum berechnen(Semantik sem, Ableitungsbaum baum
        , Interpretation z);
}
```

Die Klasse Interpretation heißt dabei bewusst nicht Zustand, nicht nur, um Verwechslungen mit der bei Turing-Maschinen und endlichen Automaten verwendeten Klasse zu vermeiden. Eine Interpretation enthält eine Zuordnung der Variablen zu ihren Werten und kann noch Hilfsinformationen enthalten. Im konkreten Fall ist das ein sogenannter Akkumulator, der sich einen Wert merken kann. Dieser wird insbesondere bei der Auswertung von Ausdrücken benötigt, da diese den eigentlichen Zustand, die Werte der Variablen nicht verändern. Selbst bei einem einfachen Ausdruck wie 5 + 4, muss sich der Wert 5 zwischengemerkt werden, um nach dem Erkennen der 4 dann die Addition durchzuführen. Die Konkrete Umsetzung für die Regel `Ausdruck -> ITerm + Ausdruck` sieht deshalb wie folgt aus.

```
    Berechnung b5 = new Berechnung() {
```

```
    @Override
    public Ableitungsbaum berechnen(Semantik sem, Ableitungsbaum baum
              , Interpretation z) {
      sem.schritt(baum.get(0), z);
      int tmp = z.getWert();
      sem.schritt(baum.get(2), z);
      z.setAkku(new Wert(tmp + z.getWert()));
      return baum; // evtl auf return verzichten oder sogar int zurueck?
    }
  };
```

Dabei wertet `schritt()` den übergebenen Teilbaum aus und bearbeitet die Interpretation. Im konkreten Fall wird der Wert von ITerm in die Variable wert der Interpretation geschrieben und dann aus dieser ausgelesen. Dann wird der rechte zu Ausdruck gehörende Teilbaum ausgelesen und der dabei berechnete, mit z.getWert() erhaltene Wert zu tmp addiert. Die Klasse Wert dient dabei dazu, dass Werte verschiedener Typen in einem Objekt gehalten werden können. Im Detail fällt auf, dass für „Aufruf“ mit get(2) der dritte Teilbaum genutzt wird. Dies ist relativ logisch, da der zweite mit get(1) zum + gehört. Die rechte Seite der betrachteten Regel besteht insgesamt aus drei Zeichen, zwei Nichtterminalen und einem Terminalen. Der Kommentar beim return-Statement deutet an, dass die Rückgabe hier nicht benötigt wird, da hier der Baum einfach rekursiv von links nach rechts abgearbeitet wird.

Die Methode `schritt()` betrachtet nebenbei nur die für den Ableitungsbaum erste Regel, sucht die dazugehörige Berechnung und führt diese aus.

Bei der Zuweisung mit der Regel `Befehl -> Variable:=Ausdruck;` findet dann die einzige echte Zustandsveränderung als Veränderung des Interpretations-Objekts statt.

```
  Berechnung bbef1 = new Berechnung() {
    @Override
    public Ableitungsbaum berechnen(Semantik sem, Ableitungsbaum baum
             , Interpretation z) {
      sem.schritt(baum.get(0), z);
      String name = z.getAkku().getStringWert();
      sem.schritt(baum.get(2), z);
      z.set(name, z.getAkku().getIntegerWert());
      System.out.println(":= :     " + z.getBelegung());
      return baum;
    }
  };
```

Im ersten Schritt wird die Variable gefunden, dabei wird ausgenutzt, dass ein Wert-Objekt auch einen String annehmen kann. Hierzu wird das Wert-Objekt mit getAkku() aus der Interpretation gelesen und der passende Wert entnommen. Sollte der Typ nicht passen, wirft die Klasse Wert eine IllegalStateException. Der Rückgabewert wird hier ebenfalls nicht benötigt.

Bei einer while-Schleife `Befehl -> while(Bedingung){Sequenz}` wird nicht einfach nur ein Teilbaum betrachtet, sondern das danach auszuführende Programm kann sich so verändern, dass erst das Innere der Schleife und dann erneut die Schleife ausgeführt werden kann.

```
  Berechnung bbef3 = new Berechnung() {
    @Override
```

```
  public Ableitungsbaum berechnen(Semantik sem, Ableitungsbaum baum
            , Interpretation z) {
    sem.schritt(baum.get(2), z);
    boolean tmp = z.getAkku().getBooleanWert();
    while(tmp) {
      System.out.println("while t: " + z.getBelegung());
      sem.schritt(baum.get(5), z);
      sem.schritt(baum.get(2), z);
      tmp = z.getAkku().getBooleanWert();
    }
    System.out.println("while f: " + z.getBelegung());
    return baum;
  }
};
```

Wieder ist zu beachten, dass „Bedingung“ das dritte Zeichen ist, das erste ist „while“, das zweite „(“.

Die Umsetzung von if `Befehl -> if(Bedingung){Sequenz}else{Sequenz}` erfolgt in der Programmierung ebenfalls wieder als if. Die rechte Seite der Regel besteht insgesamt aus 11 Zeichen.

```
Berechnung bbef2 = new Berechnung() {
  @Override
  public Ableitungsbaum berechnen(Semantik sem, Ableitungsbaum baum
          , Interpretation z) {
    sem.schritt(baum.get(2), z);
    boolean tmp = z.getAkku().getBooleanWert();
    if(tmp) {
      System.out.println("if t:    " + z.getBelegung());
      sem.schritt(baum.get(5), z);
    } else {
      System.out.println("if f:    " + z.getBelegung());
      sem.schritt(baum.get(9), z);
    }
    return baum;
  }
};
```

Die sequentielle Berechnung `Sequenz -> Befehl Sequenz` ist relativ einfach. Es werden beide Teile einfach ausgewertet. Dabei ist zu beachten, dass die Interpretation bei jeder Berechnung verändert werden kann.

```
Berechnung bseq1 = new Berechnung() {
  @Override
  public Ableitungsbaum berechnen(Semantik sem, Ableitungsbaum baum
            , Interpretation z) {
    sem.schritt(baum.get(0), z);
    sem.schritt(baum.get(1), z);
    return baum;
  }
};
```

Generell fällt auf, dass der Rückgabewert von `berechnen()` keine Rolle spielt, er könnte auch einfach null sein. Da es für andere Arten von Sprachen sinnvoll sein kann einen noch zu verarbeitenden Baum zurückzugeben, wird dies durch das Interface unterstützt.

Wie bereits angedeutet dient die Klasse Interpretation dazu Informationen zwischen den einzelnen Schritten bei der Semantik-Bearbeitung auszutauschen, was im konkreten Fall die Belegungen der Variablen (Zustand) und eine Hilfsinformation ist. Bei anderen Semantiken werden meist deutlich mehr solcher Hilfsinformationen benötigt, so dass auch der Name SemantikKonfiguration als Klassenname sinnvoll ist. Soll z. B. der Aufruf einer n-stelligen Funktion berechnet werden, muss zuerst die Semantik eines jeden Teilterms berechnet werden, um dann die Semantik der Funktion darauf anwenden zu können. Die Semantiken der Teilterme sind dann in einer Liste zu sammeln.

Es gibt mit der Klasse SemantikBasissprache Datei eine leichte Implementierungsvariante. Das Programm nutzt statt einer programmierten Grammatik eine Grammatik, die aus einer Datei gelesen wird. Die zugehörige Grammatik steht in der Datei /theoriesammlung/beispiele/kontextfreiegrammatiken/ProgspracheBasis.kfg .

Würde die Grammatik Terminalzeichen aus einfachen Zeichen nutzen, kann für folgende Programmzeile ein Problem auftreten.

```
if((not(x42 == 6) or (true and 42 == 40 + 1))) {
```

```
java.lang.IllegalStateException: kein Wert fuer Variable trueand42 gefunden
```

Die Besonderheit ist, dass die Schlüsselworte keine Terminalzeichen sind. Das hat den Vorteil, dass beliebige Variablennamen zur Verfügung stehen. Der Nachteil ist, dass die Analyse der Programme deutlich länger dauert.

5 Endliche Automaten

Dieses Kapitel zeigt Möglichkeiten verschiedene Varianten von einfachen Automaten mit einfachen Text-Dateien zu beschreiben und für diese dann interaktiv oder automatisch Worte schrittweise abzuarbeiten. Weiterhin wird gezeigt, wie Epsilon-Übergänge entfernt werden können, aus nichtdeterministischen deterministische Automaten werden und wie eine Minimierung durchgeführt wird. Zusätzlichen sind Automaten in rechtslineare Grammatiken und reguläre Ausdrücke transformierbar. Diese Transformationen sind auch in der anderen Richtung möglich.

5.1 Eingeben, Ausgeben, Simulieren und Analysieren

Automaten werden entweder aus einer Datei eingelesen

```
EndlicherAutomat ea = new EndlicherAutomat();
ea.dateiEinlesen("beispiele\\endlicheautomaten\\beispielautomat.atm");
```

oder direkt als String beschrieben. Die nachfolgende Eingabe zeigt die typischen Möglichkeiten, die ebenfalls für eine Text-Datei genutzt werden können. Es sind ein Alphabet, eine Zustandsmenge, eine Endzustandsmenge, ein Startzustand und die Überführungsfunktion(en) zu übergeben. Die Überführungsschritte sind als Tripel mit dem Ausgangszustand, dem verarbeiteten Zeichen und dem Folgezustand anzugeben. Für Epsilon-Überführungsschritte ist statt des Zeichens /eps einzugeben. Nichtdeterminismus entsteht dadurch, dass zu einem Ausgangszustand und einem Zeichen mehrere Zeilen mit verschiedenen möglichen Nachfolgezuständen angegeben werden.

```
ea.stringAlsAutomat("""
    % Kommentarmoeglichkeit (geht ohne)
    % a am Ende nichtdeterministisch mit Epsilon
    % Alphabet
    A: a b
    % Zustaende
    Z: z0 z1 z2
    % Endzustaende
    E: z1 z2
    % Startzustand
    S: z0
    % Ueberfuehrung
    % von mit  nach
      z0  a    z1
    % Uebergang mit Espilon
      z1  /eps z2
      z2  /eps z0
    % naechste zwei Zeilen nichtdeterministisch
      z2  a    z1
      z2  a    z2
      z1  b    z0
      z0  b    z0
      z0  a    z0
    """);
```

Generell gibt es mehrere Wege die Ausführung eines Automaten interaktiv zu simulieren. Die einfachste ist folgende Methode.

```
ea.wortAbarbeiten("abba");
```

Die nachfolgende Ausgabe zeigt, dass Auswahlmöglichkeiten nur angeboten werden, wenn es mehr als eine gibt.

```
Eingabe: abba  Zustand: z0
Welchen der moeglichen Folgezustaende?
```

```
(0) z1
(1) z0
0
Eingabe: bba  Zustand: z1
Epsilonschritt (e) oder erstes Zeichen abarbeiten (a): e
Welchen der moeglichen Folgezustaende?
(0) z2
(1) z0
0
Eingabe: bba  Zustand: z2
Eingabe: bba  Zustand: z0
Eingabe: ba  Zustand: z0
Eingabe: a  Zustand: z0
Welchen der moeglichen Folgezustaende?
(0) z1
(1) z0
0
Wort wurde auf diesem Weg akzeptiert
```

Alternativ kann zuerst ein Wort mit `bearbeite(.)` übergeben und mit einer der verschiedenen Varianten von `ausfueherenSimulation(.)` verarbeitet werden.

5.2 Überführung in einen deterministischen Automaten

Die üblichen Bearbeitungsschritte können für endliche Automaten ausgeführt werden, dabei wird jeweils das Automaten-Objekt selbst verändert. Es ist möglich mit `clone()` eine echte Kopie der Automaten zu erhalten, die allerdings mit den identischen Zustands- und Zeichen- (Terminal-) Objekten arbeitet.

```
  public static void main(String[] args) {
    EndlicherAutomat ea = new EndlicherAutomat();
    ea.stringAlsAutomat("""
        % Kommentarmoeglichkeit (geht ohne)
        % a am Ende nichtdeterministisch mit Epsilon
        % Alphabet
        A: a b
        % Zustaende
        Z: z0 z1 z2
        % Endzustaende
        E: z1 z2
        % Startzustand
        S: z0
        % Ueberfuehrung
        % von mit  nach
          z0  a    z1
        % Uebergang mit Espilon
          z1  /eps z2
          z2  /eps z0
        % naechste zwei Zeilen nichtdeterministisch
          z2  a    z1
          z2  a    z2
          z1  b    z0
          z0  b    z0
          z0  a    z0
```

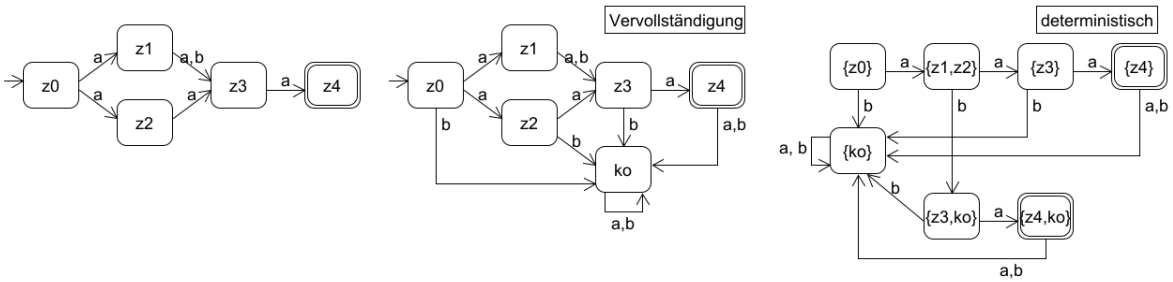
```
        """);
    System.out.println("istohneEps: " + ea.istOhneEpsilon());
    System.out.println("istdeterministisch : " + ea.istDeterministisch());
    System.out.println("istvollstaendig : " + ea.istVollstaendig());
    ea.epsilonEntfernen();
    System.out.println("\nistohneEps: " + ea.istOhneEpsilon());
    System.out.println(ea);
    ea.vervollstaendigen();
    System.out.println("\nistvollstaendig : " + ea.istVollstaendig());
    System.out.println("\n" + ea);
    ea.deterministisch();
    System.out.println("\nistdeterministisch : " + ea.istDeterministisch());
    System.out.println("\n" + ea);
  }
```

Die zugehörige Ausgabe sieht wie folgt aus. Zur Berechnung der Akzeptanz wird eine Kopie des Automaten erstellt und zunächst in eine deterministische Form umgewandelt. Für diesen Automaten werden dann die Verarbeitungsschritte als Konfigurationsfolge angegeben.

```
istohneEps: false
istdeterministisch : false
istvollstaendig : false

istohneEps: true
Zustaende: [z0, z1, z2]
Endzustaende: [z1, z2]
Alphabet: [a, b]
Start: z0
Ueberfuehrungsfunktion:
 ueber(z2, a) = z1
 ueber(z2, a) = z2
 ueber(z2, a) = z0
 ueber(z0, a) = z1
 ueber(z0, a) = z2
 ueber(z0, a) = z0
 ueber(z1, b) = z0
 ueber(z1, a) = z1
 ueber(z1, a) = z2
 ueber(z1, a) = z0
 ueber(z2, b) = z0
 ueber(z0, b) = z0


istvollstaendig : true

Zustaende: [z0, z1, z2]
Endzustaende: [z1, z2]
Alphabet: [a, b]
Start: z0
Ueberfuehrungsfunktion:
 ueber(z2, a) = z1
 ueber(z2, a) = z2
```

```
 ueber(z2, a) = z0
 ueber(z0, a) = z1
 ueber(z0, a) = z2
 ueber(z0, a) = z0
 ueber(z1, b) = z0
 ueber(z1, a) = z1
 ueber(z1, a) = z2
 ueber(z1, a) = z0
 ueber(z2, b) = z0
 ueber(z0, b) = z0


istdeterministisch : true

Zustaende: [z0_, z0_z1_z2]
Endzustaende: [z0_z1_z2]
Alphabet: [a, b]
Start: z0_
Ueberfuehrungsfunktion:
 ueber(z0_, a) = z0_z1_z2
 ueber(z0_, b) = z0_
 ueber(z0_z1_z2, a) = z0_z1_z2
 ueber(z0_z1_z2, b) = z0_
```

Beim Übergang vom nicht - deterministischen zum deterministischen Automaten wird vor der Potenzmengenkonstruktion der Automat vervollständigt. Der Ablauf ist in dem folgenden Diagramm skizziert.



Das zugehörige Programm sieht wie folgt aus.

```
public static void main(String[] args) {
  EndlicherAutomat ea = new EndlicherAutomat();
  ea.stringAlsAutomat("""
      A: a b
      Z: z0 z1 z2 z3 z4
      E: z4
      S: z0
      z0  a  z1
      z0  a  z2
      z1  a  z3
      z1  b  z3
      z2  a  z3
      z3  a  z4
      """);
```

```
    ea.deterministisch();
    System.out.println(ea);
  }
```

Die Ausgabe zeigt, dass der Zustand ko hier z0001 heißt. Dies ist ein automatisch generierter, bisher nicht vorhandener Zustand. An den Namen der Zustände ist jeweils die zugehörige Menge an Zuständen zu erkennen. Im Namen steht jeweils nachfolgend für jeden Zustand der Menge der Zustandsnamen gefolgt mit einem Unterstrich.

```
Zustaende: [z0_, z1_z2, z0001_, z3_, z3_z0001, z4_, z4_z0001]
Endzustaende: [z4_, z4_z0001]
Alphabet: [a, b]
Start: z0_
Ueberfuehrungsfunktion:
 ueber(z0_, a) = z1_z2
 ueber(z0_, b) = z0001_
 ueber(z0001_, a) = z0001_
 ueber(z3_, a) = z4_
 ueber(z4_, a) = z0001_
 ueber(z3_, b) = z0001_
 ueber(z4_, b) = z0001_
 ueber(z1_z2, b) = z3_z0001
 ueber(z1_z2, a) = z3_
 ueber(z4_z0001, a) = z0001_
 ueber(z4_z0001, b) = z0001_
 ueber(z0001_, b) = z0001_
 ueber(z3_z0001, b) = z0001_
 ueber(z3_z0001, a) = z4_z0001
```

Die Berechnung des deterministischen Automaten kann auch ohne vorherige Vervollständigung erfolgen, es gibt dann den Zustand {} aus der Potenzmenge der Zustände für einen nicht erreichbaren Zustand. Soll dieser Ansatz genutzt werden, ist die Methode deterministisch(false) mit dem Booleschen Parameter false zu nutzen.

```
    ea.deterministisch(false);
```

5.3 Automaten minimieren und als Grammatik ausgeben

Für Automaten besteht weiterhin die Möglichkeit den zugehörigen minimalen Automaten zu berechnen, die dabei genutzte Matrix zur Bestimmung der Äquivalenz sich ausgeben zu lassen (X bedeutet nicht äquivalent) und zum zugehörigen Automaten eine rechtslineare Grammatik zu generieren, die wie andere Grammatiken auch nutzbar ist. Die Grammatiken sind im Detail nur verständlich, wenn das Kapitel zu kontextfreien Grammatiken gelesen wurde. Bei den jeweiligen Berechnungen werden neue Zustände generiert, deren Namen sich aus der Berechnungsart ergeben. So werden Zustände, die aus Zustandsmengen erzeugt werden, als mit einem Unterstrich verkette Namen ausgegeben. Da die Namen sehr lang werden können, sind die Namen durch die Methode `zustandsnamenKuerzen()` in kürzere Namen umbenennbar (wäre im Beispiel unnötig).

```
    ea = new EndlicherAutomat();
    ea.stringAlsAutomat("""
        % Automat zur Minimierung aus der Vorlesung
        A: a b
```

```
        Z: z0 z1 z2 z3 z4
        E: z0 z3
        S: z0
        z0 a z1
        z0 b z3
        z1 a z2
        z1 b z4
        z2 a z0
        z2 b z2
        z3 a z4
        z3 b z0
        z4 a z2
        z4 b z4
        """);
    ea.minimieren();
    ea.zeigeMinimierungsmatrix();
    System.out.println(ea);
    ea.zustandsnamenKuerzen();
    System.out.println("\n" + ea);
    KontextfreieGrammatik kfg = ea.alsGrammatik();
    System.out.println("\n" + kfg);
```

Die zugehörige Ausgabe sieht wie folgt aus.

```
 z z z z z
 0 1 2 3 4
   X X   X  z0
     X X    z1
       X X  z2
         X  z3
---------------------------------
Zustaende: [z0_z3, z1_z4, z2_]
Endzustaende: [z0_z3]
Alphabet: [a, b]
Start: z0_z3
Ueberfuehrungsfunktion:
 ueber(z0_z3, a) = z1_z4
 ueber(z0_z3, b) = z0_z3
 ueber(z2_, a) = z0_z3
 ueber(z2_, b) = z2_
 ueber(z1_z4, b) = z1_z4
 ueber(z1_z4, a) = z2_


Zustaende: [z0006, z0007, z0008]
Endzustaende: [z0006]
Alphabet: [a, b]
Start: z0006
Ueberfuehrungsfunktion:
 ueber(z0008, b) = z0008
 ueber(z0007, a) = z0008
 ueber(z0006, a) = z0007
```

```
 ueber(z0007, b) = z0007
 ueber(z0008, a) = z0006
 ueber(z0006, b) = z0006


Terminale =[a, b]
Nichtterminale = [z0007, z0006, z0008]
Start = z0006
 (1) z0006 -> az0007
 (2) z0006 -> bz0006
 (3) z0006 -> ε
 (4) z0007 -> az0008
 (5) z0007 -> bz0007
 (6) z0008 -> az0006
 (7) z0008 -> bz0008
```

Die andere Richtung wird ebenfalls angeboten, für eine rechtslineare Grammatik kann ein zugehöriger Automat konstruiert werden. Für Regeln der Form A -> aaB wird am Ende eine Epsilon-Regel vom letzten a zum A ergänzt, hier wird in der Vorlesung ein Zustand und damit Episilon-Übergang wegoptimiert.

```
  public static void main(String[] args) {
    String grammatik =
        """
        N: A B
        T: a b
        S: A
        r1:: A -> aa | aaaA | aaB | B
        r2:: B -> bbB | b | /eps
        """;
    KontextfreieGrammatik g = new KontextfreieGrammatik();
    g.stringAlsGrammatik(grammatik);
    System.out.println("ist rechtslinear? " + g.istRechtslinear() + "\n");
    EndlicherAutomat ea = g.rechtslinearInAutomaten();
    System.out.println(ea);
  }
```

Die Ausgabe sieht wie folgt aus:

```
      ist rechtslinear? true

      Zustaende: [A, B, z0001, z0002, z0003, z0004, z0005, z0006, z0007,
      z0008, z0009, z0010]
      Endzustaende: [z0002, z0010, B]
      Alphabet: [a, b]
      Start: A
      Ueberfuehrungsfunktion:
       ueber(A, a) = z0001
       ueber(A, a) = z0003
       ueber(A, a) = z0006
       ueber(z0001, a) = z0002
       ueber(B, b) = z0008
       ueber(B, b) = z0010
       ueber(z0008, b) = z0009
       ueber(z0006, a) = z0007
```

```
ueber(z0003, a) = z0004
ueber(z0004, a) = z0005
ueber(A, ε) = B
ueber(z0005, ε) = A
ueber(z0007, ε) = B
ueber(z0009, ε) = B
```

5.4 Bearbeitungsmöglichkeiten für reguläre Ausdrücke

Zunächst ist zu beachten, dass nur reguläre Ausdrücke unterstützt werden, die aus folgender Grammatik ableitbar sind, dabei sind die geforderten Klammern zu beachten:

```
%ignorespace
N: Reg
T: a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3
T: 4 5 6 7 8 9 ( ) * + {}
S: Reg
Reg -> (Reg)* | (Reg+Reg) | RegReg | (Reg)
Reg -> {} | a | b | c | d | e | f | g | h | i | j | k | l | m
Reg -> n | o | p | q | r | s | t | u | v | w | x | y | z | 0
Reg -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Reguläre Ausdrücke können direkt aus Text eingegeben werden. Mit der Methode wird dann überprüft, ob ein Wort zur Sprache des regulären Ausdrucks gehört.

```
public static void main(String[] args) {
  RegulaererAusdruck re = RegulaererAusdruck.alsAusdruck("(a+(b)*)");
  System.out.println("re: " + re);
  System.out.println("bb: " + re.beinhaltet("bb"));
  System.out.println("aa: " + re.beinhaltet("aa"));
}
```

Die Ausgaben lauten:

```
re: (a + (b)*)
bb: true
aa: false
```

Alternativ lassen sich reguläre Ausdrücke auch aus Objekten zusammensetzen. Das Verhalten ist sonst identisch.

```
public static void main(String[] args) {
  RegulaererAusdruck re = new Leer();
  System.out.println("re: " + re);
  re = Zeichen.zeichen("a");
  System.out.println("re: " + re);
  re = new Punkt(re, Zeichen.zeichen("b"), re); // Konkatenation
  System.out.println("re: " + re);
  re = new Klammern(re);
  System.out.println("re: " + re);
  re = new Oder(Zeichen.zeichen("c"), re);
  System.out.println("re: " + re);
  re = new Stern(re);
  System.out.println("re: " + re);
}
```

Die Ausgaben lauten:

```
re: {}
re: a
re: aba
re: (aba)
re: (c + (aba))
re: ((c + (aba)))*
```

Für reguläre Ausdrücke wird die Umformung in einen endlichen Automaten und umgekehrt angeboten.

```
public static void main(String[] args) {
  RegulaererAusdruck re = RegulaererAusdruck.alsAusdruck("(a+(b)*)");
  EndlicherAutomat ea = re.alsAutomat();
  System.out.println(ea);
  EndlicherAutomat ea2 = new EndlicherAutomat();
  ea2.stringAlsAutomat("""
      % Automat_det
      Z: z0 z1 z2 z3 z4
      E: z0 z3 z4
      S: z0
      A: a b
      z0 /eps z1
      z0 /eps z2
      z1 a z3
      z2 b z4
      z4 b z4
      """);
  System.out.println(ea2.alsRegulaererAusdruck());
```

Die Ausgaben lauten:

```
Zustaende: [z0001, z0002, z0003, z0004, z0007, z0008, z0005, z0006]
Endzustaende: [z0002]
Alphabet: [a, b]
Start: z0001
Ueberfuehrungsfunktion:
 ueber(z0003, a) = z0004
 ueber(z0005, b) = z0006
 ueber(z0004, ε) = z0002
 ueber(z0006, ε) = z0008
 ueber(z0006, ε) = z0005
 ueber(z0001, ε) = z0003
 ueber(z0001, ε) = z0007
 ueber(z0008, ε) = z0002
 ueber(z0007, ε) = z0005
 ueber(z0007, ε) = z0008

((({})* + a) + b(b)*)
```

Soll dieser Ansatz für selbst erstellte Übungsaufgaben genutzt werden, ist es sinnvoll die zugehörigen Automaten zu minimieren (vorher deterministisch zu machen) und mit `istMinimalIsomorph()` auf die Akzeptanz der gleichen Sprache zu prüfen. Mit der Methode `nichtGemeinsamesWort()` kann

nach einem Wort gesucht werden, das nur von einem der Automaten akzeptiert wird. Sollte es ein solches Wort nicht geben, wird eine IllegalStateException geworfen.

```
    ea.deterministisch().minimieren().zustandsnamenKuerzen();
    ea2.deterministisch().minimieren().zustandsnamenKuerzen();
    boolean erg = ea.istMinimalIsomorph(ea2);
    Wort w = null;
    if (!erg) {
      w = ea.nichtGemeinsamesWort(ea2);
      System.out.println(w + "wird nicht gleichbehandelt");
    } else {
      System.out.println("sind sprachaequivalent\n" + ea);
    }
  }
```

Die Ausgaben lauten:

```
sind sprachaequivalent
Zustaende: [z0013, z0014, z0015, z0016]
Endzustaende: [z0013, z0014, z0015]
Alphabet: [a, b]
Start: z0013
Ueberfuehrungsfunktion:
 ueber(z0013, b) = z0015
 ueber(z0015, b) = z0015
 ueber(z0014, a) = z0016
 ueber(z0014, b) = z0016
 ueber(z0013, a) = z0014
 ueber(z0016, a) = z0016
 ueber(z0016, b) = z0016
 ueber(z0015, a) = z0016
```

Für reguläre Ausdrücke kann direkt geprüft werden, ob die Ausdrücke äquivalent sind. Falls dies nicht der Fall ist, muss für ein Gegenbeispiel erst jeweils der deterministische Automat erstellt und dann nach dem unterscheidenden Wort gesucht werden.

```
  public static void main(String[] args) {
    RegulaererAusdruck re = RegulaererAusdruck.alsAusdruck("(a+(b)*)");
    System.out.println("aequivalent: "
        + re.istAequivalent("((({})* + a) + b(b)*)"));
    RegulaererAusdruck re2 = RegulaererAusdruck.alsAusdruck("(aa+(b)*)");
    System.out.println("aequivalent: " + re.istAequivalent(re2));
    EndlicherAutomat ea = re.alsAutomat().deterministisch().minimieren();
    EndlicherAutomat ea2 = re2.alsAutomat().deterministisch().minimieren();
    Wort unterschied = ea.nichtGemeinsamesWort(ea2);
    System.out.println("Verhalten unterscheidet sich bei: " + unterschied);
    System.out.println(re + ": " + re.beinhaltet(unterschied));
    System.out.println(re2 + ": " + re2.beinhaltet(unterschied));
  }
```

Die Ausgaben lauten:

```
aequivalent: true
aequivalent: false
Verhalten unterscheidet sich bei: a
(a + (b)*): true
(aa + (b)*): false
```

Da das Alphabet zu einem regulären Ausdruck direkt aus diesem Ausdruck berechnet wird, kann der Vergleich mit einem Automaten schwierig werden. Um Zeichen zum Alphabet eines regulären Ausdrucks zu ergänzen, kann z. B. der ursprüngliche Ausdruck mit „+ xyz{}“ verlängert werden. So gehören x, y und z garantiert zum Alphabet des Ausdrucks, da aber die Konkatenation mit der leeren Menge die leere Menge ergibt, wird die Sprache nicht verändert.

```
  public static void main(String[] args) {
    RegulaererAusdruck re = RegulaererAusdruck.alsAusdruck("(b)*");
    EndlicherAutomat ea = re.alsAutomat().deterministisch().minimieren();
    EndlicherAutomat ea2 = new EndlicherAutomat();
    ea2.stringAlsAutomat("""
        Z: z0 z1
        E: z0
        S: z0
        A: a b
        z0 a z1
        z0 b z0
        z1 a z1
        z1 b z1
        """);
    try {
      System.out.println(ea.istMinimalIsomorph(ea2));
    } catch (IllegalStateException e) {
      System.err.println(e.getMessage());
    }
    re = RegulaererAusdruck.alsAusdruck("((b)* + a{})");
    ea = re.alsAutomat().deterministisch().minimieren();
    System.out.println(ea.istMinimalIsomorph(ea2));
  }
```

Die Ausgaben lauten:

```
gleiches Alphabet notwendig, this-Automaten fehlt a
true
```