Fakultät Elektrotechnik/Informatik
Bereich Informatik

Bachelorarbeit

Entwicklung einer Zwischensprache FLaAL mit LaTeX-Makros zur Erzeugung typografisch anspruchsvoller Publikationen am Beispiel von FLACI

vorgelegt von: Kappa, Jonas
Matrikelnummer 212927
Arndtstr. 25
03044 Cottbus

betreut durch: Prof. Dr. rer. nat. Christian Wagenknecht
Dr. Michael Hielscher

24. November 2019

Inhaltsverzeichnis

Abkürzungsverzeichnis

DEA deterministischer endlicher Automat

DKA deterministischer Kellerautomat

FA finite automaton

FFC FLACI to FLaAL Compiler

FLaAL Formal Languages and Automata LaTeX

FLACI Formale Sprachen, abstrakte Automaten, Compiler und Interpreter

JSON Javascript Object Notation

NEA nichtdeterministischer endlicher Automat

NKA nichtdeterministischer Kellerautomat

PA pushdown automaton

PGF portable grafics format

SVG Scalable Vector Graphics

TM Turing-Maschine

YAML YAML Ain't Markup Language

1 Einleitung

An vielen Universitäten und Hochschulen wird LaTeX genutzt, um wissenschaftliche Arbeiten zu publizieren und Unterrichtsmaterialien zu erstellen. Darunter sind auch die vielfältigen Darstellungsobjekte der theoretischen Informatik, wie z.B. die abstrakten Automaten und formalen Sprachen. Da LaTeX für einige Darstellungen jedoch wenig Unterstützung bietet, ist es für die Autoren sehr mühsam, diese Darstellungen zu erstellen.

Aufgabe der vorliegenden Arbeit ist es, ein Paket zu entwickeln, welches die Erstellung solcher Graphen bestmöglich abstrahiert, um typografisch anspruchsvolle Publikationen zu erzeugen.

In dieser Arbeit wird zuerst die Motivation der Aufgabenstellung beschrieben. Dann werden Grundlagen zur Erstellung von LaTeX-Makros gegeben. Darauf folgt der Hauptteil der Arbeit, welcher sich mit der Entwicklung des FLaAL-Paketes und des FF-Compilers beschäftigt. An einigen ausgewählten Beispielen wird die Leistungsfähigkeit von FLaAL gezeigt. Abschließend wird eine Zusammenfassung und ein Ausblick auf mögliche weitere Schritte gegeben.

Es wird davon ausgegangen, dass der Leser mit den Grundzügen von LaTeX und den Grundlagen der abstrakten Automaten aus dem Bereich der theoretischen Informatik vertraut ist.

2 Motivation

2.1 LaTeX

Standard für die meisten Wissenschaftler ist die Sprache TeX, auf welche LaTeX aufbaut. Auch viele Herausgeber von wissenschaftlichen Arbeiten sind darauf eingestellt, mit diesem Werkzeug zu arbeiten ([CT], Kap. 10). So gehört auch an der Hochschule Zittau/Görlitz LaTeX zu einem festen Bestandteil im Studiengang Informatik.

Im Gegensatz zu anderen Programmen, wie Word oder OpenOffice, verfolgt LaTeX nicht den Ansatz *What-you-see-is-what-you-get* (WYSIWYG, [di07]), zu deutsch: Was du siehst bekommst du. Wenn man in LaTeX etwas schreibt, muss der Code erst übersetzt werden, bevor man die Änderungen im Dokument sieht.

Da LaTeX seinen Code als lesbaren Text speichert, kann man Versionskontrollsysteme, wie Git, nutzen, um das jeweilige Projekt zu verwalten.

LaTeX besitzt viele Befehle, um Publikationen zu strukturieren (wie *section* und *subsection*) und das Aussehen dieser anzupassen (wie *textit* und *textbf*). Doch der Befehlssatz von LaTeX hat seine Grenzen. Um diese aufzubrechen, bietet LaTeX die Möglichkeit, eigene Befehle (Makros) zu implementieren, um so neue Funktionen hinzuzufügen. Diese können als Pakete gebunden und in ein LaTeX-Dokument geladen werden. In diesem können die Befehle des Pakets dann wie normale LaTeX-Befehle genutzt werden. Beispiele dafür sind das `xspace`-, das `color`- und das `ifthen`-Paket. Alle diese liefern Befehle, welche man ohne Einbindung dieser Pakete so nicht nutzen kann.

2.2 Aufgabenstellung

Ziel dieser Arbeit ist, die Modelle der theoretischen Informatik, z.B. die abstrakten Automaten, in LaTeX grafisch darzustellen.

Im Falle eines abstrakten Automaten bedeutet dies, dass ein gerichteter Graph erzeugt werden soll, wie in Abbildung 1 beispielhaft dargestellt. Die Definitionen der unterschiedlichen abstrakten Automaten sind im Anhang A.1, S. 38 zu finden.

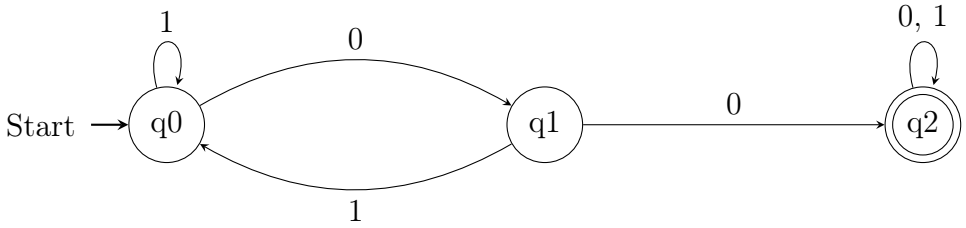

Abbildung 1: Endlicher Automat

Das Problem der Darstellung von Modellen der theoretischen Informatik in wissenschaftlichen Arbeiten ist nicht neu. Es gibt bereits Möglichkeiten, diese Graphen in LaTeX darzustellen, z.B. indem man eine Bildbearbeitungssoftware oder vektorbasierte Grafikwerkzeuge zur Hilfe nimmt, mit welchen man Bilder erzeugen kann (z.B. Photoshop). Nachdem man diese Bilder erstellt hat, kann man sie in das Dokument einfügen. Nachteilig ist jedoch, dass Programme wie Photoshop keine gute Unterstützung bieten, um abstrakte Automaten zu erstellen. Außerdem ist es nach Erstellung des Bildes nur schwer möglich, dieses innerhalb von LaTeX wieder zu ändern.

Des Weiteren gibt es die Möglichkeit, Werkzeuge wie FLACI[1] oder dessen Vorgänger AtoCC, zu nutzen. Diese unterstützen die grafische Erstellung und können das Ergebnis ebenfalls als Bild exportieren, welches dann in LaTeX eingebunden werden kann. Jedoch muss bei Änderungen das externe Werkzeug erneut bemüht werden, um ein neues Bild zu erzeugen.

Die dritte Möglichkeit ist, die Graphen direkt in LaTeX zu erzeugen. Diese kann man dann auch in LaTeX anpassen, wenn dies nötig ist. Dafür steht z.B. das Paket Tikz zur Verfügung. Dieses bietet Makros an, um Graphen zu erzeugen. Jedoch ist Tikz ein generisches Werkzeug zur Erstellung von Graphen und bietet keine spezifischen Befehle zur Erstellung von abstrakten Automaten. Außerdem befindet sich die Sprache Tikz auf einem niedrigeren Level der Abstraktion, als es für das Ergebnis dieser Arbeit gefordert ist.

Um ein Werkzeug zu schaffen, welches sowohl innerhalb von LaTeX existiert, als auch gut genug abstrahiert ist, soll die Sprache *Formal Languages and Automata to LaTeX*, kurz: FLaAL, entwickelt werden. Diese Arbeit be-

[1] https://flaci.com/home/

schäftigt sich mit dem Teil der abstrakten Automaten.

3 Grundlagen

3.1 LaTeX-Makros

Um ein fremdes Programm um eigenen Code zu erweitern, hat man die Möglichkeit ein neues Programm zu schreiben, welches auf dem Basisprogramm aufbaut. Das heißt, mit dem neuen aufgesetzten Programm kann man z.B. mehr Befehle nutzen, welche intern so verarbeitet werden, dass das Basisprogramm die Anweisungen versteht. Eine weitere Möglichkeit, ein fremdes Programm zu erweitern, sind Makros. Diese können jedoch nur dann genutzt werden, wenn das Basisprogramm diese auch vorsieht und entsprechende Werkzeuge zur Verfügung stellt.

Die Informationen über LaTeX-Makros wurden verschiedenen Quellen entnommen und hier zusammengetragen. Trotz intensiver Recherche konnte keine Literatur mit einer vollständigen Dokumentation gefunden werden. Die hilfreichsten Ausführungen dazu waren: [Pa06], [Gr], [Wi19], [to10]

3.1.1 Befehle

Wenn man alle Befehle nutzt, welche LaTeX standardmäßig zur Verfügung stellt, dann bekommt man am Ende ein Resultat, welches den generellen Regeln eines gut aussehenden Dokumentes folgt. Dennoch kann es sein, dass man Befehle benötigt, welche so nicht in LaTeX existieren. Für die Lösung des Problems gibt es den Befehl *newcommand*. Dieser erlaubt es, einen neuen Befehl zu erzeugen und diesen mit einem Programmiercode zu versehen. Der Code hinter dem Befehl ist dann das LaTeX-Makro. Die Syntax dahinter ist folgende:

```
1 \newcommand{\<NAME>}[<ANZAHL DER ARGUMENTE>]<[<STANDARDWERT>]>{<
    MAKRO CODE>}
```

Mit `NAME` legt man einen Bezeichner für den Befehl fest. Dieser darf jedoch nur aus kleinen und großen Buchstaben bestehen. Es gibt Makros, welche auch andere Zeichen enthalten dürfen, jedoch müssen diese in bestimmten Umgebungen erstellt werden und werden hier nicht näher erläutert.

Die `ANZAHL DER ARGUMENTE` gibt an, wieviele Argumente an den Befehl angehangen werden dürfen bzw. müssen. Dabei können maximal neun Argumente angegeben werden. Wenn ein `STANDARDWERT` angegeben wird, ist das erste Argument optional. Dieses muss nicht zwingend angegeben werden, damit der Befehl wie erwartet funktioniert. In diesem Fall wird der `STANDARDWERT` als Argument genutzt.

Der `MAKRO CODE` bestimmt, was mit den Daten passiert, welche der Nutzer dem Befehl übergibt. In diesem Code dürfen alle existierenden Befehle (auch eigene) genutzt werden, um das gewünschte Resultat zu erreichen. Innerhalb des Makrocodes kann mit `#<INDEX>` auf eines der Argumente verwiesen werden.

Beispielhaft wird ein Makro dargestellt. Dieses besteht aus einer Liste mit drei Elementen, wobei das erste fett, das zweite kursiv und das dritte fett und kursiv geschrieben ist:

```
1 \newcommand{\liste}[3]{%
2 \begin{itemize}
3     \item \textbf{#1}
4     \item \textit{#2}
5     \item \textbf{\textit{#3}}
6 \end{itemize}%
7 }
```

Der Befehl kann dann wie folgt genutzt werden:

```
1 \liste{Ich bin fett}{Ich bin kursiv}{Ich bin kursiv und fett}
```

Wenn er ausgeführt wird, erhält man Folgendes:

- **Ich bin fett**
- *Ich bin kursiv*
- ***Ich bin kursiv und fett***

3.1.2 Umgebungen

Konstrukte, welche mit `\begin<NAME>` anfangen und mit `\end<NAME>` enden, nennt man *Umgebungen* (s.a. Beispiel in 3.1.1). Mit Hilfe von Umge-

bungen können bestimmte Eigenschaften gesetzt werden, welche innerhalb dieser Umgebung existieren, so zum Beispiel auch Befehle. Wenn Befehle innerhalb einer Umgebungsdefinition definiert werden, dann können diese auch nur innerhalb dieser Umgebung existieren. Außerdem können Umgebungen dazu beitragen, die Übersichtlichkeit eines LaTeX-Dokumentes zu wahren.

Die Syntax des *newenvironment* Befehls sieht wie folgt aus:

```
1 \newenvironment{<NAME>}[<ANZAHL DER ARGUMENTE>][<STANDARDWERT>]{<
    DAVOR>}{<DANACH>}
```

Der `NAME` ist der Wert, welcher bei `begin` und `end` in den Klammern stehen muss. Man kann, mit einigen Ausnahmen, nur kleine und große Buchstaben benutzen. Im Gegensatz zum Namen eines Befehls, gibt man den Namen hier jedoch ohne ein Backslash an, da er später in `begin` und `end` auch ohne Backslash angegeben wird.

Genau wie bei den Befehlen kann es auch hier ein optionales Argument geben, welches mit einem `STANDARDWERT` belegt wird. Die maximale Anzahl der Argumente beträgt ebenfalls 9.

Die Werte `DAVOR` und `DANACH` werden jeweils vor dem `begin` und nach dem `end` der Umgebung ausgeführt. Wenn man also Befehle innerhalb der Umgebung definieren möchte, muss das im `DAVOR`-Block geschehen. Auch zu beachten ist, dass Argumente nur im `DAVOR`-Block zu erreichen sind. Wenn man diese Argumente auch im `DANACH`-Block nutzen möchte, muss man diese mit Hilfsbefehlen zwischenspeichern.

Beispielhaft wird eine Umgebungsdefinition dargestellt. Diese sorgt dafür, dass an Anfang und Ende des Textes in der Umgebung kleine schwarze Quadrate gesetzt werden. Das könnte z.B. als Titel für ein Kapitel o.ä. dienen.

```
1 \newenvironment{titel}[1][5ex]
2 {\rule{1ex}{1ex}\hspace{#1}}
3 {\hspace{\stretch{1}}\rule{1ex}{1ex}}
```

Die Umgebung wird dann wie folgt genutzt:

```
1 \begin{titel}[\stretch{1}]
2 Das ist ein Titel \ldots
3 \end{titel}
```

Das Ergebnis wird wie folgt ausgegeben:

■ Das ist ein Titel ... ■

3.2 LaTeX-Pakete

Benötigt man in seinem Dokument viele solcher Befehle und Umgebungen, kann es sehr unübersichtlich werden, wenn die ersten 1000 Zeilen des Dokumentes nur mit den Definitionen der unterschiedlichen Befehle und Umgebungen gefüllt sind. Des Weiteren muss man in jedem neuen Dokument, welches man anlegt, die gleichen Befehle und Umgebungen immer wieder neu definieren.

Um dieses Problem zu umgehen, kann man die Makros in separate Dateien schreiben und diese bei Bedarf in das Hauptdokument laden. Das hat den Vorteil, dass die Befehls- und Umgebungsdefinitionen nicht im Hauptdokument stehen. Damit ist das Dokument übersichtlicher. Außerdem kann man sie leicht wiederverwenden. Zusätzlich können so auch andere LaTeX-Nutzer das Paket einbinden und die darin enthaltenen Makros nutzen.

Ein Paket bietet sich sehr gut an, um anspruchsvolle Publikationen zu erzeugen. Denn mit Hilfe von Paketen können wissenschaftliche Arbeiten einheitlich erstellt werden und erhalten somit eine wiederkehrende Struktur und ein reproduzierbares Aussehen. Durch die sich wiederholenden Formateigenschaften ist es dem Leser möglich, sich mit diesen vertraut zu machen und sich in Folge dessen mehr auf den Inhalt zu konzentrieren, während die Struktur und das Aussehen unterstützend wirken.

Die unterschiedlichen Compiler von LaTeX bringen bereits Pakete mit, welche mit dem Befehl *usepackage* einfach eingebunden werden können. Dies bewirkt, dass man die Befehle und Umgebungen, welche die Pakete enthalten, in seinem Dokument nutzen kann.

3.2.1 Aufbau eines Paketes

Ein LaTeX-Paket wird immer in einer `.sty`-Datei gespeichert. Diese muss im gleichen Verzeichnis wie das `.tex`-Dokument liegen und kann mit *usepackage* eingebunden werden.

Der Aufbau so eines Paketes sieht wie folgt aus ([ov19], Kapitel 2):

- ***Identifikation***. Die Datei identifiziert sich selbst und gibt an, mit welcher Syntax sie geschrieben ist.

  Die Definition der Syntax des Dokumentes geschieht mit *NeedsTeXFormat* und gehört ganz an den Anfang des Paketes. Mit dem Befehl *ProvidesPackage* legt man den Namen fest, und es können noch zusätzliche Beschreibungen und das Veröffentlichungsdatum bereitgestellt werden.

- ***Vorläufige Deklarationen***. Hier werden alle Pakete eingebunden, welche das Paket selbst benötigt. Außerdem werden alle Befehle und Definitionen erstellt, welche in den Optionen benötigt werden.

  Um Pakete einzubinden, welche das eigene Paket benötigt, benutzt man den Befehl *RequirePackage*. Dieser ist sehr ähnlich zu dem Befehl *usepackage*, dennoch wird empfohlen, den erstgenannten zu nutzen ([ov19], Kapitel 2.2).

- ***Optionen***. Hier werden die Optionen des Paketes definiert und verarbeitet.

  Die Optionen werden mit *DeclareOption* festgelegt. Diese Optionen können beim Einfügen eines Paketes mit angegeben werden. Zum Beispiel kann es die Optionen f und k geben, welche den gesamten ausgegebenen Text eines Paketes in fett (f) oder kursiv (k) darstellt.

- ***Weitere Deklarationen***. Dies ist der Hauptteil des Paketes. Fast die gesamte Funktionalität des Paketes wird hier definiert.

  Alle Befehle und Umgebungen, welche das Paket zur Verfügung stellen soll, werden hier definiert. Dies geschieht mit *newcommand* und *newenvironment*.

3.2.2 Umgang mit Fehlern

Wenn Nutzer Eingaben tätigen, kann es vorkommen, dass die Eingabe nicht den Erwartungen entspricht. Um dem Nutzer mitzuteilen, dass seine Eingabe nicht den Erwartungen entspricht oder um zu verhindern, dass der Code

überhaupt vom LaTeX-Compiler übersetzt wird, gibt es die Befehle *PackageWarning*, *PackageWarningNoLine* und *PackageError*. Bei *PackageWarning* und *PackageWarningNoLine* werden im Log des Compilers Warnungen in der Form ausgegeben, dass die Eingabe nicht den Erwartungen entspricht. Trotz der Warnung stoppt der Übersetzungsprozess nicht und man erhält ein Ergebnis. Der Unterschied zwischen den beiden Befehlen besteht darin, dass nur der erste Befehl eine Zeilangabe tätigt. Wenn der Befehl *PackageError* genutzt wird, dann steht im Compiler-Log eine Fehlermeldung und der Ort des Auftretens. Außerdem wird der Prozess des Übersetzens gestoppt ([ov19], Kap. 3).

Ausführliche Informationen und Beispiele findet man u.a. im Artikel auf `overleaf.com` ([ov19]).

FLaAL soll in ein solches Paket gekapselt und LaTeX-Makros darin implementiert werden.

4 FLaAL Paket

4.1 Idee

Da ein Befehlssatz für die Erstellung von Modellen der theoretischen Informatik entwickelt werden und dieser in vielen verschiedenen Arbeiten zum Einsatz kommen soll, liegt es nahe, ein LaTeX-Paket für diese Befehle und Umgebungen zu erstellen. Durch die Kapselung des eigentlichen Paketes ist der Nutzer in der Lage, die Erstellung der Graphen möglichst abstrakt zu sehen. Im besten Fall muss sich der Nutzer dann nur noch um den Inhalt des Graphen und weder um dessen Aussehen, noch dessen Erstellung, Gedanken machen.

Die Sprache FLaAL hat außerdem den Vorteil, unterschiedliche Darstellungsobjekte unter diesem Paket zu vereinigen. Das wären zum einen die abstrakten Automaten, welche in dieser Arbeit behandelt werden, aber auch formale Sprachen und T-Diagramme.

Die formalen Sprachen könnten, ähnlich wie in FLACI, als Syntaxdiagramme dargestellt sein, während die T-Diagramme aus T-förmigen Bausteinen bestehen und einen Graphen formen. Das Einfügen dieser in LaTeX, ist nicht Bestandteil dieser Arbeit.

Eine weitere Möglichkeit für FLaAL bestünde darin, bereits vordefinierte Beispielgraphen, wie in der Beispielsammlung von FLACI, zu besitzen. Diese können für Lehrzwecke immer wieder genutzt werden. Das wäre z.B. der Graph für die Verdopplungsmaschine (Anhang A.6.1, S. 57). Für diese könnte es den Befehl *doublingmachine* geben, welcher genau diese Turing-Maschine erzeugt.

4.2 Anforderungen

Die Sprache FLaAL soll vor allem den komplizierten Gebrauch der LaTeX-Befehle abstrahieren. Denn mit Paketen wie Tikz ist es möglich, einen Automaten zu erzeugen. Jedoch sind die Befehle an das Zeichnerische und nicht an einen abstrakten Automaten angelehnt. Deshalb ist es das Ziel, dass der Nutzer semantische Befehle mit einem Inhalt versieht und das Paket diese

dann in ein zufriedenstellendes Ergebnis übersetzt. Außerdem soll FLaAL in gängigen TEX-Distributionen verwendet werden können. Von der TEX Users Group empfohlene Distributionen sind TEX Live, MacTEX, MiKTEX sowie ein paar Weitere ([19], Kap. Free TeX Implementations).

Zur Umsetzung der Anforderungen sollen die LATEX-Befehle *state* und *transition*, sowie die LATEX-Umgebung *transitiongraph* implementiert werden.

transitiongraph Die transitiongraph-Umgebung beinhaltet die Befehle *state* und *transition*. Der Umgebung kann zusätzlich ein optionales Argument hinzugefügt werden, mit welcher man die Art des abstrakten Automaten festlegen kann. Standardmäßig ist FA[2] (finite automaton) voreingestellt.

state Dieser Befehl ermöglicht das Erstellen von Zuständen eines Automaten. Man kann für den Zustand den Typ und einen Namen festlegen.

transition Um einen Übergang zwischen zwei Zuständen oder eine *selftransition* zu erzeugen, benötigt man den Startzustand, von welchem der Übergang ausgeht und den Zielzustand, zu welchem der Übergang hinführt. Wenn diese Zustände die gleichen sind, ist dies eine *selftransition*. Außerdem müssen die Labels der Übergänge festgelegt werden können.

4.3 Entwurf des Paketes

Die Sprache FLaAL baut auf dem Paket *Tikz* auf. „PGF is a TeX macro package for generating graphics. It is platform- and format-independent and works together with the most important TeX backend drivers, including pdftex and dvips. It comes with a user-friendly syntax layer called TikZ“([pg], erster Absatz der `README`). Frei übersetzt bedeutet das, dass Tikz eine nutzerfreundliche Oberfläche für PGF[3] (portable graphics Layer) ist, welches wiederum ein Makro ist, um Grafiken zu generieren. Des Weiteren ist PGF

[2]endlicher Automat

[3]PGF ist ein Basislayer über einem Systemlayer, welches komplexe Grafiken erzeugen kann. Auf dem Basislayer baut dann Tikz auf. Tikz abstrahiert die komplexeren Befehlsstrukturen von PGF und stellt somit ein nutzerfreundliches Frontend dar. [Ta15]

und damit auch Tikz plattformunabhängig. Dadurch wird Tikz von jedem gängigen TEX-Compiler unterstützt und eignet sich daher sehr gut für die Umsetzung unserer Anforderungen.

Wie in Kapitel 3 bereits behandelt, stellt LATEX die Möglichkeit bereit, eigene Makros zu erstellen und diese in eigene Pakete zu verpacken. Diese Eigenschaft kann man für die Sprache FLaAL nutzen und die Makros in ein Paket Namens FLaAL packen. Dadurch kann FLaAL in vielen verschiedenen Dokumenten eingesetzt werden, da die erneute Verwendung durch das Paket gewährleistet wird.

4.3.1 Deklaration von Zustandsgraphen

Die Umgebung des Übergangsgraphen definiert die Befehle *state* und *transition*, legt die Styles für das Tikz-Paket fest, beginnt und beendet eine Tikz-Umgebung und handhabt den Typ des Automaten.

Die Styles für das Tikz-Paket orientieren sich an der allgemeinen Darstellungsform. Zustände werden als Kreise repräsentiert. Zwei konzentrische Kreise stehen für einen Endzustand. Startzustände bekommen einen offenen, von links kommenden Übergang, welcher mit *Start* beschriftet ist. Die Zustände, welche Start- und Endzustand zugleich sind, vereinen die grafischen Eigenschaften von Start- und Endzustand. Die Übergänge werden mit Pfeilen zwischen den Zuständen repräsentiert. An diesen werden dann die Bedingungen für die Übergänge geschrieben.

Der Typ des abstrakten Automaten wird als optionales Argument in die *transitiongraph*-Umgebung übergeben. Standardmäßig ist das der FA. Falls ein anderes Argument als *fa* (*finite automaton*), *pa* (*pushdown automaton*) oder *tm* (TURING *machine*) angegeben wird, gibt FLaAL einen Fehler aus. Wenn nun ein Befehl innerhalb der Umgebung die Information über einen Automatentyp benötigt, nimmt er die Information aus der Umgebung. Die Syntax für den Nutzer sieht wie folgt aus:

```
1 \begin{transitiongraph}[<TYP DES AUTOMATEN>]
2     <Platz fuer andere Befehle>
3 \end{transitiongraph}
```

4.3.2 Deklaration von Zuständen

Der *state*-Befehl wird in der *transitiongraph*-Umgebung definiert und ermöglicht das Erstellen von Zuständen eines Automaten. Er bekommt ein optionales, sowie drei erforderliche Argumente. Im folgenden Codeblock kann man die Syntax des Befehls erkennen.

```
1 \state[<TYP>]{<NAME>}{<X-POS>}{<Y-POS>}
```

Der `TYP` eines Zustandes soll durch *s* (start), *f* (final), *sf* (start und final) oder *n* (normal) gekennzeichnet werden. Standardmäßig ist *n* als Typ eingestellt. Aus Gründen der Nutzerfreundlichkeit wurde entschieden, lediglich die Abkürzungen der Wörter zu nutzen. Eine andere Möglichkeit ist die Nutzung von den Schlüssel-Wert-Paaren `[start=true,final=true]`. Diese Variante ist jedoch zu lang, weshalb die Abkürzungen eine gute Alternative darstellen. Diese sind eingängig, und die Semantik dahinter ist ebenfalls verständlich. Wenn das Argument keines von den vier gültigen ist, gibt FlaAL eine Warnung aus und fällt auf den Typ *n* zurück.

Der `NAME`, welcher angegeben werden muss, ist auch zeitgleich der Schlüssel für das Tikz-Paket und muss somit eineindeutig sein. Das bedeutet, dass die Schlüssel der *Nodes* des Tikz-Graphen den Namen des Zustandes bekommen müssen. Dadurch kann Tikz später die Referenzierung auf die Zustände im *transition*-Befehl erkennen und die Übergänge erzeugen. Wenn ein Zustandsname jedoch mehrfach vorkommt, wird FLaAL einen Fehler erzeugen.

Die Positionierung der Zustände mit `X-POS` und `Y-POS` erfolgt über x- und y-Koordinaten, wobei die Werte, welche man angibt, als Millimeter interpretiert werden, da das Tikz-Paket alle Positionierungen in Millimetern vornimmt. Außerdem sollte man die vertikale und horizontale Begrenzung des Papiers beachten. Durch zu große Werte kann es passieren, dass der Graph am Ende größer als das Papier wird. Für ein DIN-A4-Papier kann man die Maximalwerte $x \approx 130$ und $y \approx 180$ annehmen, für einen Ursprung in $(0, 0)$.

Beispiele von *state*-Befehlen

```
1 \state[s]{z0}{0}{0}
2 \state{z1}{50}{0}
```

```
3 \state[f]{z2}{100}{0}
```

Für die automatische Positionierung der Zustände des Paketes wurden folgende Möglichkeiten durchdacht:

Var. 1: *Festes Gitter* Man teilt die Fläche des Papiers in ein festes Gitter ein, entlang dieser die Zustände platziert werden, in der vom Nutzer angegebenen Reihenfolge. Das ist möglich, ohne eine neue Syntax zu erzeugen, wird aber in den meisten Fällen unbrauchbar sein, da diese Variante zu unflexibel ist.

Var. 2: *Dynamisches Gitter* Dieses funktioniert im Prinzip genauso wie das feste Gitter, nur ist die Anzahl der Zeilen und Spalten abhängig von der Anzahl der Zustände. Das kann nicht mehr ohne eine Erweiterung der Syntax geschehen und wäre vermutlich in den meisten Fällen auch nicht flexibel genug.

Var. 3: *Automatische Positionierung wie in FLACI* In FLACI wird ein Automat wie folgt automatisch ausgerichtet: Zuerst werden die Zustände zufällig gemischt, und der Startzustand wird auf den ersten Platz in der Liste gesetzt. Wenn diese Liste ein Element besitzt, wird das erste Element auf $x = 150$ und $y = 150$ gesetzt. Dann werden alle freien Plätze berechnet, also ein dynamisches Gitter wird erzeugt und die Zustände werden darauf platziert. Die Übergänge zwischen den Zuständen werden neu berechnet, und anschließend wird der Wert des Graphen ermittelt. Anhand von Überkreuzungen von Übergängen wird festgestellt, wie gut der Graph ist. Je mehr Punkte, desto schlechter ist die Ausrichtung. Das Ganze wird 1000 mal durchgeführt, und am Ende wird der Graph mit den wenigsten Punkten als Ausrichtung für den Automaten gewählt. Dieses Vorgehen hat den Vorteil, dass mit hoher Wahrscheinlichkeit eine gute Ausrichtung gefunden wird. Jedoch ist dieses Verfahren abhängig von den Zuständen und den Übergängen, was wieder eine Erweiterung der Syntax für FlaAL bedeutet. Auch die zufällige Ausrichtung des Graphen ist ein Problem, da man beim Übersetzen des

Dokumentes jedes Mal eine andere Ausrichtung der Graphen erhalten würde, was im LaTeX-Dokument nicht gewünscht ist.

Var. 4: Ziel ist ein Algorithmus, welcher determiniert ist, aber dennoch einen bestmöglichen Graphen findet. Das könnte man erreichen, indem ein eigener Zufallsalgorithmus mit zehn verschiedenen vordefinierten Zahlen-Sets bereitgestellt wird. Der Nutzer kann sich beim Erstellen eines Graphen ein Set auswählen und hat somit zehn verschiedene Layouts zur Verfügung. Allerdings gewährleistet auch diese Variante nicht, dass der Nutzer am Ende nicht doch die x- und y-Werte selbst angeben muss.

Aus oben genannten Gründen wurden die ersten beiden Varianten verworfen. Die dritte Variante klingt vielversprechend, ist für die Praxis jedoch unbrauchbar. Die letzte Variante ist die wahrscheinlich beste Möglichkeit, eine zufriedenstellende, automatische Ausrichtung zu bekommen. Jedoch benötigt sie eine mehrfach verschachtelte Syntax und wird somit komplizierter. Außerdem kann die Eingabe von x- und y-Werten nicht entfallen, da auch die vierte Variante keine beste Ausrichtung garantieren kann. Daher wird die automatische Positionierung in dieser Arbeit nicht implementiert.

4.3.3 Deklaration von Zustandsübergängen

Der *transition*-Befehl benötigt drei Argumente und kann durch ein weiteres optionales Argument erweitert werden. Die Syntax sieht wie folgt aus:

```
1 \transition[<STYLE>]{<START>}{<ZIEL>}{<LABELS>}
```

Das `STYLE`-Argument kann zwei Schlüssel-Wert-Paare enthalten, *label* und *line*. Der *label*-Schlüssel bestimmt die Position der Labels an den Übergängen mit Hilfe der Schlüsselwörter *top*, *bot*, *left* und *right*. Das muss jedoch nicht zwingend angegeben werden, da die Positionierung der Labels automatisch geschieht. Die Ausrichtung der Übergänge selbst erfolgt zwar auch automatisch, benötigt jedoch mehr Korrekturen im Nachhinein als die Ausrichtung der Labels. Dafür ist der Schlüssel *line* zuständig. Durch die gleichen Schlüsselwörter wie bei *label* und dem zusätzlichen Schlüsselwort *straight* kann die

Ausrichtung der Übergänge gesetzt werden. Dabei ist jedoch zu beachten, dass *straight* lediglich bei Übergängen genutzt werden kann, welche keine *selftransitions* sind, jedoch *top* und *bot* ausschließlich von *selftransitions* genutzt werden können. Wenn die angegebenen Argumente von FLaAL nicht erkannt werden, gibt FLaAL eine Warnung aus und fällt auf die Standardbelegung zurück.

Für `START` und `ZIEL` wird jeweils der Name eines Zustandes angegeben. Wichtig hierbei ist, dass das die gleichen Namen sind, wie sie auch bei den *state*-Befehlen angegeben wurden, damit Tikz diese wiedererkennt. Wenn ein Name eines Zustands angegeben wird, welcher vorher nicht mit dem *state*-Befehl erstellt wurde, wird FLaAL eine Warnung erzeugen und den Übergang ignorieren.

Die `LABELS` müssen je nach `TYP` des Automaten unterschiedlich eingegeben werden. Das hat zur Folge, dass die Verarbeitung im Paket ebenfalls unterschiedlich geschehen muss. Für alle Typen gilt, dass jedes Label durch ein Semikolon getrennt wird. Bei FAs funktioniert das sehr gut, da deren Labels nur aus einzelnen Token bestehen. Für PAs und TMs gibt es noch eine zusätzliche Syntax. Alle Werte, welche so ein Automat pro Label benötigt, werden nochmal mit einem Komma getrennt. Für einen PA sieht das dann wie folgt aus:

```
1 \transition{q1}{q1}{A,a,AA;A,b,;B,a,;B,b,BB}
```

Wenn bei FAs oder PAs ein Epsilon zum Einsatz kommen soll, dann kann bei FAs das erste Zeichen frei gelassen werden und bei PAs kann der entsprechende Wert frei gelassen werden. Dafür wird dann ein `$\varepsilon$` eingefügt.

```
1 \transition{Z1c}{Z1b}{;a;b;c}
```

Alternativ kann man auch selbst ein Epsilon als Zeichen einfügen.

Wenn in einem Übergang ein Label doppelt angegeben wird, wird es von FLaAL ignoriert. Das heißt *{a;b;b;a;a;c;c;c;a}* ist das gleiche wie *{a;b;c}*.

Beispiele von *transition*-Befehlen

```
1 \transition{z1}{z2}{a;b;c}                    % fuer einen
    endlichen Automaten
```

```
2 \transition[line=left]{z0}{z0}{b}                    % fuer einen
    endlichen Automaten
3 \transition[label=top]{z0}{z2}{1,\$,R;0,\$,L}    % fuer eine
    Turing Maschine
```

Zusätzlich zu FLaAL selbst wurde eine kontextfreie Grammatik in FLACI erstellt, welche die Syntax von FLaAL repräsentiert. Im Anhang A.2 kann man diese einsehen. Die Sprache FLaAL ist eine LL(1)-Sprache[4], die mit FLACI überprüft wurde. Da das Übersetzen des Textes nicht vom FLaAL-Paket selbst, sondern vom LaTeX-Compiler übernommen wird, ist die kontextfreie Grammatik für diese Anwendung nicht weiter relevant.

4.4 Implementierung

4.4.1 Transitiongraph

Zuerst wurde die Umgebung *transitiongraph* erstellt. Diese überprüft, um welchen Automatentyp es sich handelt. Dafür zieht LaTeX das optionale Argument der Umgebung heran. Falls dieses Argument nicht fa, pa oder tm ist, wird ein Fehler erzeugt. Die Umgebung definiert die *tikzstyles* für die unterschiedlichen Bausteine eines Automaten, und es werden die Befehle *state* und *transition* definiert. Danach wird eine *tikzpicture*-Umgebung begonnen und im zweiten Teil der Umgebungsdefinition wieder beendet.

4.4.2 State

Der *state*-Befehl übersetzt die gegebenen Daten in ein Tikznode mit entsprechendem Namen, Aussehen und den x- und y-Werten als Millimeter für die Positionierung. Der Name des Zustandes wird weiterhin für die Referenzierung der Nodes bei den Übergängen genutzt.

4.4.3 Transition

Für den *transition*-Befehl wurden zwei Schlüssel-Wert-Paare erstellt, damit die Anzahl der Argumente nicht zu groß wird. Diese können in der Form

[4]Eine LL(1)-Sprache ermöglicht das schnelle parsen durch einen Parser. [WH, F. 7]

`key=value` im Argument genutzt werden.

Eine Herausforderung im *transition*-Befehl ist es, die Labels zu verarbeiten. Für eine Liste, die durch Semikolons getrennt ist, benötigt man eine Schleife und die Möglichkeit, die Liste an den Semikolons zu teilen. Um das Ziel zu erreichen, wurde mit den Labels des FA begonnen. Diese sind eine Liste aus Zeichen des Kelleralphabets, welche durch Semikolons getrennt sind.

Label-Verarbeitung des FA Da die Labels der Übergänge der FAs eine Liste aus Zeichen des Eingabealphabets ist, muss diese Liste auch von FLaAL verarbeitet werden. Das heißt, der Eingabestring mit den Zeichen muss an den Semikolons getrennt und wieder neu zusammengefügt werden. Das Zusammenfügen bei endlichen Automaten gestaltet sich so, dass alle Zeichen mit einem Komma und einem Leerzeichen getrennt werden. Zusätzlich müssen duplizierte Labels beachtet werden, da diese nicht doppelt auftreten sollen. Mit Hilfe der LaTeX3-Befehle wurde eine Lösung gefunden, welche im Anhang (A.4, S. 50) im Codeblock *FA-Label-Processing* zu finden ist. Wenn man einen nichtdeterministischen endlichen Automaten darstellen möchte, kann man entweder das erste Zeichen der Labels frei lassen oder ein `$\varepsilon$` an entsprechender Stelle einfügen. Die Übergänge

```
1 \transition{q0}{q1}{$\varepsilon$;a;b;c}
```

und

```
1 \transition{q0}{q1}{;a;b;c}
```

sind folglich gleich.

Label-Verarbeitung des PA Auch die Labels der Kellerautomaten sind eine durch Semikolons separierte Liste. Jedoch bestehen die einzelnen Labels wiederum aus einer durch Kommas separierten Liste. Dadurch muss eine verschachtelte Schleife genutzt werden. Dafür sind neue Variablen anzulegen. Das Prinzip der Verarbeitung der Labels entspricht dem des FA. Als Ergebnis der Verarbeitung erhält man jedoch untereinander stehende Labels der Form:

```
1 (TopOfStack,Eingabezeichen):Kellerwort
```

Damit die Labels eines PA untereinander stehen, ist ein Zeilenumbruch hinter jedem Label hinzuzufügen.

Das einfache Hinzufügen eines Zeilenumbruchbefehls brachte nicht das gewünschte Ergebnis. Zur Kontrolle wurde das Verfahren auf die Label-Verarbeitung des FA angewandt, da dessen Verarbeitungsfunktion simpler ist. Auch das brachte keinen Erfolg.

Anstatt eines einfachen Textes wurde nun eine Tabelle als Beschriftung der Übergänge implementiert, da mit dieser die Labels untereinander geschrieben werden können. Dies funktionierte zwar, der Code war jedoch zu unübersichtlich, sodass man nichts mehr richtig erkennen und zuordnen konnte.

Nach umfangreichen Tests mit verschiedenen Konfigurationen innerhalb des Tikz-Paketes wurde eine Lösung gefunden. Die Übergänge des Tikz-Paketes bieten eine *alignment*-Option an. Wenn man diese auf *center* stellt, kann man ein doppeltes Backslash als Zeilenumbruch nutzen. Bei dem FA hat das ohne Probleme funktioniert. Die Anwendung der Lösung auf einen PA erbrachte jedoch ein unerwartetes Resultat. Anstatt, dass mehrere Labels eines Übergangs in der Form (TopOfStack,Eingabzeichen):Kellerwort untereinander standen, stand ein Label dieser Form ganz oben, während die anderen Labels leer waren: (,) :. Auch das Benutzen anderer Zeilenumbruchsbefehle, wie *newline*, führten zum selben Ergebnis. Wurde der Zeilenumbruch entfernt, war der Inhalt aller Labels wieder vorhanden.

Es wird vermutet, dass der Zeilenumbruch einige Indizes zurücksetzt. Wenn man sich die Werte ausgeben lässt, welche während der Schleife immer wieder verändert werden, stellt man fest, dass diese nach dem ersten Durchlauf der Schleife auf null zurückgesetzt werden. Das könnte der Grund für die leeren Beschriftungen der Labels sein.

Da das Problem so nicht zu lösen war, wurde ein anderer Weg gewählt, der im Anhang (A.4, S. 51) zu finden ist. Es wird eine Liste erstellt, welche alle Labels sammelt, bis auf das Erste. Nachdem die Schleife dann durchlaufen und alle Labels zwischengespeichert wurden, wird jedem Label noch ein

Zeilenumbruch vorangestellt. Das Ergebnis ist wie gewünscht.

Wenn in Kellerautomaten ein Epsilon vorkommen soll, kann man entweder ein `$\varepsilon$` an entsprechender Stelle einfügen oder die entsprechende Stelle frei lassen. Leere Stellen werden durch ein Epsilon ersetzt.

Label-Verarbeitung des TM Für die TURING-Maschine gilt das Gleiche wie für die PAs. Auch für diese muss das Label zuerst zusammengestellt und in eine Liste gespeichert werden, bevor der Zeilenumbruch hinzugefügt werden kann. Anders als bei den FAs und PAs kann hier jedoch durch Freilassen von Stellen kein Epsilon eingesetzt werden. Das Paket wird auch keinen Fehler erzeugen, wenn ein `$\varepsilon$` eingefügt wird.

4.4.4 Aufgetretene Fehler

In FLACI wird als Kellerzeichen oft das Dollarzeichen genutzt. Deshalb sollte man es auch in LaTeX nutzen können. Das ist kein Problem, solange man nur ein einziges Zeichen pro Übergang nutzt. Diese Einschränkung führt dazu, dass der LaTeX-Compiler folgenden Fehler erzeugt, wenn mehr als ein Dollarzeichen in eine *transition* eingebaut wird: *TeX capacity exceeded, sorry [input stack size=5000]*. Die weitere Information verrät, dass es mit dem *math mode* in Verbindung stehen könnte. Eigentlich dürfte das kein Problem sein, da Dollarzeichen mit einem vorangestellten Backslash genutzt werden, um die Funktion des Dollars als *math mode* zu deaktivieren. Es stellte sich jedoch heraus, dass das Problem mit der TeX-Distribution zusammenhängt. In der Entwicklung wurde pdfLaTeX als LaTeX-Compiler genutzt. Bei diesem tritt das Problem auf. Folgende Distributionen funktionieren wie erwartet: TeX Live, XeLaTeX und LuaLaTeX. Distributionen, welche Probleme mit dem Dollarzeichen haben sind pdfLaTeX und LaTeX. Weitere Distributionen wurden nicht getestet. Wenn man keine Möglichkeit hat, einen funktionierenden LaTeX-Compiler zu nutzen, kann man alternativ auch das Und-Zeichen als Kellervorbelegungszeichen nutzen.

Ein weiteres Fehlverhalten, welches durch Testen aufgedeckt wurde, war die schlechte Interaktion mit der Schriftgröße des Dokumentes. Diese wirkt sich auf die Skalierung der inneren Kreise der Finalzustände aus und sorgt

dafür, dass diese sich automatisch vergrößern, wenn Platz gebraucht wird. Jedoch werden die äußeren Kreise nicht an die Größe des Inhalts angepasst. Auf diese Funktionalität Rücksicht nehmend, wurde FlaAL rückwirkend angepasst, sodass jetzt auch unterschiedliche Schriftgrößen kein Problem mehr darstellen.

4.5 Evaluation

Zum Testen wurden mehrere Testfälle mit typischen Graphen erstellt, welche bei der Entwicklung immer wieder verwendet wurden. Dadurch konnte auf Fehler möglichst schnell reagiert werden.

Für FLaAL gab es unterschiedliche Testdaten. Da sich die Syntax im Verlauf des Projektes mehrfach geändert hatte, mussten die Testdaten dementsprechend angepasst werden. Ein aktuelles Beispiel dafür ist:

Testgraph

```
1  \begin{figure}
2      \centering
3      \begin{transitiongraph}[fa]
4          \state[s]{q0}{0}{0}
5          \state{q1}{20}{40}
6          \state[f]{q2}{40}{0}
7          \transition[label=left]{q0}{q1}{a;b}
8          \transition[line=right,label=bot]{q0}{q2}{$\varepsilon$;a
               ;b;c}
9          \transition{q1}{q1}{a;b;c}
10         \transition[line=right,label=right]{q2}{q2}{a;b}
11         \transition[line=left,label=right]{q2}{q1}{$\varepsilon$}
12     \end{transitiongraph}
13     \caption{Tikzgraph}
14     \label{graph:first_graph}
15 \end{figure}
```

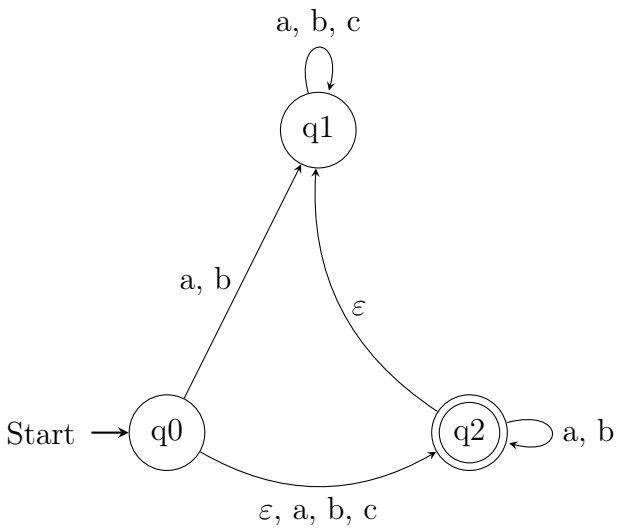

Abbildung 2: Testgraph

Es wurden immer mehrere Kombinationen von optionalen Argumenten genutzt (Z. 4-12), damit mit einem Testgraphen möglichst viele Varianten getestet werden konnten. Abbildung 2 zeigt das Resultat.

Für unterschiedliche Automatentypen gab es unterschiedliche Testdaten, da das optionale Argument der *transitiongraph*-Umgebung (*fa*, *pa*, *tm*), sowie die Labelverarbeitung dieser getestet werden musste.

Die Ergebnisse wurden begutachtet, und es wurde entschieden, ob sie den Anforderungen entsprachen. Wenn nicht, wurde der Code so lange angepasst, bis das Ergebnis zufriedenstellend war. Erst danach konnte weitergearbeitet werden.

4.6 Möglichkeiten und Grenzen von FLaAL

Möglichkeiten FLaAL bietet die Möglichkeit, Graphen von abstrakten Automaten zu zeichnen. Diese orientieren sich am Aussehen der Graphen von FLACI und besitzen alle Eigenschaften, welche ein Übergangsgraph benötigt, wie z.B. unterschiedliche Typen von Zuständen und Beschriftungsarten der Übergänge. Die Syntax ist weitestgehend so stark abstrahiert worden, dass man sich nur noch Gedanken über die Positionierung der Zustände und teilweise der Übergänge des Graphen machen muss.

Grenzen Die Zustände des Graphen der abstrakten Automaten können sich nicht selbst positionieren, d.h. der Nutzer muss sich mit der Ausrichtung des Graphen beschäftigen. Auch die Ausrichtung der Übergänge muss teilweise vom Nutzer nachkorrigiert werden.

Da LaTeX ein Werkzeug für wissenschaftliche Arbeiten ist, werden die Ergebnisse in einem Papierformat ausgegeben. Der Platz eines Graphen ist durch das Papierformat begrenzt. Damit der Graph innerhalb der Papiergrenzen bleibt, müssen maximale x- und y-Werte beachtet werden. Die Maße, welche man für ein DIN A4-Papier beachten sollte, sind für $x \approx 130$ und $y \approx 180$.

Einige LaTeX-Distributionen können nicht zuverlässig mit dem Dollarzeichen umgehen. Wenn mehr als eines in einem Übergang genutzt werden soll, übersetzen einige LaTeX-Distributionen dies nicht und geben stattdessen einen Fehler aus.

Formale Sprachen und T-Diagramme sind in FLaAL noch nicht implementiert.

5 FLACI to FLaAL Compiler

5.1 Anforderungen

Es wird ein Programm FFC (FLACI to FLaAL Compiler) benötigt, welches eine Definition eines abstrakten Automaten im `JSON`-Format (Javascript Object Notation) als Eingabe entgegennimmt und nach LaTeX in die Zielsprache FLaAL übersetzt. Die Ausgabe soll dabei dem Erscheinungsbild in FLACI möglichst nahe kommen.

5.2 Entwurf

Es gibt verschiedene Varianten der Anwendungsform für einen Compiler von FLACI nach FLaAL.

1. ***CLI Anwendung*** (Command Line Instruction). Dieser Anwendung übergibt man eine `JSON`-Datei und bekommt als Resultat eine Datei mit dem LaTeX-Code.

2. ***Anwendungsprogramm für den Desktop***. Diese Anwendung wird installiert und stellt eine grafische Oberfläche zur Verfügung. Die `JSON` kann über die grafische Oberfläche ausgewählt und vom Programm übersetzt werden. Die Ausgabe ist innerhalb des Programms möglich.

3. ***Website***. Diese funktioniert ähnlich wie ein Desktopanwendungsprogramm, wird jedoch im Browser ausgeführt.

4. ***In FLACI integriert***. FFC wird als Paket in FLACI eingebunden und kann von dort aus genutzt werden.

Variante 1: CLI Anwendung Eine CLI-Anwendung ist für einen Compiler eine ausreichende Anwendungsform, angesichts der Tatsache, dass viele kompilierte Sprachen (z.B. Java mit Javac) ebenfalls nur Kommandozeilencompiler nutzen. Durch einen simplen Aufruf in der Konsole mit Angabe einer JSON-Datei ist der Aufwand für den Anwender sehr gering.

Variante 2: Desktopanwendung Ein Vorteil der grafischen Oberfläche ist die Abstraktion vom eigentlichen Compiler. Der Nutzer zieht seine Datei per Drag and Drop in das Fenster und drückt einen Knopf zum Kompilieren. Das Ergebnis wird im Programm abgebildet und kann bei Bedarf abgespeichert werden. Für unsere Anforderungen ist eine grafische Oberfläche jedoch unverhältnismäßig. Da in unserem Compiler nur wenig Argumente übergeben werden sollen, ist die Übersichtlichkeit auch in der Konsole noch gegeben.

Variante 3: Browseranwendung Die Anwendung im Browser bietet den gleichen Komfort, welchen auch eine Desktopanwendung liefert, jedoch ohne dass der Nutzer neue Software installieren muss. Eine extra Website bietet ebenfalls die Möglichkeit der Verlinkung von FLACI zum Compiler. Dennoch sollte man von einer externen Website absehen, da es ein weiteres Tool wäre, welches jedoch kein Alleinstellungsmerkmal besitzt.

Variante 4: In FLACI integriert Da FFC die Funktionalität bietet, JSONs von FLACI nach FLaAL zu übersetzen, ist dieses Tool stark an FLACI gebunden. Deshalb ist eine Integration in FLACI eine logische Schlussfolgerung. Dafür muss FFC z.B. als Javascriptfunktion entwickelt werden, welche dann in den Quellcode von FLACI eingearbeitet und durch einen Knopfdruck aufgerufen wird.

Aus den vier vorgestellten Varianten wurde die vierte Variante als beste herausgearbeitet. Sie bietet eine komfortable Nutzung für den Anwender. Dieser braucht keine extra Installation und kann weiterhin FLACI ohne zusätzliche Software nutzen. Es wurde festgelegt, FFC als eine Javascriptfunktion zu entwickeln, jedoch mit einer Konsolenanwendung als Wrapper, damit der Compiler getestet werden kann.

Es wird eine Funktion benötigt, welche eine `JSON` von FLACI und zusätzliche Optionen als Argumente nimmt. Die `JSON` wird übersetzt und der LaTeX-Code zurückgegeben. Die Übersetzung erfolgt so, dass die Skalierung des Graphen angepasst wird und die entsprechenden LaTeX-Befehle generiert werden. Zusätzlich zu der *transitiongraph*-Umgebung erstellt der Compiler

noch eine *figure*-Umgebung. Diese dient dazu, den so entstandenen Graphen in LaTeX gut zu platzieren. Auch die Krümmung der Übergänge soll im Compiler beachtet werden.

Um das Verhalten des Compilers anpassen zu können, kann der Funktion ein *options*-Objekt übergeben werden. Die Optionen sind *width*, *height* und *flipY*. *width* und *height* sind für die Skalierung des Graphen wichtig, da die Fläche, auf welcher der Graph dargestellt wird, in FLACI potenziell unendlich groß sein kann. Der Graph muss so skaliert werden, dass er eine maximale Höhe und Breite hat, da eine DIN A4-Seite nur beschränkt Platz bietet. Wenn der Graph jedoch kleiner als die angegebene *width* und *height* ist, soll der Graph nicht skaliert werden.

Da die y-Werte bei einer `SVG`[5] (Scalable Vector Graphics) von oben nach unten größer werden, während die y-Werte sich beim Tikz-Paket genau umgekehrt verhalten, gibt es noch die Möglichkeit, in den Optionen die *flipY*-Eigenschaft auf *true* zu setzen und somit die y-Werte umzukehren. Dadurch ergibt sich ein Graph, welcher genauso aussieht wie in FLACI.

5.3 Aufbau der JSONs von FLACI

Mit Hilfe von `jsonschema.net` [ja18] werden die JSONs der unterschiedlichen Automatentypen in das `JSON`-Schema[6] gebracht. Das `JSON`-Schema-Tool übersetzt das so entstandene Schema in ein menschenlesbares `YAML`-Format (YAML Ain't Markup Language). Für die `JSON` einer Turing-Maschine von FLACI ist dies beispielhaft im Anhang A.3 dargestellt. Da die `JSON`s der Automaten alle sehr ähnlich sind, reicht ein Beispiel aus. Die einzigen Unterschiede liegen in den Labels der Übergänge.

Aufbau der JSON Alle Automaten sind als Objekt aufgebaut. Dieses hat die Eigenschaften *name*, *description*, *type* und *automaton*.

name ist ein String, welcher vom Nutzer auf FLACI beliebig gewählt werden kann. Ebenso kann der Nutzer die ***description*** beliebig wählen. Der

[5] Die Graphen in FLACI werden durch eine `SVG` dargestellt.

[6] „[The] `JSON` Schema asserts what a `JSON` document must look like, ways to extract information from it, and how to interact with it“[Wr].

type des Automaten wird dadurch festgelegt, welcher Automat auf FLACI ausgewählt wird. Das kann *DEA*, *NEA*, *DKA*, *NKA* oder *TM* sein. Die Eigenschaft ***automaton*** enthält ein Objekt, welches die eigentliche Definition des Automaten beinhaltet.

automaton Der Automat hat die Eigenschaften *simulationInput*, *Alphabet*, *StackAlphabet* und *States*.

Der ***simulationInput*** enthält Informationen über das Wort, welches dem Automaten übergeben wird. Das ***Alphabet*** enthält die Symbole, welche in der Simulationseingabe verwendet werden dürfen, das ***StackAlphabet*** die Symbole für den Keller der Kellerautomaten. In den ***States*** sind alle Zustände und deren Übergänge in einem Array enthalten.

States Die Elemente des Arrays sind Zustände. Jeder dieser Zustände wird durch ein Objekt repräsentiert. Dieses besteht aus *ID*, *Name*, *x*, *y*, *Final*, *Radius*, *Transitions* und *Start*.

Die ***ID*** identifiziert den Zustand und wird in FLACI genutzt, um die Zustände in den Übergängen zu referenzieren. ***Name*** ist ein String, welcher im Graphen innerhalb des Kreises abgebildet wird. ***x*** und ***y*** beschreiben die Position des Zustandes in dem Koordinatensystem. Der Datentyp der Eigenschaften ***Final*** und ***Start*** ist der *Boolean*. Durch diese beiden Eigenschaften wird festgelegt, ob ein Zustand ein Start- und/oder ein Finalzustand ist. Die ***Transitions*** werden in einem Array dargestellt.

Transitions Das *Transitions*-Array enthält Objekte mit Informationen über die Übergänge. Ein Übergang besteht immer aus *Source*, *Target*, *x*, *y* und *Labels*.

Source ist die *ID* des Zustandes, von welchem der Übergang ausgeht, während ***Target*** die *ID* des Zustandes ist, zu welchem der Übergang hinführt. Das kann ein anderer oder derselbe Zustand sein. ***x*** und ***y*** werden für die Berechnung der Bezierkurven der Übergänge benötigt. ***Labels*** ist eine Eigenschaft, die für unterschiedliche Automatentypen unterschiedlich ist.

Labels Für DEAs und NEAs sind die Labels ein Array, bestehend aus Strings. Wenn einer dieser Strings leer ist, dann steht das für ein Epsilon. Das kann jedoch nur bei einem NEA auftreten.

Die Labels der Kellerautomaten sind, genau wie bei DEAs und NEAs, in einem Array enthalten. Jedes Element dieses Arrays ist wiederum ein Array (Subarray). An erster und zweiter Position dieses Subarrays stehen Strings. Die dritte Position enthält ein weiteres Array (Subsubarray), welches wiederum aus Strings besteht. Das Subarray der NKAs und DKAs kann leere Strings besitzen, ebenso kann das Subsubarray dieser leer sein, welche jeweils das Epsilon repräsentieren.

Eine TURING-Maschine nutzt eine weitere Methode, um Labels darzustellen. Wie bei Kellerautomaten sind auch hier Labels als Subarrays in einem Array repräsentiert. Die Subarrays besitzen drei Strings und kein Subsubarray.

Es gibt Eigenschaften, welche nicht immer in einer `JSON` von FLACI auftreten. Es wurde sich nur auf die wichtigsten, für den Compiler relevanten Eigenschaften konzentriert. Das sind:

- *Name* und *automaton* der `JSON`
- *States* des *automaton*
- *ID*, *name*, *x*, *y*, *Final*, *Start* und *Transitions* der *States*
- *Source*, *Target*, *x*, *y* und *Labels* der *Transitions*

5.4 Implementierung

5.4.1 Skalierung des Graphen

Eine wichtige Funktion, welche der Compiler erfüllen muss, ist das Skalieren des Graphen auf die Größe des jeweiligen Papierformates. Standardmäßig ist das DIN A4-Format voreingestellt. Dazu wurde eine Funktion entwickelt, welche im Anhang A.5, S. 54 abgebildet ist.

Diese Funktion bekommt als Argumente *st*, *width* und *height*. Für Zustände, die skaliert werden sollen, steht *st*. Durch *width* und *height* wird die maximale Größe des Graphen angegeben.

Zum Skalieren der Graphen, werden zuerst die Zustände geklont, um nicht auf dem Originalobjekt zu arbeiten (Z. 2-6). Dann werden die Werte auf der x-Achse so verschoben, dass der Zustand mit dem kleinsten x-Wert auf $x = 0$ liegt. Das geschieht, indem allen x-Werten der kleinste x-Wert mit -1 multipliziert hinzu addiert wird (Z. 8-21). Für die y-Werte gilt das Gleiche (Z. 23-36).

Im nächsten Schritt muss der größte x- und y-Wert gefunden werden (Z. 38-58). Für diese Werte müssen die Verhältnisse zu der jeweiligen maximalen Breite bzw. Höhe berechnet werden (Z. 60-61). Wenn diese Verhältnisse kleiner oder gleich null sind, ist der Graph innerhalb oder genau auf der maximal zulässigen Grenze für das Resultat. Wenn jedoch mindestens ein Verhältnis größer als null ist, muss der Graph abwärts skaliert werden. Da das grundlegende Aussehen des Graphen nicht verändert werden soll, sind beide Achsen gleich stark zu stauchen. Die Stärke der Stauchung beider Achsen wird dabei von der Achse vorgegeben, die stärker gestaucht werden muss. Damit kann gewährleistet werden, dass die maximale Breite und Höhe des Graphen nicht überschritten wird (Z. 63-81).

Als Ergebnis werden die Zustände in skalierter Form innerhalb der angegebenen Grenzen zurückgegeben (Z. 82).

5.4.2 Austausch des Dollarzeichens

Während der Implementierung des FLaAL-Paketes wurde festgestellt, dass einige LaTeX-Distributionen nicht mit mehreren Dollarzeichen in einem *transition*-Befehl umgehen können (s. 4.4.4). Dieses Verhalten muss auch im Compiler berücksichtigt werden. Deshalb kann den Optionen des Compilers noch ein *swapDollar* (Typ: Boolean) hinzugefügt werden, welches die automatische Ersetzung der Dollarzeichen durch Und-Zeichen veranlasst. Standardmäßig ist dieses auf `false` gestellt.

5.4.3 Konsolenanwendung

Um FFC komfortabel zu testen, wurde eine Konsolenanwendung entwickelt, welche FFC implementiert. Zur Ausführung dieser Anwendung benötigt man `NODE JS`[7]. Wenn man in den Projektordner navigiert, kann man das Programm mit `node index.js` ausführen. Zur Übersetzung von einer `JSON` nach LaTeX benötigt es Argumente, welche einen abstrakten Automaten zum Übersetzen angeben. Dafür hat man zwei Möglichkeiten.

Die erste Möglichkeit besteht darin, einen von fünf Automaten auszuwählen, welcher im Code der Anwendung enthalten ist. Dafür kann man das Flag *-a* (*automaton*) und dahinter eine Zahl von 0 bis 4 angeben. Dann wird der entsprechende Automat nach FLaAL übersetzt und eine `graph.txt` wird ausgegeben.

Die zweite Möglichkeit ist das Flag *-fn* (*filename*). Hinter diesem Flag kann man eine `JSON`-Datei angeben, welche im gleichen Verzeichnis wie die `index.js` liegt: `node index.js -fn myautomaton.json`. Abschlieend wird diese `JSON`-Datei eingelesen, vom Compiler übersetzt und eine `graph.txt` mit dem LaTeX-Code ausgegeben.

Außerdem existieren noch die Flags *-fy* (*flip Y*) und *-sd* (*swap Dollar*). Das Flag *-fy* veranlasst den Compiler alle y-Werte der Zustände zu invertieren. Durch *-sd* werden alle Dollarzeichen durch Und-Zeichen ersetzt. Da einige LaTeX-Compiler Probleme mit dem Dollarzeichen haben, kann das sinnvoll sein.

5.5 Evaluation

Genau wie FLaAL wurde FFC parallel zu der Entwicklung getestet. Die Testdaten wurden alle von FLACI direkt genommen. Dafür wurde entweder ein Automat erstellt, um bestimmte Grenzfälle zu erzeugen oder ein Beispielautomat genommen, um auch etwas größere Automaten zu testen und deren Abdeckung zu garantieren.

Die Kontrolle erfolgte einerseits durch Begutachten des ausgegebenen

[7] `NODE JS` ist eine Javascript-Laufzeitumgebung, welche Javascript ausserhalb des Browsers ausführt. `https://nodejs.org/en/`

LaTeX-Codes des Compilers, andererseits durch das Einfügen der Ausgabe in eine LaTeX-Datei mit anschließendem Übersetzen dieser. Wenn das übersetzte Resultat dann dem Erscheinungsbild in FLACI entsprach, war der Test erfolgreich.

5.6 Möglichkeiten und Grenzen von FFC

Möglichkeiten FFC ermöglicht es, abstrakte Automaten von FLACI nach FLaAL zu übersetzen.[8]

Der Compiler kann die Graphen der abstrakten Automaten skalieren und diese an der x-Achse um 180° drehen.

Durch einige LaTeX-Distributionen gesetzte Grenzen der Darstellung der Dollarzeichen können umgangen werden, da der Compiler die Dollarzeichen in den Automaten durch Und-Zeichen ersetzen kann.

Wenn gebogene Übergänge existieren, werden diese vom Compiler berücksichtigt und entsprechend nach FLaAL übersetzt.

Grenzen Da FlaAL noch keine formalen Sprachen und T-Diagramme unterstützt, können diese von FFC auch nicht übersetzt werden.

[8]FFC wurde in FLACI implementiert und kann wie folgt genutzt werden:
Zuerst muss die Bearbeitungsoberfläche eines Graphen aufgerufen werden. Im Leistenmenü oberhalb des Graphen befindet sich der Menüpunkt *konvertieren*, welchen man anklickt. Durch Auswählen des Unterpunktes *Automat für LaTeX konvertieren* bekommt man den LaTeX-Code. Diesen fügt man dann in LaTeX ein.

6 Beispiele

In den Anlagen sind die Beispiele zunächst als Graph und nachfolgend als LaTeX-Code dargestellt.

6.1 Verdopplungsmaschine

Ein Beispiel für einen mit FFC und FLaAL erstellten Automaten ist im Anhang A.6.1 (S. 57) abgebildet. Dieser Automat ist aus der Beispielsammlung von FLACI entnommen und wurde mit FFC übersetzt. In FLACI werden statt des Und-Zeichens Dollarzeichen genutzt, was in manchen LaTeX-Distributionen jedoch nicht möglich ist. Deshalb hat FFC alle Dollarzeichen durch Und-Zeichen ersetzt. Mit Hilfe des *flipY*-Flags hat der Compiler den Graphen an der x-Achse gespiegelt, sodass der Graph in LaTeX genau wie in FLACI aussieht.

6.2 Notensprache NKA

Im Beispiel *Notensprache NKA* (Anhang A.6.2, S. 59) kann man sehen, dass das Paket auch mit einer sehr großen Anzahl an Labels umgehen kann. Ab einer bestimmten Größe kann es jedoch sein, dass der Graph nicht mehr komplett auf eine Seite passt oder die Seitenzahl überdeckt bzw. innerhalb des Graphen liegt. Auch hier wurden die Dollarzeichen durch Und-Zeichen, sowie Bindestriche durch den Ausdruck `$\relbar$` ersetzt.

6.3 DEA

Das DEA Beispiel (Anhang A.6.3, S. 61) zeigt einen endlichen Automaten, welcher komplett in LaTeX erstellt wurde, d.h. er wurde nicht von FFC aus FLACI übersetzt. Dieser Graph zeigt, wie Startzustände, welche gleichzeitig Finalzustände sind, in FLaAL dargestellt werden. Auch die gebogenen Übergänge, welche zwischen *q0* und *q1* auftreten, werden repräsentiert. Ebenfalls kann man sehen, dass Labels, welche in den *transition*-Befehlen mehrfach vorkommen, ignoriert werden.

7 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, eine Sprache FLaAL zu entwickeln, welche die Erstellung von Darstellungsobjekten der theoretischen Informatik in LaTeX abstrahiert.

Im Vordergrund stand die Anwenderfreundlichkeit, d.h. der Nutzer sollte seinen Fokus hauptsächlich auf den Inhalt und weniger auf die Erstellung des Graphen richten. Auch die Syntax eines Paketes sollte nicht zu kompliziert sein, um den Nutzer nicht vom Inhalt abzulenken.

Das Ergebnis dieser Arbeit ist ein Werkzeug, welches auf einfache Art und Weise abstrakte Automaten erzeugen kann. Man kann Zustände und Übergänge mit beliebiger Anzahl an Labels erstellen.

Jedoch besitzt dieses Paket einige Grenzen. Mit der entwickelten Syntax ist es nicht möglich, eine automatische Positionierung der Zustände sinnvoll durchzuführen. Um das zu ermöglichen, müsste eine komplexere Syntax gewählt werden. Auch der begrenzte Platz eines Papierformats kann Probleme bereiten. Außerdem kann FLaAL bis jetzt nur die abstrakten Automaten abdecken.

Nachteilig ist, dass die Wartbarkeit des Paketes kaum gewährleistet ist, da schon kleinste komplexe Aufgaben unübersichtlichen Code erzeugen.

Abschließend ist festzuhalten, dass die Erstellung von abstrakten Automaten mit LaTeX, meiner Auffassung nach, schwerer ist als mit grafischen Werkzeugen wie FLACI. Das liegt vor allem an der Syntax von LaTeX, die gleich bleibt, auch wenn die Befehle stark abstrahiert werden. Dagegen können in FLACI erstellte Graphen mit Hilfe von FFC leicht nach FLaAL übersetzt und dort durch die abstrakte Syntax von FLaAL einfach angepasst werden.

Der nächste Schritt für das Paket wäre beispielsweise, ein neues Darstellungsobjekt zu implementieren. Dabei könnte mit den T-Diagrammen begonnen werden, da diese von der Konstruktion her ähnlich zu den Automaten sind. Alternativ kann man die Implementierung der vollautomatischen Positionierung der Zustände der abstrakten Automaten in Angriff nehmen.

8 Quellen

Das FLaAL-Paket und der FF-Compiler werden auf Github gehostet: https://github.com/TrueRushHunt3r/FLaAL.

Literatur

[Pa06] Partosch, G.: Einfache Makros in TEX und LATEX, 2006, URL: https://www.staff.uni-giessen.de/partosch/unterlagen/Makros.pdf.

[di07] dictionary.com: Wysiwyg | Definition of Wysiwyg at Dictionary.com, 2007, URL: https://www.dictionary.com/browse/wysiwyg.

[to10] topskip: Why do LaTeX internal commands have an @ in them?, 2010, URL: https://tex.stackexchange.com/questions/6240/why-do-latex-internal-commands-have-an-in-them.

[WH14] Wagenknecht, C.; Hielscher, M.: Formale Sprachen, abstrakte Automaten und Compiler. Springer, 2014.

[Ta15] Tantau, T.: TikZ and PGF: Manual for version 3.0. 1a, 2015.

[ja18] jackwootton: JSON Schema Tool, 2018, URL: https://jsonschema.net.

[ov19] overleaf: Writing your own package, 2019, URL: https://de.overleaf.com/learn/latex/Writing_your_own_package.

[19] TEX Resources on the Web, 2019, URL: http://www.tug.org/interest.html#free.

[Wi19] Wikibooks: LaTeX/Macros, 2019, URL: https://en.wikibooks.org/wiki/LaTeX/Macros.

[CT] CTAN: Was sind TEX und seine Freunde?, URL: https://www.ctan.org/tex/.

[Gr] Graz, T.: Anwendung für Fortgeschrittene, URL: https://latex.tugraz.at/latex/fortgeschrittene?s[]=newcommand.

[pg] pgf-tikz: pgf – A Portable Graphic Format for TeX, URL: https://github.com/pgf-tikz/pgf.

[WH] Wagenknecht, C.; Hielscher, M.: LL(k) Sprachen und LL(1) Forderungen, URL: https://web1.hszg.de/cwagenknecht-lehre/TI/SAuC/ADuTI2/Woche5/5_Slides.pdf.

[Wr] Wright, A.; Andrews, H.; Hutton, B.; Dennis, G.: JSON Schema: A Media Type for Describing JSON Documents, URL: https://json-schema.org/draft/2019-09/json-schema-core.html.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Cottbus, den 24. November 2019

_____

Jonas Kappa

A Anhang

A.1 Definitionen der Darstellungsobjekte

A.1.1 DEA

„Ein [deterministischer endlicher Automat (DEA)] (auch EA) ist ein Quintupel $M = (Q, \sum, \delta, q_0, E)$, mit

Q ... endliche Menge der Zustände,
$\sum$ ... Eingabealphabet, $Q \cap \sum = \emptyset$,
δ ... Überführungsfunktion (*totale* Funktion), $Q \times \sum \to Q$,
q_0 ... Startzustand, $q_0 \in Q$, und
E ... endliche (nichtleere) Menge der Endzustände, $E \subseteq Q$.

“([WH14], Definition 6.1, S. 63)

A.1.2 NEA

„Ein nichtdeterministischer endlicher Automat, kurz: NEA, wird durch ein Quintupel $M = (Q, \sum, \delta, q_0, E)$ definiert, wobei bis auf δ die Bedeutungen der Symbole aus der Definition des DEA übernommen werden. Die Überführungsdefinition eines NEA ist wie folgt definiert:

$$\delta : Q \times E \to \mathcal{P}(Q)$$

“([WH14], Definition 6.2, S. 71)
Daraus folgt:

Q ... endliche Menge der Zustände,
$\sum$ ... Eingabealphabet, $Q \cap \sum = \emptyset$,
δ ... Überführungsfunktion, $Q \times E \to \mathcal{P}(Q)$,
q_0 ... Startzustand, $q_0 \in Q$, und
E ... endliche (nichtleere) Menge der Endzustände, $E \subseteq Q$.

A.1.3 NKA

„Ein nichtdeterministischer Kellerautomat, kurz: NKA oder KA, [...] ist durch ein 7-Tupel $M = (Q, \sum, \Gamma, \delta, q_0, k_0, E)$ definiert. Die verwendeten

Symbole haben folgende Bedeutungen:

Q	...	endliche Menge der Zustände
$\sum$	...	Eingabealphabet
Γ	...	Kelleralphabet
δ	...	partielle Überführungsfunktion
q_0	...	Anfangszustand, $q_0 \in Q$
k_0	...	Kellervorbelegungszeichen, $k_0 \in \Gamma$
E	...	Menge von Endzuständen, $E \subseteq Q$

"([WH14], Definition 8.1, S. 130)
Ein NKA kann auch durch ein 6-Tupel definiert werden. Dabei wird die Menge der Endzustände weggelassen, sodass man $M = (Q, \sum, \Gamma, \delta, q_0, k_0)$ erhält. Ein 6-Tupel-NKA akzeptiert ein Eingabewort, wenn das Wort vollständig gelesen wurde und der Keller des Automaten leer ist.
([WH14], Abschnitt 6, S. 134)
Mit FlaAL können beide Definitionen von NKAs dargestellt werden.

A.1.4 DKA

„Ein [deterministischer Kellerautomat (DKA)] ist wie ein NKA definiert. Allerdings gibt es drei Abweichungen:

1. $\delta : Q \times (\sum \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$, anstelle von $\mathcal{P}_{endlich}(Q \times \Gamma^*)$ auf der rechten Seite bei NKA. Die Funktionswerte sind also Paare und keine Mengen.

2. Wenn $\delta(q, \varepsilon, A)$ definiert ist, dann ist für alle $a \in \sum$ der Funktionswert $\delta(q, a, A)$ undefiniert oder umgekehrt.

3. Es gibt eine Menge von Endzuständen $E \subseteq Q$, d.h. es wird immer die NKA-Definitionsform als 7-Tupel $M = (Q, \sum, \Gamma, \delta, q_0, k_0, E)$ zugrunde gelegt. Die bei NKA zulässige 6-Tupel-Definition darf für DKA nicht verwendet werden.

"([WH14], Definition 8.2, S. 148)

A.1.5 TM

„Eine TURING-Maschine (TM) ist ein 7-Tupel $M = (Q, \sum, \Gamma, \delta, q_0, \$, E)$.

Q	...	endliche Menge von Zuständen
$\sum$	...	Eingabealphabet
Γ	...	Bandalphabet, wobei $\sum \subseteq \Gamma \setminus \{\$\}$
δ	...	partielle Überführungsfunktion: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, N, R\}$. Für eine nichtdeterministische TM - kurz: NTM - gilt: $Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, N, R\})$.
q_0	...	Anfangszustand, $q_0 \in Q$
$\$$	...	Bandvorbelegungszeichen, kurz: Blankzeichen, mit $\$ \in \Gamma \setminus \sum$
E	...	endliche Menge von Endzuständen, mit $E \subseteq Q$

“([WH14], Definition 12.1, S. 227)

A.2 Kontextfreie Grammatik FLaAL

```
1  FLaAL                              -> transitiongraph
2  transitiongraph                    -> \begin{transitiongraph}
                  aTypeOptional tgspecification
                  \end{transitiongraph}
3  aTypeOptional                      -> [ aType ] | EPSILON
4  aType                              -> fa | pa | tm
5  tgspecification                    -> state
                  tgspecificationTransitionsEnabled | EPSILON
6  tgspecificationTransitionsEnabled -> state
                  tgspecificationTransitionsEnabled | transition
                  tgspecificationStatesDisabled | EPSILON
7  tgspecificationStatesDisabled      -> transition
                  tgspecificationStatesDisabled | EPSILON
8  state                              -> \state stateTypeOptional {
                  stateName } { number } { number }
9  stateTypeOptional                  -> [ stateType ] | EPSILON
10 stateType                          -> n | s | f | sf
11 stateName                          -> charLettersOnly string
12 transition                         -> \transition { stateName } {
                  stateName } { Labels }
13 Labels                             -> string continueLabel |
                  EPSILON
14 continueLabel                      -> ; string continueLabelFA |
                  , string , string continueLabelPATM
15 continueLabelFA                    -> ; string continueLabelFA |
                  EPSILON
16 continueLabelPATM                  -> ; string , string , string
                  continueLabelPATM | EPSILON
17 string                             -> char string | EPSILON
18 char                               -> charLettersOnly | 0 | 1 | 2
                  | 3 | 4 | 5 | 6 | 7 | 8 | 9 | - | \$ |
                  dollarEnvironment
19 charLettersOnly                    -> 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 | 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
20 dollarEnvironment                  -> $ dollarCommand $
```

```
21 dollarCommand                    -> \mid | \varepsilon |
                    \epsilon
22 number                           -> negative digit
                    numberContines
23 numberContines                   -> digit numberContines | .
                    numberAfterPoint | EPSILON
24 numberAfterPoint                 -> digit numberAfterPoint |
                    EPSILON
25 digit                            -> 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                    7 | 8 | 9
26 negative                         -> - | EPSILON
```

A.3 JSON-Datei von FLACI über Turingmaschinen

```
definitions: {}
$schema: http://json-schema.org/draft-07/schema#
$id: http://example.com/root.json
type: object
title: The Root Schema
required:
- name
- description
- type
- automaton
properties:
  name:
    $id: '#/properties/name'
    type: string
    title: The Name Schema
    default: ''
    examples:
    - TMGraph
    pattern: ^(.*)$
  description:
    $id: '#/properties/description'
    type: string
    title: The Description Schema
    default: ''
    examples:
    - ''
    pattern: ^(.*)$
  type:
    $id: '#/properties/type'
    type: string
    title: The Type Schema
    default: ''
    examples:
    - TM
    pattern: ^(.*)$
  automaton:
    $id: '#/properties/automaton'
    type: object
```

```
39      title: The Automaton Schema
40      required:
41      - simulationInput
42      - Alphabet
43      - StackAlphabet
44      - States
45      properties:
46        simulationInput:
47          $id: '#/properties/automaton/properties/simulationInput'
48          type: array
49          title: The Simulationinput Schema
50          items:
51            $id: '#/properties/automaton/properties/simulationInput
                  /items'
52            type: string
53            title: The Items Schema
54            default: ''
55            examples:
56            - '1'
57            - '1'
58            - '1'
59            pattern: ^(.*)$
60        Alphabet:
61          $id: '#/properties/automaton/properties/Alphabet'
62          type: array
63          title: The Alphabet Schema
64          items:
65            $id: '#/properties/automaton/properties/Alphabet/items'
66            type: string
67            title: The Items Schema
68            default: ''
69            examples:
70            - '1'
71            pattern: ^(.*)$
72        StackAlphabet:
73          $id: '#/properties/automaton/properties/StackAlphabet'
74          type: array
75          title: The Stackalphabet Schema
76          items:
```

```
77          $id: '#/properties/automaton/properties/StackAlphabet/
                items'
78          type: string
79          title: The Items Schema
80          default: ''
81          examples:
82          - $
83          - '1'
84          pattern: ^(.*)$
85        States:
86          $id: '#/properties/automaton/properties/States'
87          type: array
88          title: The States Schema
89          items:
90            $id: '#/properties/automaton/properties/States/items'
91            type: object
92            title: The Items Schema
93            required:
94            - ID
95            - Name
96            - x
97            - y
98            - Final
99            - Radius
100           - Transitions
101           - Start
102           properties:
103             ID:
104               $id: '#/properties/automaton/properties/States/
                      items/properties/ID'
105               type: integer
106               title: The Id Schema
107               default: 0
108               examples:
109               - 1
110             Name:
111               $id: '#/properties/automaton/properties/States/
                      items/properties/Name'
112               type: string
```

```
113                 title: The Name Schema
114                 default: ''
115                 examples:
116                 - q0
117                 pattern: ^(.*)$
118               x:
119                 $id: '#/properties/automaton/properties/States/
                        items/properties/x'
120                 type: integer
121                 title: The X Schema
122                 default: 0
123                 examples:
124                 - 140
125               y:
126                 $id: '#/properties/automaton/properties/States/
                        items/properties/y'
127                 type: integer
128                 title: The Y Schema
129                 default: 0
130                 examples:
131                 - 110
132               Final:
133                 $id: '#/properties/automaton/properties/States/
                        items/properties/Final'
134                 type: boolean
135                 title: The Final Schema
136                 default: false
137                 examples:
138                 - false
139               Radius:
140                 $id: '#/properties/automaton/properties/States/
                        items/properties/Radius'
141                 type: integer
142                 title: The Radius Schema
143                 default: 0
144                 examples:
145                 - 30
146               Transitions:
```

```
147                 $id: '#/properties/automaton/properties/States/
                        items/properties/Transitions'
148                 type: array
149                 title: The Transitions Schema
150                 items:
151                   $id: '#/properties/automaton/properties/States/
                          items/properties/Transitions/items'
152                   type: object
153                   title: The Items Schema
154                   required:
155                   - Source
156                   - Target
157                   - x
158                   - y
159                   - Labels
160                   properties:
161                     Source:
162                       $id: '#/properties/automaton/properties/
                              States/items/properties/Transitions/items
                              /properties/Source'
163                       type: integer
164                       title: The Source Schema
165                       default: 0
166                       examples:
167                       - 1
168                     Target:
169                       $id: '#/properties/automaton/properties/
                              States/items/properties/Transitions/items
                              /properties/Target'
170                       type: integer
171                       title: The Target Schema
172                       default: 0
173                       examples:
174                       - 2
175                     x:
176                       $id: '#/properties/automaton/properties/
                              States/items/properties/Transitions/items
                              /properties/x'
177                       type: integer
```

```
178                     title: The X Schema
179                     default: 0
180                     examples:
181                     - 0
182                   y:
183                     $id: '#/properties/automaton/properties/
                            States/items/properties/Transitions/items
                            /properties/y'
184                     type: integer
185                     title: The Y Schema
186                     default: 0
187                     examples:
188                     - 0
189                   Labels:
190                     $id: '#/properties/automaton/properties/
                            States/items/properties/Transitions/items
                            /properties/Labels'
191                     type: array
192                     title: The Labels Schema
193                     items:
194                       $id: '#/properties/automaton/properties/
                              States/items/properties/Transitions/
                              items/properties/Labels/items'
195                       type: array
196                       title: The Items Schema
197                       items:
198                         $id: '#/properties/automaton/properties/
                                States/items/properties/Transitions/
                                items/properties/Labels/items/items'
199                         type: string
200                         title: The Items Schema
201                         default: ''
202                         examples:
203                         - '1'
204                         - $
205                         - R
206                         pattern: ^(.*)$
207             Start:
```

```
208                 $id: '#/properties/automaton/properties/States/
                        items/properties/Start'
209                 type: boolean
210                 title: The Start Schema
211                 default: false
212                 examples:
213                 - true
```

A.4 FLaAL Code

FA-Label-Verarbeitung

```
% Label Processing for
% Finite Automata
\ExplSyntaxOn

\NewDocumentCommand{\fa@label@processing}{O{;}m}
{
    \fa_make:nn {#1} {#2}
}

\cs_new_protected:Npn \fa_make:nn #1 #2
{
    \seq_set_split:Nnn \l_fa_args_seq { #1 } { #2 }
    \seq_pop_left:NN \l_fa_args_seq \l_fa_temp_tl
    \StrLen{\l_fa_temp_tl}[\falength]
    \seq_clear:N \l_fa_used_labels
    \ifthenelse{\equal{\falength}{0}}{
        $\varepsilon$
    }{
        \seq_put_right:NV \l_fa_used_labels \l_fa_temp_tl
        \l_fa_temp_tl
    }
    \seq_map_inline:Nn \l_fa_args_seq {
        \tl_set:Nn \tl_fa_ele_in_list {false}
        \seq_map_inline:Nn \l_fa_used_labels {
            \ifthenelse{\equal{##1}{####1}}{
                \tl_set:Nn \tl_fa_ele_in_list {true}
            }{}
        }
        \ifthenelse{\equal{false}{\tl_fa_ele_in_list}}{
            ,\space ##1
            \seq_put_right:Nx \l_fa_used_labels {##1}
        }{}
    }
}

\tl_new:N \tl_fa_ele_in_list
```

```
\seq_new:N \l_fa_used_labels
\seq_new:N \l_fa_args_seq
\tl_new:N \l_fa_temp_tl

\ExplSyntaxOff
```

PA-Label-Verarbeitung

```
% Label Processing for
% Pushdown Automata
\ExplSyntaxOn

\NewDocumentCommand{\pa@label@processing}{O{;}m}
{
    \pa_make:nn {#1} {#2}
}

\cs_new_protected:Npn \pa_make:nn #1 #2
{
    % clearing the used labels for new use
    \seq_clear:N \l_pa_used_labels
    % split by semicolon
    \seq_set_split:Nnn \l_pa_args_seq { #1 } { #2 }
    % pop first element
    \seq_pop_left:NN \l_pa_args_seq \l_pa_temp_tl
    % put the label in the used labels
    \seq_put_right:NV \l_pa_used_labels \l_pa_temp_tl
    % split first element by comma
    \seq_set_split:NnV \l_pa_ele_args_seq {,} {\l_pa_temp_tl}
    % pop first element
    \seq_pop_left:NN \l_pa_ele_args_seq \l_pa_ele_first_tl
    % pop second element
    \seq_pop_left:NN \l_pa_ele_args_seq \l_pa_ele_second_tl
    \StrLen{\l_pa_ele_second_tl}[\paseclength]
    % pop third element
    \seq_pop_left:NN \l_pa_ele_args_seq \l_pa_ele_third_tl
    \StrLen{\l_pa_ele_third_tl}[\pathirdlength]
    % result is (first,second):third
    (\l_pa_ele_first_tl,\int_compare:nTF {\paseclength=0}{$\
        varepsilon$}{\l_pa_ele_second_tl}):\int_compare:nTF {\
```

```
        pathirdlength=0}{$\varepsilon$}{\l_pa_ele_third_tl}
32
33      % clearing the processed list, to be ready to be filled again
34      \seq_clear:N \l_pa_proc_args
35      % going trough all remaining labels
36      \seq_map_inline:Nn \l_pa_args_seq {
37          % set the pa element in list flag to false
38          \tl_set:Nn \tl_pa_ele_in_list {false}
39          % search for matching, already used label
40          \seq_map_inline:Nn \l_pa_used_labels {
41              \ifthenelse{\equal{##1}{####1}}{
42                  \tl_set:Nn \tl_pa_ele_in_list {true}
43              }{}
44          }
45          % if element is not in List add it and continue with
                default procedure
46          \ifthenelse{\equal{false}{\tl_pa_ele_in_list}}{
47              % isn't in use already so add to the used label list
48              \seq_put_right:Nx \l_pa_used_labels {##1}
49              % continue with standard procedure
50              % split first element by comma
51              \seq_set_split:Nnn \l_pa_ele_args_seq {,} {##1}
52              % pop first element
53              \seq_pop_left:NN \l_pa_ele_args_seq \l_pa_ele_first_
                    tl
54              % pop second element
55              \seq_pop_left:NN \l_pa_ele_args_seq \l_pa_ele_second_
                    tl
56              \StrLen{\l_pa_ele_second_tl}[\paseclength]
57              % pop third element
58              \seq_pop_left:NN \l_pa_ele_args_seq \l_pa_ele_third_
                    tl
59              \StrLen{\l_pa_ele_third_tl}[\pathirdlength]
60              % adding labels to the proccessed labels
61              \seq_put_right:Nx \l_pa_proc_args {(\l_pa_ele_first_
                    tl,\int_compare:nTF {\paseclength=0}{$\varepsilon
                    $}{\l_pa_ele_second_tl} ):\int_compare:nTF {\
                    pathirdlength=0}{$\varepsilon$}{\l_pa_ele_third_
                    tl}}
```

```
62          }{}
63      }
64      % print it below each other
65      \seq_map_inline:Nn \l_pa_proc_args {\\##1}
66
67  }
68
69  \tl_new:N \tl_pa_ele_in_list
70  \seq_new:N \l_pa_used_labels
71
72  \seq_new:N \l_pa_ele_args_seq
73  \seq_new:N \l_pa_args_seq
74  \seq_new:N \l_pa_proc_args
75  \tl_new:N \l_pa_temp_tl
76  \tl_new:N \l_pa_ele_first_tl
77  \tl_new:N \l_pa_ele_second_tl
78  \tl_new:N \l_pa_ele_third_tl
79
80  \ExplSyntaxOff
```

A.5 FFC Code

Skalierungsfunktion

```
function scale(st, width, height) {
    let states = [];
    // clone the objects
    for (const state of st) {
        states.push(clone(state));
    }
    // shift all states on x axes so most left is on x=0
    let minX = null;
    for (const s of states) {
        if (!minX) {
            minX = s;
            continue;
        }
        if (s.x < minX.x) {
            minX = s;
        }
    }
    let val = minX.x;
    for (let s of states) {
        s.x += val * -1;
    }
    // shift all states on y axes so most bottom ist on y = 0
    let minY = null;
    for (const s of states) {
        if (!minY) {
            minY = s;
            continue;
        }
        if (s.y < minY.y) {
            minY = s;
        }
    }
    val = minY.y;
    for (let s of states) {
        s.y += val * -1;
    }
```

```
37      // find max X value
38      let maxX = null;
39      for (const s of states) {
40          if (!maxX) {
41              maxX = s;
42              continue;
43          }
44          if (s.x > maxX.x) {
45              maxX = s;
46          }
47      }
48      // find max Y value
49      let maxY = null;
50      for (const s of states) {
51          if (!maxY) {
52              maxY = s;
53              continue;
54          }
55          if (s.y > maxY.y) {
56              maxY = s;
57          }
58      }
59
60      let xratio = (maxX.x - width) / width;
61      let yratio = (maxY.y - height) / height;
62
63      if (xratio > 0 || yratio > 0) {
64          if (xratio >= yratio) {
65              let scaler = maxX.x;
66              for (const s of states) {
67                  let p = (s.x * 100) / scaler;
68                  s.x = Math.round(((width * p) / 100) * 1000) /
                        1000;
69                  p = (s.y * 100) / scaler;
70                  s.y = Math.round(((width * p) / 100) * 1000) /
                        1000;
71              }
72          } else {
73              let scaler = maxY.y;
```

```
74              for (const s of states) {
75                  let p = (s.x * 100) / scaler;
76                  s.x = Math.round(((height * p) / 100) * 1000) /
                        1000;
77                  p = (s.y * 100) / scaler;
78                  s.y = Math.round(((height * p) / 100) * 1000) /
                        1000;
79              }
80          }
81      }
82      return states;
83  }
```

A.6 Beispiele

A.6.1 Verdopplungsmaschine

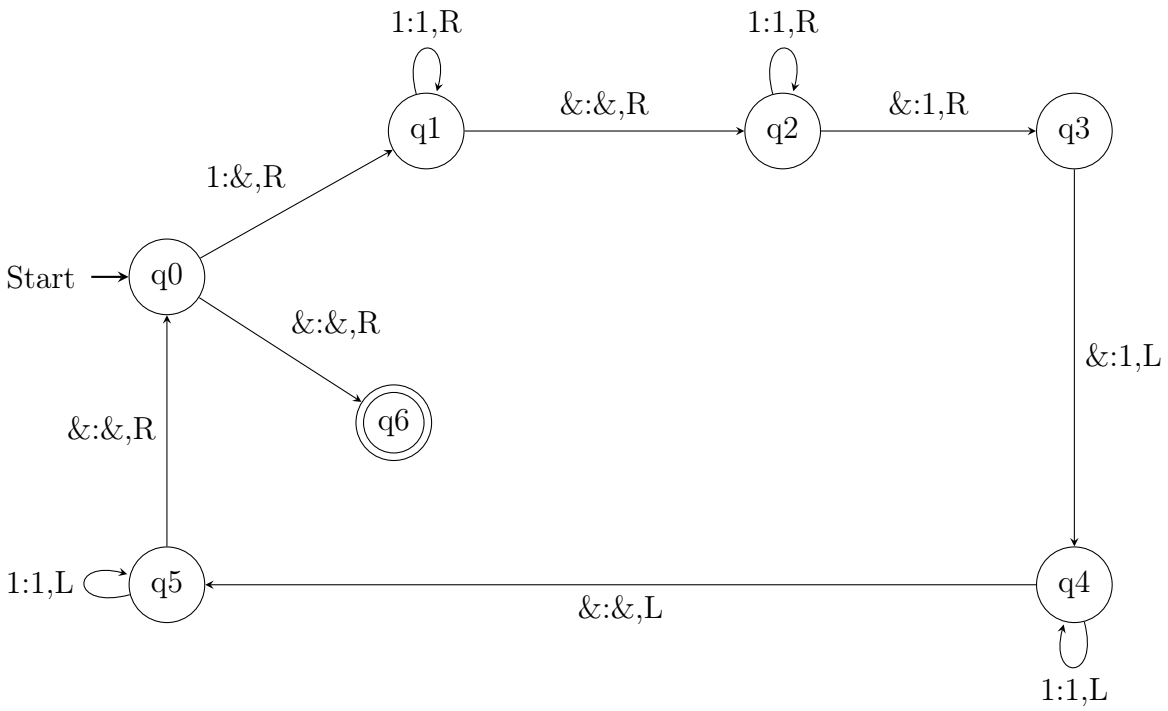

Abbildung 3: Verdopplungsmaschine

Verdopplungsmaschine LaTeX Code

```
\begin{figure}
    \centering
    \begin{transitiongraph}[tm]
        \state[s]{q0}{0}{-19.286}
        \state{q3}{120}{0}
        \state[f]{q6}{30}{-38.571}
        \state{q5}{0}{-60}
        \state{q1}{34.286}{0}
        \state{q4}{120}{-60}
        \state{q2}{81.429}{0}
        \transition{q0}{q1}{1,\&,R}
        \transition{q0}{q6}{\&,\&,R}
```

```
13          \transition{q3}{q4}{\&,1,L}
14          \transition[line=left]{q5}{q5}{1,1,L}
15          \transition{q5}{q0}{\&,\&,R}
16          \transition{q1}{q1}{1,1,R}
17          \transition{q1}{q2}{\&,\&,R}
18          \transition{q4}{q5}{\&,\&,L}
19          \transition[line=bot]{q4}{q4}{1,1,L}
20          \transition{q2}{q2}{1,1,R}
21          \transition{q2}{q3}{\&,1,R}
22      \end{transitiongraph}
23      \caption{Verdopplungsmaschine}
24      \label{graph:Verdopplungsmaschine}
25  \end{figure}
```

A.6.2 Notensprache NKA

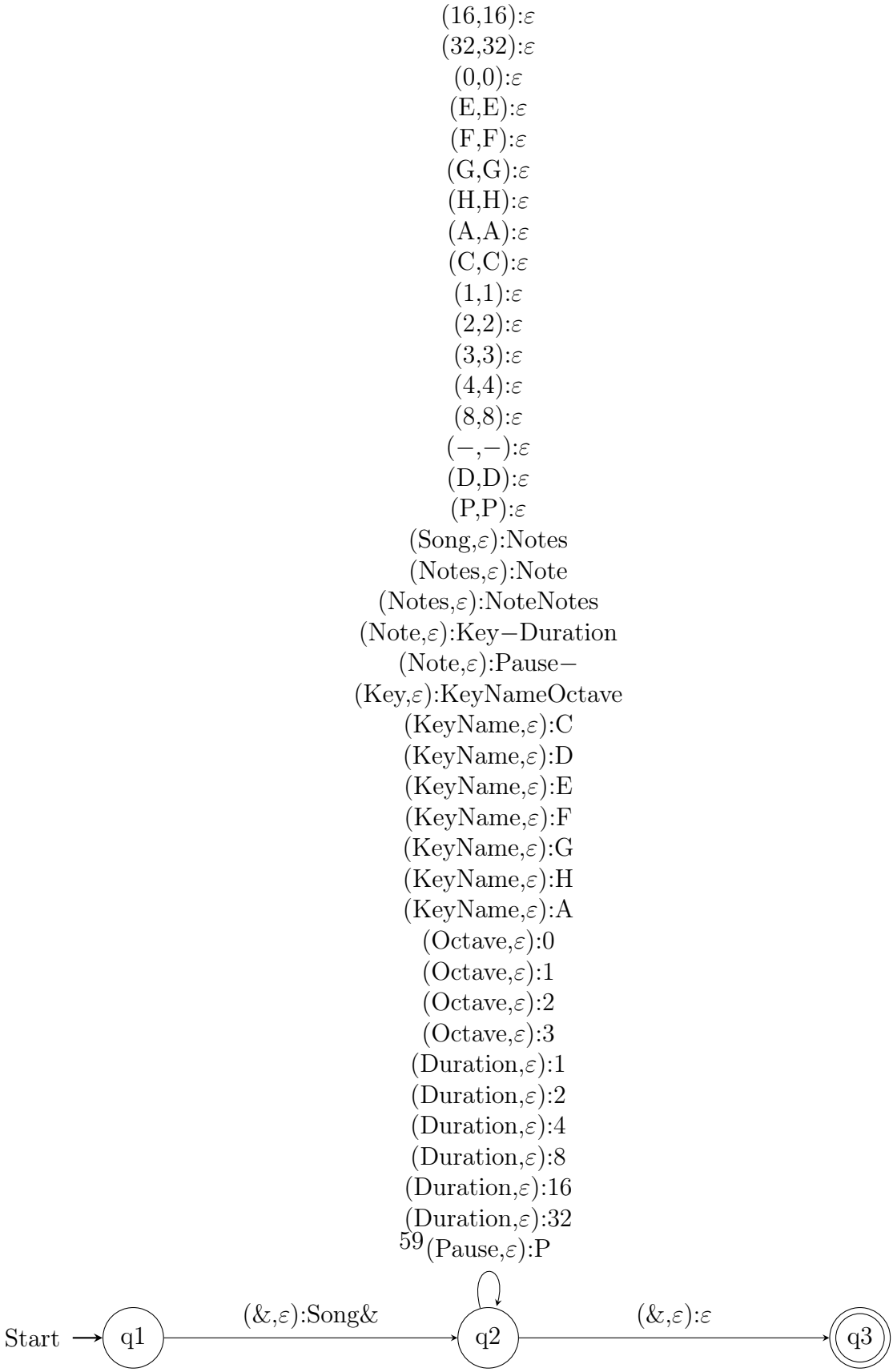

Abbildung 4: Notensprache NKA

Notensprache LaTeX Code

```
\begin{figure}
    \centering
    \begin{transitiongraph}[pa]
        \state[s]{q1}{0}{0}
        \state[f]{q3}{120}{0}
        \state{q2}{58.605}{0}
        \transition{q1}{q2}{\&,,Song\&}
        \transition{q2}{q3}{\&,,}
        \transition{q2}{q2}{16,16,;32,32,;0,0,;E,E,;F,F,;G,G,;H,H
            ,;A,A,;C,C,;1,1,;2,2,;3,3,;4,4,;8,8,;$\relbar$,$\
            relbar$,;D,D,;P,P,;Song,,Notes;Notes,,Note;Notes,,
            NoteNotes;Note,,Key$\relbar$Duration;Note,,Pause$\
            relbar$;Key,,KeyNameOctave;KeyName,,C;KeyName,,D;
            KeyName,,E;KeyName,,F;KeyName,,G;KeyName,,H;KeyName,,
            A;Octave,,0;Octave,,1;Octave,,2;Octave,,3;Duration
            ,,1;Duration,,2;Duration,,4;Duration,,8;Duration,,16;
            Duration,,32;Pause,,P}
    \end{transitiongraph}
    \caption{Notensprache NKA}
    \label{graph:Notensprache_NKA}
\end{figure}
```

A.6.3 DEA

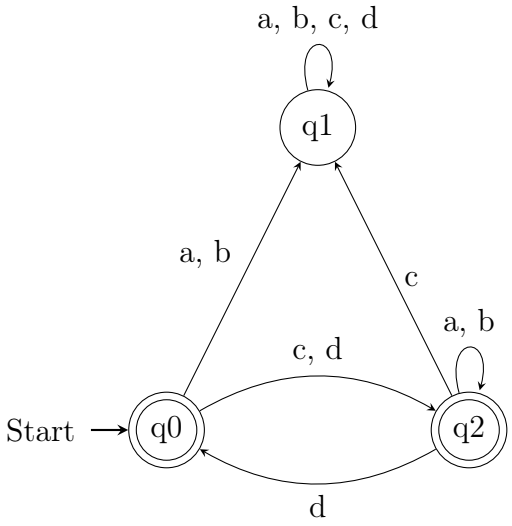

Abbildung 5: DEA

DEA LaTeX Code

```
\begin{figure}
    \centering
    \begin{transitiongraph}[fa]
        \state[sf]{q0}{0}{0}
        \state{q1}{20}{40}
        \state[f]{q2}{40}{0}

        \transition{q0}{q1}{a;b}
        \transition[line=left]{q0}{q2}{c;d}
        \transition[line=left]{q2}{q0}{d;d;d;d}
        \transition{q1}{q1}{a;b;a;c;d;a;b;c;d}
        \transition{q2}{q2}{a;b}
        \transition[label=right]{q2}{q1}{c}
    \end{transitiongraph}
    \caption{DEA}
    \label{graph:dea}
\end{figure}
```