Bachelorarbeit

E-Learning-Unterstützung für Theoretische Informatik

Eine Anwendung zur Visualisierung der Konzepte regulärer Sprachmodelle

von
Tom Kranz

Betreuung
Prof. Dr. Christoph Kreitz
Lehrstuhl „Theoretische Informatik“
Prof. Dr.-Ing. habil. Ulrike Lucke
Lehrstuhl „Komplexe Multimediale Anwendungsarchitekturen“

Institut für Informatik und Computational Science
Universität Potsdam

19. November 2018

Zusammenfassung

Es wurde festgestellt, dass Studierende der Informatik besondere Probleme mit Lehrveranstaltungen zur theoretischen Informatik haben. In dieser Arbeit wird eine computergestützte Lernanwendung konzipiert und ein Prototyp vorgestellt, mithilfe derer Studierenden die Konzepte regulärer Sprachformalismen nähergebracht werden sollen. Dazu werden die Probleme im Kontext aktueller Forschung eingeordnet und unterschiedliche Lösungsansätze betrachtet. Dabei werden auch mögliche Synergien mit Lernanwendungen aufgezeigt. Danach werden bereits existierende Anwendungen vorgestellt und ihre Eignung für den gewünschten Einsatzzweck geprüft. Anschließend werden die Ziele der hier beschriebenen Anwendung mit Rücksicht auf Bedürfnisse von Lernenden und Lehrenden geschärft. Insbesondere werden die verschiedenen Kompetenzen zusammengetragen, die aktuell durch Übungsaufgaben gefestigt werden sollen und in einer Lernanwendung abgebildet werden müssten. Darauf aufbauend wird die Implementierung der Anwendung konzipiert und die prototypische Webanwendung vorgestellt. Diese umfasst eine C++-Bibliothek, die mittels emscripten und WebAssembly in den JavaScript-Code integriert ist, und drei Aktivitäten: Eine zum Veranschaulichen der Zugehörigkeit eines Wortes zur Sprache eines endlichen Automaten, eine zum Konstruieren eines deterministischen endlichen Automaten nach Vorgabe einer Sprachspezifikation und eine zur Transformation endlicher Automaten in reguläre Ausdrücke. Schließlich werden verbleibende Unzulänglichkeiten des Prototypen angesprochen.

Inhaltsverzeichnis

1 Theoretische Informatik hat ein Imageproblem

Als Tutor und Kommilitone hat der Autor dieser Arbeit die Erfahrung gemacht, dass Studierende der informatiknahen Fachrichtungen das Erlernen der Grundlagen der theoretischen Informatik mehrheitlich eher als Last denn als Bereicherung empfinden. Dies spiegelt sich im Lernerfolg wider, der vom Lehrpersonal nur vereinzelt festgestellt wird. Sowohl im persönlichen Austausch während der Übungen als auch in der Auswertung schriftlicher Tests lässt sich bei vielen Studierenden kaum Verständnis auch nur einfachster Zusammenhänge feststellen.

Die Ergründung der Ursachen dafür ist Gegenstand aktueller Forschung. So wird in [KAPG07] der Einfluss des Abstraktionsgrads der vermittelten Fähigkeiten untersucht, indem als Alternative zu klassischen Lehrmethoden ein Ansatz präsentiert wird, in dem die Anwendung der Modellierfähigkeiten an einem praktischen Beispiel, der Erstellung eines Spiels, im Vordergrund steht. Ein vorläufiges Ergebnis dieser Untersuchung ist eine erhöhte Motivation im klassischen Umfeld leistungsschwächerer Teilnehmer und Hinweise darauf, dass klassisch stärkere Teilnehmer den Ansatz ablehnen könnten. In [BK18] werden das Verständnis und die Anwendung bestimmter Formalismen als ein Faktor dafür behandelt, dass Studierenden das Beweisen schwerfällt und ein Ansatz untersucht, der das Erlernen von Beweisfähigkeiten durch einen rigoros formalen – und maschinell verifizierbaren – Einstieg und einen schrittweisen Übergang zu informaleren Argumentationsketten erleichtern soll. In einer kleinen Gruppe wurde dabei bei der Hälfte der Teilnehmer eine Verbesserung der Fähigkeiten festgestellt. Auch das in [Rob10] begründete Modell des *Learning-Edge-Momentums*[1] bildet einen Teil der Motivation für diese Arbeit. Das Modell erklärt, vereinfacht gesagt, die Neigung der Prüfungsergebnisse von Informatikkursen zu bimodalen Verteilungen dadurch, dass die Menge an Zusammenhängen in der Informatik im Vergleich zu Fächern mit normalverteilten Prüfungsergebnissen besonders hoch ist. Studierende erhalten entweder den „Schwung“ beim Begreifen der Konzepte aufrecht, um die Prüfung überdurchschnittlich gut zu bewältigen oder verlieren aber durch Nichtverstehen einzelner Konzepte so viel „Schwung“, dass sie Zusammenhänge nicht mehr durchdringen können und schließlich komplett versagen, wodurch mittelmäßige Ergebnisse dann zu Ausreißern werden.

[Zuk16] nennt als grundlegende Konzepte in Theoretische-Informatik-Kursen Objekte zur Beschreibung formaler Sprachen – Grammatiken, reguläre Ausdrücke und Automaten. Mangelhaftes Verständnis für deren Struktur und Arbeitsweise ist ein starker Reibungspunkt, an dem Lernenden der Schwung für darauf aufbauende Konzepte ver-

[1] Frei übersetzt: Wissenserwerbsschwung

lorengehen kann. Es erscheint daher sinnvoll, den Umgang mit diesen Konzepten so intuitiv wie möglich zu gestalten, um vor allem Studienanfängern den Einstieg in die theoretische Informatik zu erleichtern, die zwar Potential für Verständnis aufweisen, aber von der Abstraktheit der Präsentation zunächst abgeschreckt werden. Vor allem bei regulären Sprachmodellen bietet sich dabei Computerunterstützung an, da zusätzlich zu neuen Formen der Präsentation bei vielen Aufgaben auch eine automatische Verifikation stattfinden und unter Umständen sogar konstruktive Kritik generiert werden kann. So könnte der Umgang mit den benötigten Formalismen in selbstständiger Auseinandersetzung mit Übungsaufgaben gefestigt werden, ohne Gefahr zu laufen, sich Fehler anzueignen, die ansonsten erst in der nächsten Bewertung einer schriftlichen Aufgabenbearbeitung[2] auffallen würden. Eine solche Bewertung kann momentan überhaupt nur geschehen, wenn Aufgaben bearbeitet und eingereicht werden, wofür vielen Teilnehmern jedoch vor allem gegen Semesterende die Motivation fehlt. Falls eine Aufgabe bearbeitet wird, vergehen unter Umständen mehrere Tage, bis eine Bewertung vorliegt. In dieser Zeit werden gegebenenfalls wieder neue Aufgabenstellungen präsentiert, die auf den vorherigen aufbauen, für die aber noch keine Kritik vorliegt, sodass sich Unsicherheiten bei der Bearbeitung immer mehr verstärken, bis die Motivation schließlich versiegt. Dem können computergestützte Aufgabenstellungen mit ihrer unmittelbaren Auswertung entgegenwirken und insbesondere unsicheren Studierenden helfen, den Schwung aufrechtzuerhalten.

Darauf aufbauend wird in dieser Arbeit die Entwicklung einer Webanwendung begründet, mithilfe derer Kursteilnehmer selbstständig Aufgaben bearbeiten können. Sie soll eine anschauliche Darstellung der grundlegenden Konzepte formaler Sprachmodelle ermöglichen und durch Verifikation von Eingaben Verständnisfehler frühzeitig aufdecken. Das soll helfen, die Last des Erlernens der benötigten Formalismen zu verringern und Schwung für das Durchdringen der weiteren Grundlagen der theoretischen Informatik aufzubauen.

[2] In Theoretische-Informatik-Kursen an der Universität Potsdam werden wöchentlich fakultative Hausaufgaben angeboten und über das Semester verteilte obligatorische Testate gefordert.

2 Lösungsansätze

Welche Möglichkeiten gibt es überhaupt, die Konzepte hinter formalen Sprachen anschaulicher und effektiver einzuführen? Dieser Abschnitt soll grobe Denkanstöße geben und eine Einordnung der zu entwickelnden Anwendung ermöglichen. Eine tiefergehende Auseinandersetzungen mit den Strategien ist nicht Teil dieser Arbeit.

Senkung des Themenumfangs

Eine Forderung, die traditionell Studierenden zugeschrieben wird, ist eine Verkleinerung des Curriculums, um für die schwierig zu begreifenden Themengebiete Freiraum zu schaffen. An der Universität Potsdam sind die Grundlagen der theoretischen Informatik im Studiengang *Informatik/Computational Science* auf zwei Module verteilt, die mit je sechs ECTS-Punkten vergütet werden: *Modellierungskonzepte der Informatik* (TI1) und *Effiziente Algorithmen* (TI2). Hier wird der in [Zuk16] definierte Kompetenzbereich „Formale Sprachen und Automaten“ vollständig abgedeckt und Teile der Bereiche „Modellierung“, „Diskrete Strukturen, Algebra und Logik“ und „Algorithmen und Datenstrukturen“ vermittelt (vgl. [ICS13]).

Die beiden Module lassen sich zusammen ungefähr mit dem Modul *Einführung in Berechenbarkeit, Komplexität und formale Sprachen* des zweiten Beispielstudiengangs in den Empfehlungen vergleichen, das mit acht hypothetischen ECTS-Punkten vergütet wird. Die darüber hinausgehend vermittelten Kompetenzen lassen die Vergütung des Themenkomplexes mit zwölf ECTS-Punkten durchaus angemessen und eine Reduktion des Inhalts nicht erforderlich erscheinen. Andere Studiengänge könnten diesbezüglich zu unterschiedlichen Ergebnissen kommen, aber unabhängig davon soll die zu entwickelnde Anwendung nicht dazu dienen, Wissen zu vermitteln, das nicht ohnehin schon Teil der Vorlesung ist.

Medieneinsatz während der Vorlesung

Dem Einsatz digitaler Unterstützung in der Lehre wird, vor allem im Schulbereich, momentan eine Renaissance aufgezwungen. Im Informatikstudium der Universität Potsdam ist fast jede Vorlesung auf vorbereitete Präsentationen aufgebaut, mit gemischten Auswirkungen auf die Aufmerksamkeit der Hörer. In den Vorlesungen zur TI1 und TI2 werden Präsentationen jedoch nicht einfach nur präsentiert, sondern auch mit ihnen interagiert. Als Teil der in [KKB14] beschriebenen Umorganisation wurden sie eingeführt, um Zeit zu sparen und Rückfragen aus dem Publikum mehr Aufmerksamkeit schenken

zu können. Dabei wurde vor allem Wert darauf gelegt, dass der Vorteil von Tafelbildern, das Geschriebene schnell und den Bedürfnissen der Hörerschaft entsprechend mehr oder weniger formlos zu ergänzen, erhalten bleibt.

Für den nach Verständnis suchenden Hörer hat diese Methode den Vorteil, dass dem Dozenten mehr Spielraum bleibt, den zu vermittelnden Stoff in Kontext einzubetten. Einerseits bleibt mehr Zeit, um Zusatzinformationen zu vermitteln, die in einem klassischen Tafelbild keinen Platz hätten. Andererseits kann zu beliebigen Themenbereichen gesprungen werden, wenn der Bedarf besteht, Querverbindungen deutlich zu machen, wodurch auch verlorengegangenes Learning-Edge-Momentum zumindest teilweise wieder angestoßen werden kann. Auch lassen sich Einsatzgebiete des vermittelten Wissens am Rechner demonstrieren, um dem Stoff weiteren Kontext zu verleihen. Letzlich hängt es aber vom Stil und der persönlichen Einstellung des Dozenten ab, ob ein Medieneinsatz sinnvoll ist. Die zu entwickelte Anwendung soll zunächst nicht darauf ausgelegt sein, in Vorlesungen eingesetzt zu werden.

Bessere Betreuung

Die eben erwähnte Umorganisation zielte auf eine effektivere Implementierung des *Cognitive-Apprenticeship*-Ansatzes für die Lehre der Grundlagen der theoretischen Informatik ab. Dabei sollte den Studierenden ermöglicht werden, ihr Verständnis beim gemeinsamen Bewältigen typischer Aufgabenstellungen mit den Dozenten und Tutoren zu entwickeln. In [KKB14] wird dadurch für TI1 bei gleichbleibenden Anforderungen von einer Reduktion der Durchfallquoten von grob einem Viertel auf weniger als ein Zehntel berichtet.

Die *Art* der Betreuung scheint also bereits einen merklichen Einfluss auf den Kompetenzerwerb zu haben. Dennoch bleiben in Übungen auch Fragen ungeklärt, sodass engagierte Tutoren regelmäßig die vorgesehenen Zeiten überziehen, um Verständnisprobleme zu klären. Angesichts des sinkenden Betreuungsverhältnisses an deutschen Universitäten (vgl. [DW18]) lässt sich *mehr* Betreuung in absehbarer Zeit aber wohl nicht implementieren. Prinzipiell könnte die zu entwickelnde Anwendung analog zum eben dargestellten Medieneinsatz in der Vorlesung dazu dienen, das Erstellen aufwändiger Tafelbilder beim gemeinsamen Lösen von Aufgaben zu erleichtern, um mehr Zeit für den persönlichen Austausch zu lassen. Ein solcher Einsatz steht aber nicht im Vordergrund der Betrachtungen in dieser Arbeit und müsste von den Lehrenden separat vorbereitet werden.

Learning by doing

Hausaufgaben gehören in vielen Lehrveranstaltungen zum Lehrangebot dazu, in manchen sind sie sogar Voraussetzung für die Prüfungszulassung. Anders als beim betreuten bzw. angeleiteten Bearbeiten von Aufgaben ist der Studierende hier auf seine eigene Interpretation der nötigen Kompetenzen angewiesen. Bereits in [HH05] wird aber auf die

Tabelle 2.1: Teilnehmer- und Tutorenzahlen in TI1 in den letzten vier Jahren

Jahr	Teilnehmer	Tutoren	Verhältnis
2014	159	5	31,8
2015	146	5	29,2
2016	165	5	33
2017	239	5	47,8

Quelle Teilnehmerzahlen: [PULS]

Schwierigkeiten einer für die Studierenden gewinnbringenden Bewertung hingewiesen. So sind sie ihren eventuellen Fehlinterpretationen hilflos ausgeliefert, während sie die Aufgaben bearbeiten. Dadurch festigen sie womöglich ineffektive Herangehensweisen an ansonsten effektive Veranschaulichungen der Konzepte. Auch wenn der eben angeführte *Cognitive-Apprenticeship*-Ansatz genau an dieser Stelle Abhilfe schaffen soll, kann in den vorgesehenen Übungen selbst bei optimalem Betreuungsverhältnis nicht jede Art der Aufgabenbewältigung in der Tiefe besprochen werden, die das schriftliche Lösen der Aufgaben erfordert. Hier liegt der hauptsächliche Einsatzzweck der zu entwickelnden Anwendung: Dem Teilnehmer soll eine Hilfestellung gegeben werden, die ihm die Unsicherheit bezüglich der Korrektheit des eigenen Vorgehens nimmt.

Eine weitere Schwierigkeit beim Anbieten von Hausaufgaben besteht darin, sie auch zu bewerten. Mithilfe von Tabelle 2.1 lässt sich erahnen, dass das in neun Stunden pro Woche und Tutor kaum realisierbar ist. In TI1 wurden daher Gruppenabgaben von zwei bis vier Studierenden auf einmal verlangt[3] und 2017 wurden Hausaufgaben schließlich fakultativ, was eine erhebliche Verringerung des Arbeitsaufwands mit sich brachte. Dementsprechend verschlimmerten sich die eingangs erwähnten Symptome der Ahnungslosigkeit, sodass der Austausch in den Übungen ohne gewaltige Mühen des Tutors nur noch von ausgewählten leistungsstarken Teilnehmern getragen wurde. Eine Lernanwendung kann einerseits die Versorgung der Teilnehmer mit Übungsmaterial erhöhen, indem sie in ihren Einsatzgebieten die manuelle Korrektur überflüssig macht. Andererseits kann sie für mehr Motivation sorgen, indem sie die Verzögerung der „Belohnung“, also der Bewertung, nach der Bearbeitung von einigen Tagen auf einige Bruchteile von Sekunden reduziert.

[3] Zusätzlich sollten Gruppenabgaben zur gegenseitigen Hilfestellung außerhalb der festen Übungen animieren. Häufig endete die Hilfestellung jedoch beim Punkteerwerb für die Gruppe.

3 Vorhandene Arbeiten

Simulatoren für endliche Automaten gibt es im Internet zuhauf. Diese gehen aber für gewöhlich nicht über eine Visualisierung der Abarbeitungsprinzipien der Automatenmodelle hinaus, wie beispielsweise [Dic] und [Bur]. Es gibt kaum die Möglichkeit, Algorithmen zu üben; so kann zum Beispiel [Dot] zwar DEAs, NEAs, reguläre Ausdrücke, kontextfreie Grammatiken und Turingmaschinen simulieren, aber die einzige angebotene Transformation ist die vollautomatische Minimierung eines gegebenen DEAs.

Das „interactive visualization and teaching tool for formal languages“ [RF06, Seite ix][4] *JFLAP* wurde hingegen von vornherein für die Unterstützung der Lehre konzipiert. Es bietet vielfältige Methoden zur Veranschaulichung von Sprachmodellen: Eingabewörter können „Schritt für Zustand“ abgearbeitet und verschiedene reguläre Sprachformalismen ineinander überführt werden. Dabei geschehen jedoch manche Schritte noch automatisch, was den Anwender wieder in eine passive Rolle versetzt und die Möglichkeiten des Mediums nicht völlig ausschöpft. Auch das Prüfen auf Äquivalenz zweier regulärer Sprachbeschreibungen wird unterstützt, jedoch werden dabei über die binäre Antwort hinaus keine zusätzlichen Informationen ausgegeben. JFLAP wurde als Desktopanwendung in Java geschrieben und ist somit auf Computer beschränkt, die die volle *Java Runtime Environment* anbieten. Die JFLAP-Lizenz gestattet zwar eine Modifikation des Codes, schränkt aber die Weiterverwendung aller Teile der modifizierten Version auf völlig kostenlose Produkte ein. Das könnte von manchen Nutzern als Einschränkung ihrer Freiheit betrachtet werden (vgl. Definition freier Software nach [FSF]).

Weiterhin existierte *Grail*, das laut [RW95] formale Sprachkonzepte in einer C++-Bibliothek implementiert. Diese Bibliothek war keine freie Software und dazu in oberflächlichen Tests nicht lauffähig. Das darauf aufbauende *Grail+* hingegen war lauffähig und erweitert den Funktionsumfang sogar noch. Es bietet über die Bibliothek hinaus Kommandozeilenprogramme, die einzelne Operationen auf Sprachmodellen ausführen und über Pipes hintereinandergeschaltet werden können. Jedoch ist Grail+ laut [Câm17] explizit ebenso unfreie Software. *JGrail*, beschrieben in [Coo05], ist eine in Java geschriebene Oberfläche für die Kommandozeilenprogramme. Obwohl die aktuelle Version Schaltflächen beinhaltet, die die Visualisierung von Automaten versprechen, sind die dafür benötigten Grail+-Programme nicht verfügbar. Damit bietet JGrail keine funktionalen Vorteile gegenüber der Kommandozeilennutzung von Grail+. Diese Programme bieten zudem keine Möglichkeit, den Umgang mit den Formalismen interaktiv zu üben und sind somit als Lernanwendungen nur bedingt geeignet.

[4] „interaktives Visualisierungs- und Lehrwerkzeug für formale Sprachen“

4 Ziele dieser Lernanwendung

Aufbauend auf den Leistungen und Defiziten der vorhandenen Anwendungen und Erfahrungen mit der Lehre von TI1 und TI2 sollen hier Ziele formuliert werden, die eine Lernanwendung erreichen sollte. Dazu wird sich im Folgenden mit den zwei hauptsächlichen Anwenderrollen befasst – Studierende und Lehrende.

4.1 Aus Sicht der Studierenden

Als „Studierender“ wird hier jeder Anwender bezeichnet, der die Anwendung hauptsächlich zum Erwerb und zur Vertiefung der in der Lehrveranstaltung vermittelten Kompetenzen nutzt.

Intuitive Oberfläche

Eine Lernanwendung sollte nicht selbst zum Lehrinhalt werden müssen. Daher ist die oberste Priorität während der Anwendungsentwicklung, die Interaktionsmöglichkeiten so selbsterklärend wie möglich zu gestalten. Grafische Benutzeroberflächen (*GUIs*) haben gegenüber kommandozeilenbasierten[5] den Vorteil, dass sie Hilfestellungen als Teil der Oberfläche anbieten können. Dazu sollen Schaltflächen mit kontextabhängigen Symbolen beitragen, die beschreiben, welche Funktion eine Schaltfläche gerade erfüllt. Wo möglich, können kurze Hilfetexte die Funktion einer Schaltfläche explizieren, wenn der Benutzer sich diesbezüglich unsicher ist.

Die GUI sollte sich auf die zur Erfüllung ihres jeweiligen Einsatzzwecks nötigen Bedienelemente beschränken. Einerseits soll der Teilnehmer nicht durch eine Fülle irrelevanter Optionen abgelenkt oder überfordert werden. Andererseits ist es unter Umständen nötig, gewisse Funktionalitäten zu verstecken, um noch nicht behandelten Stoff nicht vorwegzunehmen.

In der Vorlesung etablierte Konventionen sollten die Grundlagen und Richtlinien für die Visualisierungen von Modellen sein. Das erleichtert den Einstieg in die Anwendung und weitere zur Benutzung zwingend notwendige Hilfestellungen können eingespart werden. Auch wird der Eindruck vermieden, die Aufgabenbearbeitung in der Anwendung wäre vom Curriculum unabhängig. Unter Umständen bietet es sich trotzdem an, eine Möglichkeit zum Erklären der Konventionen zumindest anzubieten, um die Vorlesung und Übung zu ergänzen.

[5] Gemeint sind hier zustandslose Programme, die auf Eingaben lediglich mit Ausgaben reagieren. Ein auf *ncurses* basierendes Programm hat demensprechend eine GUI.

Klarer Nutzen

Zu jeder Zeit sollte ersichtlich sein, welche Kompetenz durch das Bearbeiten einer Aufgabe erworben oder gefestigt wird. Durch die klare Zielsetzung und das Versprechen unmittelbarer Auswertung soll eine Erwartungshaltung entstehen, die Motivation stiftet, sich mit den Aufgaben zu befassen.

Das Einhalten von Konventionen ist dabei bereits ein Grundstein, denn es ermöglicht eine intuitive Zuordnung der Aufgabe zum Inhalt der Vorlesung. Weitere Unterstützung können klar formulierte Titel in der GUI liefern, die den Kontext des Dargestellten abstecken. Ergänzend können optionale Erklärungen angeboten werden, die den Nutzen der Aufgabe für den Kompetenzerwerb deutlich machen.

Zeitnahe Auswertung

Sofern mit aktuellem Stand der Technik möglich, sollten Lösungsversuche sofort auf ihre Korrektheit überprüft werden. Kritik und Verbesserungsvorschläge sollten ebenfalls nach Möglichkeit nicht lange auf sich warten lassen. Das ist ein zentraler Punkt in der Rechtfertigung des Computereinsatzes.

Verfügbarkeit

Smartphones scheinen heutzutage allgegenwärtig. Das verführt zu der Annahme, eine Lernanwendung in Form einer App würde maximale Verfügbarkeit gewährleisten, jedoch benutzen laut [Bit17] nur 95 % der Deutschen zwischen 14 und 29 ein Smartphone. In dieser Zahl gibt es jedoch auch eine Unterteilung in verschiedene Betriebssysteme, die die Erstellung und Unterhaltung einer Lern-App verkompliziert (vgl. [Sta18]). Desktop-Computer stehen hingegen jedem Studierenden in den universitätseigenen Computerpools zur Verfügung. Eine reine Desktop-Anwendung schränkt jedoch wiederum den Aktionsradius von Studierenden ohne Laptop-Computer ein. Eine Webanwendung würde vom Zielgerät lediglich das Vorhandensein eines Webbrowsers verlangen und wäre somit von allen Studierenden an Desktop-Rechnern und von denjenigen mit mobilen Geräten an beliebigen Orten benutzbar.

Bedarfsgerechte Granularität

Fortgeschrittenen Lernenden sollte die Möglichkeit gegeben werden, Aufgabenteile zu überspringen, in denen sie bereits sicher sind. Das beugt einerseits einem Motivationsabfall durch Langeweile vor und hält den Kompetenzerwerb stabil an den Grenzen der vorhandenen Fähigkeiten. Im Umkehrschluss sollten Neueinsteigern nicht zu viele Schritte abgenommen werden, um das Learning-Edge-Momentum aufrechtzuerhalten. Daher liegt eine grundsätzlich feingranulare Konzeption der Aufgaben nahe, die Optionen zum Überspringen von Details vorsieht.

4.2 Aus Sicht der Lehrenden

Als „Lehrender“ wird hier jeder Anwender bezeichnet, der die Anwendung hauptsächlich dazu einsetzt, Studierenden den Erwerb und das Vertiefen von Kompetenzen zu erleichern. Das schließt die Integration in den Lehrplan und die Bereitstellung von Aufgaben für die Anwendung ein, ist aber nicht darauf beschränkt.

Intuitive Oberfläche

Für Lehrende ist die Fähigkeit der Anwendung, sich selbst zu erklären, unter Umständen von noch größerer Bedeutung als für Studierende. Immerhin müsste für eine schwierig zu durchschauende Anwendung zunächst entweder eine gewisse Abstraktion wie in [BK18] geschaffen werden, was zusätzlichen Aufwand für Lehrende bedeutet oder es müsste eine Einführung konzipiert und in den Lehrplan integriert werden, was bei bereits dicht gepackten Veranstaltungen schon ein Ausschlusskriterium darstellen kann.

Passende Aufgabentypen

Um den Einsatz in einer Lehrveranstaltung abwägen zu können, muss klar sein, welche Art von Unterstützung eine Anwendung überhaupt bieten kann. Wie schon in Kapitel 2 umrissen, soll die zu entwickelnde Anwendung hauptsächlich für das Selbststudium anhand von Übungsaufgaben benutzt werden.

In [HH05] wurden für die Entwicklung der Lernanwendung die in der technischen Informatik vorkommenden Aufgabentypen untersucht, um diejenigen zu identifizieren, die sich sinnvoll in eine Lernanwendung integrieren lassen. Die erarbeiteten Kategorien lassen sich jedoch schlecht auf die theoretische Informatik übertragen, weswegen auf Grundlage der Übungsaufgaben der TI1 folgende Aufgabentypen erarbeitet wurden:

Wortzugehörigkeitsprobleme Hierbei werden deklarative Sprachbeschreibungen, endliche Automaten, reguläre Ausdrücke, Kellerautomaten, Turingmaschinen oder Grammatiken (im Folgenden nur „Sprachbeschreibungen“) und eine Liste von Zeichenketten vorgegeben und gefordert, festzustellen, ob die Zeichenketten in der beschriebenen Sprache enthalten sind.
Dieser Aufgabentyp lässt sich prinzipiell für alle aufzählbaren Sprachen durch einen Computer unterstützen, da die Abarbeitungsmechanismen der Automaten beziehungsweise die Ableitungsmechanismen der Grammatiken berechenbar sind und dem Anwender zumindest eine Visualisierung geboten werden kann. Für endliche Automaten, reguläre Ausdrücke, Kellerautomaten und reguläre und kontextfreie Grammatiken lässt sich ein Lösungsvorschlag mittels der in [HMU06, Seiten 154 f., 303 ff.] beschriebenen Algorithmen sogar vollautomatisch prüfen.

Sprachbeschreibungen Hier werden Sprachbeschreibungen in beliebiger Form, meist eher formlos, vorgegeben und eine äquivalente Beschreibung in einer spezifizierten Form gefordert.

Für reguläre Sprachen lassen sich Lösungsvorschläge mittels des in [HMU06, Seite 159] beschriebenen Algorithmus vollautomatisch prüfen, sofern sie und die geforderte Sprache mittels endlicher Automaten oder regulärer Ausdrücke beschrieben werden. Nichtreguläre Sprachen lassen diese Prüfung nicht zu.

Transformationen Diese Aufgaben verlangen die Anwendung von Algorithmen, um verschiedene gleichmächtige Formen von Sprachbeschreibungen ineinander zu überführen.
Da es sich um Algorithmen handelt, kann ihre Ausführung durch den Computer unterstützt werden. So lassen sich beispielsweise die Tabelleneinträge für den in [HMU06, Seiten 155 ff.] beschriebenen Algorithmus während des Ausfüllens auf Richtigkeit überprüfen.

Argumentationen Häufig sollen gewisse Eigenschaften der benutzten Objekte nachgewiesen werden. Zu diesem Zwecke wird eine schlüssige Argumentation verlangt, die sich auf in der Vorlesung erworbene Kenntnisse stützt.
Arbeiten wie [SC07] und [BKK16] begründen, dass Computerunterstützung für das Erlangen von Beweiskompetenzen möglich ist. Die Grundlage dafür bildet hier *Coq*, ein interaktiver Theorembeweiser, der ursprünglich nicht für die Lehre konzipiert wurde (siehe [Coq]). Dementsprechend muss für die verschiedenen Themenbereiche eine Formalisierung für Coq vorbereitet werden, damit es für Lernende ohne umfassendes Wissen über Beweisassistenten benutzbar ist. Die zu entwickelnde Anwendung wird keine Unterstützung beim Entwerfen von Beweisen bieten.

Personalisierungsmöglichkeiten

Sollen im Rahmen der unterstützten Aufgabentypen eigene Aufgabenstellungen benutzt werden, müssen diese in die Anwendung gelangen. Dafür kann ein Austauschformat für die einzelnen Szenarien definiert werden, das die externe Vorbereitung einer Aufgabenstellung und das anschließende Importieren durch Studierende ermöglicht. Darin sollten Aufgabentyp, die gewünschte Lösung beziehungsweise der gewünschte Ausgangspunkt und eventuell ein Aufgabentext repräsentierbar sein.

Es ist auch denkbar, dass Lehrende eigene Programmteile schreiben, die sich in die Anwendung integrieren. Diese können genau auf die Anforderungen der gewünschten Übungsaufgaben abgestimmt werden und die Vielfalt an unterstützten Aufgabentypen unter Umständen noch erweitern. Die Lizenz des Programms sollte dem Benutzer deswegen mindestens die Freiheiten zur Modifikation der Anwendung und zur Verfügbarmachung der modifizierten Version einräumen, was den in [FSF] beschriebenen Freiheiten 1 und 3 entspricht. Daher erscheint die Erstellung der Anwendung als freie Software naheliegend.

5 Implementierungsmöglichkeiten

Wie können die gesetzten Ziele in einem ausführbaren Programm umgesetzt werden? Um diese Frage zu beantworten, findet in diesem Abschnitt eine Auseinandersetzung mit verschiedenen Architekturen und Programmieransätzen statt.

5.1 Architektur

In Abschnitt 4.1 wurde bereits begründet, dass es sinnvoll ist, eine Lernanwendung als Webanwendung zu realisieren und ergründet, was eine GUI anbieten sollte, um dem Zweck der Anwendung gerecht zu werden. So eine GUI könnte auf der *Model-View-Controller*-Architektur (*MVC*) basieren, wie sie in [GHJV95, Seiten 4 f.] beschrieben wird: Das Model könnte die Charakteristika der aktuellen Aufgabe sowie den aktuellen Stand des Lösungsversuchs des Benutzers beinhalten. Die View würde dem Benutzer seinen aktuellen Stand, Möglichkeiten, diesen zu manipulieren und bei Bedarf Informationen zur Aufgabenstellung präsentieren, indem sie das *Document Object Model* (*DOM*) der Webseite anpasst, sobald das Model Änderungen am Zustand meldet. Der Controller würde die Eingaben des Benutzers während der Aufgabenbearbeitung als Fortschritt im Model festhalten, indem Eingabecallbacks im DOM mit Methoden des Controllers belegt werden. Diese grundlegende Einteilung würde die Voraussetzung dafür schaffen, dass einzelne Komponenten unabhängig voneinander getestet und in verschiedenen Szenarios wiederverwendet werden können.

Eine Alternative zum MVC wäre die *Model-View-Presenter*-Architektur (*MVP*) nach [Fowo6], in der die View lediglich eine Ansammlung von Präsentationsobjekten ist und die Logik zur Manipulation der Präsentation in einen Presenter verlagert wird. Die View nimmt zwar Benutzereingaben entgegen, reicht sie aber mehr oder weniger direkt zur Verarbeitung an den Presenter weiter. Die direkte Weitergabe von Eingaben an den Presenter, die sogenannte *Passive-View*-Variante des MVP, bietet sich bei einer Webanwendung besonders an, da dann View und DOM zusammengefasst werden können und der Presenter, wie im MVC der Controller, durch Eingabecallbacks die Benutzerinteraktionen abfangen kann. Über das DOM kann dem MVP auch ein *Supervising-Controller*-Aspekt verliehen werden, denn beispielsweise werden Benutzereingaben in Textfelder zunächst in den Attributen des Textfelds gespeichert und das DOM kann so selbst als Model dienen. Diese Sichtweise sollte vermieden werden, wenn eine klare Trennung zwischen Model und View das Ziel ist.

JavaScript-Frameworks bieten in der Regel auch eigene Interpretationen und Implementierungen von MVC, MVP oder anderen Architekturen, erhöhen aber die Komplexität der Anwendung und bedeuten eine Bindung an genau ein Framework, wie in

[Rey17] beschrieben. Es wird daher der Einfachheit halber auf Frameworks wie *AngularJS*, *Ember.js* und *Backbone.js* verzichtet.

5.2 Programmieransätze

Prinzipiell wäre es möglich, für jeden Aufgabentyp das Verhalten von Sprachformalismen, das gerade für eine Aufgabe benötigt wird, im Model zu implementieren. Das würde aber die Wiederverwendbarkeit des Codes erschweren und unter Umständen zu duplizierten Codefragmenten zwischen den einzelnen Models führen. Es wirkt daher zweckmäßig, das Verhalten der Formalismen ein einziges Mal als Bibliothek zu implementieren und diese an den Stellen einzusetzen, an denen sie gebraucht wird. Sie kann demnach komplett von der eigentlichen Anwendung abgekapselt entwickelt und gewartet werden.

Die Objekte, die dafür modelliert werden müssen – endliche Automaten, reguläre Ausdrücke, reguläre Grammatiken – sind grundlegend mathematischer Natur und eine Beschreibung mithilfe des funktionalen Programmierparadigmas liegt daher nahe. Im Bachelorstudium kam jedoch der funktionalen Programmierung nicht viel Aufmerksamkeit zu und der Aufwand für die Entwicklung einer in diesem Stil geschriebenen Bibliothek wird demnach vom Autor als zu hoch eingeschätzt. Als Programmiersprachen bieten sich zunächst solche an, die sich ohne großen Aufwand in eine Webanwendung integrieren lassen. Das ist zum einen JavaScript selbst und Sprachen wie *CoffeeScript* und *TypeScript*, die sich direkt zu JavaScript kompilieren lassen. Zum anderen können über die Kompilierung ins *WebAssembly*-Format auch beispielsweise *C*, *C++*, *Rust* und *Go* für die Programmierung von Webanwendungen benutzt werden. C und C++ gehören nach [Cas18] zu den beliebtesten und nach [Lex16] zu den verbreitetsten dieser Sprachen und ermöglichen außerdem eine Integration in viele andere Einsatzbereiche. Ein möglicher künftiger Betreuer des Codes wird demnach mit hoher Wahrscheinlichkeit bereits Erfahrung mit diesen Sprachen haben. C++ ist eine objektorientierte Sprache und bietet daher eine einfache Möglichkeit, die verschiedenen Formalismen als Klassen zu implementieren. Da C++ auch funktionale Sprachelemente und Konzepte unterstützt, ermöglicht die Wahl dieser Sprache auch eine künftige Überarbeitung, um die Implementierung näher an die mathematische Beschreibung zu bringen, ohne die JavaScript-Schnittstelle verändern zu müssen.

Für die Anwendung selbst kommen JavaScript und die dazu kompilierbaren Sprachen in Frage. Wegen ähnlicher Argumente wie für C++ ist aber hier die direkte Implementierung in JavaScript vorzuziehen, um den Wartungsaufwand möglichst gering zu halten.

6 Details der gewählten Implementierung

Aufbauend auf den Betrachtungen der vorherigen Kapitel wurden die C++-Bibliothek *reglibcpp* und eine prototypische Webanwendung entwickelt. Die Webanwendung ist unter `regapp.tomkra.nz` verfügbar. Eine ausführliche API-Dokumentation der Bibliothek kann unter `reglibcpp.tomkra.nz` eingesehen werden. Auf den Seiten ist auch je das entsprechende *Git*-Repository verlinkt, aus dem der unter der *GNU General Public License* stehende Quelltext bezogen werden kann. Außerdem sind Webanwendung, Dokumentation und Quellcode an diese Arbeit angehängt. Ergänzend dazu werden in diesem Abschnitt die Feinheiten der Implementierung dargestellt und die Motivation hinter einzelnen Entscheidungen erklärt.

6.1 reglibcpp

Die C++-Bibliothek ist in je eine Header- und eine Implementierungs-Datei pro Sprachkonzept aufgeteilt, die in Tabelle 6.1 beschrieben sind. Außerdem ist sie mit einer *CMakeList.txt*-Datei ausgestattet, die die Kompilierung mithilfe des Build-Datei-Generators *CMake* ermöglicht. Dadurch kann die Bibliothek auf vielen Plattformen entweder direkt in ein CMake-Build integriert oder zumindest automatisch kompiliert und separat eingebunden werden.

Bei der Benutzung muss darauf geachtet werden, dass formale Sprachen mithilfe der Bibliothek über *Unicode*-Codepunkte definiert werden. Somit stehen momentan zwar 137 374 Zeichen zur Verfügung, aber beispielsweise können kombinierte Symbole wie das Deutsche-Flagge-Emoji 🇩🇪, bestehend aus den Unicode-Zeichen `REGIONAL INDICATOR SYMBOL LETTER D` 🇩 und `REGIONAL INDICATOR SYMBOL LETTER E` 🇪, nicht Elemente eines Eingabealphabets Σ sein, jedoch durchaus Teil der über den einzelnen Zeichen definierten Sprache[6]: 🇩🇪 $\in \{🇩, 🇪\}^*$ aber $|🇩🇪| = 2$. Das ist jedoch eine Eigenschaft der Darstellung von Unicode und wird in [Uni18] näher ausgeführt.

Da die vier verschiedenen Formalismen gleichmächtig sind, wurden für `dfa`-, `expression`- und `gnfa`-Objekte implizite Umwandlungen in `nfa`-Objekte implementiert. Das heißt, dass Funktionen wie der Gleichheitsoperator `.operator==()` oder `reg::findShortestWord()` – beide werden später erklärt – nur für `nfa`-Argumente implementiert sein müssen und für alle anderen Objekte analog funktionieren.

[6] Das Estnische-Flagge-Emoji 🇪🇪 kann auch Teil der Sprache über diesem Alphabet sein.

Tabelle 6.1: Header-Dateien, die zur Benutzung der C++-Bibliothek nötig sind und die darin enthaltenen Klassen

Datei	Klassendefinition	Repräsentiert
fabuilder.h	`fabuilder` (finite **a**utomaton **builder**)	„Konkreter Erbauer“ und „Direktor“ für `nfa`- und `dfa`-Instanzen
nfa.h	`nfa` (**n**ondeterministic **f**inite **a**utomaton)	Nichtdeterministische endliche Automaten mit ε-Übergängen
dfa.h	`dfa` (**d**eterministic **f**inite **a**utomaton)	Deterministische endliche Automaten
expression.h	`expression[::exptr]` (regular **expression** [**p**oin**t**er])	Reguläre Ausdrücke
gnfa.h	`gnfa` (**g**eneralized **n**ondeterministic **f**inite **a**utomaton)	Verallgemeinerte nichtdeterministische endliche Automaten

6.1.1 Implementierung des Erbauer-Entwurfsmusters

Zur Instanziierung von `dfa`- und `nfa`-Objekten wird eine Vereinfachung des Erbauer-Entwurfsmusters in Anlehnung an [GHJV95, Seiten 97 ff.] bemüht. Diese Vereinfachung umfasst das Weglassen der Erbauer-Schnittstelle und das Zusammenlegen des Direktors mit dem konkreten Erbauer in der Klasse `fabuilder`. Das Erbauer-Muster ermöglicht die Erstellung von endlichen Automaten in einem einheitlichen Prozess und die Möglichkeit, während des Erbauungsprozesses Optimierungen und auch Umformungen anzuwenden.

In Anlehnung an [HMU06] sind NEAs und DEAs grundlegend ähnlich aufgebaut. Sie sind geordnete Fünf-Tupel aus endlicher Zustandsmenge Q, endlicher Eingabesymbolmenge Σ, der Zustandsübergangsfunktion δ, einem Startzustand $q_0 \in Q$ und der Menge der akzeptierenden Zustände $Q_F \subseteq Q$. Der Unterschied liegt in der Zustandsübergangsfunktion, die im deterministischen Fall vom Typ $Q \times \Sigma \to Q$ ist und im nichtdeterministischen Fall vom Typ $Q \times \Sigma \cup \{\varepsilon\} \to \mathcal{P}(Q)$. Hier bezeichnet ε das leere Wort und $\mathcal{P}(Q)$ die Potenzmenge der Zustandsmenge. Da $\{\{q\}|q \in Q\}$ eine zu Q isomorphe Untermenge von $\mathcal{P}(Q)$ und Σ direkt eine Untermenge von $\Sigma \cup \{\varepsilon\}$ ist, lässt sich mithilfe des dafür benutzten Isomorphismus $\varphi : Q \to \mathcal{P}(Q)$ jeder DEA als NEA auffassen.[7] Weiterhin ist entscheidbar, ob ein NEA ohne Veränderung der Zustandsmenge als DEA aufgefasst werden kann, indem für jeden Übergang geprüft wird, ob er deterministisch ist. Fällt dieser Test positiv aus, lässt sich mithilfe des Umkehrisomorphismus ein DEA mit Zustandsübergangsfunktion $\varphi^{-1} \circ \delta$ und Alphabet Σ bauen.

Auf diesem Prinzip basiert die Erbauung von endlichen Automaten mithilfe von `fabuilder`, der zunächst annimmt, einen NEA zu konstruieren. Es können Zustände zu

[7] Die Rückrichtung ist ebenso wahr, aber aufwendiger zu berechnen.

Q oder Q_F hinzugefügt oder als q_0 festgelegt werden, Symbole zu Σ hinzugefügt und für beliebige Paare aus $Q \times \Sigma \cup \{\varepsilon\}$ Zustände zur Zielmenge hinzugefügt werden. Wird dann versucht, ein `dfa`-Objekt zu instanziieren, wird der erwähnte Test durchgeführt und im Erfolgsfall sofort der DEA konstruiert. Andernfalls wird, in Abhängigkeit von einem beim Methodenaufruf festgelegten Parameter, entweder eine Exception geworfen oder per Teilmengenkonstruktion der Nichtdeterminismus beseitigt. Für die Instanziierung von `nfa`-Objekten müssen dementsprechend keine Tests oder Transformationen durchlaufen werden.

Grundsätzlich könnte diese Konstruktion auch auf verallgemeinerte NEAs ausgeweitet werden, indem für Paare aus $Q \backslash Q_F \times Q \backslash \{q_0\}$ je ein regulärer Ausdruck als Übergang festgelegt würde, um die Übergangsfunktion in Sinne von [Sipo6, Seite 72] zu definieren. Das würde jedoch noch mehr Transformationen bei der Konstruktion der spezielleren Automaten bedingen und außerdem haben sich die existierenden Methoden zur Konstruktion von `gnfa`-Objekten bisher als ausreichend erwiesen. Diese werden im Abschnitt 6.1.5 erläutert.

Das Erbauen durch Definition von $(Q, \Sigma, \delta, q_0, Q_F)$ wird ergänzt durch Operationen, die die beschriebene Sprache direkt beeinflussen. So kann die Sprache des künftigen Automaten mit der eines anderen vereinigt oder geschnitten werden, indem der Produktautomat nach dem Vorbild von [HMU06, Seiten 43 f.] gebildet wird. Unter der Bedingung, dass δ deterministisch definiert wurde, kann auch das Komplement gebildet werden, indem Q_F durch $Q \backslash Q_F$ ersetzt wird. Wie bei der Instanziierung eines `dfa`-Objekts wird auch hier entweder eine Exception geworfen oder eine Teilmengenkonstruktion ausgeführt, wenn δ nichtdeterministische Merkmale aufweist.

Zuletzt gibt es noch „kosmetische“ Operationen, die die Sprache des sich im Aufbau befindlichen Automaten nicht verändern. Dazu gehören die explizite Ausführung der Teilmengenkonstruktion, die Eliminierung von unerreichbaren und nichtproduzierenden Zuständen, das Totalisieren von δ im deterministischen Sinne,[8] das Verschmelzen äquivalenter Zustände bei deterministischem δ und die Durchnummerierung, sprich Namensänderung, der Zustände.

6.1.2 Nichtdeterministische endliche Automaten mit ε-Übergängen

NEAs werden im Rahmen der Implementierung allgemein zu ε-Übergängen befähigt. Sind diese nicht gewünscht, müssen sie bei der Erbauung durch die dem Nutzer zugewandte Software explizit ausgeschlossen und beim Darstellen einfach weggelassen werden. Entsteht ein `nfa`-Objekt aus einem `gnfa`-Objekt, lassen sich ε-Übergänge nicht vermeiden, was aber im Kontext von regulären Ausdrücken ohnehin nicht mehr nötig sein sollte.

`nfa`-Objekte bieten die Möglichkeit, die dazugehörigen Tupelelemente Q, Σ und q_0 zu inspizieren. δ und die dadurch implizierte erweiterte Zustandsüberführungsfunktion

[8] Gemeint ist hiermit, dass Zustandsübergänge, die nicht explizit definiert sind, also bei nichtdeterministischer Interpretation in die leere Menge führen, in einen eventuell eigens kreierten Müllzustand geleitet werden.

$\hat{\delta}$ lassen sich durch Eingabe von Zuständen und Symbolen beziehungsweise Wörtern auswerten und für die Zugehörigkeit zu Q_F steht ein Prädikat zur Verfügung. Außerdem kann zu jedem Zustand die ε-Hülle berechnet werden.

Die Auswertung von δ, $\hat{\delta}$ und der ε-Hülle ergibt dabei immer eine Zustandsmenge, die, in Abhängigkeit von der Art der Eingaben, auf unterschiedliche Weise ausgedrückt wird. Zum einen kann das ein Array von booleschen Werten mit Länge $|Q|$ sein, das eine interne Ordnung der Zustände nutzt, um Teilmengen von Q dadurch zu kodieren, dass am Index jedes Zustands ein wahrer oder falscher Wert steht, um die Zugehörigkeit oder Nichtzugehörigkeit zur Teilmenge zu signalisieren. Zum anderen kann das eine Menge von String-Referenzen sein, die auf die internen Repräsentationen der Zustandsnamen zeigen. Wird ein Zustand in der Eingabe über seinen Index identifiziert, kommt es zur ersten Ausgabeform und bei Identifikation über seinen Namen zur zweiten. Es ist zudem möglich, diese Ausgaben wiederum als Eingaben für $\hat{\delta}$ und die ε-Hüllenbildung zu benutzen, wenn die Ergebnisse für die Einzelemente vereinigt werden sollen. Hierbei wird wieder die Form ausgegeben, die eingegeben wurde.

Auf sprachlicher Ebene steht eine Implementierung des Vergleichsoperators `==` zur Verfügung, die zwei NEAs auf die Gleichheit ihrer Sprachen prüft. Diese ist derzeit über eine Teilmengenkonstruktion und den Aufruf der des entsprechenden Vergleichsoperators der `dfa`-Klasse realisiert. Mithilfe von Schnitt-, Vereinigungs- und Komplementoperationen und dem Vergleich mit einem Objekt, das die leere Sprache beschreibt, lässt sich somit auch die Teilmengenrelation zwischen den Sprachen zweier NEAs prüfen. Diese könnte künftig über die Vergleichsoperatoren `<`, `<=`, `>` und `>=` zur Repräsentation von $\subset$, $\subseteq$, $\supset$ und $\supseteq$ direkt implementiert werden, muss aber derzeitig noch über diesen Umweg geschehen.

Als externe Funktion wurde bereits die Funktion zum Suchen des kürzesten akzeptierten Wortes eines NEAs erwähnt. Diese ordnet in einem der Breitensuche nicht unähnlichen Prozess jedem Zustand ein Wort zu, das ihn mit den wenigsten Zustandsübergängen erreicht, bis einem akzeptierenden Zustand begegnet wird. Weil ε-Übergänge dazugezählt werden, kann dabei auch ein Wort gefunden werden, das mehr Zeichen enthält als das tatsächlich kürzeste Wort.

6.1.3 Deterministische endliche Automaten

Wie in Abschnitt 6.1.1 erwähnt, könnten DEAs mithilfe von φ^{-1} als NEAs implementiert werden. Der Effizienz halber wurde der direkte Weg gewählt, die Übergangsfunktion als $Q \times \Sigma \to Q$ zu implementieren. Dementsprechend gibt es keine Methode zur Berechnung der ε-Hülle und δ und $\hat{\delta}$ liefern und akzeptieren nur einzelne Zustände. Diese sind, wieder abhängig von der Eingabekodierung, als Index von oder als Referenz auf Zustandsnamen kodiert.

Der in Abschnitt 6.1.2 erwähnte Test auf Gleichheit von Sprachen ist wie in [HMU06, Seite 155] beschrieben umgesetzt und bildet die Grundlage für den Vergleich aller Formalismen. Die restlichen Aussagen zu Vergleichen treffen auch für `dfa`-Objekte zu. Die externe Funktion zum Suchen des kürzesten akzeptierten Wortes ist für `dfa`-Objekte

überladen, da sie für DEAs effizienter zu berechnen ist und immer das tatsächlich kürzeste Wort liefert. Die Überladungsauflösung von C++ verhindert aber die Auswahl dieser Funktion für Objekte anderer Klassen, was auch von Vorteil ist, da ansonsten eine Teilmengenkonstruktion nötig wäre.

6.1.4 Reguläre Ausdrücke

Die Struktur von RAs ist als Binärbaum umgesetzt. Die einzelnen `expression`-Objekte sind entweder Knoten, die Zeiger auf ihre maximal zwei Kinder verwalten und die Funktion einer Alternation, Verkettung oder Kleene-Hülle erfüllen oder sie sind Blätter, die die leere Menge $\emptyset$, das leere Wort ε oder einzelne Symbole darstellen. Um die Baumstruktur effizient zu handhaben, wird praktisch nur mit Zeigern auf Knoten gearbeitet, die deswegen ein eigenes Typenalias `expression::exptr` bekommen.

Da Blätter, wenn sie einmal erstellt wurden, keine andere Aufgabe mehr erfüllen, als Blätter zu sein und diese mit hoher Wahrscheinlichkeit an vielen Stellen gebraucht werden, ist für Blätter in Anlehnung an das Einzelstück-Entwurfsmuster nach [GHJV95, Seiten 127 ff.] die Erstellung von Instanzen eingeschränkt. Bäume werden ohnehin nur über statische Funktionen der Klasse `expression` erstellt, die Zeiger auf die Wurzel zurückgeben, was eine solche Einschränkung auch praktikabel macht. Diese Funktionen kombinieren entweder bereits vorhandene Bäume, um Knoten-Objekte anzulegen, auf die ein Zeiger ausgegeben wird oder sie liefern Zeiger auf die Einzelstück-Blatt-Ausdrücke. Bei der Kombination von Bäumen können auch unterschiedliche Grade der Optimierung hinzugeschaltet werden, die die Semantik von Alternation, Verkettung und Kleene-Hülle ausnutzen, um unter Umständen einen kleineren Baum zu generieren als bei der bloßen Erweiterung von Bäumen. Mithilfe der ersten Optimierungsstufe werden die Speicheradressen der `expression`-Objekte zurategezogen, um gewisse Identitäten zu finden, die die Verkettung und die Alternation vereinfachen. Zusätzlich wird bei der Kleene-Hüllenbildung eines Kleene-Hüllen-RAs keine zusätzliche Hülle gebildet. Die zweite Optimierungsstufe vergleicht `expression`-Objekte mithilfe der impliziten `nfa`-Konversion und deren Vergleichsoperator semantisch, um Identitäten und auch Teilmengenbeziehungen festzustellen.

Weiterhin ist ein Parser implementiert, der eine Zeichenkette einliest und eine Baumstruktur baut, die den darin kodierten RA abbildet. Dieser Parser basiert auf [LL09] und die Implementierung selbst könnte auch für Lehrzwecke interessant sein, um kontextfreien Grammatiken einen Praxisbezug zu verleihen. Welche Symbole hierbei als Operatoren beziehungsweise als leere Menge oder leeres Wort interpretiert werden, wird über statische Attribute der `expression`-Klasse festgelegt. Die Standardeinstellung orientiert sich an den Konventionen in [HMU06], die `(` und `)` für Klammern, `+` für die Alternation, `*` für die Kleene-Hülle und $\emptyset$ und ε für die leere Menge und das leere Wort vorsehen. Über diese Attribute wird auch die textuelle Repräsentation eines `expression`-Objekts gesteuert, wenn diese angefordert wird.

Um die Baumstruktur, die in einem Objekt verwurzelt ist, zu traversieren, kann die darin kodierte Operation abgefragt, über die Liste von Kindern iteriert und von Blättern,

die Symbole oder das leere Wort kodieren, das entsprechende Symbol extrahiert werden. Dabei wird der ε-Ausdruck als Symbol aufgefasst, der die leere Zeichenkette "" kodiert, während der $\emptyset$-Ausdruck eine eigene Operation darstellt.

6.1.5 Verallgemeinerte nichtdeterministische Automaten

In Abschnitt 6.1.1 wurde bereits angedeutet, dass die `gnfa`-Klasse auf den VNEAs aus [Sipo6] beruht. Wenn die Quintupel-Notation für endliche Automaten aus [HMUo6] beibehalten wird, ist demnach $|Q_F| = 1$ und $\delta : Q \backslash Q_F \times Q \backslash \{q_0\} \to \mathcal{R}$, wenn $\mathcal{R}$ die Menge der regulären Ausdrücke bezeichnet. Sipser gibt auch die erste von zwei Methoden zur Instanziierung von `gnfa`-Objekten an: Die „Umwandlung“ eines `nfa`- oder eines `dfa`-Objekts. Im weiteren Verlauf wird jedoch klar, dass für beliebige Eingabealphabete Σ und reguläre Ausdrücke R über diesem Alphabet auch das Tupel $(\{q_0\}, \Sigma, (q_0, q_F) \mapsto R, q_0, \{q_F\})$ ein VNEA ist. Das liefert die zweite Konstruktionsmethode, die einen `expression::exptr` nimmt und den dadurch repräsentierten regulären Ausdruck zum einzigen Übergang zwischen den einzigen Zuständen q_0 und q_F macht.

Somit dürfte der schlussendliche Zweck der `gnfa`-Klasse bereits gut umrissen sein: Die Brücke zwischen endlichen Automaten und regulären Ausdrücken zu schlagen. Der Weg zwischen den beiden Extremen führt über die Eliminierung von Zuständen und Verknüpfung von RAs hin beziehungsweise die Spaltung von regulären Ausdrücken und das Einschieben neuer Zustände zurück. Der Hinweg kann unterteilt werden in das Überbrücken von Übergängen und die Entfernung des Quellzustandes dieser Übergänge. Beim Überbrücken wird ein im Quellzustand startender Übergang zuerst mit jedem Übergang verkettet, der in den Quellzustand führt und dann entfernt. Sind alle Übergänge überbrückt, kann der Quellzustand entfernt werden. Sind alle Zustände bis auf q_0 und q_F entfernt, wird ein `expression::exptr` ausgegeben, der einen RA repräsentiert, der dieselbe Sprache beschreibt wie der Ausgangsautomat. In die andere Richtung wird der Wurzelknoten der den RA repräsentierenden Baumstruktur entfernt und je nach dessen Operation neue Zustände nach dem in [HMUo6, Abb. 3.17] dargestellten Plan eingefügt, bis nur noch Blätter übrig sind, die nach Abb. 3.16 derselben Quelle interpretiert werden. Ergebnis ist ein NEA, der dieselbe Sprache beschreibt wie der ursprüngliche RA.

Wie bei den anderen Automaten kann von `gnfa`-Objekten der Startzustand abgefragt werden, dazu die Menge der entfernbaren Zustände $Q \backslash \{q_0, q_F\}$ und der akzeptierende Zustand. Bei der Abfrage von δ müssen die Elemente eines Zustandspaares angegeben werden und das Ergebnis ist ein `expression::exptr`, der den RA repräsentiert, der vom ersten zum zweiten Zustand führt. Außerdem gibt es eine Methode, die die Menge der Paare aus $Q \times Q$ ermittelt, die durch einen spaltbaren Knoten-Ausdruck verbunden werden.

6.2 Webanwendung

Im Prototypen der Anwendung sollen zunächst die drei erstgenannten Aufgabentypen aus Abschnitt 4.2 vertreten sein: Wortzugehörigkeitsproblem, Sprachbeschreibungen und Transformationen. Diese sollen als Demonstrationen dienen und eine Grundlage für Erweiterungen bieten.

Hier werden alle Automaten mittels der Graphrepräsentation ihrer Übergangsfunktion dargestellt. Diese ist mithilfe der in [Fra+16] vorgestellten JavaScript-Bibliothek *Cytoscape.js* umgesetzt. Für die vorgestellten Aufgaben wäre auch eine Tabellendarstellung denkbar, aber als aufwendiger umzusetzen eingestuft, da Cytoscape.js bereits viele Aspekte der Darstellung abstrahiert. Insbesondere die Implementierung zu Abschnitt 6.2.3, in der die Trennung von Darstellung und Geschäftslogik besonders gut gelungen ist, bietet sich als Einstiegspunkt zur Erweiterung durch eine alternative Ansicht an. Dafür müsste nur eine View implementiert werden, die die Methoden der vorhandenen View definiert und damit eine Tabelle verwaltet. Zusätzlich werden GUI-Komponenten aus *Material Design Lite* (siehe [MDL])verwendet, um der Anwendung Optik und Haptik zu verleihen, die Anwendern bereits aus anderen Web- und Mobilanwendungen bekannt sein dürften. Die Benutzung verschiedener Sprachen wird durch *l10n.js* (siehe [Gre18]) ermöglicht. Die Lizenzen der Bibliotheken schränken die Freiheiten, die die GPL den Benutzern der Anwendung einräumt, durch ihre Inklusion nicht ein.

Das Format für die Aufgabenspezifikation ist allen Aufgaben gemein und prinzipiell kann jede Spezifikation für jede Aufgabe verwendet werden. Es wurde die *JavaScript Object Notation* gewählt, weil sie ein verbreitetes Datenaustautschformat ist und gerade von JavaScript sehr einfach eingebunden werden kann. Für Informationen zum Format ist [JSON17] zu konsultieren – auf diese Quelle beziehen sich auch die hier erwähnten kursiv gedruckten Begriffe. Eine Spezifikationsdatei sollte ein *Object* mit den *Members* `re`, `machine`, `handwritten` und `description` beschreiben; für ein Beispiel siehe Auflistung 7.1. Die *Members* haben dabei folgende Bedeutungen:

`re` Beschreibt eine reguläre Sprache mittels eines regulären Ausdrucks in Textform, der von reglibcpp geparst werden kann. Basiert eine Aufgabenstellung auf einem Automaten, kann der Ausdruck automatisch in einen NEA umgewandelt werden. `re` ist optional, wenn `machine` oder `handwritten` gegeben sind. In Aufgaben, die nicht explizit einen regulären Ausdruck erfordern, haben diese beiden `Members` Vorrang.

`machine` Beschreibt eine reguläre Sprache mittels einer Graphbeschreibung eines Automaten. Diese Beschreibung ist das Ergebnis der Cytoscape.js-Methoden `.elements()` und `.jsons()` auf einem entsprechenden Graphobjekt, das bereits einen Übergangsgraphen beschreibt. `machine` ist optional, wenn `re` oder `handwritten` gegeben sind. Ohne besondere Einschränkungen hat `handwritten` Vorrang vor `machine`.

handwritten Beschreibt eine reguläre Sprache mittels einer Beschreibung der Komponenten eines Automaten. Diese ist als ein *Object* mit den `Members` `states`, `alphabet`, `transitions` und `accepting` definiert. `states` ist ein *Array* aus *Strings*, die die Namen der Zustände des Automaten beschreiben. Der erste Eintrag wird dabei als Startzustand festgelegt. `alphabet` ist ein *Array* aus *Strings*, die die Eingabesymbole des Automaten beschreiben. Das leere Wort ε wird dabei durch den leeren *String* `""` dargestellt, obwohl es streng genommen nicht zu Σ gehört. Es definiert aber einen entsprechenden Eintrag für die Übergangsfunktion. `transitions` ist ein *Array* aus je einem `Array` pro `states`-Eintrag, das je eine *Number* pro `alphabet`-Eintrag beinhaltet, die den Index des Zustands innerhalb von `states` beschreibt, die bei einem Übergang vom zum äußeren *Array* korrespondierenden Zustand mittels des zum inneren `Array` korrespondierenden Symbols erreicht wird. Für nichtdeterministische Übergänge kann diese *Number* auch ein *Array* aus *Numbers* sein, die wieder Indices innerhalb von `states` beschreiben. `accepting` ist ein `Array` aus Indices von `states`-Einträgen, die die Menge der akzeptierenden Zustände bilden.

description Ist ein *Object* mit *Members*, die Übersetzungen der Freitextbeschreibung einer Sprache als *String* beschreiben. Die Namen dieser *Member* korrespondieren dabei zu Sprachkürzeln wie `de` oder `en`. Der Name `""` ist reserviert für eine Standardbeschreibung, die angezeigt wird, wenn für eine geforderte Sprache kein *Member* vorhanden ist.

6.2.1 Wortzugehörigkeitsproblem

Hier werden zu einem vorgegebenen Automaten Wörter über dessen Alphabet generiert und es soll festgestellt werden, ob sie zur Sprache des Automaten gehören. Unterstützung erhalten Studierende dabei in Form einer interaktiven Prüfung ihrer Herangehensweise. So ist es möglich, für jedes zu verarbeitende Symbol des generierten Wortes die erreichten Zustände zu markieren und diesen Vorschlag prüfen zu lassen. Ist kein Symbol mehr übrig, können diejenigen Markierungen entfernt werden, die nicht auf akzeptierenden Zuständen liegen, was wiederum geprüft wird. Darauf basierend wird dann eine Entscheidung bezüglich der ursprünglichen Fragestellung gefordert und geprüft. Es können jedoch zu jedem Zeitpunkt Entscheidungen eingereicht werden, die dann entweder bejaht oder mit einem Hinweis zur Korrektur beantwortet werden. In Abb. 7.1 ist die GUI dieser Aktivität am Beispiel eines NEAs dargestellt, der Wörter über dem Alphabet $\{0, 1\}$ akzeptiert, die keine 1 und eine gerade Anzahl 0 enthalten.

Es wurde das in Abschnitt 5.1 besprochene Passive-View-MVP-Muster als Grundlage verwendet. Der Presenter nimmt über seine Methoden Benutzereingaben entgegen und weist Aktualisierungen im Model an, die er dann der View zur Darstellung übergibt. Diese verwaltet hierbei einen Cytoscape.js-Graphen als Darstellung des zugrundeliegenden Automaten und einige Textfelder, die die aktuelle Konfiguration erläutern. Das Model hält eine reglibcpp-Repräsentation des Automaten sowie den abgearbeiteten, abzuarbeitenden und aktuell abgearbeiteten Teil des Eingabeworts vor. Außerdem ver-

waltet es eine Liste der im vorherigen Schritt erreichten und noch nicht betrachteten Zustände, der vom Benutzer als im nächsten Schritt erreichbar markierten Zustände, des aktuell betrachteten zuvor erreichten Zustands und der von diesem erreichbaren und noch nicht markierten Zustände. Auf Anweisung des Presenters werden Zustände in die Liste als erreichbar markierter Zustände geschrieben, falls sie in der Liste der vom aktuell betrachteten Zustand aus erreichbaren Zustände standen und anschließend geprüft, ob diese Liste abgearbeitet ist. Ist das der Fall, wird der aktuell betrachtete Zustand aus der Liste noch nicht betrachteter Zustände entfernt und geprüft, ob diese Liste abgearbeitet ist und gegebenenfalls das nächste Eingabesymbol betrachtet. Schließlich bleiben entweder keine erreichbaren Zustände oder keine Symbole mehr übrig. Im ersten Fall wird die Schaltfläche „Wort ist in Sprache“ direkt ausgeblendet, ansonsten wird dazu eingeladen, nicht-akzeptierende erreichte Zustände durch Antippen auszusortieren und danach einen verbleibenden akzeptierenden Zustand zu suchen. Bleibt nach dem Aussortieren kein Zustand mehr übrig, wird wieder die bejahende Schaltfläche deaktiviert, ansonsten wird nach Auswahl eines akzeptierenden erreichten Zustands die andere Schaltfläche deaktiviert. Das Betätigen dieser Schaltflächen lässt den Presenter im Model nachfragen, ob der Automat das Eingabewort akzeptiert und die View veranlassen, die entsprechenden Meldungen auszugeben.

6.2.2 Sprachbeschreibungen

Die Aktivität „DEAs bauen“ repräsentiert die Klasse der Sprachbeschreibungsaufgaben. Hierbei besteht die Aufgabe darin, zu einer im Freitext vorgegebenen Sprachspezifikation einen DEA zu bauen, der genau die spezifizierte Sprache akzeptiert. Dazu werden in einem Cytoscape.js-Graphen Zustände erstellt, festgelegt, ob sie Startzustand beziehungsweise akzeptierend sein sollen und Übergänge zwischen den Zuständen definiert. Anschließend kann der Lösungsvorschlag geprüft werden. Der Erfolgsfall wird bestätigt und, falls möglich, mit einem Hinweis zur Optimierung versehen. Im Fehlerfall wird ein Zeuge für die Ungleichheit der geforderten und der durch den DEA beschriebenen Sprache generiert und ausgegeben. Sollte ein partieller oder ein nichtdeterministischer Automat gebaut worden sein, wird dies ebenfalls als Fehler angemerkt.

Bei der Implementierung wurde auf eine Unterteilung in verschiedene Klassen verzichtet, da die Darstellung von vornherein stark mit dem Modell gekoppelt ist. Das Graphobjekt verwaltet die dargestellten Knoten und Kanten und die entsprechenden Markierungen mithilfe entsprechender Callback-Funktionen selbst. Für die interne Repräsentation der gewünschten Sprachspezifikation und des Lösungsvorschlags ist reglibcpp zuständig. Der Vergleich findet mithilfe der in Abschnitt 6.1.2 erläuterten Methoden statt. Ein Zeuge wird mithilfe der Mengenoperationen aus Abschnitt 6.1.1 und der Wortsuchfunktion generiert, indem das kürzeste akzeptierte Wort der Differenz von Lösungsvorschlag und tatsächlicher Lösung[9] gesucht wird.

[9] Sollte diese leer sein, wird die Differenz umgedreht. Diese kann dann nicht mehr leer sein, denn ansonsten läge ein Erfolgsfall vor.

6.2.3 Transformationen

Als Demonstration dieses Aufgabentyps wurde die Umwandlung endlicher Automaten in reguläre Ausdrücke durch Zustandseliminierung gewählt, wie sie in [HMU06, Seiten 99 f.] beziehungsweise [Sip06, Seite 73] dargestellt ist. Dabei wird ein endlicher Automat vorgegeben, als verallgemeinerter NEA dargestellt und nach und nach müssen Zustände entfernt werden. Dazu müssen ihre ein- und ausgehenden Kanten durch äquivalente umleitende Übergänge ersetzt werden, indem paarweise Vorgänger- und Nachfolgezustände gewählt und ein neuer regulärer Ausdruck vorgeschlagen wird. Dieser wird nach erfolgreicher Äquivalenzprüfung mit den vorhandenen Übergängen als einziger Übergang zwischen den gewählten Zuständen eingetragen.

Das Modell verwaltet dafür eine reglibcpp-Darstellung des VNEA, eine Tabelle mit Übergangsausdrücken und im Bedarfsfall den aktuell zu eliminierenden Zustand, ausgewählten Vorgänger und Nachfolger und Listen aller Vorgänger und Nachfolger des zu eliminierenden Zustands und für den ausgewählten Vorgänger noch nicht bearbeiteter Nachfolger. Es können Zustände übergeben werden, die kontextabhängig als einer der drei ausgewählten Zustände festgelegt werden. Sind zu eliminierender, Vorgänger- und Nachfolgezustand ausgewählt, kann ein regulärer Ausdruck übergeben werden, der dann auf Äquivalenz zu den vorhandenen Übergängen geprüft wird. Im Erfolgsfall wird der angebotene Ausdruck in eine Tabelle eingetragen, die dem Automatenobjekt schließlich als Vorschlag für die Eliminierung des Zustands übergeben wird, wodurch diese dann in den Automaten übernommen werden. Danach werden die drei ausgewählten Zustände zurückgesetzt und die Übergangstabelle neu aus dem Automatenobjekt extrahiert. Das alles geschieht auf Anweisung des Presenters, der mithilfe von Callbacks über Interaktionen mit der View informiert wird und diese nach Veränderungen im Model wieder konsistent macht. Wieder hält die View ein Graphobjekt vor, das je nach Kontext entweder den gesamten VNEA darstellt oder nur eine Gegenüberstellung von ein- und ausgehenden Kanten des zu eliminierenden Zustands. Außerdem verwaltet sie einen Eingabedialog für Ausdrucksvorschläge, der auch noch einmal die relevanten Übergänge für ein Zustandstripel auflistet, da er unter Umständen den Graphen verdeckt. Sollte der Ausdruck nicht äquivalent oder fehlgeformt sein, werden hier auch entsprechende Meldungen angezeigt.

7 Demonstration der Anwendung

Wortzugehörigkeit

Für Abb. 7.1 wurde ein unnötig komplizierter NEA als Grundlage verwendet und das Wort 0001 generiert. Das Teilwort 00 führte zur Zustandsmenge $\{q_0, q_2\}$ und die Verarbeitung des Symbols 0 führt von q_0 aus zu $\{q_1, q_2, q_3\}$, wovon q_1 und q_2 bereits markiert wurden. Die Markierung von q_3 würde die Betrachtung von q_0 als Ausgangszustand beenden. Von q_2 wird dann nur noch $\{q_2\}$ erreicht und die Betrachtung würde durch die vorherige Markierung automatisch beim Auswählen von q_2 beendet werden. Daraufhin werden $\{q_1, q_2, q_3\}$ als zuvor erreicht markiert und müssen für die Verarbeitung von 1 betrachtet werden. Es ist jederzeit möglich, einen Lösungsvorschlag abzugeben.

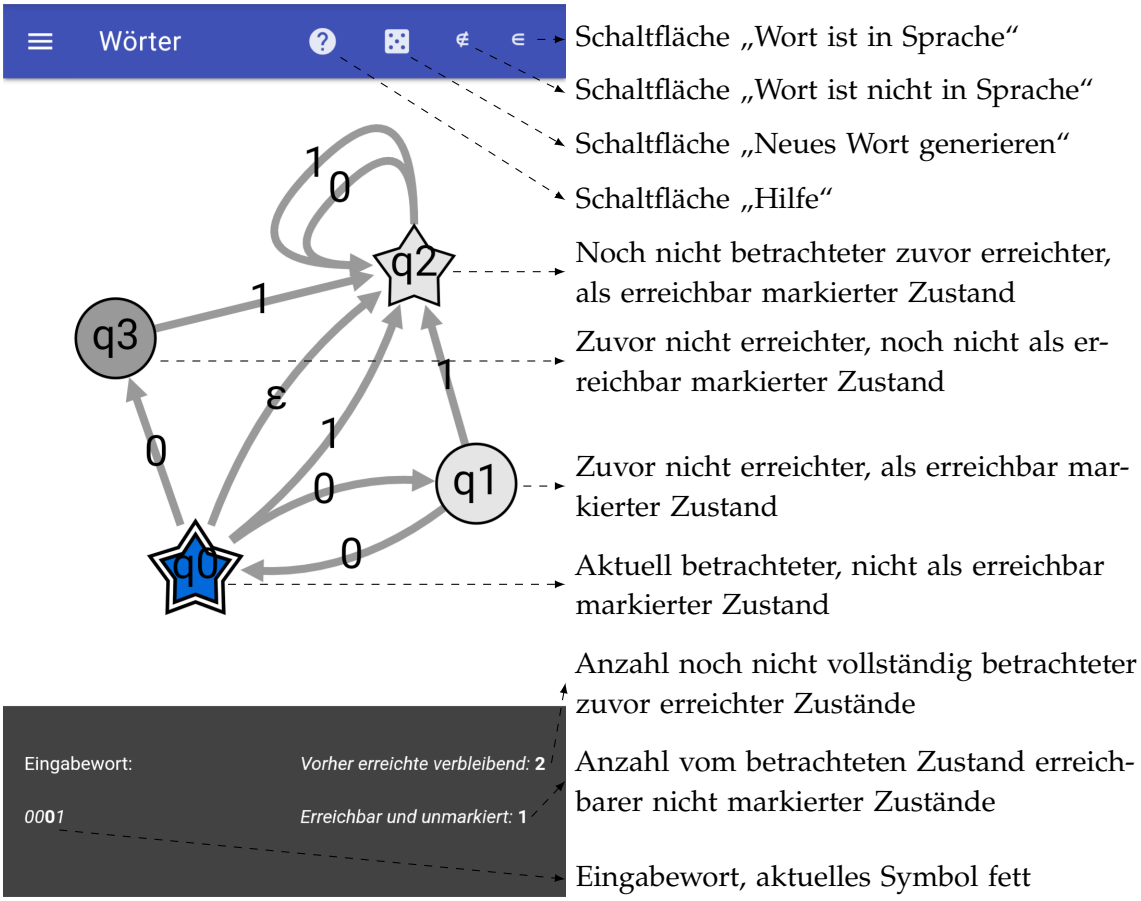
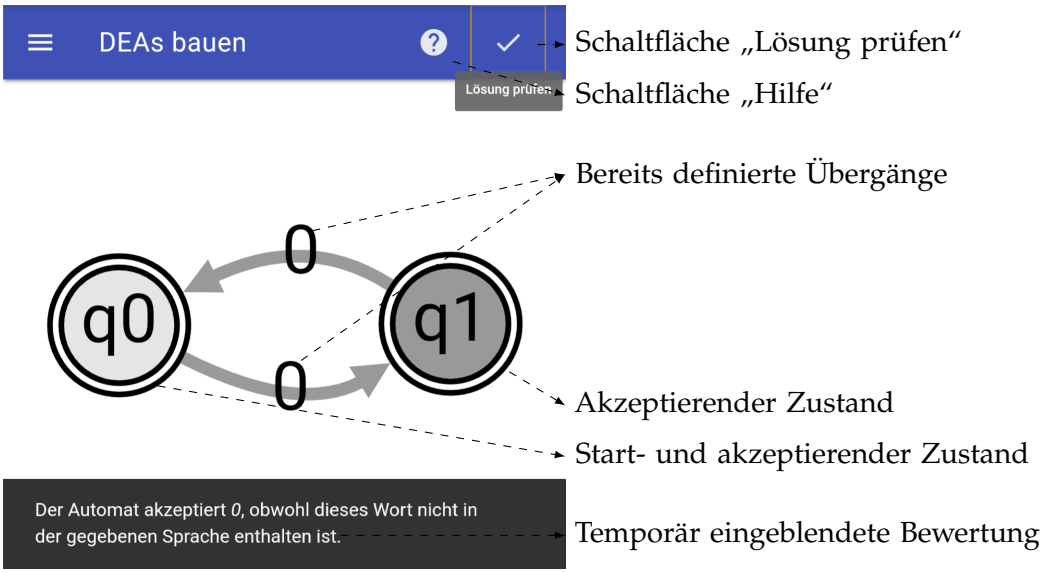


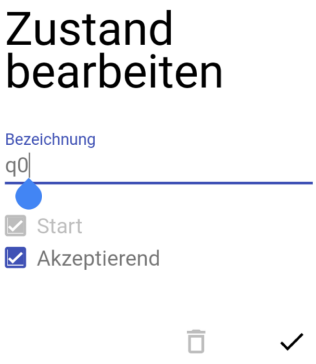
Abbildung 7.1: Screenshot der Aktivität „Wortzugehörigkeit“

DEAs bauen

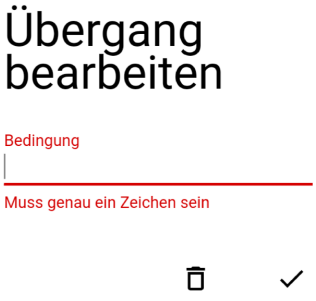
Für Abb. 7.2 wurde für die Sprache $\{w \in \{0\}^* \mid |w| \bmod 2 = 0\}$ ein nicht ganz korrekter DEA gebaut und in Abb. 7.2a als Lösung vorgeschlagen, was die in Abschnitt 6.2.2 erklärte Zeugensuche angestoßen hat. Auswählen von q_0 führte zum Dialog, der in Abb. 7.2b zu sehen ist. Dort kann sein Name und seine Zugehörigkeit zu Q_F eingestellt werden. Zu beachten ist, dass Startzustände diesen Status nur durch Designieren eines anderen Zustands verlieren und bis dahin auch nicht gelöscht werden können – die entsprechenden Schaltflächen sind deaktiviert. Die Generierung eines neuen oder das Auswählen eines vorhandenen Übergangs führt zum in Abb. 7.2c dargestellten Dialog. Dort kann das Symbol festgelegt und der Übergang auch wieder entfernt werden. Hier ist zu beachten, dass ein Symbol festgelegt werden *muss*, wenn der Dialog ohne Löschen des Zustands verlassen werden soll.



(a) Nach einem Lösungsvorschlag



(b) Zustandsbearbeitung

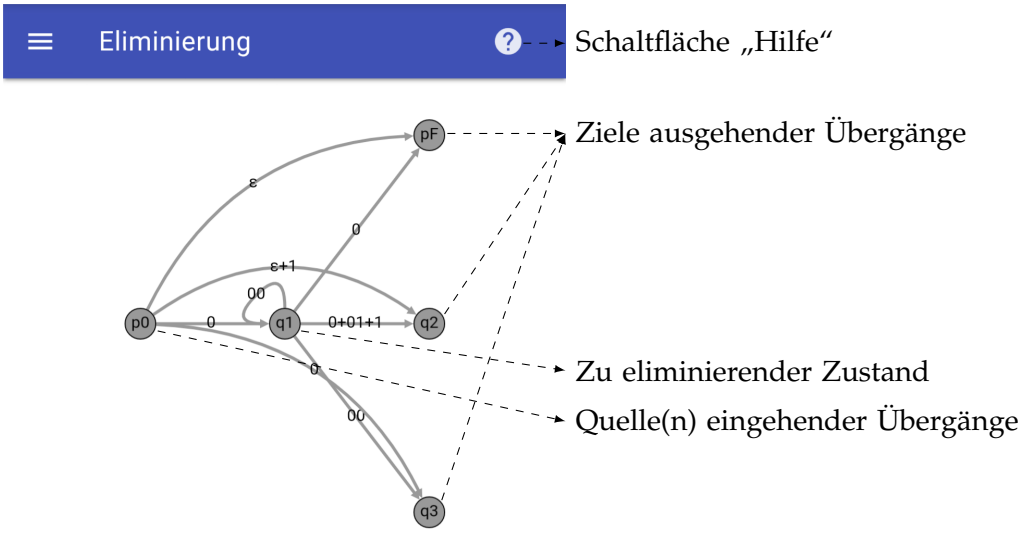


(c) Übergangsbearbeitung

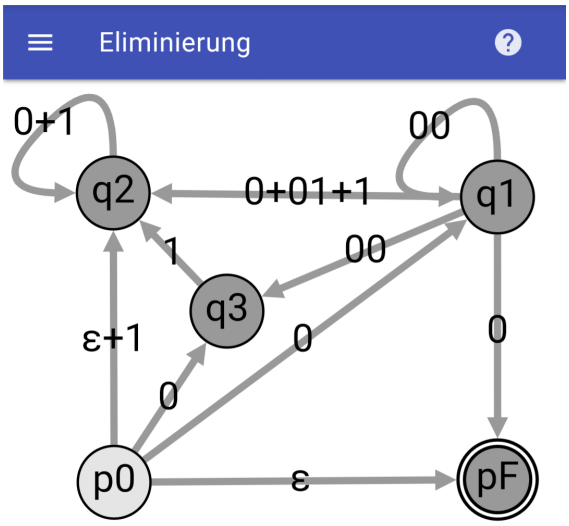
Abbildung 7.2: Screenshots der Aktivität „DEAs bauen“

Zustandseliminierung

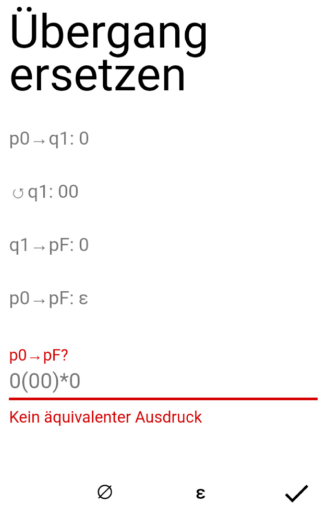
Für diese Demonstration wurde der NEA aus Abb. 7.1 geladen und automatisch in einen VNEA umgewandelt. Für Abb. 7.3b wurde q_0 entfernt, um die regulären Ausdrücke als Kantenmarkierungen hervorzuheben. Das Auswählen von q_1 zur Eliminierung führte zu Abb. 7.3a, von wo aus p_F ausgewählt wurde, um zum Dialog in Abb. 7.3c zu gelangen. Dort wird ein abgelehnter Vorschlag für die Ersetzung dargestellt – richtig wären beispielsweise $0(00)^*0 + \varepsilon$ oder auch $(00)^*$. Die Schaltflächen „$\varnothing$“ und „ε“ fügen die entsprechenden Zeichen ins Textfeld ein.



(a) Nach der Eliminierung von q_0, nach der Auswahl von q_1 zur Eliminierung



(b) Nach der Eliminierung von q_0



(c) Übergangsersetzung $p_0 \to p_F$

Abbildung 7.3: Screenshots der Aktivität „Zustandseliminierung“

Aufgabenspezifikation

Als Beispiel für das in Abschnitt 6.2 beschriebene Format ist in Auflistung 7.1 eine Möglichkeit angegeben, den NEA zu spezifizieren, der die Grundlage für Abb. 7.1 und Abb. 7.3 bildet.

```
{
  "handwritten" : {
    "nondeterministic": true,
    "states" : [
      "q0", "q1", "q2", "q3"
    ],
    "alphabet" : [ "", "0", "1" ],
    "transitions" : [
      [ [2], [1, 3] , 2 ],
      [ [], 0 , 2 ],
      [ [], 2 , 2 ],
      [ [], [] , 2 ]
    ],
    "accepting" : [ 0 ]
  },
  "description" : {
    "" : "{<i>w</i>&isin;{0,1}<sup>*</sup> | |<i>w</i>|<sub>0</sub> mod
        2 = 0 &and; |<i>w</i>|<sub>1</sub> = 0}",
    "en" : "The set of words consisting of an even number of 0s and
       containing no 1s.",
    "de" : "Alle Wörter, die aus einer geraden Anzahl 0en bestehen und
       keine 1en enthalten."
  }
}
```

Auflistung 7.1: Beispielspezifikation eines NEAs

8 Fazit und verbleibende Problemstellungen

Eine Evaluation der Anwendung durch Lernende und Lehrende steht noch aus. Geplant ist, sie in der Lehrveranstaltung *Modellierungskonzepte der Informatik* an der Universität Potsdam einzusetzen, um die Übungen zu unterstützen. Auch wäre eine rigorose Einordnung und gegebenenfalls Überarbeitung der Ziele der Lernanwendung unter lehrdidaktischen Gesichtspunkten sinnvoll. So könnten auch Experimente zur Einschätzung der Wirksamkeit der Anwendung beim Erhalt von Learning-Edge-Momentum oder beim allgemeinen Wissenszuwachs entstehen.

Das Produkt dieser Arbeit ist ein Prototyp, der noch nicht alle umsetzbaren Übungsaufgaben aus der Lehrveranstaltung abbildet; diese sollten bei Bedarf hinzugefügt werden. Die abgebildeten Übungen sind außerdem im Rahmen der technischen Möglichkeiten noch nicht zur Gänze ausgeschöpft. So gibt es keine Aktivitäten, um NEAs oder reguläre Ausdrücke zur Beschreibung von Sprachen zu bauen und es gibt keine Möglichkeit, das „Detail“ der Vollständigkeit des zu bauenden DEAs zu „überspringen“. Umgekehrt fehlen der Aktivität zur Zustandseliminierung noch Details wie die manuelle Auswahl ein- und ausgehender Kanten und die entsprechende Überprüfung dieser Auswahl. Es wären sogar Hinweise auf gewisse Suboptimalitäten eingegebener regulärer Ausdrücke möglich, wenn die Parser-Funktionen beim Konstruieren eines Ausdrucks aus der Eingabe einen kürzeren Ausdruck bauen könnten. Außerdem wurden keine Anstrengungen gemacht, Übungskonzepte für die Anwendung zu formulieren, die noch keine analoge Entsprechung haben oder wegen ihrer Aufwendigkeit haben können, aber mit der Computerunterstützung umsetzbar wären.

Auf ausführliche Dokumentation des Webanwendungs-Codes wurde wegen der fehlenden Definition einer Programmierschnittstelle verzichtet, diese könnte aber in Zukunft noch entstehen. Die Programmierschnittstelle von reglibcpp ist hingegen ausführlich dokumentiert. Dieser Code leidet lediglich an den in Abschnitt 6.1 erwähnten Einsatzstellen unter der Ineffizienz von Algorithmen, wie sie typischerweise im Bachelorstudium gelehrt werden. Eine Ersetzung durch Algorithmen, die dem aktuellen Stand der Forschung entsprechen und Vergleiche mit der aktuellen Implementierung wären wünschenswert. Außerdem wird die Korrektheit der Implementierungen der Algorithmen nicht garantiert; diese zu beweisen, könnte Inhalt eine künftigen Arbeit werden. Auch wurden zwar automatisierte Tests für einzelne Teile der Bibliothek verfasst, jedoch decken diese nicht die gesamte Programmierschnittstelle ab und entsprechen auch ansonsten nicht den Anforderungen des modernen Software-Testing.

Der Aufwand, der mit der Erstellung der Bibliothek verbunden war, überstieg in vielerelei Hinsicht die anfänglichen Erwartungen. Zunächst wurde diese unabhängig von der Anwendung in Java implementiert, da diese Programmiersprache ähnlich objektorien-

tiert ist wie C++ und die Zielplattform sich zunächst nur auf das *Android*-Betriebssystem beschränkte. Als die allgemeinere Browser-Plattform als Ziel ins Auge gefasst wurde, musste eine Entscheidung zwischen einer Java-zu-JavaScript-Transpilierung, wie in [Paw15] beschrieben, einer Neuimplementierung in JavaScript und einer Neuimplementierung in einer Sprache mit einer breiteren Auswahl an Compilerzielplattformen getroffen werden. Die Entscheidung fiel aufgrund der in Abschnitt 5.2 genannten Argumente auf die Implementierung in C++. Die in Abschnitt 6.1 beschriebene Unicode-Unterstützung, die für das Parsen von regulären Ausdrücken unerlässlich ist, musste dabei erst separat implementiert werden. Da ein Java-`String` UTF-16-kodiert ist, während C++-`std::string` auf der Byte-orientierten UTF-8-Kodierung basiert, haben Symbole außerhalb des ASCII-Zeichensatzes dort standardmäßig keine feste Länge. Das erschwert das Parsen von Symbolen wie ε und $\emptyset$ gegenüber Java, wo diese Symbol noch in der *Basic Multilingual Plane* liegen und somit, wie ASCII-Symbole auch, eine feste Länge von zwei Bytes haben. Daraufhin wurde als interne Repräsentation UTF-32 gewählt, womit *alle* Unicode-Codepunkte einheitlich vier Bytes lang sind, während die normale Programmierschnittstelle mit dem standardmäßigen UTF-8 arbeitet und diese mithilfe der C++-Standardbibliothek übersetzt.

Danach stellte sich die Integration der Bibliothek in JavaScript als schwieriger heraus, als es die Dokumentation von emscripten zunächst erahnen lässt. Zwar findet eine automatische Konvertierung von JavaScript-Strings in `std::string`-Objekte statt, jedoch nur für Zeichenketten, die sich ausschließlich aus dem *Latin1*-Zeichensatz zusammensetzen lassen. Das schließt griechische Buchstaben und das echte Leere-Menge-Symbol aus, sofern es nicht durch den Buchstaben Ø ersetzt werden soll. Um vom vollen Umfang von Unicode Gebrauch machen zu können, mussten Adapter-Objekte definiert werden, die mit `std::wstring`-Objekten arbeiten. Diese können vermittels ihres zwei Byte breiten Basistypen `wchar_t` UTF-16-Codeeinheiten direkt abbilden und mithilfe der C++-Standardbibliothek in `std::string` mit UTF-8-Kodierung umgewandelt werden. Da eine unmittelbare Umwandlung in die interne UTF-32-Kodierung nicht angeboten wird, müssen diese Adapter zwingend auf die UTF-8-Schnittstelle zurückgreifen, die selbst nichts weiter tut, als UTF-8 zu UTF-32 zu konvertieren und die echten Funktionen aufzurufen. Außerdem werden Objekte beim Übergang von der Bibliothek in den JavaScript-Code zumeist kopiert, obwohl die C++-Schnittstelle eine Referenz-Semantik vorsieht. Das reduziert die Performanz beim Aufrufen von Bibliotheksfunktionen und kann den Nebeneffekt haben, dass nicht mehr auf das Objekt zugegriffen wird, das per Referenz übergeben werden sollte, sondern auf eine Kopie davon. Zudem werden beinahe alle Objekte, die von der Bibliothek ausgegeben werden, vom JavaScript-Garbage-Collector ignoriert und ihre Lebenszeiten müssen manuell verwaltet werden. Das kann vor allem bei den unerwarteten Objekt-Kopien schnell zu Speicherlecks führen. Diese spezielle Einschränkung könnte aber mit künftigen WebAssembly-Revisionen gelindert werden, wenn dem Standard ein eigener Garbage-Collector hinzugefügt wird.

Literaturverzeichnis

[Bit17] Bitkom. *Anteil der Smartphone-Nutzer in Deutschland nach Altersgruppe im Jahr 2017*. Aug. 2017. URL: https://de.statista.com/statistik/daten/studie/459963/umfrage/anteil-der-smartphone-nutzer-in-deutschland-nach-altersgruppe/ (besucht am 9. Sep. 2018).

[BK18] Sebastian Böhne und Christoph Kreitz. „Learning how to Prove: From the Coq Proof Assistant to Textbook Style". In: Proceedings 6th International Workshop on *Theorem proving components for Educational software*, Gothenburg, Sweden, 6 Aug 2017. Herausgegeben von Pedro Quaresma und Walther Neuper. Band 267. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2018, Seiten 1–18. DOI: 10.4204/EPTCS.267.1.

[BKK16] Sebastian Böhne, Maria Knobelsdorf und Christoph Kreitz. „Mathematisches Argumentieren und Beweisen mit dem Theorembeweiser Coq". In: *Hochschuldidaktik der Informatik, HDI 2016 - 7. Fachtagung des GI-Fachbereichs Informatik und Ausbildung / Didaktik der Informatik, 13.-14. September 2016 an der Universität Potsdam, Germany*. Herausgegeben von Andreas Schwill und Ulrike Lucke. Band 10. Commentarii informaticae didacticae. Universitätsverlag Potsdam, 2016, Seiten 69–80. ISBN: 978-3-86956-376-3. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:kobv:517-opus4-93511.

[Bur] Carl Burch. *Finite automaton simulation*. URL: http://www.cs.cmu.edu/~cburch/survey/dfa/index.html (besucht am 26. Apr. 2018).

[Câm17] Cezar Câmpeanu. *Grail+ Software*. 25. Jan. 2017. URL: http://www.csit.upei.ca/~ccampeanu/Grail/ (besucht am 15. Nov. 2018).

[Cas18] Stephen Cass. *The 2018 Top Programming Languages*. IEEE Spectrum. 31. Juli 2018. URL: https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages (besucht am 26. Okt. 2018).

[Coo05] Aaron Cooper. *JGrail Specification*. Technischer Bericht CSIT-1. University of Prince Edward Island, 28. Jan. 2005.

[Coq] Action for Technological Development Coq. *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (besucht am 14. Sep. 2018).

[Dic] Kyle Dickerson. *Automaton Simulator*. URL: http://automatonsimulator.com/ (besucht am 26. Apr. 2018).

[Dot] David Doty. *Automaton Simulator*. URL: http://web.cs.ucdavis.edu/~doty/automata/ (besucht am 26. Apr. 2018).

[DW18] Dieter Dohmen und Lena Wrobel. *Entwicklung der Finanzierung von Hochschulen und Außeruniversitären Forschungseinrichtungen seit 1995*. Forschungsinstitut für Bildungs- und Sozialökonomie, 22. Aug. 2018. URL: https://www.fibs.eu/referenzen/publikationen/publikation/entwicklung-der-finanzierung-von-hochschulen-und-ausseruniversitaeren-forschungseinrichtungen-seit-1995/.

[Fow06] Martin Fowler. *GUI Architectures*. 18. Juli 2006. URL: https://www.martinfowler.com/eaaDev/uiArchs.html (besucht am 24. Okt. 2018).

[Fra+16] Max Franz, Christian Tannus Lopes, Gerardo Huck, Yue Dong, Selçuk Onur Sümer und Gary D. Bader. „Cytoscape.js: a graph theory library for visualisation and analysis“. In: *Bioinformatics* 32.2 (2016), Seiten 309–311.

[FSF] Free Software Foundation. *What is free software?* URL: https://www.gnu.org/philosophy/free-sw.html (besucht am 20. Okt. 2018).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[Gre18] Eli Grey. *l10n.js. Passive localization JavaScript library*. 16. März 2018. URL: http://purl.eligrey.com/l10n.js (besucht am 15. Nov. 2018).

[HH05] Norman Hendrich und Klaus von der Heide. „Automatische Überprüfung von Übungsaufgaben“. In: *DeLFI 2005: 3. Deutsche e-Learning Fachtagung Informatik, der Gesellschaft für Informatik e.V. (GI) 13.-16. September 2005 in Rostock*. Herausgegeben von Jörg M. Haake, Ulrike Lucke und Djamshid Tavangarian. Band 66. LNI. GI, 2005, Seiten 295–305. ISBN: 3-88579-395-4. URL: https://dl.gi.de/20.500.12116/15199.

[HMU06] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[ICS13] Fakultätsrat der mathematisch-naturwissenschaftlichen Fakultät der Universität Potsdam. *Studien- und Prüfungsordnung für das Bachelorstudium im Fach Informatik/Computational Science und das Masterstudium im Fach Computational Science an der Universität Potsdam*. 23. Jan. 2013. URL: https://www.uni-potsdam.de/de/studium/konkret/rechtsgrundlagen/studienordnungen/informatikcomputational-science.html (besucht am 14. Nov. 2018).

[JSON17] T. Bray, Herausgeber. *The JavaScript Object Notation (JSON) Data Interchange Format*. Dez. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/rfc/rfc8259.txt.

[KAPG07] Laura Korte, Stuart Anderson, Helen Pain und Judith Good. „Learning by game-building: a novel approach to theoretical computer science education“. In: *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2007, Dundee, Scotland, UK, June 25-27, 2007*. Herausgegeben von Janet Hughes, D. Ramanee Peiris und Paul T. Tymann. ACM, 2007, Seiten 53–57. ISBN: 978-1-59593-610-3. DOI: 10.1145/1268784.1268802. URL: http://doi.acm.org/10.1145/1268784.1268802.

[KKB14] Maria Knobelsdorf, Christoph Kreitz und Sebastian Böhne. „Teaching theoretical computer science using a cognitive apprenticeship approach“. In: *The 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Atlanta, GA, USA - March 05 - 08, 2014*. Herausgegeben von J. D. Dougherty, Kris Nagel, Adrienne Decker und Kurt Eiselt. ACM, 2014, Seiten 67–72. ISBN: 978-1-4503-2605-6. DOI: 10.1145/2538862.2538944. URL: http://doi.acm.org/10.1145/2538862.2538944.

[Lex16] Vincent Lextrait. *The Programming Languages Beacon*. 5. März 2016. URL: http://www.lextrait.com/vincent/implementations.html (besucht am 26. Okt. 2018).

[LL09] Martin Lange und Hans Leiß. „To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm“. In: *Informatica Didactica* 8 (2009).

[MDL] *Material Design Lite*. 27. Juni 2017. URL: https://getmdl.io/ (besucht am 15. Nov. 2018).

[Paw15] Renaud Pawlak. *JSweet: insights on motivations and design. A transpiler from Java to JavaScript*. EASYTRUST™, 16. Nov. 2015. URL: http://www.jsweet.org/papers-and-publications/.

[PULS] PULS-Team. *Potsdamer Universitätslehr- und Studienorganisationsportal*. URL: https://puls.uni-potsdam.de (besucht am 7. Sep. 2018).

[Rey17] Camilo Reyes. *The MVC Design Pattern in Vanilla JavaScript*. 14. Juli 2017. URL: https://www.sitepoint.com/mvc-design-pattern-javascript/ (besucht am 24. Okt. 2018).

[RF06] Susan H. Rodger und Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones & Bartlett Learning, 2006. ISBN: 0763738344. URL: https://www2.cs.duke.edu/csed/jflap/jflapbook/jflapbook2006.pdf.

[Rob10] Anthony V. Robins. „Learning edge momentum: a new account of outcomes in CS1“. In: *Computer Science Education* 20.1 (2010), Seiten 37–71. DOI: 10.1080/08993401003612167. URL: https://doi.org/10.1080/08993401003612167.

[RW95] Darrell R. Raymond und Derick Wood. *The Grail Papers: Version 2.3*. Technischer Bericht HKUST-CS95-17. The Hong Kong University of Science & Technology, Mai 1995. URL: http://hdl.handle.net/1783.1/32.

[SC07] Jakub Sakowicz und Jacek Chrząszcz. „Papuq: a Coq assistant“. In: *Workshop on Proof Assistants and Types in Education (PATE 2007)*. 2007, Seiten 79–96.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. 2nd. Thomson Course Technology, 2006. ISBN: 0-534-95097-3.

[Sta18] StatCounter. *Mobile Operating System Market Share in Germany*. Aug. 2018. URL: http://gs.statcounter.com/os-market-share/mobile/germany/2018 (besucht am 9. Sep. 2018).

[Uni18] The Unicode Consortium. *The Unicode Standard*. Version 11.0.0. Mountain View, CA: The Unicode Consortium, 5. Juni 2018.

[Zuk16] Olaf Zukunft. *Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen (Juli 2016)*. GI e.V., 2016. URL: https://dl.gi.de/handle/20.500.12116/2351.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „E-Learning-Unterstützung für Theoretische Informatik“ („Eine Anwendung zur Visualisierung der Konzepte regulärer Sprachmodelle“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Magdeburg, den 19. November 2018,

_____

(Tom Kranz)