INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Eine Toolbox zur Visualisierung von Algorithmen der theoretischen Informatik

Isabel Wingen

Bachelorarbeit

Beginn der Arbeit:	20. Dezember 2016
Abgabe der Arbeit:	20. März 2017
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Jörg Rothe

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 20. März 2017

_____

Isabel Wingen

Zusammenfassung

Das Ziel dieser Bachelorarbeit war es, ein Programm, welches Werkzeuge zum Simulieren und Visualisieren von Grammatiken und Automaten zur Verfügung stellt, weiterzuentwickeln und zu verbessern. Diese *Toolbox* ist in Java programmiert und so angelegt, dass eine leichte Erweiterung möglich ist. Der Fokus dieser Toolbox lag hauptsächlich auf Mitteln, die für die Entwicklung eines Compilers hilfreich sind.

Es bestand der Wunsch, die Toolbox als Hilfmittel in der Lehre zur Vorlesung *Einführung in die theoretische Informatik* für den Bereich der formalen Sprachen einzusetzen. Hierfür war der Funktionsumfang aber nicht ausreichend, da viele Algorithmen zu Grammatiken, wie Umformung in Chomsky-Normalform, fehlten. Desweiteren sollte es möglich sein, vollführte Änderung an den Objekten rückgängig zu machen. Die ausgeführten Algorithmen sollten in LaTeX-Dateien exportiert werden können, so dass die Toolbox auch für die Lehrenden beim Erstellen von Foliensätzen helfen kann.

Das Programm und die Benutzeroberfläche wurden so umgearbeitet, dass ein paralleles Arbeiten mit mehren Objekten möglich ist.

Als wichtigste Änderung sei die Umwandlung der Objekte in unmodifierzierbare Objekte genannt. Dies erleichterte die Speicherung von vorherigen Versionen und machte eine Wiederherstellung möglich.

Außerdem wurden folgenden Algorithmen implementiert:

Grammatik

- Entfernen von λ-Regeln
- Entfernen von einfachen Regeln
- Umformen in Chomsky-Normal-Form
- CYK-Algorithmus
- Automatisches und interaktives Finden eines Pfades
- Umwandeln in einen Kellerautomat

Kellerautomat

- Umwandeln in eine äquivalente Grammatik
- Interaktives Finden eines Pfades.

Die gewünschten Ziele konnten umgesetzt werden.

Die Toolbox ist nun intuitiver zu bedienen und gut als Lehrmittel einzusetzen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Dieser Bachelorarbeit liegt eine von Fabian Ruhland im Zuge seiner Bachelorarbeit im Sommersemester 2016 entwickelte Toolbox [Ruh16] (im Folgenden als Version 1.0 bezeichnet) zur Visualisierung von Automaten und Grammatiken zu Grunde. Der Fokus dieser Arbeit lag auf Automaten und der Entwicklung eines Systems, das leicht erweiterbar ist. Die in der Toolbox 1.0 enthaltenen Algorithmen sind besondern hilfreicht für die Entwicklung eines Compilers.

In dieser Arbeit wird die Toolbox nun erweitert. Die Toolbox 1.0 stellte nur einen kleinen Teil der Algorithmen auf dem Gebiet der formalen Sprache zur Verfügung. Der Funktionsumfang soll nun in dem Maße vergrößert werden, dass die Toolbox als unterstützendes Mittel in der Lehre der Vorlesung *Einführung in die theoretische Informatik* eingesetzt werden kann. Außerdem soll die Benutzerfreundlichkeit erhöht werden. Die Toolbox soll zum Verständnis der Studierenden beitragen. Hierzu ist eine nachvollziehbare und schrittweise Durchführung der Algorithmen wünschenswert. Es soll Spaß machen, die Toolbox zu benutzen. Ohne viel Aufwand soll es möglich sein, zu experimentieren und auszuprobieren, wie die Algorithmen sich auswirken. Zu diesem Zweck soll die Toolbox eine Möglichkeit zur Verfügung stellen, Änderungen rückgängig zu machen. Eine Überarbeitung der Benutzeroberfläche soll zu mehr Komfort beitragen.

Der Quellcode des Programmes findet sich unter
`https://github.com/Winis04/STUPS-Toolbox-2.0`.

1.2 Theoretische Grundlagen

In diesem Abschnitt werden zunächst grob die theoretischen Grundlagen der formalen Sprachen erläutert, die zum Verständnis der Toolbox beitragen. Für eine genauere Ausführung sei auf [Rot16] verwiesen.

1.2.1 Alphabet und Wörter

Ein Alphabet ist eine endliche, nichtleere Menge von Symbolen [Rot16]. Ein Wort über einem Alphabet ist eine endliche Kombination aus Symbolen aus dem Alphabet.

1.2.2 Formale Sprache

Eine *formale Sprache* über einem Alphabet Σ ist eine Menge von Wörtern über Σ.

1.2.3 Grammatik

Eine kontextfreie *Grammatik* ist ein Quadrupel $G = (N, \Sigma, S, R)$.

Hierbei sei

Σ – ein Alphabet. Elemente aus diesem Alphabet werden *Terminale* genannt,

N – eine Menge von Nichtterminalen. Dies sind Symbole, die nicht in Σ sind und die durch andere Terminale und Nichtterminale ersetzt werden können,

R – eine Menge von *Produktionsregeln*.$P \subset (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$

S – das *Startsymbol* einer Grammatik, $S \in N$.

Das Anwenden einer Regel wird *Ableiten* genannt und durch $\vdash$ beschrieben. Eine Folge von Nichtterminalen und Terminalen ist eine *Konfiguration*. Das Startsymbol S ist die Startkonfiguration. Durch Ableiten von Symbolen in einer Konfiguration erhält man eine neue Konfiguration. Sobald diese nur noch Terminale enthält, wurde ein Wort in der von G erzeugten Sprache gefunden.

Die durchlaufenen Konfigurationen sind der *Ableitunspfad* dieses Wortes.

Die von G erzeugte Sprache wird mit $L(G)$ bezeichnet.

Eine Grammatik heißt *kontextfrei*, falls für jede Regel $(p, q) \in P$ gilt: $p \in N$.

Die Menge der Sprachen, die von kontextfreien Grammatiken erzeugt werden, ist die Menge der kontextfreien Sprachen.

1.2.4 Kellerautomat

Ein Kellerautomat besteht aus

- einer Menge von Zuständen,
- einem Eingabe-Alphabet,
- einem Stack,
- einem Stack-Alphabet,
- einem initialen Stacksymbol,
- und einer Überführungsfunktion δ.

Ein Kellerautomat befindet sich immer in einem Zustand und bekommt ein Wort als Eingabe. Durch die Überführungsfunktion kann er in einen anderen Zustand übergehen und dabei ein Symbol der Eingabe lesen und den Stack verändern. Ein Kellerautomat akzeptiert ein Wort, falls der Stack leer ist, nachdem die Eingabe komplett abgearbeitet worden ist.

1.2.5 Äquivalenz

Zu jeder kontextfreien Grammatik G gibt es einen äquivalenten Kellerautomaten M, d. h. M akzeptiert die von G erzeugte Sprache, und umgekehrt.

2 Die Toolbox

2.1 Typen

Die Toolbox 1.0 konnte mit Grammatiken und Automaten umgehen. Hierzu gehören insbesondere, aber nicht ausschließlich

- die Entwicklung eines eigenen Datentypes für Grammatiken und Automaten
- das Parsen von Grammatiken und Automaten mittels SableCC[1]
- das Speichern von Grammatiken und Automaten in Datein
- First- und Follow-Set einer Grammatik
- Vervollständigung und Entfernen von λ-Übergängen von Automaten

2.2 Erweiterbarkeit

Das von Fabian Ruhland entwickelte System legt Wert darauf, leicht weiterentwickelbar zu sein. Hierfür wurde ein Plugin-System entworfen, das es einem anderen Programmieren erleichtert, neue Features hinzuzuentwickeln, ohne sich mit dem bereits bestehenden Code vertraut zu machen. Es gibt hierbei verschiedene Arten von Plugins, die automatisch eingebunden werden:

- CLI-Plugins – Kommandozeilenprogramme
- SimpleFunction Plugins – GUI-Funktionen, die keine Nutzereingabe benötigen
- ComplexFunction Plugins – GUI-Funktionen, die Nutzerinteraktion benötigen
- DisplayPlugins – Darstellung von Datentypen

2.3 Mechanismen

Das Programm sollte sowohl über die Konsole als auch über eine Benutzeroberfläche bedient werden können. Hierfür gibt es die beiden zentralen Klassen *CLI* für die Konsole und *GUI* für die grafische Benutzeroberfläche. Die Plugins haben die Methoden *getInputType* und *getOutputType*. Diese lieferen eine Klasse, von der das Plugin eine Instanz als Eingabe bekommt bzw. als Ausgabe zurückgibt. Sobald der Nutzer ein Kommando in der Konsole (oder der GUI) durchführt, ruft die Klasse CLI (oder die Klasse GUI) die *execute*-Methode des jeweiligen Plugins auf und übergibt das aktuelle Objekt der passende Klasse. Von jeder Klasse ist immer nur ein Objekt gespeichert. Das von der *execute*-Methode zurückgelieferte Objekt überschreibt das bisherige.

[1] hier Referenz

2.4 GUI

In Abbildung 1 ist die Benutzeroberfläche der Toolbox 1.0 im Grammar-Modus dargestellt. Die aktuelle Grammatik wird zum Bearbeiten angezeigt. Um Automaten zu bearbeiten, muss man unter *Choose Plugin* den Punkt *Automaton* wählen.

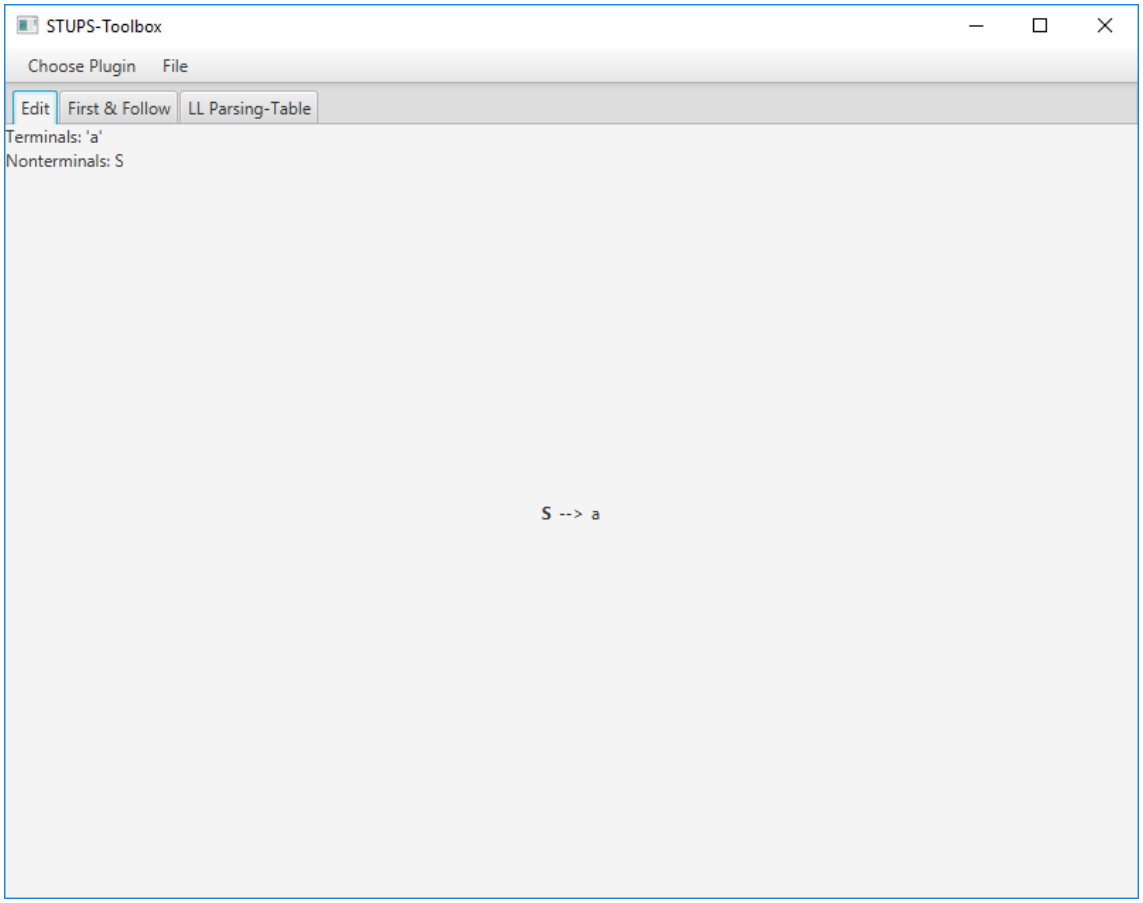

Abbildung 1: Die GUI der Toolbox 1.0

3 Benutzerebene

Viele der Änderung betreffen die Funktionalität der Toolbox und sind für den Benutzer direkt ersichtlich. Im Folgenden wird auf diese Änderungen auf Nutzerebene eingegangen.

3.1 GUI-Design

Die Benutzeroberfläche (*GUI*) ist modernisiert worden. Sie hat bunte Elemente erhalten, die sich per CSS-Stylesheet anpassen lassen. Außerdem ist die Bedienung intuitiver gestaltetet worden. Eine Abbildung der GUI findet sich in Abb. 2.

Aufbau Beim Design der GUI wurde sich an bekannten IDEs, wie zum Beispiel IntelliJ Idea[2], orientiert. Im linken Teil der GUI befindet sich ein Baum mit drei Unterknoten: Grammar, PushDownAutomaton (Kellerautomat) und Automaton. Klappt man diese Unterknoten auf, erhält man eine Übersicht über alle momentan gespeicherten Objekte dieses Types. Eine Abbildung der GUI findet sich in Abb. 2.

Bedienung Über den Baum lassen sich die verschiedenen Objekte auswählen und werden im rechten Teil dargestellt. Im unteren rechten Teil befinden sich die ComplexFunctionPlugins. Diese arbeiten immer auf dem aktuell ausgewählten Typ und sind über Tabs auswählbar.

Die SimpleFunctionPlugins lassen sich durch Rechtsklick auf das Objekt im Baum, auf dem sie durchgeführt werden sollen, aufrufen. Es gibt nun SimpleFunctionPlugins, die auf allen Objekten im Baum arbeiten. Hierzu gehören *Print*, *Rename* und *Undo*, welches eine Änderung rückgängig macht.

Vorteile Die Bedienung ist intuitiver geworden und ermöglicht es, schnell zwischen den verschiedenen Datentypen zu wechseln, ohne bestehende Änderungen zu verlieren. Auch zwischen Objekten desselben Datentypes kann leicht gewechselt werden und diese sind dadurch gut vergleichbar. Die Funktion, eine Änderung rückgängig zu machen, ermöglicht es, Algorithmen auszuprobieren und danach die ursprünglichen Daten ohne viel Aufwand wieder herzustellen.

Alle vorgenommen Einstellungen, wie CSS-Stylesheets oder das Symbol für das leere Wort, werden gespeichert, so dass eine komfortable, persönliche Modifizierung möglich ist.

3.2 Feature-Übersicht

Im Folgenden werden die neuen Features kurz erklärt. Hierbei geht es hauptsächlich um die Ausführung der Befehle, eine genaue Erklärung der Algorithmen findet sich in Kapitel 6.

3.2.1 Ausgabe

Damit die Durchführung der Algorithmen transparent ist, ist eine dokumentierte Ausführung möglich. Hierbei wurden die Algorithmen in Schritte eingeteilt, die sich an den Schritten der Algorithmen in [Rot16] orientieren. Auf Wunsch ist eine Ausgabe dieser Dokumentation möglich.
An dieser Stelle sei bemerkt, dass es in der vorgebebenen Zeit nicht möglich war, für alle bereits bestehenden Funktionen eine Ausgabe zu erstellen.

[2] `https://www.jetbrains.com/idea/`

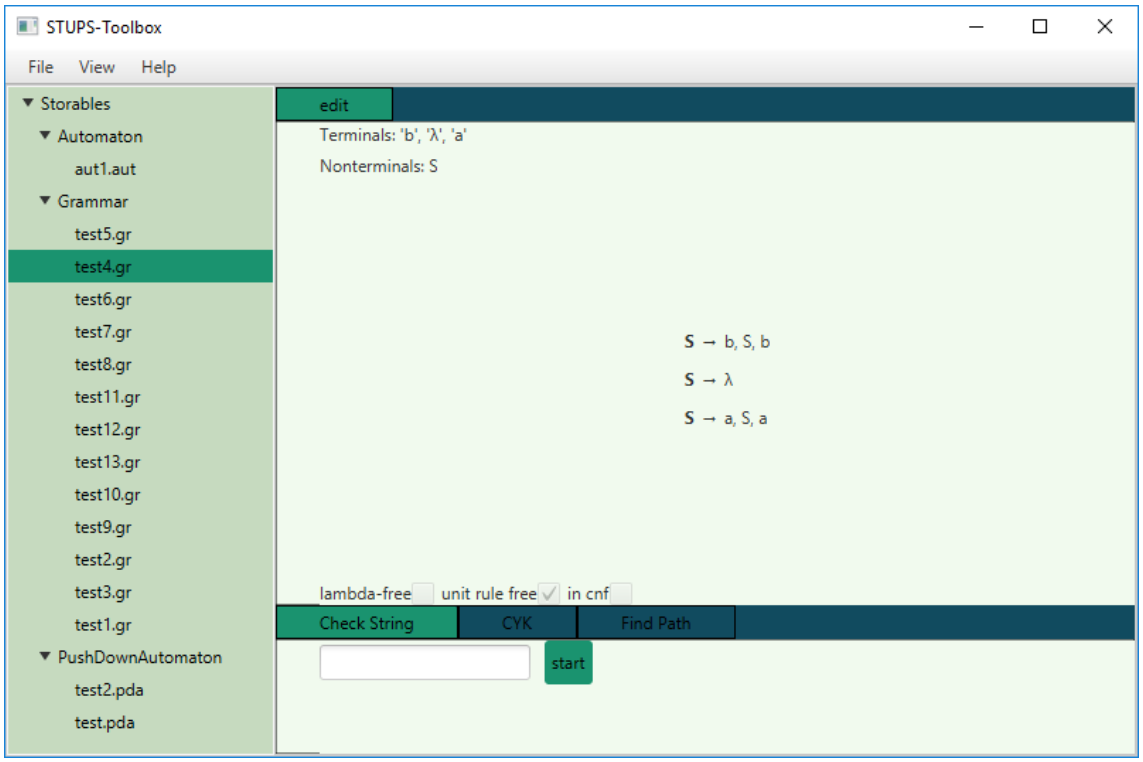

Abbildung 2: Die GUI der Toolbox 2.0

Für die Ausgabe gibt es den sogenannnten *Printmodus*. Dieser hat drei Zustände: *no*, *console* und *latex*. Im Modus *no* erfolgt keine Ausgabe, im Modus *console* erfolgt eine Ausgabe auf die Konsole und im Modus *latex* eine LaTeX-konforme Ausgabe in eine kompilierbare LaTeX-Datei. Printmodus *no* und *console* werden durch den Befehl *print-mode type* aufgerufen. Die *GUI* befindet sich standardmäßig im Printmodus *no*.

Der Printmodus *Latex* besitzt dabei einige Besonderheiten, auf die im nächsten Abschnitt genauer eingegangen wird.

LaTeX Kombilierbarer LaTeX-Code wird nur erzeugt, wenn die Toolbox sich im Printmodus *latex* befindet. Dieser muss manuell gestartet und auch wieder beendet werden. Der Printmodus *latex* wird über das Kommando *print-mode latex path/to/file* oder in der GUI über *Export* → *Latex* → *Start Latex Export* gestartet. Das Beenden des LaTeX-Modus geschieht durch den Wechsel in einen anderen *Printmodus* per Konsolenbefehl *print-mode no* oder *print-mode console*, durch das Öffnen oder Schließen der GUI oder in der GUI über *Export* → *Latex* → *End Latex Export*.

Während das Programm im Printmodus *latex* ist, werden alle durchgeführten Befehle mit Ausgabe in die ausgewählte Datei geschrieben.

3.2.2 Grammatiken

Entfernen von λ-Regeln λ-Regeln sind Regeln der Form $A \to \lambda$. λ-Regeln lassen sich per SimpleFunctionPlugin *Remove λ-Rules* oder per Konsolenbefehl *rlr* entfernen.

Entfernen von einfachen Regeln Einfache Regeln (*Unit Rules*) sind Regeln der Form $A \to B$. Sie lassen sich per SimpleFunctionPlugin *Eliminate Unit Rules* oder per Konsolenbefehl *eur* entfernen.

Chomsky-Normal-Form (CNF) Die CNF ist eine Normalform, die von dem Linguisten Noam Chomsky erfunden wurde [Cho59]. Eine Grammatik lässt sich per SimpleFunctionPlugin *CNF* oder per Konsolenbefehl *cnf* in CNF umformen.

CYK-Algorithmus Der CYK-Algorithmus löst in kubischer Zeit das Wortproblem für kontextfreie Sprachen. Der CYK-Algorithmus wurde über ein ComplexFunctionPlugin implementiert. Hierzu muss im Bereich der ComplexFunctionPlugin der Tab „CYK“ ausgewählt und in dem Textfeld das zu untersuchende Wort eingegeben werden. Zu Beachten ist, dass die einzelnen Buchstaben durch Leerzeichen getrennt werden müssen. Der Button „Go“ startet den Algorithmus.

Vereinfachung Per SimpleFunctionPlugin lässt sich eine Grammatik vereinfachen. Hierbei werden Nichtterminale erkannt und entfernt, die redundant, nicht erreichbar oder ohne weiterführende Regeln sind. Falls die Grammatik in Chomsky-Normalform ist, bleibt diese Eigenschaft bestehen.

Umwandlung in Kellerautomat Eine Grammatik lässt sich per SimpleFunctionPlugin *To PDA* oder Konsolenbefehl *topda* in einen Kellerautomaten umwandeln.

Interaktives Ableiten (nur GUI) Als ComplexFunctionPlugin wurde die Möglichkeit implementiert, eine interaktive Ableitung zu erstellen.

3.2.3 Kellerautomaten

Kellerautomaten wurden als neuer Datentyp eingeführt, da sie die kontextfreien Grammatiken gut ergänzen aufgrund der bestehenden Äquivalenzbeziehung.

Umwandeln in Grammatik Ein Kellerautomat lässt sich per SimpleFunctionPlugin *To Grammar* oder Konsolenbefehl *togrammar* in eine äquivalente Grammatik umwandeln.

Bearbeitung (nur GUI) Die Regeln eines Kellerautomatens sind durch *Buttons* dargestellt. Durch Rechtsklick oder Doppelklick auf eine Regel, öffnet sich zunächst ein Menü, in dem ausgewählt werden kann, ob die Regel bearbeitet oder gelöscht werden soll. Wählt man Ersteres, öffnet sich ein Dialog zum Bearbeiten.

Schritt-für-Schritt Durchlauf (nur GUI) Das ComplexFunctionPlugin „Check String“ liefert dem Benutzer die Möglichkeit, interaktiv zu prüfen, ob ein Wort in der Sprache des Automaten liegt. Hierfür bietet das Plugin ein Textfeld und einen „Start“- und „Undo“-Button an. In das Textfeld muss das Wort eingegeben werden, auch hier wieder Trennung der Buchstaben durch Leerzeichen. Der Button „Start“' startet das Plugin. Nun verändern sich die Buttons, die die Regeln des Automatens darstellen. Es sind immer nur die Regel auswählbar, die auch tatsächlich auf den aktuellen Input und *Top of Stack* passen. Durch Klick auf eine Regel wird diese angewandt. Dies kann so lange gemacht werden, bis ein Pfad gefunden wurde, oder keine Regel mehr anwendbar ist.

3.3 Sonstiges

- Der Hilfe-Text (aufrufbar durch den Konsolenbefehl *help*) wurde so formatiert, dass er übersichtlicher ist. Hierzu gehörte das Einrücken der Beschreibungen und eine Sortierung nach dem Alphabet.
- Das leere Wort wird nun nicht mehr als Bestandteil der Terminalmenge angezeigt. Ebenso muss beim Parser einer Grammatik das leere Wort nicht mehr in der Terminalmenge inkludiert werden.
- „epsilon“, „lambda“ und „->“ wurden in der GUI durch entsprechende ASCII-Zeichen ersetzt.
- Durch „u“ und „d“ lässt sich die Schrift in der GUI vergrößern bzw. verkleinern.
- In der GUI gibt es jetzt Tooltips, die die Bedienung erklären.

4 Build-Prozess

Als Build-Prozess wird die automatische Erstellung eines fertigen, ausführbaren Programmes bezeichnet. Bisher wurde dies über ein *Makefile* umgesetzt. Dies hatte den Nachteil, dass das Ausführen nicht plattformunabhängig gewesen ist. Außerdem war es unter anderem aufgrund der externen Bibliotheken nicht möglich, eine JAR erstellen zu lassen.

4.1 Gradle

Als Build-Tool wurde sich für Gradle entschieden. Es ist plattformunabhängig und verfügt über die Möglichkeit mit *Groovy*[3] ausführbare Skripte zu erstellen, was im Folgenden wichtig wird. Die Umstellung auf Gradle erforderte einige Anpassungen des von SableCC erzeugten Codes.

Für die Umstellung musste zunächst der Aufbau der Packages angepasst werden. Gradle setzt eine Trennung von Source-Code und Resourcen voraus. Des Weiteren erwartet Gradle folgende Ordnersturktur:

src

- **main**
  - **java**
  - **resource**
- **test**
  - **java**
  - **resource**

Es wurde außerdem ein weiterer Source-Ordner *GeneratedSource* angelegt, der die von Gradle generierten Dateien enthält. Gradle sorgt dafür, dass alle benötigten Bibiliotheken heruntergeladen werden und zum *Java-Classpath* hinzugefügt werden. Durch ein Plugin[4] kann eine ausführbare .jar-Datei erzeugt werden, die die importierten Bibliotheken enthält. Dies war vorher nicht möglich.

4.1.1 Gradle und SableCC

Das Zusammenspiel von Gradle und SableCC gestaltete sich nicht einfach. SableCC ist ein Parser-Generator, welcher von Etienne M. Gagnon entwickelt wurde.[5] Durch SableCC werden Java-Dateien und Resourcen erzeugt. Leider vermischt SableCC diese. Dies hat zur Folge, dass die Resourcen nicht mehr gefunden werden, da auf sie mit *getResource* zugegriffen wird und sie sich nicht im Resource-Ordner befinden, wo Java die Dateien nach Umstellung auf Gradle sucht.

Es war also nötig, Groovy-Code 1 zu schreiben, welcher nach dem Build-Prozess die benötigten Resourcen in Unterordner des Resource Ordner von *GeneratedSources* kopiert und anschließend die entsprechende *getResource* Zeilen auf diesen Unterordner anpasst.

Die Toolbox besitzt dabei folgenden Abhängigkeiten:

JUNG2 [6] zur Darstellung von Automaten

JUnit 4 [7] zum Testen von Code

Apache Commons-IO [8] zum Bearbeiten von Ordnern

Reflections [9] Erstellen von Objekten zur Laufzeit, auch in der .jar

SableCC Parser-Generator

Apache Commons Lang [10] zum Überschreiben der *equals* und *hashCode*-Methoden

[4] `https://github.com/johnrengelman/shadow`
[5] `http://www.sablecc.org/`
[6] `http://jung.sourceforge.net/`
[7] `http://junit.org/junit4/`
[8] `http://commons.apache.org/proper/commons-io/`
[9] `https://github.com/ronmamo/reflections`
[10] `https://commons.apache.org/proper/commons-lang/`

Listing 1: Groovy-Script für den Gradle-SableCC-Workaround

```
task doAdjustments() << {
    new File("src/main/sablecc") . eachFile ( ) { file ->
      String name = file.getName().split("\\.")[0]
      String contents = new File('generated-src/main/java/'+name+'/
          lexer/Lexer.java').getText( 'UTF-8' )
      contents = contents.replaceAll( 'lexer.dat', '/'+name+'/lexer.dat
          ')
      new File( 'generated-src/main/java/'+name+'/lexer/Lexer.java' ).
          write( contents, 'UTF-8' )

      contents = new File('generated-src/main/java/'+name+'/parser/
          Parser.java').getText( 'UTF-8' )
      contents = contents.replaceAll( 'parser.dat', '/'+name+'/parser.
          dat')
      new File( 'generated-src/main/java/'+name+'/parser/Parser.java' )
          .write( contents, 'UTF-8' )
      copy {
        from 'generated-src/main/java/'+name+'/parser/parser.dat'
        into 'generated-src/main/resources/'+name
      }
      copy{
       from 'generated-src/main/java/'+name+'/lexer/lexer.dat'
       into 'generated-src/main/resources/'+name
      }
    }
  }
}
```

4.1.2 Build

Folgende Befehle sind durch die bereitgestellte ausführbare Gradle-Datei aufrufbar:

- gradle shadowJar
  Erstellt eine ausführbare Jar in build/libs.
- gradle sableCC
  Startet SableCC und erstellt die notwendigen Parser. Dieser Befehl ist in dem Befehl *build* enthalten.
- gradle javadoc
  Erstellt die Dokumentation.
- gradle eclipse bzw. gradle idea
  Ermöglicht es, das Programm als Eclipse-Projekt bzw. IntelliJ-Projekt zu importieren.
- gradle build
  Ein kompletter Build.

5 Entwicklung

Wie schon am Anfang erwähnt, ist dies ein fortgeführtes Projekt. Die Weiterentwicklung eines bereits bestehenden Programmes bietet andere Herausforderungen als die alleinige Entwicklung. Einige Aspekte des Programmes waren nicht mit den Anforderungen dieser Arbeit kompatibel und mussten geändert werden. Im Folgenden wird auf einige notwendige Änderungen und Verbesserungen eingegangen.

5.1 Aufbau der Toolbox

Wie oben erwähnt, benötigt Gradle eine strikte Trennung von Source und Resource. Dies wurde durch Anpassung der Packages erreicht. Außerdem war es nötig, *CLI* und *GUI* aus dem Wurzelverzeichnis in ein eigenes Package zu verschieben, sodass diese Klasse von anderen importiert werden können. Im Folgenden wird kurz auf die einzelnen Packages und ihre Aufgaben eingegangen.

Automaton Simulator Dieses Package beeinhaltet die Models, die benötigt werden, um Automaten darzustellen. Die Logik befindet sich in der Klasse *AutomatonUtil*. Die Klasse *Visitor* stellt die Schnittstelle zu SableCC her und ist für das Parsen von Automaten aus Dateien zuständig.

Grammar Simulator Dieses Package beinhaltet die Models, die benötigt werden, um Grammatiken darzustellen und Algorithmen auf diesen duchrzuführen. Genau wie bei den Automaten stellt *Visitor* die Verbindung zu SableCC her und *GrammarUtil* beinhaltet die Logik.

PushDownSimulator Dieses Package beinhaltet die Models für Kellerautomaten (Pushdown Automatons). *PushDownAutomatonUtil* enthält Funktionen für Kellerautomaten und *Visitor* ist für das Parsen von Kellerautomaten aus Dateien zuständig.

CLIPlugins In diesem Package finden sich die Kommandozeilenprogramme. Jedes der Plugins muss die abstrakte Klasse *CLIPlugin* erweitern, damit es automatisch eingebunden wird. Genaueres dazu findet sich in [Ruh16].

GUIPlugins

- **ComplexFunctionPlugins** Hier finden sich GUIPlugins, die eine Benutzereingabe benötigen.
- **DisplayPlugins** Die Klassen in diesem Package sind für die Darstellung der Modelle zuständig.
- **SimpleFunctionPlugins** SimpleFunctionPlugins operieren auf einem Objekt und benötigen keine weitere Nutzereingabe.

Main Das Package *Main* beinhaltet unter anderem die Klassen *CLI* und *GUI*, welche vorher im Wurzelverzeichnis waren. Diese beiden Klassen sind das Herzstück des Programmes. Außerdem findet sich in dem Package die Klasse *StateController*, die dafür sorgt, dass sich das Programm nach einem Neustart im selben Zustand befindet wie es beendet wurde, und die Klasse *Content*, die die gespeicherten Objekte

enthält und verwaltet. Durch diese beiden neuen Klassen wurde eine größere Aufgabentrennung erreicht.

Print Dieses Package ist für jegliche Art von Schreiboperationen zuständig. Die wichtigeste Klasse ist die Klasse *Printer*. Die Enumeration *Printmode* enthält die Typen *NO* für kein Drucken, *CONSOLE* für Ausgabe auf das Terminal und *LATEX* für die Ausgabe in eine LaTeX-Datei.

5.2 Umgang mit Objekten

Vor der Arbeit bestand der Wunsch, die an Objekten vollführten Änderungen rückgängig machen zu können. Hierzu ist es nötig, das Objekt in seinem *Vorher-Zustand* zu speichern. Leider hatte die Toolbox 1.0 diese Möglichkeit nicht, da jeder Algorithmus auf demselben Objekt arbeitete und dieses veränderte. Es war also nicht (einfach) möglich, eine *Vorher-Version* zu speichern.

Um dies zu ändern, wurde zunächst eine Methode eingeführt, die eine *deep-copy* des Objektes anlegt, also ein Objekt, welches die gleichen Eigenschaften, aber nicht die selben Referenzen hat. Der Nachteil hieran war, dass vor jeder Änderung manuell eine frühere Version gespeichert werden musste. Daher wurden im nächsten Schritt die Objekte *immutable* gemacht, d. h. die Objekte sind nach dem Erstellen nicht mehr veränderbar. Der Programmierer wird gezwungen, für jede Änderung ein neues Objekt anzulegen. Dies erscheint auf den ersten Blick unpraktisch, stellte sich jedoch als keine große Unannehmlichkeit heraus. Diese Veränderung liefert direkt mehrere Vorteile:

- Eine Objektreferenz auf die Vorgängerversion muss direkt im Konstruktor bestimmt werden.
- Beim Verändern der Regelmenge einer Grammatik oder eines Kellerautomaten ist es nun nicht mehr nötig, die Menge der Terminale und Nichtterminale anzupassen, da diese automatisch im Konstruktor aus der Regelmenge bestimmt werden.
- Es kommt zu keinen Konflikten mehr mit der Hashcode-Berechnung. Verändert sich ein Objekt, so verändert sich auch sein Hashcode. Daher sollten Objekte in HashSets nicht mehr verändert werden. Dies war in der alten Toolbox nicht gewährleistet.

Es war nicht nur nötig, die Variablen der Klasse mit dem Kennwort *final* zu kennzeichnen und die Setter-Methoden zu entfernen, es musste auch dafür gesorgt werden, dass in bestehenden Listen oder Sets keine Elemente hinzugefügt oder gelöscht werden. Hierfür wurde in den Getter-Methoden mithilfe von *Collections.unmodifiableSet* bzw *Collections.unmodifiableList*[11] unmodifizierbare Mengen bzw. Listen zurückgegeben.

Aufgrund der recht komplexen, bestehenden Algorithmen auf Automaten, welche häufig auf die Setter-Methoden zugreifen, war es nicht möglich, in der gegebenen Zeit die Automaten unmodifizierbar zu machen. Daher wird weiterhin bei Automaten mit der *deep-copy*-Methode gearbeitet.

[11] `https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`

Als weiteres Feature ist nun das parallele Arbeiten mit mehren Objekten eines Types möglich. Dazu wurde ein sogenannter *Store* eingeführt, der die Objekte für den Nutzer aufbewahrt und verwaltet.
Beim Laden eines Objektes in der Konsole muss es zunächst manuell im Store gespeichert werden. Es gibt weiterhin genau ein aktuelles Objekt jeden Types. Der Nutzer kann diese jedoch mit dem Befehl *switch type name* (bzw in der GUI durch den Baum) ändern. Die Plugins operieren immer noch auf diesem aktuellen Element.

Die Objekte, die im Store gespeichert werden, müssen das Interface *Storable* implementieren. Eine verkürzte Version findet sich in Listing 2.

Listing 2: Das Interface *Storable*

```
public interface Storable extends Printable {
    //a deep-copy of the object.
    Storable deep_copy();
    //[...]
    String getName();

    void printToSave(String path) throws IOException;

    Storable restoreFromFile(File file) throws Exception;

    //returns the previous version of this object
    Storable getPreviousVersion();

    @Override
    void printLatex(String space);

    @Override
    void printConsole();
}
```

Durch das Interface wird gewährleistet, dass jede Klasse eine *deep-copy*-Methode hat. Bei Grammar und PushDownAutomaton wird hier einfach ein neues Objekt erstellt. Dieses ist automatisch wegen der Unmodifizierbarkeit eine echte Kopie. Die Methoden *printToSave* und *restoreFromFile* kümmern sich um das Laden und Speichern des Objektes. Wie im vorherigen Abschnitt erklärt, liefert die Methode *getPreviousVersion* die Vorgängerversion des Objektes. *printLatex* und *printConsole* sind Methoden aus dem Interface Printable und werden genauer in Abschnitt 5.6 erklärt.

Das Interface hat auch den großen Vorteil gegenüber der Toolbox 1.0, dass Grammar, PushDownAutomaton, Automaten, usw. ein gemeinsames Interface implementieren und daher einen gemeinsamen Supertyp haben, der nicht Object ist.

Im Zuge dieser Änderungen wurde gleichzeitig die Vergleichbarkeit von Komponenten von Grammatiken, Automaten und Kellerautomaten ermöglicht. Hierfür wurden die *equals*- und *hashCode*-Methode mit Hilfe der Bibliothek Apache Commons-lang[12] überschrieben. Das Überschreiben der *hashCode*-Methode ist wichtig, da es sonst zu Unregelmäßigkeiten im Verhalten von *HashSets* kommt.

[12] https://commons.apache.org/proper/commons-lang/

Bei einigen Objekten war dies technisch jedoch zunächst gar nicht möglich. So zum Beispiel bei Nichtterminalen, die Referenzen auf die Regeln, bei denen sie auf der linken Seite stehen, enthielten. Dadurch gab es zyklische Referenzen und ein sinnvolles Überschreiben der *equals*-Methode war nicht möglich. Daher wurde die Klasse *Rule* eingeführt, die eine Produktionsregel einer Grammatik beschreibt. Die Grammatik selbst enthält jetzt ein *Set* mit allen ihren Regeln. Die Nichtterminale selbst enthalten keine Regeln mehr. Hierdurch ist der Vergleich von zwei Nichtterminalen sehr einfach geworden. Sie sind genau dann gleich, wenn sie den gleichen Namen haben.

Diese Vergleichbarkeit erleichtert die Durchführung vieler Algorithmen, da nun die Gleichheit verschiedener Objekte nicht mehr von ihrer Objektreferenz abhängig ist.

Hierzu ein kleines Beispiel:

Wollte man in der Toolbox 1.0 eine neue Regel $A \rightarrow a, B, C$, mit A, B, C Nichtterminal und a Terminal, zu einer Grammatik hinzufügen, musste man zunächst aus der Menge der Nichtterminale das Objekt suchen, welches den gewünschten Namen „A“ hat. Dann musste jedes Element auf der rechten Seite in der Menge der Terminale und Nichtterminale gesucht und genau dieses Objekt einer Liste hinzugefügt werden. Diese Liste wurde dann der Regelmenge von Nichtterminal „A“ hinzugefügt.

Listing 3: Das Erstellen einer Regel

```
public Grammar addRule(Grammar g) {
        List<Symbol> symbols = new ArrayList<>();
        symbols.add(new Terminal("a"));
        symbols.add(new Nonterminal("B"));
        symbols.add(new Nonterminal("C"));
        Rule rule = new Rule(new Nonterminal("A"),symbols);
        Set<Rule> rules = new HashSet<Rule>(grammar.getRules());
        rules.add(rule);
        return new Grammar(grammar.getStartsymbol,rules,grammar.
            getName(),grammar);
}
```

Mit den Änderung ist dies viel einfacher geworden. Mit dem Code in Listing 3 wird eine neue Regel erstellt und der Grammatik hinzugefügt. Wie man sieht, ist es nicht nötig, die Referenz der Symbole herauszufinden; man kann einfach neue Objekte erstellen.

Hier wird nochmal die Unmodifizierbarkeit deutlich. Es ist nicht möglich, die Regel direkt der Menge der Regeln hinzuzufügen. Stattdessen wird eine temporäre Regelmenge erstellt und dann eine neue Grammatik, die diese neue Regelmenge hat.

5.3 Persistierung der Objekte

Wie im vorherigen Abschnitt festgestellt wurde, stellt das Interface *Storable* Methoden zum Speichern und Laden zur Verfügung. Beim Beenden des Programmes werden die vollführten Änderung automatisch gespeichert und beim nächsten Start wieder geladen.

Das Programm *merkt* sich Änderungen in einer *config*-Datei. Die *config*-Datei sorgt dafür, dass das Programm im selben Zustand startet, wie es beendet wurde. In ihr wird unter anderem der Ordner des aktuellen Workspaces, der Name des leeren Wortes, das der User ausgewählt hat und das aktuelle CSS-Stylesheet gespeichert. Den Aufbau der *config*-Datei sieht man in Listing 4.

Listing 4: Die *Config*-Datei

```
WORKSPACE = workspace/
STYLESHEET = /blue.css
NULLSYMBOL = lambda
TOOLTIPS = true
```

Um einen zentralen Ort für die Speicherung der Objekte zu haben, wurde ein sogenannter *Workspace* eingeführt. Auch hier wurde sich wieder an einer IDE orientiert. Alle Dateien des Workspaces sind in der GUI in einer Baumstruktur dargestellt. Der Workspace hat ein Pendant auf der Festplatte. Standardmäßig ist dies der Ordner *workspace* im Verzeichnis des Programms. Auf Wunsch kann der Workspace, momentan nur in der GUI, gewechselt werden.

Damit eine problemlose Persistierung möglich ist, wird bereits in der GUI verhindert, dass ungültige Namen für Terminale und Nichtterminale eingegeben werden. Hierzu gehören bei Terminalen der Gebrauch des Hochkommas und bei Nichtterminalen alles außer Buchstaben, Zahlen und dem Unterstrich.

5.4 Kellerautomaten

Kellerautomaten (PDA) wurden als neuer Datentyp implementiert. Ein Kellerautomat besteht aus einer Menge von Zuständen (states), einem Eingabe-Alphabet (inputAlphabet), einem Stack-Alphabet (stackAlphabet), einem Startzustand (startState), einem Bottom-Symbol für den Stack (initialStackLetter), einer Menge von Produktionsregeln (rules) und dem aktuellen Zustand, in dem sich der PDA befindet (currentState). Ein PDA akzeptiert in dieser Implementierung ein Wort, wenn der Stack leer ist.
Die Darstellung der Klasse in Listing 5 ist stark verkürzt, um nur die wichtigsten Aspekte zu zeigen.

Es ist nicht nötig, dem PDA Zustände, Input- und Stack-Alphabet zu übergeben, da diese im Konstruktor bestimmt werden. Dies ist nur dank der Unmodifizierbarkeit möglich, da sich die genannten Mengen nach Erstellung des Objekts nicht mehr verändern.

Die Unmodifizierbarkeit wird auch bei der Betrachtung der Getter- und Setter-Methoden deutlich:
Aufgrund der fehlenden Setter-Methoden ist es nicht möglich, einzelne Werte zu verändern. Auch das Hinzufügen in *List* und *Set* wurde durch den Aufruf von *Collections.unmodifiableList* bzw. *Collections.unmodifiableSet* in den Getter-Methoden verhindert.

Listing 5: Die Klasse *Pushdownautomaton*

```
public class PushDownAutomaton {
    private final HashSet<State> states;
    private final HashSet<InputLetter> inputAlphabet;
    private final HashSet<StackLetter> stackAlphabet;
    private final State startState;
    private final StackLetter initialStackLetter;
    private final String name;
    private final PushDownAutomaton previousPDA;
    private final List<PDARule> rules;
    public PushDownAutomaton(State startState, StackLetter
        initialStackLetter, List<PDARule> rules, String name,
        PushDownAutomaton previousPDA) {
        this.states = new HashSet<>(rules.stream().map(PDARule::
            getComingFrom).collect(Collectors.toSet()));
        states.addAll(new HashSet<>(rules.stream().map(PDARule::
            getGoingTo).collect(Collectors.toSet())));
        states.add(startState);
        this.inputAlphabet = new HashSet<>();
        rules.stream().map(PDARule::getReadIn).forEach(inputAlphabet::
            add);
        this.stackAlphabet = new HashSet<>();
        rules.stream().map(PDARule::getOldToS).forEach(stackAlphabet::
            add);
        rules.stream().map(PDARule::getNewToS).forEach(list -> list.
            forEach(stackAlphabet::add));
        this.startState = startState;
        this.initialStackLetter = initialStackLetter;
        stackAlphabet.add(initialStackLetter);
        this.rules = rules;
        this.name = name;
        this.previousPDA = previousPDA;
    }
    public List<PDARule> getRules() {
        return Collections.unmodifiableList(new ArrayList<>(rules));
    }
    //weitere Getter
}
```

5.5 Ausführbare JAR

Es war bei der Toolbox 1.0 unter anderem wegen der Erzeugung von Objekten zur Laufzeit mit Hilfe des *URLClassLoaders*[13] nicht möglich, eine ausführbare JAR zu erstellen. Dies ist durch die Verwendung von Reflections und die im Abschnitt 4.1 erwähnten Änderungen möglich geworden.

Die Codeausschnitt 5.5 aus der Klasse *CLI* zeigt beispielsweise die Veränderung beim Laden der CLI-Plugins. Der Code ermöglicht nicht nur die Ausführung als JAR, er ist auch kürzer und übersichtlicher.

[13] https://docs.oracle.com/javase/7/docs/api/java/net/URLClassLoader.html

Listing 6: Toolbox 1.0 – Erstellen der Plugins

```
try {
 String packagePath = Thread.currentThread()
   .getContextClassLoader()
   .getResources("CLIPlugins")
   .nextElement().getFile().replace("%20", " ");
 File[] classes = new File(packagePath).listFiles();
 URLClassLoader urlClassLoader = URLClassLoader
   .newInstance(new URL[]{new URL("file://" +
         packagePath)});
 for(File file : classes) {
   if(file.getName().endsWith(".class")
      && !file.getName().equals("CLIPlugin.class")
      && !file.getName().contains("$")) {
       plugins.add((CLIPlugin) urlClassLoader
       .loadClass("CLIPlugins." + file.getName()
       .substring(0, file.getName().length() - 6))
       .newInstance());
   }
 }
} catch(Exception e) {
 e.printStackTrace();
}
```

Listing 7: Toolbox 2.0 – Erstellen der Plugins

```
Reflections refl=new Reflections("CLIPlugins");
Set<Class<? extends CLIPlugin>> s=refl.
        getSubTypesOf(CLIPlugin.class);
s.forEach(r -> {
  try {
    CLIPlugin plugin = r.newInstance();
     plugins.add(plugin);
  } catch (InstantiationException |
        IllegalAccessException e) {
    e.printStackTrace();
  }
});
```

5.6 Ausgabe

Die Toolbox 1.0 besaß eine Ausgabe auf die Konsole. Diese war starr geregelt und daher nicht erweiterbar. Da aber auch eine Ausgabe in .tex-Dateien möglich sein sollte, musste die Ausgabe überarbeitet werden. Das Ziel hierbei war, dass in Zukunft auch andere Ausgabe-Modi (html, xml) leicht hinzu programmiert werden können. Dazu wurde zunächst ein neues Packages *Print* erstellt.

5.6.1 Das Interface *Printable*

Wenn man ein Objekt drucken möchte, soll man sich keine Gedanken darüber machen müssen, in welcher Form das geschieht. Man möchte die Ausgabe nicht selber formatieren. Es wäre wünschenswert, wenn das Objekt selbst weiß, wie es zu drucken ist. Daher wurde das Interface *Printable* eingeführt. Dieses stellt zwei Methoden zur Verfügung: *printConsole*, welches für die Ausgabe auf die Konsole zuständig ist, und *printLatex*, welches für die LaTeX-konforme Ausgabe sorgt.

Das Interface *Storable* erweitert *Printable*, da alle speicherbaren Objekte auch druckbar seien sollen.

5.6.2 Die Klasse *Printer*

Die *Printer*-Klasse ist für die Ausgabe von Objekten und Umformungen zuständig und stellt einen wichtigen Teil des Programmes dar. In diesem Abschnitt werden kurz die wichtigsten Funktionen erläutert.

Listing 8: Die Klasse *Printer*

```
public class Printer {
    private static PrintMode printmode=PrintMode.CONSOLE;
    private static BufferedWriter writer=new BufferedWriter(new
        OutputStreamWriter(System.out));
    public static void print(Printable printable) {
        switch (printmode) {
            case NO:
                break;
            case LATEX:
                printable.printLatex(getSpace(deepness));
                break;
            case CONSOLE:
                printable.printConsole();
                break;
        }
    }
    public static void printEnumeration(ArrayList<Printable> printables
        , String[] pointdescriptions, String[] texts, String title) {
        switch(printmode) {
            case NO:
                break;
            case CONSOLE:
                printEnumerationConsole(printables,pointdescriptions,
                    texts);
                break;
            case LATEX:
                printEnumerationLatex(printables,toLatex(
                    pointdescriptions),toLatex(texts),toLatex(title));
                break;
        }
    }
   //...
}
```

Listing 8 stellt die zentralen Komponenten der Klasse dar. Die Klasse *Printer* besitzt verschiedene Printmodi, die angeben, wie und wohin geschrieben wird. Dazu gibt es einen *BufferedWriter*, der passend initialisiert wird. Idealerweise sollte der Writer auf die Konsole schreiben, wenn der Printmodus *CONSOLE* ist und in eine Datei, wenn der Printmodus *LATEX* ist. Dies ist jedoch nicht festgelegt, sodass beispielsweise die Möglichkeit besteht, Konsolenausgabe in eine Datei zu schreiben.

Die Methode *print(Printable printable)* sollte zum Schreiben von *printable* Objekten genutzt werden. Anhand des Printmodus wird entschieden, welche der Methoden *printConsole()* und *printLatex(String space)* des Printables aufgerufen werden soll. Diese Art der Ausgabe ist vor dem direkten Aufruf von *printConsole()* und *printLatex(String space)* zu bevorzugen, da so die Verantwortung des richtigen Druckens an den Printer abgegeben wird.

Eine weitere Methode ist *printEnumeration*. Diese gibt eine Aufzählung aus. Auch hier gibt es wieder eine Version für die Konsole und eine für Latex. Es soll eine Aufzählung geschrieben werden, deren einzelne Punkte die Überschriften in *pointdescriptions* haben.

Dann folgt die Ausgabe eines beschreibenden Textes aus dem dritten Argument *texts* und die Ausgabe eines Elementes aus dem ersten Argument *printables*. Der Titel der Aufzählung ist *title*; der Titel wird hierbei als *Section* angegeben.

5.7 Generierung von LaTeX-Code

Ein weitere Funktion der Toolbox ist das Generieren von LaTeX-Code, welcher die vom Nutzer durchgeführten Schritte darstellt.

Wichtig hierbei war, darauf zu achten, dass LaTeX-Sonderzeichen erkannt und angepasst werden, damit die entstandene LaTeX-Datei kompilierbar ist.

Die LaTeX-Generierung ist nicht automatisch aktiviert. Der Nutzer muss den *LaTeX*-Modus starten. Durch Starten des LaTeX- Modus wird die Präambel, in Listing 9 dargestellt, geschrieben.

Listing 9: Die Präambel der erstellten LaTeX-Dateien

```
%this document was generated by the STUPS Toolbox 2.0
\documentclass{article}
\usepackage{amssymb}
\usepackage{amsmath,amsthm}
\usepackage[ngerman,english]{babel}
\usepackage{tikz}
\usetikzlibrary{automata,positioning}

\begin{document}
```

Die Datei muss nun noch beendet werden. Dies geschieht wieder nicht automatisch nach Beendigung eines Befehls, sondern es wurde sich bewusst dafür entschieden, dass der Nutzer den LaTeX-Modus manuell beenden muss. Hierdurch ist es möglich, mehrere Algorithmen oder Objekte in eine Datei zu schreiben.

Erst das Beenden des LaTeX-Modus beendet die Datei und schließt den BufferedWriter.

Ein Herausforderung stellte die Ausgabe von Automaten dar. Die Automaten sollten möglichst automatisch so dargestellt werden, dass sie gut und übersichtlich aussehen und nicht mehr viel nachträgliche Verbesserung benötigen.

Zur Darstellung der Automaten wird das Paket TikZ[14] genutzt. Die Zustände sind Knoten und die Übergänge beschriftete Kanten. TikZ erwartet zuerst eine Aufzählung aller Knoten. Die Positionen werden relativ zueinander angegeben. Um ein gutes Ergebnis zu erhalten, werden die Zustände von der Toolbox alphanumerisch sortiert. Das ist häufig eine sinnvolle Reihenfolge, da der Startzustand meist mit $z0'$ bezeichnet wird, und die nachfolgenden Zustände aufsteigend nummeriert werden.

Dann müssen die Kanten angegeben werden. Hierbei gibt es für Kanten von einem Knoten zu sich selbst, die Option *loop below* oder *loop above*. Es wurde sich für *loop below* entschieden. Bei Kanten, die von einem Zustand zu einem anderen Zustand gehen, muss

[14] https://www.ctan.org/pkg/pgf

eine Richtung, in die sich die Kante beugt, und ein Winkel angegeben werden. Als Richtung wurde *bend left* gewählt. Dies bewirkt, dass "Vorwärtskanten", also Kanten, die zu einem Knoten weiter hinten in der Aufzählung gehen, über den Knoten sind und "Rückwärtskanten" unterhalb. Die Winkel werden mit Hilfe der Länge der Kante $length$ und der maximalen Länge $maxLength$ berechnet. Hierbei gibt es zwei Fälle:

1. $maxLength = 1$. Der Winkel jeder Kante wird auf $45°$ gesetzt
2. $maxLength > 1$. Der Winkel einer Kante wird auf

   $$length * 90/maxLength°$$

   gesetzt. Dies bewirkt eine gleichmäßige Fächerung auf dem Winkelintervall $[90/maxLength, 90]$.

```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on
      grid,auto]

  \node[state,initial]   (z0)    {z0};
  \node[state]   (z1)  [right of=z0]    {z1};
  \node[state]   (z2)  [right of=z1]    {z2};
  \node[state,accepting]   (z3)  [right of=z2]    {z3};
\path[->]
(z0)    edge [bend left=60] node {b}  (z2)
        edge [bend left=90] node {b}  (z3)
        edge [bend left=30] node {a}  (z1)
(z1)    edge [bend left=30] node {b}  (z2)
        edge [bend left=60] node {c}  (z3)
(z2)    edge [bend left=30] node {b}  (z3)
        edge [bend left=60] node {b}  (z0)
(z3)    edge [loop below]   node {d}  (z3)
        edge [bend left=60] node {a}  (z1)
;
\end{tikzpicture}
```

Abbildung 3: LaTeX-Code für einen Automaten

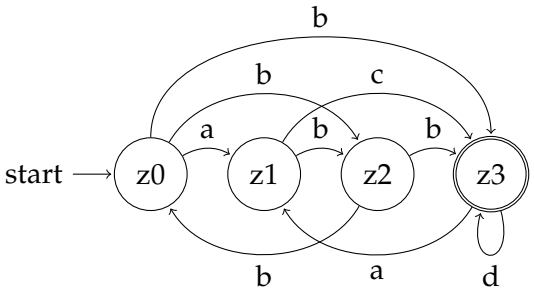

Abbildung 4: Erstellter Automat

5.8 Testen

Das Testen beanspruchte einen großen Teil der Arbeitszeit, da das Testen größtenteils manuell durchgeführt werden musste. Dies hatte mehrere Gründe. Zum einen ist es schwer bis gar nicht möglich, rechnerisch zu überprüfen, ob zwei Grammatiken äquivalent sind. Die Umformungen, wie z.B. Entfernen von einfachen Regeln, Entfernen von λ-Regeln, mussten manuell auf ihre Korrektheit bzgl. der Äquivalenz überprüft werden. Zum anderen erwies sich der LaTeX-Code als große Fehlerquelle. Es mussten alle Eventualitäten bedacht werden, so dass keine Zeichen geschrieben werden, mit denen LaTeXnicht umgehen kann. Das reine Überprüfen des Kompilieren reichte nicht aus, es musste weiterhin auf die korrekte und ästhetische Schreibweise geachtet werden.

Jedoch wurden, soweit wie möglich, JUnit-Tests verwendet. Methoden, die eine Grammatik auf gewisse Eigenschaften, wie keine einfache Regeln oder λ - Freiheit, überprüfen, konnten getestet werden. Ebenso, ob Algorithmen, die diese Eigenschaften herbei führen sollen, dies auch tatsächlich tun.

Auch das Verhalten von *equal* und *toHash* wurde ausführlich per JUnit getestet.

6 Theoretische Aspekte der Algorithmen

Beim Umsetzen der Algorithmen in echten Code kam es zu einigen Schwierigkeiten, da die Algorithmen aus didaktischen Gründen so geschrieben sind, dass sie für Menschen leicht umsetzbar sind, jedoch nicht so einfach für Computer. Ein prägnantes Beispiel hierfür ist:
Finde alle Kreise und entferne sie.
Für einen Menschen sind die Kreise in einem Graphen leicht ersichtlich, ein Computer muss erst eine Breitensuche laufen lassen und Rückwärtskanten ermitteln.

Alle Beispiele in den folgenden Abschnitten wurden durch die Toolbox generiert und mit leichten Modifizierungen übernommen.

Die Algorithmen sind, wenn nicht anders angegeben, dem Skript [Rot16] zur Vorlesung *Einführung in die theoretische Informatik* an der Heinrich-Heine-Universität entnommen.

Da angedacht ist, die Toolbox in der Lehre zu verwenden, wurde darauf geachtet, die tatsächlich Umsetzung dicht an den Algorithmen im Skript zu belassen und die Durchführung nachvollziehbar zu gestalten.

6.1 Kellerautomaten

6.1.1 Überprüfen eines Strings

Durch das *ComplexFunctionPlugin CheckStringPDAPlugin* ist es möglich, zu überprüfen, ob ein PDA ein Wort akzeptiert. Hierfür wurden die Produktionsregeln in der GUI als Buttons dargestellt. Nach Start des Plugins ist es möglich, mit diesen Buttons einen Durchlauf zu simulieren. Es sind immer nur die Regeln auswählbar, die auch mit dem aktuellen Input und dem *Top of Stack* kompatibel sind. Intern wurde der Durchlauf durch eine weitere Klasse *RunThroughInfo* ermöglicht. Diese ist eine Konfiguration eines Durchlaufs. Der PushDownAutomat hat als einzige veränderbare Variable den aktuellen Zustand in dem er sich befindet. In der *RunThroughInfo* wird der Stack, der einzulesende Input, der aktuelle Zustand und die vorherige *RunThroughInfo* gespeichert. Wird eine Regel vom Nutzer ausgewählt, wird diese auf die aktuelle *RunThroughInfo* angewandt und dadurch entsteht eine neue Konfiguration, welche die alte Konfiguration als Vorgänger hat.

6.1.2 Umwandeln in eine Grammatik

Es ist möglich, einen Kellerautomaten in eine äquivalente Grammatik umzuformen und umgekehrt (6.2.7). Dafür muss der Kellerautomat zunächst in einen äquivalenten Kellerautomaten, der nur Regeln der Form $z_i a_i A_i \to z_j B_{i_1} B_{i_2}$ hat, umgewandelt werden. Eine Möglichkeit, dies zu tun, ist in Algorithmus 1 angegeben.

Nun kann Algorithmus 6.1.2 angewandt werden. Hierbei ist zu beachten, dass z' und z'' *beliebige* Zustände aus Z sind, d. h. es müssen alle möglichen Kombinationen durchlaufen werden. Es sei außerdem erwähnt, dass bei dem Algorithmus Nichtterminale entstehen,

Algorithm 1 Umwandeln eines PDAs in einen PDA mit kurzen Regeln

Input: PDA P mit Regeln $zaA \to z'B_1B_2 \dots B_k$ mit $k > 2$
Output: PDA ohne Regeln $zaA \to z'B_1B_2 \dots B_k$ mit $k > 2$
for all Regel $zaA \to z'B_1B_2 \dots B_k$ mit $k > 2$ **do**
Wähle neue Zustände $z_1, z_2, \dots z_{k-2}$
Ersetze diese Regel durch die folgenden neuen δ-Regeln:

$$\begin{aligned} zaA &\to z_1B_{k-1}B_k \\ z_1\lambda B_{k-1} &\to z_2B_{k-2}B_{k-1} \\ &\vdots \\ z_{k-2}\lambda B_2 &\to z'B_1B_2 \end{aligned}$$

end for
return P

die nie erreicht werden. Das geht aus dem Algorithmus nicht klar hervor. Die Toolbox entfernt diese Zustände nach Durchführung.

Des Weiteren sei bemerkt, dass die im Algorithmus 6.1.2 verwendeten Klammern in den Nichtterminalen weggelassen worden sind, da der Parser diese nicht in Nichtterminalen erlaubt. In Terminalen sind Sonderzeichen wie +, -, &, #, usw. erlaubt, in Nichtterminalen allerdings nicht. Da der Algorithmus aber aus Terminalen Nichtterminale macht, mussten diese Sonderzeichen eliminiert werden. Dies geschieht, indem jeder einzelne Char eines Nichtterminales überprüft wird, und wenn dieser nicht erlaubt ist, wird er zu einem Integer gecastet. Zahlen in Nichtterminalen sind erlaubt. Anstatt des nicht erlaubten Chars wird dann dieser korrespondierende Integerwert geschrieben.

Der Beweis der Äquivalenz wird hier nicht erneut geführt, da er sich bereits in [Rot16, S.75-77] befindet.

Algorithm 2 PDA in Grammatik umwandeln

Input: PDA $M = (\Sigma, \Gamma, Z, \delta, z_0, \#)$ mit $k \leq 2$ für alle Regeln $zaA \to z'B_1B_2 \dots B_k$
Output: Eine äquivalente Grammatik $G = (\Sigma, \{S\} \cup Z \times \Gamma \times Z, S, P)$
$G = (\Sigma, \{S\} \cup Z \times \Gamma \times Z, S, P)$
P besteht dabei aus folgenden Regeln

$$\begin{aligned} &1.S \to (z_0, \#, z) \text{ für jedes } z \in Z \\ &2.(z, A, z') \to a, \text{ falls } (z', \lambda) \in \delta(z, a, A) \\ &3.(z, A, z') \to a(z_1, B, z'), \text{ falls } (z_1, B) \in \delta(z, a, A) \\ &4.(z, A, z') \to a(z_1, B, z'')(z'', C, z'), \text{ falls } (z_1, BC) \in \delta(z, a, A) \end{aligned}$$

Hierbei sind $z, z_1 \in Z, A, B, C \in \Gamma, a \in \Sigma \cup \{\lambda\}$ und $z', z'' \in Z$ beliebig.
return G

6.2 Algorithmen für Grammatiken

6.2.1 Entfernen von einfachen Regeln

Zu jeder Grammatik mit einfachen Regeln (unit Rules, $A \to B$ mit $A, B \in N$) gibt es eine Grammatik ohne einfache Regeln, die dieselbe Sprache erzeugt.

1. Entferne alle Zyklen und ersetze die Nichtterminale eines Zykels durch dasselbe Symbol
2. Nummeriere die Nichtterminale so, dass aus $A_i \to A_j$ $i < j$ folgt
3. Für $k = n-1, n-2, \ldots, 1$ (rückwärts) eliminiere die Regel $A_k \to A_l$ mit $l < k$ so: Füge jede Regel, die A_l als linke Seite hat, als neue Regel mit A_k als linke Seite zu P hinzu.

Das Finden der Zykel wurde durch eine Tiefensuche implementiert. Die einfachen Regeln werden als gerichteter Graph betrachtet. Jedes Nichtterminal ist ein Knoten, eine Kante zwischen zwei Knoten A und B besteht genau dann, wenn die Grammatik die Regel $A \to B$ enthält.

Nach dem Durchführen der Tiefensuche können Rückwärtskanten identifiziert werden. Nun wird eine Rückwärtskante entfernt, indem die beiden Knoten zu einem Knoten verschmelzen, und es wird erneut eine Tiefensuche durchgeführt. Dies wird solange gemacht, bis es keine Rückwärtskanten mehr gibt. Dann gibt es auch keine Zykel mehr.

Beispiel – Entfernen einfacher Regeln

Before

$G = (\{0,1\},\ \{S, D, B, A, E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to D \mid BD \mid ABD,\\ &D \to 0,\\ &B \to A,\\ &A \to B \mid ED \mid BB,\\ &E \to 1\} \end{aligned}$$

Step 1

Remove circles.

$G = (\{0,1\},\ \{S, D, A, E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to D \mid AD \mid AAD,\\ &D \to 0,\\ &A \to ED \mid AA,\\ &E \to 1\} \end{aligned}$$

Step 2
Number the nonterminals.
D=2 S=1

Step 3
Remove unit rules beginning by the highest number.
$G = (\{0, 1\},\ \{S, A, D, E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to AD \mid 0 \mid AAD, \\ &A \to ED \mid AA, \\ &D \to 0, \\ &E \to 1\} \end{aligned}$$

6.2.2 Entfernen von λ-Regeln

Zu jeder Grammatik mit λ-Regeln gibt es eine Grammatik ohne λ-Regeln, die dieselbe Sprache erzeugt. Hierbei akzeptieren wir Regeln $S \to \lambda$, damit $\lambda \in L(G)$ möglich ist.

1. Bestimme zuerst die Menge aller Nichtterminale, die auf das leere Wort abgeleitet werden können
$$N_\lambda = \{A \in N \mid A \vdash_G^* \lambda\}$$
2. Füge für jede Regel der Form
$$B \to uAv \text{ mit } B \in N, A \in N_\lambda \text{ und } uv \in (N \cup \lambda)^+$$
zusätzlich die Regel $B \to uv$ zu P hinzu.
3. Entferne alle Regeln $A \to \lambda$ aus P.

Regel 2 muss dabei auch für alle neu entstandenen Regeln angewandt werden.

Dies wurde im Hintergrund mit einer Queue[15] umgesetzt. Zuerst werden alle Regeln, die ein Symbol enthalten, welches sich auf das leere Wort ableiten lässt (im Folgenden *Nullsymbol* genannt), zu der Queue hinzugefügt. Nun wird immer eine Regel vom Anfang der Queue entfernt und ein Nullsymbol gelöscht, so dass eine Regel entsteht, die noch nicht betrachtet wurde. Die neue Regel wird am Ende der Queue eingefügt. Zusätzlich wird auch die ursprüngliche Regel wieder hinzugefügt, falls sie weitere Nullsymbole enthält. Sobald keine Änderungen mehr auftreten bzw. die Queue leer ist, stoppt der Algorithmus.

6.2.3 Chomsky-Normal-Form

Eine Grammatik ist in Chomsky-Normalform, falls jede Regeln eine der folgenden Formen hat:

[15] `https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html`

- $A \to BC$ mit $A, B, C \in N$
- $A \to a$ mit $A \in N, a \in \Sigma$

Zu jeder Grammatik gibt es eine äquivalente Grammatik in Chomsky-Normal-Form.

1. Regeln $A \to a$ mit $A \in N$ und $a \in \Sigma$ sind in CNF und werden übernommen. Alle anderen Regeln sind von der Form: $A \to x$ mit $x \in (N \cup \Sigma)^*$ und $\mid x \mid \geq 2$.
2. Füge für jedes $a \in \Sigma$ ein neues Nichtterminal X_a zu N hinzu, ersetze jedes Vorkommen von $a \in \Sigma$ durch X_a und füge zu P die Regel $X_a \to a$ hinzu.
3. Nicht in CNF sind nun nur noch Regeln der Form

   $$A \to B_1 B_2 \dots B_k,$$

   wobei $k \geq 3$ und jedes B_i ein Nichtterminal ist: Jede solche Regel wird ersetzt durch die Regeln

   $$\begin{aligned} A &\to B_1 C_2, \\ C2 &\to B_2 C_3, \\ &\dots \\ C_{k-2} &\to B_{k-2} C_{k-1}, \\ C_{k-1} &\to B_{k-1} B_k, \end{aligned}$$

   wobei $C_2, C_3, \dots, C_{k-1}$ neue Nichtterminale sind.

Bemerke, dass auch hier wieder im zweiten Schritt ungültige Nichtterminalsymbole entstehen können. Sollte dies passieren, wird stattdessen ein unbesetzter Großbuchstabe als Nichtterminal gewählt.

Beispiel – Chomsky Normal Form

Before

$G = (\{0, 1\},\ \{S, A, D, E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to AD \mid 0 \mid AAD, \\ &A \to ED \mid AA, \\ &D \to 0, \\ &E \to 1\} \end{aligned}$$

Step 1

Rules in form A → a are already in chomsky normal form and we keep them.

Step 2

In every other rule replace every appearance of a terminal a with a new nonterminal.
$G = (\{0,1\},\ \{S, A, D, E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to AD \mid 0 \mid AAD, \\ &A \to ED \mid AA, \\ &D \to 0, \\ &E \to 1\} \end{aligned}$$

Step 3

In every rule that contain more than two nonterminals, add a new nonterminal that points to the end of the rule.
$G = (\{0,1\},\ \{S, A, P_0, D, E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to AP_0 \mid AD \mid 0, \\ &A \to ED \mid AA, \\ &P_0 \to AD, \\ &D \to 0, \\ &E \to 1\} \end{aligned}$$

6.2.4 CYK-Algorithmus

Der Cocke-Younger-Kasami-Algorithmus (CYK) löst das Wortproblem für kontextfreie Sprachen in kubischer Zeit.

Algorithm 3 CYK-Algorithmus

Input: $G = (\Sigma, N, S, P)$ kfG in CNF, $x = a_1 a_2 \dots a_n \in \Sigma^*$
Output: eine Tabelle T, deren Zellen Mengen von Nichtterminalen sind.

```
for i = 1 to n do
    T(i, 0) = {A ∈ N | A → a_i ist Regel in P}
end for
for j = 1 to n − 1 do
    for i = 1 to n − j do
        T(i, j) = ∅
        for k = 0 to j − 1 do
            T(i, j) = T(i, j) ∪ {A ∈ N | es gibt eine Regel A → BC in P und B ∈
            T(i, k) und C ∈ T(i + k + 1, j − k − 1)}
        end for
    end for
end for
return T
```

Dieser Algorithmus wurde genau so in der Klasse *GrammarUtils* umgesetzt. Dafür wurde eine neue Klasse *Matrix* geschrieben, die eben diese Tabelle T darstellt.

Beispiel – CYK-Algorithmus
$test6.gr = (\{0,1\},\ \{S,A,B,D,E\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to AB \mid AD \mid 0,\\ &A \to ED \mid AA,\\ &B \to AD,\\ &D \to 0,\\ &E \to 1\} \end{aligned}$$

Tabelle 1: CYK

B, S				
A				
		B, S		
A		A		
E	S, D	E	S, D	S, D
1	0	1	0	0

$$\begin{aligned} S &\vdash AB\\ &\vdash EDB\\ &\vdash 1DB\\ &\vdash 10B\\ &\vdash 10AD\\ &\vdash 10EDD\\ &\vdash 101DD\\ &\vdash 1010D\\ &\vdash 10100 \end{aligned}$$

6.2.5 Finden eines Pfades

Es wurde die Möglichkeit implementiert, unter bestimmten Umständen einen Pfad zu finden.

Zum einen liefert der CYK-Algorithmus indirekt einen Pfad zu einem Wort in der Sprache, indem zurückverfolgt wird, warum das Startsymbol in der oberen linken Ecke steht. Bei einer kontextfreien Grammatik kann also mit geringem Rechenaufwand ein Pfad gefunden werden.

Zum anderen wurde eine Breitensuche implementiert. Eine Regel ist das Kind einer anderen Regel, wenn sie durch Ableiten des ersten Nichtterminals entsteht. Durch die Breitensuche wird gewährleistet, dass man nicht in einem unendlichen Pfad landet, der nie auf das gesuchte Wort führt. Die Breitensuche wird nur durchgeführt, wenn vorher mit

dem CYK-Algorithmus gewährleistet werden konnte, dass das Wort in der Sprache ist und somit tatsächlich ein Pfad existiert.

Allerdings dauert die Breitensuche bei komplizierten Grammatiken zu lange. Daher wurde eine obere Grenze eingeführt, die der Benutzer selber bestimmt. Sobald diese Anzahl an Regeln betracht wurde, bricht der Algorithmus ab und liefert kein Ergebnis.

Beispiel

Grammar $G = (\{(,), *, +, a\},\ \{S\},\ S,\ R)$ with

$$R = \{S \to (S) \mid S * S \mid S + S \mid a\}$$

Path

$$\begin{aligned} S &\vdash S * S \\ &\vdash a * S \\ &\vdash a * (S) \\ &\vdash a * (S + S) \\ &\vdash a * (a + S) \\ &\vdash a * (a + a) \end{aligned}$$

6.2.6 Sonderregel für das leere Wort

Da verkürzende Regeln nicht erlaubt sind, ist $\lambda \notin L(G)$. Dies ist jedoch nicht immer wünschenswert, daher ist die Regel $S \to \lambda$ erlaubt, jedoch darf S dann auf keiner rechten Seite einer Regel vorkommen. Im Skript wurde nur der Fall erklärt, wie man $\lambda \in L(G)$ erreicht, ohne die oben beschriebene Bedingung zu verletzen. Im Algorithmus für das Entfernen von Lambda-Regeln stand jedoch, dass man die Sonderregel anwenden soll, wenn $\lambda \in L(G)$ gilt. Dies erscheint widersprüchlich. Daher wurde der Algorithmus leicht modifiziert und wird *vor* dem Entfernen der λ-Regeln durchgeführt, falls $\lambda \in L(G)$ gilt.

1. Füge ein neues Nichtterminalsymbol S_0 hinzu
2. Ersetze jedes Vorkommen von S in einer rechten Seite einer Regel durch S_0.
3. Füge Regel von S zu S_0 hinzu

Nun kann der Algorithmus zum Entfernen von λ-Regeln problemlos und korrekt durchgeführt werden.

Beispiel - Sonderregel für das leere Wort

Before

$H = (\{b, a\},\ \{S\},\ S,\ R)$ with

$$R = \{S \to bSb \mid \lambda \mid aSa\}$$

Special Rule for Empty Word

Add a nonterminal according to the special rule for empty word.
$H = (\{a, b\},\ \{S, S_0\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to \lambda \mid aS_0a \mid bS_0b, \\ &S_0 \to \lambda \mid aS_0a \mid bS_0b\} \end{aligned}$$

Step 1

Calculate the nullable set.
$\{S, S_0\}$

Step 2

For every rule that contains a nullable nonterminal, add that rule without this nonterminal.
$H = (\{a, b\},\ \{S, S_0\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to aa \mid bb \mid \lambda \mid aS_0a \mid bS_0b, \\ &S_0 \to \lambda \mid aS_0a \mid bb \mid aa \mid bS_0b\} \end{aligned}$$

Step 3

Remove lambda-rules.
$H = (\{a, b\},\ \{S, S_0\},\ S,\ R)$ with

$$\begin{aligned} R = \{&S \to aa \mid bb \mid \lambda \mid aS_0a \mid bS_0b, \\ &S_0 \to aS_0a \mid bb \mid aa \mid bS_0b\} \end{aligned}$$

6.2.7 Umwandeln zu PDA

Eine kontextfreie Grammatik kann anhand des Algorithmus 4 zu einem äquivalenten Kellerautomaten umgeformt werden.

Algorithm 4 Grammatik in PDA umwandeln

Input: kontextfreie Grammatik $G = (\Sigma, N, S, P)$
Output: Kellerautomat $M = (\Sigma, N \cup \Sigma, \{z\}, \delta, z, S)$
Definiere δ wie folgt:

- Ist $A \to q$ eine Regel in P mit $A \in N$ und $q \in (N \cup \Sigma)^*$, so sei $(z, q) \in \delta(z, \lambda, A)$.
- Für jedes $a \in \Sigma$ sei $(z, \lambda) \in \delta(z, a, a)$

return M

7 Fazit und Ausblick

7.1 Herausforderungen

Die Weiterentwicklung eines bereits bestehenden Programmes unterscheidet sich stark von der eigenen Entwicklung eines komplett neuen Programmes. Obwohl der Aufbau der Arbeit modular ist, war es trotzdem nötig, sich in den fremden Code einzuarbeiten. Besonders die Umstrukturierung der GUI machte dies notwendig. Trotz der guten Vorarbeit und umfangreichen Dokumentation, war der fremde Code nicht immer leicht zu verstehen. Vor allem benötigt das Einarbeiten in Legacy-Code viel Zeit. Auch musste man mit Designentscheidungen arbeiten, die man so selber nicht getroffen hätte.

Der Arbeitsaufwand, welcher neben dem eigentlichen Programmieren anfällt, wurde zunächst unterschätzt und war unerwartet hoch. Die Umstellung auf Gradle erzwang eine intensive Auseinandersetzung mit gängigen Programmierpraktiken, dazu gehörte insbesondere die Trennung von Source-Code und Ressourcen. Auch ist es bei so einem großen Projekt wichtig, den Überblick zu bewahren. Hierzu gehörte es, eine größere Kapselung zu erreichen, d.h. jeder Klasse nur eine Aufgabe zuzuweisen. Diese war vorher nicht in dem Maße gegeben, wie sie jetzt vorhanden ist. Als Beispiel sei hierfür die Klasse *CLI* genannt, die vorher auch die Daten enthielt und verwaltete, was jetzt von der Klasse *Content* übernommen wird.

7.2 Ausblick

Auf Grund der begrenzten Zeit konnten leider nicht alle wünschenswerten Features umgesetzt werden. Da die Toolbox weiterhin modular und leicht erweiterbar ist, besteht eine Fülle von Funktionen, die noch implementiert werden können. Die hervorgehobenen Punkte werden als besonders wichtig erarchtet.

- Turing-Maschinen
  - Speichern und Laden
  - **Visualisierung**
  - **animierter Durchlauf**
  - Algorithmen
  - Komplexitätsberechnung
- Grammatiken
  - Kontextsensitive Grammatiken (erfordert eine Umstrukturierung der Klassen *Rule* und *Grammat* und eine Anpassung des Parsers)
  - reguläre Ausdrücke zu Grammatik umwandeln
- Sonstiges
  - Automaten unmodifizierbar machen
  - für die Algorithmen von Automaten LaTeX-Ausgabe erzeugen

- reine GUI-Funktionen für die Konsole umsetzen
- **Programm mit Parametern aufrufen**, so dass die übergebenen Kommandos direkt ausgeführt werden können. Ermöglicht automatisiertes Ausführen.
- Notizen zu Objekten anlegen
- Autovervollständigung von Befehlen
- LaTeX-Generierung: Einstellung, ob *Section*, *Subsection* oder *Paragraph*

7.3 Fazit

Diese Arbeit setzt die von Fabian Ruhland nahtlos fort. Das Ziel war es, die Toolbox komfortabler in der Benutzung zu machen und den Funktionsumfang bezüglich Grammatiken zu erweitern. Dies ist gelungen.

Weiterhin gilt jedoch, dass die Toolbox konstruktiv beim Lernen von Grammatiken und Automaten angewendet werden muss. Die LaTeX-Funktion und die schrittweise Durchführung der Algorithmen ermöglichen es, ein größeres Verständnis für diese zu erlangen. Auch das Berechnen eines Pfades einer Grammatik ist hilfreich. Auf Grund der recht umfangreichen Features lässt sie sich leicht missbrauchen; diese Toolbox liefert, bis auf Turing-Maschinen, beinahe alles, was in der Vorlesung *Theoretische Informatik* im Bereich der formalen Sprachen gelehrt wird.

Literatur

[Cho59] CHOMSKY, Noam: On certain formal properties of grammars. In: *Information and Control* 2 (1959), Nr. 2, 137 - 167. http://dx.doi.org/http://dx.doi.org/10.1016/S0019-9958(59)90362-6. – DOI http://dx.doi.org/10.1016/S0019–9958(59)90362–6. – ISSN 0019–9958

[Rot16] ROTHE, Jörg: *Theoretische Informatik: Formale Sprachen und Automaten, Berechenbarkeit und NP-Vollständigkeit*. Skript, 2016. – Online erhältlich unter http://ccc.cs.uni-duesseldorf.de/~rothe/INFO4/main.pdf; abgerufen am 12. März 2017.

[Ruh16] RUHLAND, Fabian: *Eine Toolbox zur Simulation und Visualisierung von Automaten und Grammatiken*, Heinrich-Heine Universiät Düsseldorf, Bachelorarbeit, 2016. – Online erhältlich unter https://github.com/kasebr0t/Bachelorarbeit_STUPS-Toolbox; abgerufen am 21.Dezember 2016.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Algorithmenverzeichnis