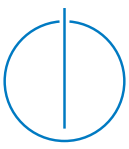
FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Kontextfreie Grammatiken in AutomataTutor

Martin Helfrich



FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Kontextfreie Grammatiken in AutomataTutor

Context Free Grammars in AutomataTutor

Autor:	Martin Helfrich
Aufgabensteller:	Univ.-Prof. Dr. Dr. h.c. Javier Esparza
Betreuer:	Julia Krämer, Salomon Sickert
Abgabedatum:	15.09.2017

Ich versichere, dass ich diese Bachelorarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15.09.2017 Martin Helfrich

Zusammenfassung

AutomataTutor ist ein bestehendes Online-Übungstool zum Bearbeiten von Aufgaben der Automatentheorie, das Nutzer durch individuelles Feedback unterstützt. Im Rahmen dieser Arbeit wird AutomataTutor um neue Aufgabentypen zu kontextfreien Grammatiken erweitert. Für jeden Aufgabentyp wird beschrieben, wie Lösungsversuche bewertet werden und das Feedback generiert wird. Dafür wird ein neu entwickelter Algorithmus zum Testen der Äquivalenz von zwei kontextfreien Grammatiken verwendet. Dieser limitierte Äquivalenztest zählt die Wörter beider Grammatiken der Länge nach auf und sucht auf diese Weise ein Gegenbeispiel für die Äquivalenz. Weiterhin wird die Umsetzung der Erweiterung von AutomataTutor dokumentiert. Dazu gehören sowohl die Benutzerschnittstelle zum Bearbeiten der Aufgaben, als auch die Implementierung der Bewertungsalgorithmen. Außerdem wird ein Experiment durchgeführt, mit dem getestet wird, ob die neuen Aufgabentypen für Studierende hilfreich sind.

Inhaltsverzeichnis

1 Einleitung

1.1 AutomataTutor

AutomataTutor ist ein online Übungstool, mit dem Studierende die Grundlagen der Automatentheorie erlernen können. Es bietet die Möglichkeit, Probleme zu stellen, die dann von den Nutzern selbständig gelöst werden können. Dabei werden sie durch individuelles Feedback unterstützt und können ihren Fortschritt anhand einer Note nachvollziehen. Das Kurssystem von AutomataTutor ermöglicht es Kurse anzubieten, in denen Nutzer Aufgabengruppen gestellt bekommen. Dabei lässt sich für jede Aufgabe eine Maximalpunktzahl und die Anzahl an erlaubten Lösungsversuchen festlegen. Bisher werden Aufgaben unterstützt, bei denen ein DFA (Determininistic Finite Automata), ein NFA (Nondeterministic Finite Automata) oder ein regulärer Ausdruck konstruiert werden muss.

Das Tool basiert auf Arbeiten von Alur et al. [Alu+13] [Dan+15], die sich mit der automatischen Generierung von Feedback bei DFA-Konstruktionen beschäftigen. Die offizielle Version von AutomataTutor ist erreichbar unter www.automatatutor.com und der Quelltext ist einsehbar auf GitHub. Die Webseite besteht aus einem C#-Backend und einem Scala-Frontend, das das Lift-Webframework verwendet. Das Frontend ist zuständig für User- und Kursmanagement und bietet eine Benutzeroberfläche zum Erstellen und Bearbeiten von Aufgaben. Die Korrektur und Benotung der Abgaben erfolgt durch das Backend.

Vorteile von AutomataTutor

- individuelles Feedback
- flexibles Arbeiten (Ort / Zeit)
- ressourcenschonende Betreuung der Studierenden

Der größte Vorteil von AutomataTutor ist das individuelle Feedback. Es unterstützt die Nutzer beim Lernen, indem sie auf die von ihnen begangenen Fehler hingewiesen werden. Sie müssen also nicht ihre Lösung mit einer Musterlösung vergleichen, sondern werden automatisch auf den fehlerhaften Teil aufmerksam gemacht.

Da es sich bei AutomataTutor um ein Online-Tool handelt, sind die Nutzer nicht an einen Ort oder eine Zeit gebunden. Sie können sich selber den Zeitraum auswählen, in dem sie üben wollen. Auch können mit AutomataTutor Online-Kurse optimal realisiert werden.

Zusätzlich ist der Einsatz von AutomataTutor für die Aufgabensteller sehr praktisch, da das individuelle Feedback ohne das Tool von Hand erstellt werden müsste. Es wird also Zeit und Geld bei der Betreuung der Lernenden eingespart.

1.2 Erweiterung für kontextfreie Grammatiken

Im Moment ist AutomataTutor auf Aufgaben zu DFAs, NFAs und reguläre Ausdrücke beschränkt. Wegen der zuvor beschriebenen Vorteile liegt aber eine Erweiterung auf andere Gebiete der Automatentheorie nahe.

Kontextfreie Grammatiken haben viele Anwendungen in der Informatik. Beispiele sind das Beschreiben der Syntax von Programmiersprachen und das Parsen von XML-Dokumenten. Ziel dieser Bachelorarbeit ist die Erweiterung von AutomataTutor durch Aufgaben zu kontextfreien Grammatiken. Hierdurch sollen Studierende bei ihrem Lernprozess zu diesem wichtigen Thema besser unterstützt werden.

1.3 Vergleichbare Angebote

Das Online Tutoring System Grammar oder (OTSG) der École polytechnique fédérale de Lausanne ist eine Website, auf der Aufgaben zu kontextfreien Grammatiken bearbeitet werden können. Die Grundlage für die Webseite ist ein Algorithmus zum automatischen Vergleichen von Grammatiken [Mad+15]. Dieser wird in Abschnitt 4.2.4 genauer behandelt. Abbildung 1.1 zeigt das Layout von OTSG.

Eine Anmeldung zur Verwendung der Webseite ist nicht erforderlich. Deshalb kann der Fortschritt eines Nutzers aber nicht gespeichert werden und es wird auch kein Kurssystem angeboten. Das Angebot beschränkt sich ausschließlich auf das Themengebiet der kontextfreien Sprachen. Für Außenstehende ist es nicht möglich, eigene Aufgaben zu stellen.[1] Die Tabelle 1.1 zeigt die Unterschiede und Gemeinsamkeiten von AutomataTutor und OTSG.

[1] Auf der Webseite von OTSG gibt es keinen Hinweis, wie neue Aufgaben erstellt werden können. Es werden auch keine Ansprechpartner genannt. (Stand: 08.09.2017)

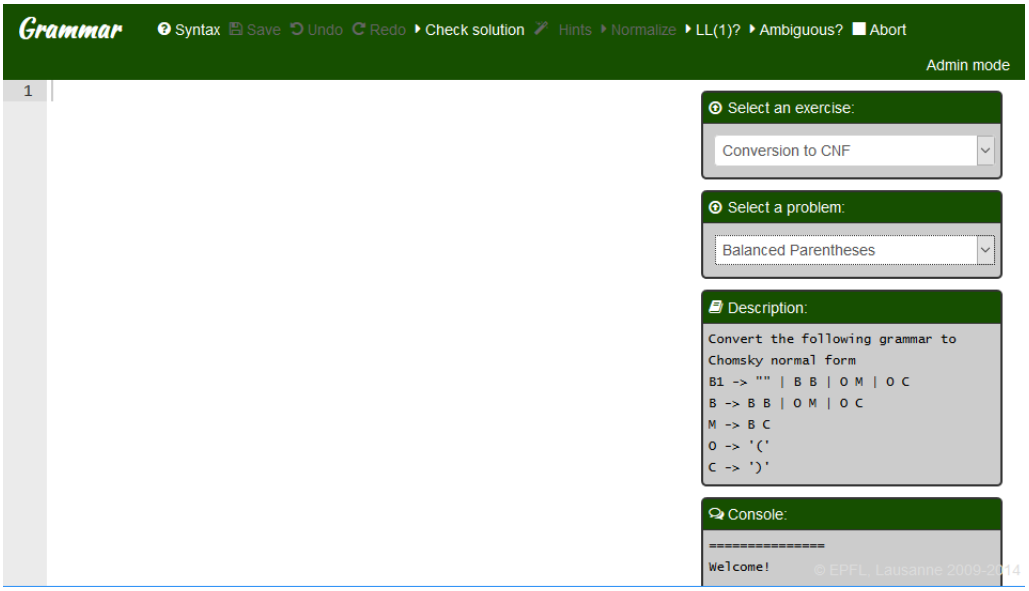


Abbildung 1.1: Layout von OTSG

	AutomataTutor	**OTSG**
Aufgabentyp: *Beschreibung nach Grammatik*	ja	ja
Aufgabentyp: *CYK-Algorithmus*	ja	ja
Aufgabentyp: *Grammatik nach CNF*	ja	ja
Aufgabentyp: *Grammatik nach GNF*	nein	ja
Aufgabentyp: *Wort ableiten*	nein	ja
Aufgabentyp: *Wörter für Grammatik*	ja	nein
individuelles Feedback	ja	ja
Kurssystem	ja	nein
Lernfortschritt wird gespeichert	ja	nein
Anmeldung	nötig	nicht nötig
weitere Themengebiete	DFAs, NFAs, RegEx	keine
eigene Aufgaben	ja	nein

Tabelle 1.1: Angebot von AutomataTutor und OTSG

1.4 Struktur der Arbeit

Im Kapitel 2 werden die Grundlagen für die Arbeit behandelt. Anschließend werden in Kapitel 3 die vier neuen Aufgabentypen beschrieben. Gleichzeitig wird dabei für jeden Aufgabentypen erklärt, wie das individuelle Feedback generiert wird. In Kapitel 4 werden dann die benötigten Algorithmen behandelt. Darunter befindet sich auch der neu entwickelte limitierte Äquivalenztest für kontextfreie Grammatiken. Die Implementierung der Algorithmen im Backend und die Umsetzung der Aufgabentypen im Frontend wird in Kapitel 5 dokumentiert. Im Kapitel 6 erfolgt die Auswertung der Erweiterung von AutomataTutor, indem das dazu durchgeführte Experiment analysiert wird. Am Ende werden in Kapitel 7 noch mögliche Erweiterungen aufgezeigt.

2 Grundlagen

2.1 Grundbegriffe

Ein **Alphabet** Σ ist eine endliche Menge an Zeichen (z.B. $\{0, 1\}$ oder $\{a, b, ..., z\}$). Ein **Wort** über Σ ist eine endliche Folge von Zeichen aus Σ. Anfang und Ende werden durch Anführungszeichen (“”) gekennzeichnet (z.B. “01101” oder “*abca*”). $|w|$ bezeichnet die **Länge** eines Wortes w (z.B. $|$“*abca*”$|$ = 4). ε ist das einzige Wort mit Länge 0. Es wird das **leere Wort** genannt. Die **Konkatenation** zweier Wörter u und v ist uv. Das Konkatenieren von “*abc*” und “*de*” ergibt beispielsweise “*abcde*”. Σ^* ist die **Menge aller Wörter** über Σ. Eine Teilmenge $L \subseteq \Sigma^*$ ist eine **(formale) Sprache**. Die **Konkatenation** für zwei Sprachen A und B ist $AB = \{uv \mid u \in A \wedge v \in B\}$. Durch das Konkatenieren der Sprachen A={“*a*”, “*ab*”} und B={“*c*”} ergibt sich somit AB={“*ac*”, “*abc*”}. A^* ist die Konkatenation von beliebig vielen Instanzen der Sprache A.

2.2 Kontextfreie Grammatiken

Definition 2.2.1. Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ ist ein 4-Tupel.

- V ist die endliche Menge an Nichtterminalen (oder Variablen).
- Σ (das Alphabet) ist die endliche Menge an Terminalen. Σ ist disjunkt von V.
- $P \subseteq V \times (V \cup \Sigma)^*$ ist die endliche Menge an Produktionen.
- $S \in V$ ist das Startsymbol.

Ein Beispiel ist die kontextfreie Grammatik

$$G_1 = (\{S, X\}, \{a, b, c\}, \{(S, aSb), (S, \varepsilon), (S, X), (X, XX), (X, c)\}, S).$$

Die Produktionen können auch folgendermaßen geschrieben werden:

$$S \to aSb \mid X \mid \varepsilon$$
$$X \to XX \mid c$$

Durch Anwenden der Produktion $S \to aSb$ lässt sich das Nichtterminal S durch aSb ersetzen.

Die kontextfreie Sprache $L(G)$ ist die Menge aller Wörter über dem Alphabet Σ, die sich aus dem Startsymbol durch wiederholte Anwendung von Produktionen ableiten lassen. Für G_1 entsteht das Wort "*aacbb*" durch die Ableitung:

$$S \to aSb \to aaSbb \to aaXbb \to aacbb$$

Somit gilt $aacbb \in L(G_1)$.

Chomsky-Normalform

Definition 2.2.2. Eine kontextfreie Grammatik G ist in Chomsky-Normalform (CNF), wenn alle Produktionen eine der Formen

$$A \to a \quad \text{oder} \quad A \to BC$$

haben.

Jede kontextfreie Grammatik $G = (V, \Sigma, P, S)$ kann in eine Grammatik G' in Chomsky-Normalform umgewandelt werden, sodass $L(G') = L(G) \setminus \{\varepsilon\}$ [Weg93]. Dafür werden nach Wegener diese vier Schritte ausgeführt:

1. `TERM`
   Entferne alle Nichtterminale $a \in \Sigma$, die in einer rechten Seite der Länge ≥ 2 vorkommen, durch Ersetzen mit der neuen Variablen X_a und Hinzufügen der Produktion $X_a \to a$ zu P.

2. `BIN`
   Ersetze jede Produktion der Form

   $$A \to B_1 B_2 \dots B_k (k \geq 3)$$

   durch

   $$A \to B_1 C_2, \quad C_2 \to B_2 C_3, \quad \dots \quad C_{k-1} \to B_{k-1} B_k$$

   und füge C_2 bis C_{k-1} als neue Variablen zu V hinzu.

3. `DEL`
   Entferne Produktionen der Form $A \to \varepsilon$ (ε-Produktionen). Erweitere dafür P induktiv zu einer Obermenge mit folgender Regel:

   Wenn $B \to \varepsilon$ und $A \to \alpha B \beta$ in P, dann füge $A \to \alpha\beta$ hinzu.

   Dann entferne alle ε-Produktionen, da sie nicht mehr benötigt werden.

4. `UNIT`
   Entferne Produktionen der Form $A \to B$ (Kettenproduktionen). Erweitere dafür P induktiv zu einer Obermenge mit folgender Regel:

   Sind $A \to B$ und $B \to \alpha$ in P und $\alpha \neq A$, dann füge $A \to \alpha$ hinzu.

   Dann entferne alle Kettenproduktionen, da sie nicht mehr benötigt werden.

Die Reihenfolge dieser vier Schritte beeinflusst, wie viele Produktionen die resultierende Grammatik G' hat. Die oben angegebene Reihenfolge von Wegener resultiert in $\mathcal{O}(|P|^3)$ Produktionen. Bei anderen Reihenfolgen kann sich aber auch eine exponentielle Größenänderung ergeben [LL09].

2.3 Eingabeformat

Um Aufgaben zu kontextfreien Grammatiken zu ermöglichen, muss ein Eingabeformat festgelegt werden. Der Syntax zum Eingeben einer Grammatik als Text ist:

- eine Produktion pro Zeile mit der Form
  `<NT> -> <RechteSeite1> | <RechteSeite2> | ...`
- erste Variable ist Startsymbol
- mögliche Terminale: `abcdefghijklmnopqrstuvwxyz()[]{}`
- mögliche Variablen: Großbuchstabe gefolgt von beliebig vielen Großbuchstaben oder Zahlen
- aufeinanderfolgende Variablen: durch Leerzeichen getrennt
  z.B: “`S -> A B`”
- leeres Wort: eine leere rechte Seite einer Produktion
  z.B: “`S -> `”

Die Beispielgrammatik G_1 ergibt sich dann durch folgende Eingabe:

```
S -> aSb | X |
X -> X X | c
```

Dieses Format hat den Vorteil, dass Produktionen sehr einfach aufschrieben werden können. Die Nichtterminale und Terminale müssen nicht explizit definiert werden, da sie automatisch erkannt werden. Auch ist das Zusammenfassen von Produktionen mit identischer linker Seite durch das Zeichen “`|`” möglich.

Allerdings ist es etwas ungewohnt, dass bei aufeinanderfolgenden Variablen nicht “`AB`” sondern “`A B`” geschrieben werden muss. Dieses Regelung ist aber sinnvoll, da sonst keine längeren und aussagekräftigen Variablennamen möglich wären.

3 Aufgabentypen und Feedbackgenerierung

Es wurden vier neue Aufgabentypen mit kontextfreien Grammatiken für AutomataTutor entworfen. Im Folgenden werden diese vorgestellt.

3.1 Wörter für Grammatik

Der Aufgabentyp *Wörter für Grammatik* dient der ersten Annäherung an Grammatiken. Ziel der Aufgabe ist es, die Bestandteile einer Grammatik zu verstehen. Dadurch entsteht ein erstes Bild von der durch eine Grammatik erzeugten Sprache.

Für eine gegebene Grammatik G muss der Nutzer Wörter finden, die innerhalb oder außerhalb von $L(G)$ liegen. Die Anzahl, der zu findenden Wörter innerhalb und außerhalb der von G erzeugten Sprache, ist durch den Aufgabensteller bestimmbar.

Angaben:

- G: kontextfreie Grammatik
- n: Anzahl der zu findenden Wörter innerhalb von G
- m: Anzahl der zu findenden Wörter außerhalb von G

Beispiel:

Für die Grammatik mit den Produktionen:

$$S \to aSb \mid X$$
$$X \to cX \mid c$$

Gib 3 Wörter an, die von der Grammatik erzeugt werden und gib 3 Wörter an, die nicht von der Grammatik erzeugt werden!

Feedback

Ein Lösungsversuch besteht aus den Wortfolgen $in_1, ..., in_n$ und $out_1, ..., out_m$. Für die Wörter $in_1, ..., in_n$ wird überprüft, ob sie auch tatsächlich innerhalb der Sprache $L(G)$ liegen. Bei den Wörtern $out_1, ..., out_m$ wird stattdessen überprüft, dass sie nicht Teil der Sprache $L(G)$ sind. Die Anzahl an unterschiedlichen Wörtern, die korrekt zugeordnet wurde, sei x. Dann wird der Lösungsversuch mit

$$\lfloor x * maxGrade/(n+m) \rfloor$$

benotet. Zusätzlich wird für jedes Wort w, falls zutreffend, einer dieser Sätze ausgegeben:

1. Du hast fälschlicherweise behauptet, dass $w \in L(G)$. (Tipp: Der Prefix w' von w ist noch zu einem Wort in $L(G)$ erweiterbar.)
2. Du hast fälschlicherweise behauptet, dass $w \notin L(G)$.
3. Du hast das Wort w mehrfach verwendet. (Du kannst pro Wort nur einmal Punkte bekommen.)

3.2 Beschreibung nach Grammatik

Der Aufgabentyp *Beschreibung nach Grammatik* soll den Lernenden helfen, ein besseres Verständnis für kontextfreie Grammatiken zu erlangen. Sie sollen ein Gefühl für die Mächtigkeit der Typ-2-Grammatiken in der Chomsky-Hierarchie entwickeln.

Für eine gegebene Beschreibung einer kontextfreien Sprache muss eine Grammatik entworfen werden, die genau diese Sprache erzeugt. Der Aufgabensteller gibt neben der Beschreibung auch eine kontextfreie Grammatik an, mit der die Abgaben intern auf Äquivalenz getestet werden.

Angaben:

- B: Beschreibung der kontextfreien Sprache (eventuell mit Beispielen)
- G' (intern): Referenzgrammatik für Äquivalenztest

Beispiel:

Gib eine kontextfreie Grammatik an, die alle gültigen Klammerausdrücke aus den Zeichen "`()[]`" erzeugt. Erzeugt werden müssen zum Beispiel "`()[](())`" und "`[([])]`", nicht aber "`[(])`".

Feedback

Es muss die Äquivalenz von der angegebenen Grammatik G und der Musterlösung G' überprüft werden. Dafür werden die x kürzesten Wörter in $L(G) \cup L(G')$ in 3 Gruppen aufgeteilt:

- $A = L(G) \cap L(G')$ sind die von beiden Grammatiken erzeugten Wörter.
- $B = L(G) \setminus L(G')$ sind die Wörter, die erkannt werden, obwohl sie nicht erkannt werden sollten.
- $C = L(G') \setminus L(G)$ sind die Wörter, die nicht erkannt werden, obwohl sie erkannt werden sollten.

Mit dem Parameter x lässt sich der Rechenaufwand beeinflussen. Die Note ergibt sich dann folgendermaßen:

$$grade = \left\lfloor \frac{|A|}{|A| + |B| + |C|} * maxGrade \right\rfloor$$

Zusätzlich werden, falls zutreffend, diese Sätze ausgegeben:

1. Deine Grammatik erkennt Wörter, die nicht erkannt werden sollen (z.B. b).
2. Deine Grammatik erkennt Wörter nicht, die erkannt werden sollen (z.B. c).

Dabei sind $b \in B$ und $c \in C$ möglichst kurze Beispielwörter.

3.3 Grammatik nach CNF

Viele Algorithmen arbeiten mit Grammatiken in CNF. Die Umwandlung von einer beliebigen kontextfreien Grammatik in CNF stellt somit eine wichtige Grundlage dar. Der Aufgabentyp *Grammatik nach CNF* ermöglicht es Studierenden, diese Umwandlung selbständig zu üben.

Für eine gegebene Grammatik G muss der Nutzer eine Grammatik G' in CNF angeben, die die Sprache $L(G) \setminus \{\varepsilon\}$ erzeugt.

Angaben:

- G: kontextfreie Grammatik

Beispiel:

Für die Grammatik G mit den Produktionen:

$$S \to SS \mid S \mid (S) \mid [S] \mid \varepsilon$$

Gib eine CNF G' an, so dass $L(G') = L(G) \setminus \{\varepsilon\}$.

Feedback

Zuerst wird überprüft, ob die Grammatik der Abgabe in CNF vorliegt. Falls sie es nicht ist, wird eine Hinweise mit einer Produktion ausgegeben, die die Normalform verletzt. Abgaben dieser Art werden nicht als Lösungsversuch gewertet.

Ansonsten wird das Feedback genau wie beim Aufgabetyp *Beschreibung nach Grammatik* generiert (siehe Abschnitt 3.2).

3.4 CYK-Algorithmus

Der Cocke-Younger-Kasami-Algorithmus entstand in den 1960er Jahren und löst das Wortproblem für kontextfreie Sprachen in CNF (siehe Abschnitt 4.1). Dieser Aufgabentyp hilft beim Einüben des Algorithmus.

Für eine gegebene Grammatik in CNF und ein Wort muss der Nutzer die CYK-Tabelle ausfüllen.

Angaben:

- G: kontextfreie Grammatik in CNF
- w: zu prüfendes Wort

Beispiel:

Für die Grammatik mit den Produktionen:

$$S \to XY \mid DD \mid d$$
$$Y \to SX$$
$$D \to d$$
$$X \to a$$

Führe den CYK-Algorithmus aus!

Feedback

Die Einträge der CYK-Tabelle werden zeilenweise überprüft. Dabei wird mit der untersten Zeile begonnen und solange fortgefahren, bis eine inkorrekte Zeile gefunden wurde. Wenn die ersten x Zeilen korrekt waren, wird die Abgabe mit

$$\left\lfloor \frac{x}{|w|} * maxGrade \right\rfloor$$

benotet. Wenn eine fehlerhafte Zeile gefunden wurde, werden für jedes Feld alle zutreffenden Sätze ausgegeben:

1. In Feld (a, b) fehlt mindestens ein Nichtterminal.
2. In Feld (a, b) sind Nichtterminale, die dort nicht hingehören.

Dabei sind a und b die Bezeichner des entsprechenden Feldes in der CYK-Tabelle.

4 Algorithmen

4.1 Wortproblem

Das Problem, bei dem entschieden werden muss, ob ein Wort w mit Länge n zu einer gegebenen Sprache gehört, heißt Wortproblem. Bei kontextfreien Grammatiken lässt es sich durch den CYK-Algorithmus entscheiden, der in den 1960er Jahren unabhängig von Cocke [CS70], Younger [You67] und Kasami [Kas65] entwickelt wurde. Dieser hat die Laufzeit $\mathcal{O}(n^3)$ und benötigt $\mathcal{O}(n^2)$ Speicherplatz. Er setzt voraus, dass die Grammatik $G = (V, \Sigma, P, S)$ in CNF vorliegt und verwendet dynamische Programmierung.

Der CYK-Algorithmus beruht auf der Idee, dass für Grammatiken in CNF längere Wörter (Länge > 1) nur durch Kombination von zwei kürzeren Wörtern entstehen können.

Algorithmus

$w_{i,j} := w_i \cdots w_{i+j-1}$ ist das Teilwort ab Index i mit Länge j. Die Menge an Nichtterminalen, die $w_{i,j}$ erzeugen können, sei $V_{i,j}$. Zu Beginn sind alle $V_{i,j}$ leer und zusammen ergeben sie die CYK-Tabelle.

$V_{1,5}$				
$V_{1,4}$	$V_{2,4}$			
$V_{1,3}$	$V_{2,3}$	$V_{3,3}$		
$V_{1,2}$	$V_{2,2}$	$V_{3,2}$	$V_{4,2}$	
$V_{1,1}$	$V_{2,1}$	$V_{3,1}$	$V_{4,1}$	$V_{5,1}$
w_1	w_2	w_3	w_4	w_5

Tabelle 4.1: CYK-Tabelle für Wort der Länge 5

Algorithmus 1: CYK-Algorithmus

```
1  n ← |w|;
2  for i = 1 to n do                                          // fill line 1
3      forall (l → r) ∈ P do
4          if r = w_i then
5              V_{i,1} ← V_{i,1} ∪ {l};
6  for j = 2 to n do                                          // fill line j
7      for i = i to n − j + 1 do
8          for k = i to j − 1 do
9              V_{i,j} ← V_{i,j} ∪ {l ∈ V | ∃B, C ∈ V : (l → BC) ∈ P ∧ B ∈ V_{i,k} ∧ C ∈
                V_{i+k,j−k}};
10 return (S ∈ V_{1,n});
```

4.2 Limitierter Äquivalenztest für kontextfreie Grammatiken

Die Frage, ob zwei kontextfreie Grammatiken dieselbe Sprache erzeugen, ist unentscheidbar [Sch92]. Da die Aufgabentypen *Beschreibung nach Grammatik* und *Grammatik nach CNF* aber einen Äquivalenztest benötigen, wurde ein limitierter Äquivalenztest entworfen.

Gegeben seien die Grammatiken $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$ in CNF. Der limitierte Äquivalenztest soll die Äquivalenz $L(G_1) = L(G_2)$ widerlegen, indem er ein Gegenbeispiel findet.

Die Idee des Tests ist es, alle Wörter der Sprachen $L(G_1)$ und $L(G_2)$ der Länge nach aufzuzählen. Sei $L_k(G)$ die Menge der Wörter in $L(G)$ mit Länge k. Dann gilt offensichtlich:

$$L_k(G) = \{w \mid w \in L(G) \wedge |w| = k\} \tag{4.1}$$

$$\bigcup_{k=0}^{\infty} L_k(G) = L(G) \tag{4.2}$$

Zu Beginn testet der Algorithmus, ob das leere Wort bereits ein Gegenbeispiel für die Äquivalenz ist. Dann werden $L_1(G_1)$ und $L_1(G_2)$ berechnet und überprüft, ob beide Mengen äquivalent sind. Wenn dies nicht der Fall ist, wird ein Gegenbeispiel ausgegeben. Ansonsten wird der Äquivalenztest beliebig lange mit immer längeren Wörtern fortgeführt.

Algorithmus 2: limitierter Äquivalenztest

```
1  if (ε ∈ G1) ≠ (ε ∈ G2) then                                  // Test ε
2      return ε;
3  k ← 1;
4  while «TERMINATIONCONDITION» is false do
5      W1 ← generateWordsWithLength(G1, k, dp1);                // W1 = Lk(G1)
6      W2 ← generateWordsWithLength(G2, k, dp2);                // W2 = Lk(G2)
7      forall w in W1 do                                        // Try: find w ∈ (W1 \ W2)
8          if w ∉ W2 then
9              return w;
10     forall w in W2 do                                        // Try: find w ∈ (W2 \ W1)
11         if w ∉ W1 then
12             return w;
13     k ← k + 1;
```

Wenn die zwei Sprachen äquivalent sind, kann kein Gegenbeispiel für die Äquivalenz gefunden werden. Deshalb muss der Algorithmus durch eine Abbruchbedingung limitiert werden. Es bieten sich beispielsweise Kriterien wie die maximal zu überprüfende Wortlänge oder ein Zeitlimit an.

Die Wörter mit einer bestimmten Länge werden durch die Funktion *generateWordsWithLength* generiert. Diese beruht wie der CYK-Algorithmus auf der Beobachtung, dass bei kontextfreien Grammatiken in CNF alle Wörter, die mehr als 1 Zeichen lang sind, nur durch Produktionen der Form $C \to AB$ entstehen können. Wieder wird mit dynamischer Programmierung gearbeitet.

Für eine gegebene Grammatik $G = (V, \Sigma, P, S)$ beschreibt die Menge $dp[X][k]$ alle Wörter, die durch das Nichtterminal X erzeugt werden können und Länge k haben. Diese Mengen lassen sich als eine DP-Tabelle darstellen:

Länge	S	V_1	...	V_m
1	$dp[S][1]$	$dp[V_1][1]$	...	$dp[V_m][1]$
2	$dp[S][2]$	$dp[V_1][2]$	...	$dp[V_m][2]$
3	$dp[S][3]$	$dp[V_1][3]$	...	$dp[V_m][3]$
⋮	⋮	⋮	⋱	⋮
k	$dp[S][k]$	$dp[V_1][k]$	...	$dp[V_m][k]$

Bei der ersten Iteration wird die erste Zeile der DP-Tabelle berechnet. Bei der i-ten Itera-

tion werden dann die schon berechneten Mengen (in den Zeilen 1 bis $i-1$) verwendet, um die i-te Zeile zu berechnen.

Zum Berechnen von $dp[C][length]$ wird für jede Produktion der Form $C \to AB$ und Schnittposition $k \in \{1, 2, ..., length - 1\}$ die Konkatenation von $dp[A][k]$ mit $dp[B][length - k]$ betrachtet.

Funktion generateWordsWithLength(G, l, dp)

```
1  if l = 1 then
2      forall v ∈ V do
3          dp[v][1] ← {t ∈ Σ | (v → t) ∈ P};
4  else
5      forall (C ∈ V) do
6          forall (C → AB) ∈ P do
7              for k = 1 to l − 1 do            // Try all length combinations!
8                  forall w1 ∈ dp[A][k] do                          // |w1| = k
9                      forall w2 ∈ dp[B][l − k] do                  // |w2| = l − k
10                         dp[C][l] ← dp[C][l] ∪ {w1w2};            // |w1w2| = l
11 return dp[S][l];                                                 // return L_l(G)
```

Beispiel:

Für die Beispielgrammatik $G = (\{S, A, B, T\}, \{a, b\}, P, S)$ mit den Produktionen P

$$S \to SS \mid AB \mid AC$$
$$A \to a$$
$$B \to b$$
$$C \to SB$$

ergibt sich nach 6 Iterationen folgende DP-Tabelle:

Länge	S	A	B	C
1	$\emptyset$	$\{a\}$	$\{b\}$	$\emptyset$
2	$\{ab\}$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$	$\{aba\}$
4	$\{aabb, abab\}$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\{aabbb, ababb\}$
6	$\{aaabbb, aababb, aabbab, abaabb, ababab\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Wie ist die Menge $dp[S][6]$ entstanden?
Es gibt 3 mögliche Produktionen, die die Variable S ersetzen:

1. $S \to AB$
   Für A und B gibt es nur Wörter der Länge 1. Somit kann aus dieser Produktion kein Wort der Länge 6 entstehen.

2. $S \to AC$
   Für A gibt es nur genau das Wort "a", das erzeugt werden kann. Somit muss nur die Kombination $dp[A][1]\, dp[C][5]$ betrachtet werden. Es ergeben sich die Wörter "$aaabbb$" und "$aababb$".

3. $S \to SS$
   Für S gibt es bisher 2 verschiedene Längen von Wörtern, die erzeugt werden können. Aus diesem Grund muss sowohl die Kombination $dp[S][2]\, dp[S][4]$ als auch die Kombination $dp[S][4]\, dp[S][2]$ betrachtet werden. Es ergeben sich die Wörter "$abaabb$", "$ababab$" und "$aabbab$".

4.2.1 Korrektheit

Zuerst wird gezeigt, dass die Funktion `generateWordsWithLength` die dp-Tabelle korrekt ausfüllt. Nach dem k-ten Aufruf der Funktion müssen die ersten k Zeilen der Tabelle ausgefüllt sein. Es muss also gelten:

$$\forall 1 \leq i \leq k : \forall X \in V : dp[X][k] = L_k\left((V, \Sigma, P, X)\right) \tag{4.3}$$

Im Folgenden wird mit Induktion über $k > 0$ gezeigt, dass Aussage 4.3 wahr ist.

1. **Induktionsanfang**
   Für $k = 1$ berechnet die Funktion für alle Variablen (Zeile 2) alle Wörter der Länge 1, da alle Produktionen der Form $A \to a$ überprüft werden (Zeile 3). Somit gilt:

   $$\forall X \in V : dp[X][1] = L_1\left((V, \Sigma, P, X)\right)$$

2. **Induktionshypothese** (I.H.)
   Nach dem k-ten Aufruf der Funktion gilt Aussage 4.3.

3. **Induktionsschritt** ($k \to k+1$)
   Durch die I.H. sind die ersten k Zeilen der dp-Tabelle schon vor dem nächstem Funktionsaufruf ausgefüllt. Da die Funktion die Inhalte dieser Zeilen auch nicht mehr verändert, muss nur gezeigt werden:

   $$\forall X \in V : dp[X][k+1] = L_{k+1}((V, \Sigma, P, X))$$

   Da die Grammatik G in CNF vorliegt, können Wörter, die länger als 1 Zeichen sind, nur durch Produktionen der Form $A \to BC$ entstehen. Für jede Variable A (Zeile 5) müssen alle passenden Produktionen $A \to BC$ (Zeile 6) betrachtet werden. Dabei müssen alle Längenkombinationen für B und C ausprobiert werden, die in Wörtern der Länge $k+1$ resultieren (Zeile 7). Da die Mengen $L_i((V, \Sigma, P, B))$ und $L_{k+1-i}((V, \Sigma, P, C))$ bereits korrekt berechnet wurden (I.H.), entstehen beim Konkatenieren (Zeilen 8 bis 10) alle Wörter aus $L_{k+1}((V, \Sigma, P, A))$. □

Nun soll die Korrektheit des limitierten Äquivalenztests aufgezeigt werden. Für zwei Grammatiken G_1 und G_2 soll dieser ein Gegenbeispiel w finden, so dass $w \in L(G_1) \cap L(G_2)$ und $|w| \leq m^*$.

Wenn es ein solches Gegenbeispiel gibt, hat es eine bestimmte Länge m und liegt somit in genau einer der Mengen $L_m(G_1)$ und $L_m(G_2)$. Der Algorithmus überprüft der Reihe nach alle Längen bis zum Limit m^* mit Hilfe der Funktion `generateWordsWithLength`. Somit findet er das Gegenbeispiel.

Wenn es kein Gegenbeispiel gibt, werden für jede Länge die selben Wörter generiert und der limitierte Äquivalenztest terminiert.

4.2.2 Laufzeit

Im Folgenden soll die Laufzeit des limitierten Äquivalenztests untersucht werden. Diese hängt allerdings von der Datenstruktur ab, die für die Wortmengen verwendet wird. Um die Analyse zu vereinfachen, wird angenommen, dass diese sowohl zum Einfügen, als auch zum Suchen nur konstante Zeit benötigt. Diese entspricht der amortisierten Laufzeit bei Verwendung einer Hash-Tabelle.

Beim limitierten Äquivalenztest werden die dp-Tabellen für die Grammatiken G_1 und G_2 bis zu einer bestimmten Länge x konstruiert.

Lemma 4.2.1. Hat eine Grammatik G in CNF n Produktionen der Form $A \to BC$, dann wird die dp-Tabelle bis zur Zeile x konstruiert in:

$$\mathcal{O}(n * x^2 * |\Sigma|^x + |V| * |\Sigma|)$$

Beweis. Der Worst-Case ist eine Grammatik, bei der jede der n Produktionen für jede Länge alle Wörter generieren kann. Die Ausgefüllte dp-Tabelle sieht dann folgendermaßen aus:

Länge	V_1	V_2	...	$V_{\|V\|}$
1	Σ	Σ	...	Σ
2	Σ^2	Σ^2	...	Σ^2
3	Σ^3	Σ^3	...	Σ^3
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x	Σ^x	Σ^x	...	Σ^x

Das Berechnen der ersten Zeile benötigt $\mathcal{O}(|V| * |\Sigma|)$. Beim Berechnen der Zeile i müssen für jede der n Produktionen alle Längenkombinationen ausprobiert werden. Dabei entsteht jedes mal Σ^i und es ergibt sich somit die Laufzeit:

$$\sum_{l=1}^{i-1} \left(\mathcal{O}(n * |\Sigma|^i) \right)$$

Somit ist die Gesamtlaufzeit für das Erstellen der Tabelle:

$$\sum_{i=2}^{x} \left(\sum_{l=1}^{i-1} \left(\mathcal{O}(n * |\Sigma|^i) \right) \right) + |V| * |\Sigma| = \mathcal{O}(n * x^2 * |\Sigma|^x + |V| * |\Sigma|)$$

□

Der limitierte Äquivalenztest konstruiert für beide Grammatiken die dp-Tabelle. Sei n_x die Anzahl der Produktionen der Form $A \to BC$ in der Grammatik G_x und $n^* = \max(n_1, n_2)$, dann ergibt sich insgesamt die Laufzeit:

$$\mathcal{O}(n^* * x^2 * |\Sigma|^x + |V| * |\Sigma|) \tag{4.4}$$

4.2.3 Vorteile und Nachteile

Vorteile:

- leicht zu implementieren
- beliebige Genauigkeit/Laufzeit
- ermöglicht individuelles Feedback
- gut geeignet für kleine Grammatiken

Nachteile:

- Wortlänge beschränkt
- kann Äquivalenz nur widerlegen

Der vorgestellte limitierte Äquivalenztest ist sehr kurz und durch die Verwendung von dynamischer Programmierung leicht zu implementieren. Da er nicht sehr komplex ist, lässt er sich auch gut debuggen.

Ein weiterer Vorteil des Algorithmus ist, dass er beliebig viele Wörter überprüfen kann. Je länger er läuft, desto kleiner ist die Chance, dass er kein Gegenbeispiel findet, obwohl es eins gibt. Für den Einsatz in AutomataTutor ist dies sehr praktisch, da das Tool nur begrenzte Ressourcen für jede Anfrage hat.

Außerdem kann der limitierte Äquivalenztest leicht so erweitert werden, dass er nicht nur ein Gegenbeispiel liefert, sondern alle Gegenbeispiele bis zu einer bestimmten Wortlänge. Dadurch lässt sich näherungsweise bewerten, wie ähnlich sich zwei Grammatiken sind (vgl. Abschnitt 3.2). Zusammen mit einem Gegenbeispiel als Hilfestellung entsteht so das individuelle Feedback für AutomataTutor.

Da AutomataTutor von Studierenden zum Üben verwendet wird, sind alle Grammatiken in den Aufgaben relativ klein. Die Aufgaben wären sonst für die Nutzer nicht mehr ohne Hilfsmittel lösbar. Für diese Grammatiken ist ein Äquivalenztest bis zu einer bestimmten Wortlänge fast immer ausreichend.

Ein Nachteil des limitierten Äquivalenztests ist, dass er wegen der Längenbeschränkung manche Gegenbeispiele nicht finden kann. Für einen menschlichen Betrachter ist es oft leicht möglich, solche Gegenbeispiele anhand der Produktionen zu konstruieren.

G_1 :	G_2 :
$S \to aSb \mid bSa \mid \varepsilon$	$S \to aSb \mid bSa \mid \varepsilon \mid dddddd \ldots d$

Für diese beiden Grammatiken wird der limitierte Äquivalenztest kein Gegenbeispiel finden und die Äquivalenz annehmen. Das kann in AutomataTutor dazu führen, dass Lösungsversuche akzeptiert werden, die offensichtlich nicht korrekt sind.

Außerdem kann der Algorithmus die Äquivalenz nur widerlegen aber nicht zeigen, da das Äquivalenzproblem für zwei kontextfreie Grammatiken unentscheidbar ist [Sch92]. Es gibt allerdings Ansätze, die versuchen, die Äquivalenz zu beweisen (siehe Abschnitt 4.2.4).

4.2.4 Vergleich mit anderen Lösungen

Der Äquivalenztest von Madhavan et al. [Mad+15] kombiniert das Finden von Gegenbeispielen mit dem Versuch, die Äquivalenz von zwei Grammatiken zu beweisen.

Der Äquivalenzbeweis verwendet Inferenzregeln, um Invarianten in den Grammatiken zu finden. Dieser ist allerdings für beliebige kontextfreie Grammatiken nicht vollständig.

Gegenbeispiele werden mit einer Bijektion von natürlichen Zahlen auf Ableitungsbäume gefunden. Damit werden für eine der Grammatiken Wörter generiert, die maximal eine bestimmte Länge haben. Dann wird mit Hilfe des CYK-Algorithmus getestet, ob diese auch von der anderen Grammatik erzeugt werden. Anders als der vorgestellte limitierte Äquivalenztest, der die kürzesten Wörter testet, werden zufällige Wörter generiert. Dadurch können längere Gegenbeispiele bei komplexeren Grammatiken (zum Beispiel von Programmiersprachen) besser gefunden werden. Für die Übungsaufgaben auf AutomataTutor ist das Generieren der kürzesten Wörter aber ausreichend. Außerdem hat der limitierten Äquivalenztest den Vorteil, dass er immer das kürzeste Gegenbeispiel findet und es dann als Hilfestellung ausgeben kann.

5 Implementierung

5.1 Backend

5.1.1 Verwendete Bibliothek

AutomataTutor verwendet die von Microsoft bereitgestellte Bibliothek AutomataDotNet. Diese stellt Algorithmen und Datenstrukturen für die Arbeit mit regulären Ausdrücken, DFAs/NFAs und auch kontextfreien Grammatiken bereit. Die Klassen für kontextfreie Grammatiken sind:

- *GrammarSymbol*
  ist eine abstrakte Klasse, die alle Elemente von Produktionen (also Terminale und Nichtterminale) zusammenfasst.
- *Nonterminal*
  repräsentiert ein Nichtterminal.
- *Exprinal*
  repräsentiert ein Terminal.
- *Production*
  repräsentiert eine Produktion mit linker und rechter Seite.
- *ContextFreeGrammar*
  repräsentiert eine kontextfreie Grammatik. Sie stellt zwei Methoden zum Umwandeln in CNF (Chomsky Normal Form) und GNF (Greibach Normal Form) bereit.
- *GrammarParser*
  erstellt eine kontextfreie Grammatik anhand eines Textes.

Änderungen an der Bibliothek

Leider war die Verwendung vieler Teile der AutomataDotNet-Bibliothek für CFGs nicht möglich, da viele wichtige Methoden, Attribute und Klassen als unsichtbar (privat)

deklariert sind. Beispielsweise ist ein Zugriff auf die komplette Klasse *GrammarParser* nicht möglich.

Dieses Problem wurde jedoch umgangen, indem die benötigten Teile des auf GitHub veröffentlichen Quellcodes neu zum Backend hinzugefügt wurden. Dies wiederum bedeutet, dass für alle Berechnungen mit kontextfreien Grammatiken nicht die eingebundene AutomataDotNet DLL-Datei verwendet wird. Stattdessen wird der entsprechende Quellcode komplett neu kompiliert und ist somit besser anpassbar.

Verändert wurde neben der Sichtbarkeit einiger Klassen und Methoden auch der Parser. So wurden die Sonderzeichen `"()[]"` als mögliche Terminale neu hinzugefügt, um Aufgaben zu Klammerausdrücken zu ermöglichen. Weiterhin wurden neue Ausnahmen beim Parsen von Grammatiken eingeführt, um den Nutzern bei syntaktisch falschen Eingaben besseres Feedback geben zu können. Der Nutzer bekommt nun auch einen Hinweis auf die Position in seiner Eingabe, die nicht eingelesen werden konnte.

Für die beiden implementierten Algorithmen (vgl. Kapitel 4) müssen die Grammatiken in CNF (Chomsky Normal Form) vorliegen. Der durch die Bibliothek bereitgestellte Algorithmus entfernt bei der Umwandlung in CNF alle ε-Produktionen. Dabei geht die Information verloren, ob die ursprüngliche Grammatik das leere Wort beinhaltet. Aus diesem Grund wurde die Klasse *ContextFreeGrammar* um ein Attribut *emptyString* erweitert, das angibt, ob die Grammatik das leere Wort erzeugt oder nicht. Es wird vor der Umwandlung in CNF automatisch gesetzt.

5.1.2 Algorithmen für Grammatiken

Alle im Zuge dieser Bachelorarbeit implementierten Algorithmen sind in der Klasse *GrammarUtilities* zusammengefasst. Sie bietet mehrere statische Methoden an:

- Die Methode *getEquivalentCNF* generiert eine CNF für eine gegebene Grammatik. Sie verwendet den von AutomataDotNet bereitgestellten Algorithmus und kümmert sich zusätzlich um die Akzeptanz des leeren Wortes.
- Die Methode *getPrefixClosure* generiert für eine Grammatik G den Präfix-Abschluss $G' = (V', \Sigma, P', S')$. Dieser akzeptiert alle Wörter, die ein Präfix von mindestens einem Wort aus $L(G)$ sind.
- Die Methode *getCNFPrefixClosure* generiert für eine Grammatik G den Präfix-Abschluss in CNF.
- Methoden für das Wortproblem (vgl. 4.1):
  - Die Methode *cyk* implementiert den CYK-Algorithmus. Anders als im Algorithmus beschrieben gibt sie aber die CYK-Tabelle zurück.

  - Die Methode *isWordInGrammar* ruft die Methode *cyk* für eine Grammatik G und ein Wort w auf und entscheidet anhand der CYK-Tabelle, ob $w \in G$.
  - Die Methode *longestPrefixLength* führt den CYK-Algorithmus für den Präfix-Abschluss einer Grammatik G und ein Wort w aus. Dafür wird erst der Präfix-Abschluss $G' = (V', \Sigma, P', S')$ mit der Methode *getCNFPrefixClosure* generiert und damit die Methode *cyk* aufgerufen. Anhand der CYK-Tabelle wird dann der längste Präfix von w gefunden, der noch ein Präfix von G ist. Also: $max(k \in \mathbb{N}_0) : S' \in V_{1,k}$

- Methoden für den limitierten Äquivalenztest (vgl. Abschnitt 4.2):
  - Die Methode *generateWordsWithLength* ist eine Hilfsmethode für den limitierten Äquivalenztest. Sie generiert für eine CNF alle ableitbaren Wörter einer bestimmten Länge.
  - Die Methode *findDifference* führt den limitierten Äquivalenztest solange aus, bis ein Gegenbeispiel für die Äquivalenz gefunden wurde oder alle Wörter bis zu einer bestimmten Länge überprüft wurden.
    $\rightarrow$ Abbruchbedingung: Wortlänge
  - Die Methode *findDifferenceWithTimelimit* führt den limitierten Äquivalenztest solange aus, bis sie ein Gegenbeispiel findet oder ein angegebenes Zeitlimit überschritten wird.
    $\rightarrow$ Abbruchbedingung: Zeitlimit

Implementierungsdetails: CYK-Algorithmus

Die Mengen in der CYK-Tabelle wurden mit HashSets implementiert. Außerdem werden zur Laufzeitoptimierung vor dem eigentlichen Algorithmus 2 Lookup-Tabellen generiert:

- *LookupT*
  liefert für ein Zeichen $t \in \Sigma$ die Menge der Nichterminale, die dieses Zeichen erzeugen.
- *LookupNT*
  liefert für ein Paar (B, C) aus Nichtterminalen die Menge der Nichterminale, die diese Kombination erzeugen können.

5.1.3 Feedbackgenerierung

Alle Methoden zum Generieren von Feedback sind in der Klasse *GrammarGrading* zusammengefasst.

- *gradeWordsInGrammar(*$G, W_{in}, W_{out}, maxGrade$*)*
  Für ein Problem vom Type *Wörter für Grammatik* der Grammatik G berechnet die Methode das Feedback. Die maximale Punktzahl ist *maxGrade* und es wird behauptet:

  $$W_{in} \in L(G) \wedge W_{out} \notin L(G)$$

  Das Generieren von Feedback ist in Abschnitt 3.1 beschrieben.

- *gradeGrammarEquality(*$G_{sol}, G_{att}, maxGrade, time$*)*
  Die Methode generiert das Feedback für die Aufgabentypen *Beschreibung nach Grammatik* und *Grammatik nach CNF*. Dabei ist G_{sol} die Lösung und G_{att} der Lösungsversuch. Es wird behauptet:

  $$L(G_{sol}) = L(G_{att})$$

  Intern wird der limitierte Äquivalenztest mit dem Timelimit *time* durchgeführt. Das Generieren von Feedback für die Äquivalenzaufgaben ist beschrieben in Abschnitt 3.2.

- *gradeCYK(*$G, w, table, maxGrade, withExample$*)*
  Die Methode generiert das Feedback für den Aufgabentypen *CYK-Algorithmus*. Es wird überprüft, ob *table* die korrekt ausgefüllte CYK-Tabelle für das Wort w ist. In Abschnitt 3.4 wird beschrieben, wie das Feedback generiert wird.

5.2 Frontend

5.2.1 Verbesserung der Benutzeroberfläche

Ein Kurs besteht aus beliebig vielen Aufgabengruppen, die mehrere Ausgaben zusammenfassen. Für jede Aufgabe lässt sich die Maximalpunktzahl und die Anzahl an Lösungsversuchen festlegen. Als Annäherung an das Frontend von AutomataTutor wurde das Erstellen von Aufgabengruppen verbessert. Zuvor musste jedes Problem nacheinander der Problemgruppe hinzugefügt werden. Jetzt ist es möglich, mehrere Probleme gleichzeitig auszuwählen. Dadurch wird beim Kursmanagement Zeit gespart. Das neue Layout ist in Abbildung 5.1 zu sehen.

5.2.2 Wörter für Grammatik

Die Benutzeroberfläche zum Bearbeiten der Aufgabe (siehe Abbildung 5.2) besteht aus 3 Teilen:

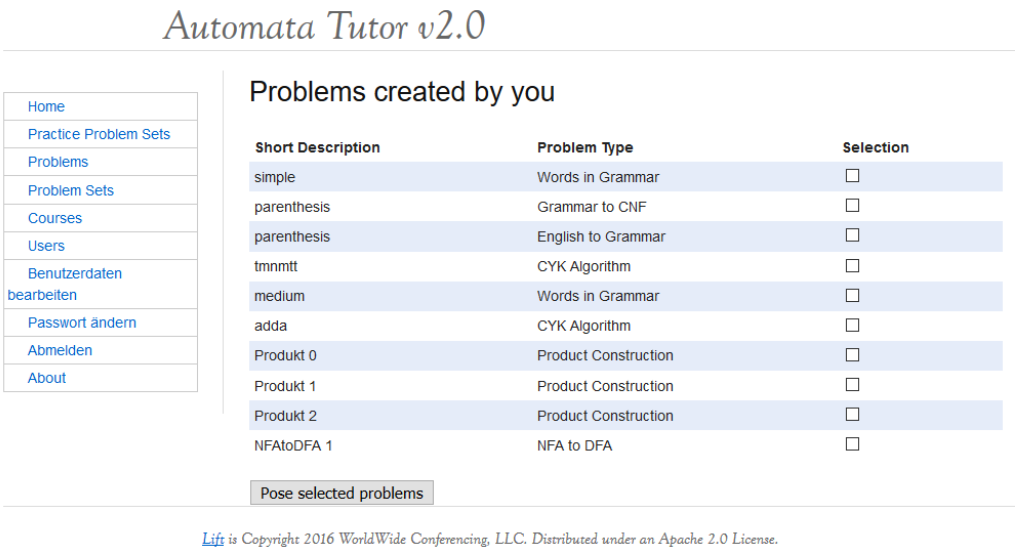


Abbildung 5.1: neue Benutzeroberfläche für das Erstellen von Aufgabengruppen

1. **Syntax**
   Es wird das Eingabeformat für Grammatiken erklärt (vgl. Abschnitt 2.3).

2. **Aufgabe**
   Die Grammatik G und die Parameter n und m werden als Aufgabenstellung ausgegeben.

3. **Eingabeformular**
   Es stehen $n+m$ Eingabefelder bereit. Diese sind auf 75 Zeichen begrenzt, um zu verhindern, dass der Server beim Prüfen einer Abgabe zu lange rechnet.

5.2.3 Beschreibung nach Grammatik

Die Benutzeroberfläche zum Bearbeiten der Aufgabe (siehe Abbildung 5.3) besteht aus 3 Teilen:

1. **Syntax**
   Es wird das Eingabeformat für Grammatiken erklärt (vgl. Abschnitt 2.3).

2. **Aufgabe**
   Die Beschreibung der kontextfreien Sprache wird ausgegeben.

3. **Eingabeformular**
   Die zu entwerfende Grammatik wird in ein großes Textfeld eingegeben.

5.2.4 Grammatik nach CNF

Die Benutzeroberfläche zum Bearbeiten der Aufgabe (siehe Abbildung 5.4) besteht aus 3 Teilen:

1. **Syntax**
   Es wird das Eingabeformat für Grammatiken erklärt (vgl. Abschnitt 2.3).

2. **Aufgabe**
   Die Grammatik G, die in CNF umgeformt werden soll, wird ausgegeben.

3. **Eingabeformular**
   Die umgeformte Grammatik wird in ein großes Textfeld eingegeben.

5.2.5 CYK-Algorithmus

Die Benutzeroberfläche zum Bearbeiten der Aufgabe (siehe Abbildung 5.5) besteht aus 3 Teilen:

1. **Syntax**
   Es wird das Eingabeformat für Grammatiken erklärt (vgl. Abschnitt 2.3).

2. **Aufgabe**
   Die kontextfreie Grammatik G und das Wort w werden ausgegeben.

3. **Eingabeformular**
   Über dem Wort w ist eine passende CYK-Tabelle sichtbar. Jedes Feld der Tabelle ist mit dem Zahlenpaar (x, y) beschriftet und stellt ein Eingabefeld bereit. In diesem werden alle Nichtterminale angegeben, die das Teilwort "$w_x, w_{x+1}, ..., w_{y-1}, w_y$" erzeugen können. Die Nichtterminale werden mit einem Leerzeichen getrennt.

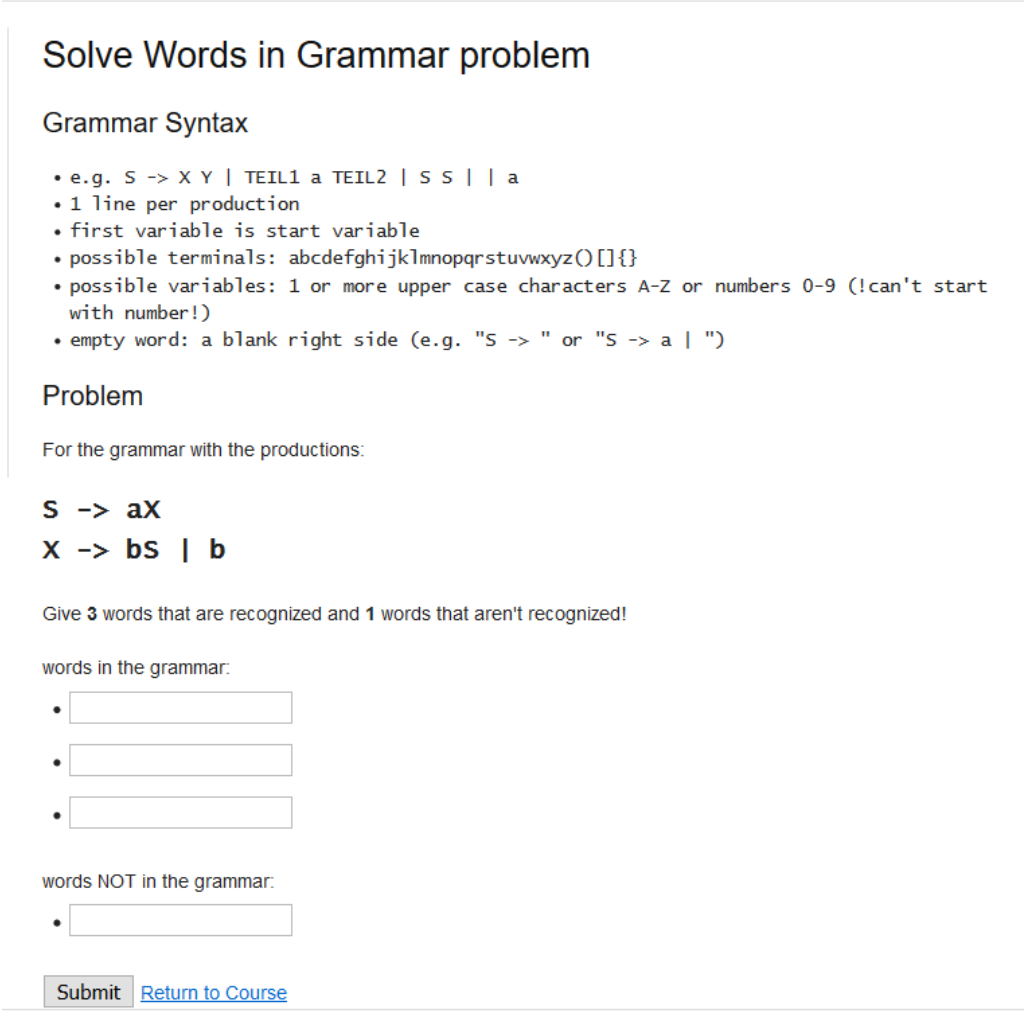


Abbildung 5.2: Lösen eines “Wörter für Grammatik”-Problems

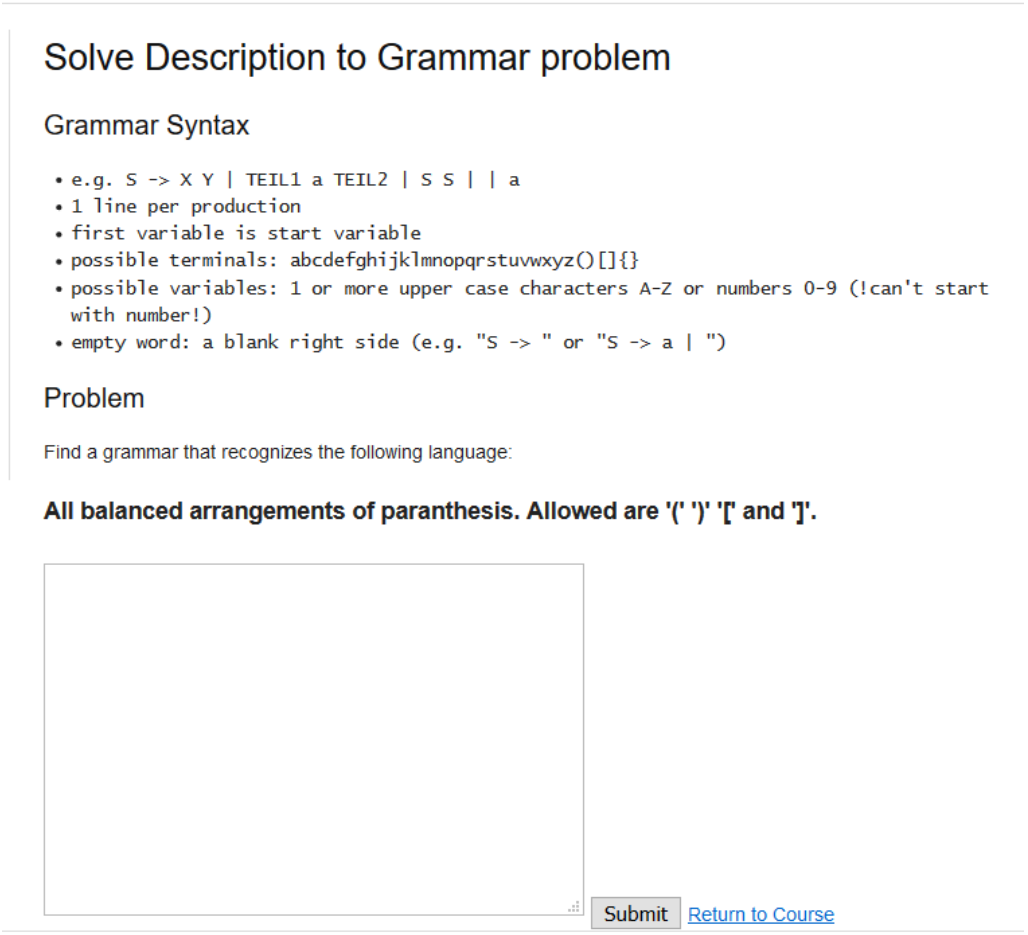


Abbildung 5.3: Lösen eines “Beschreibung nach Grammatik”-Problems

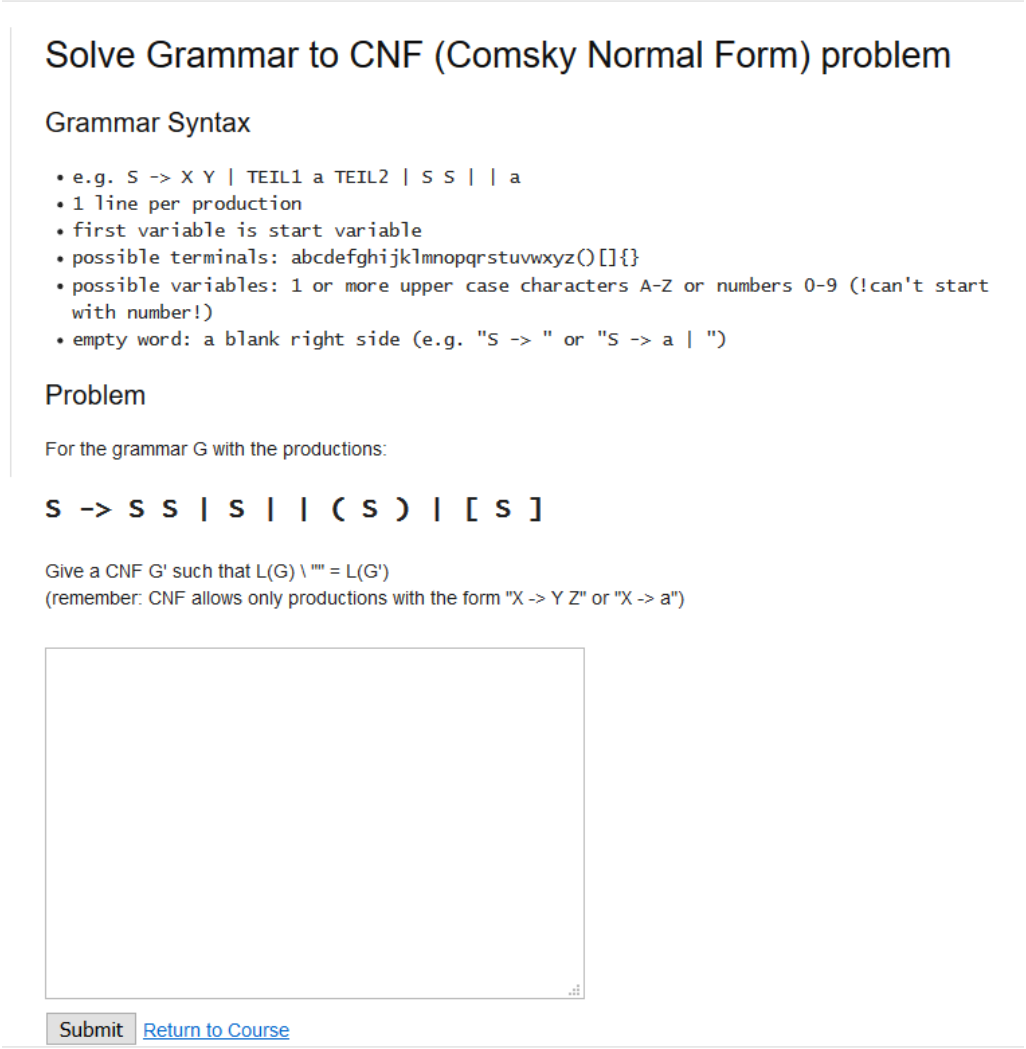


Abbildung 5.4: Lösen eines “Grammatik nach CNF”-Problems

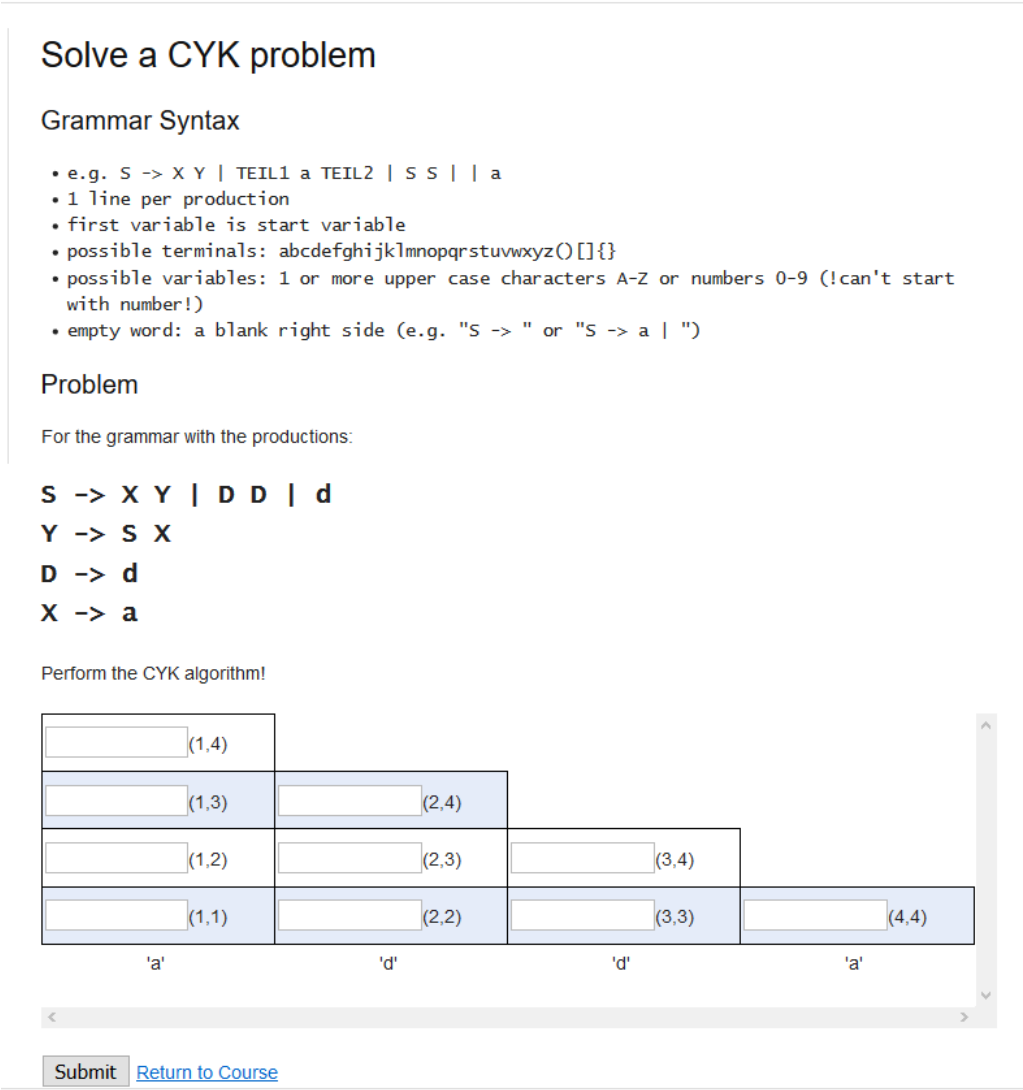


Abbildung 5.5: Lösen eines “CYK-Algorithmus”-Problems

6 Auswertung

Zum Auswerten der neuen Aufgabentypen zu kontextfreien Grammatiken wird ein Experiment mit anschließendem Feedback-Fragebogen durchgeführt. Ziel ist es herauszufinden, ob die neuen Aufgabentypen für Studenten hilfreich sind.

Teilnehmer sind 18 Studierende der Technischen Universität München, die die Vorlesung "Einführung in die theoretische Informatik" besuchen.

6.1 Ablauf

Die Teilnehmer werden zufällig in Test- und Kontrollgruppe aufgeteilt. Danach folgen 5 Phasen:

1. **Vortest**
   Alle Teilnehmer bearbeiten folgende CYK-Aufgabe auf Papier.

   > Führe den CYK-Algorithmus für das Wort "yxyab" und die Grammatik $G = (V, \Sigma, P, S)$ aus. Die Produktionen P sind:
   >
   > $$\begin{aligned} S &\to XY \mid YY \mid a \mid b \\ X &\to SY \mid x \\ Y &\to SS \mid AX \mid BY \mid y \\ A &\to AY \mid a \\ B &\to SB \mid b \end{aligned}$$

2. **Haupttest**
   Der Haupttest besteht aus 2 CYK-Aufgaben. Diese werden von der Kontrollgruppe auf Papier bearbeitet, während die Testgruppe AutomataTutor verwendet.

Führe den CYK-Algorithmus für das Wort “adda” und die Grammatik $G = (V, \Sigma, P, S)$ aus. Die Produktionen P sind:

$$S \to XY \mid DD \mid d$$
$$Y \to SX$$
$$D \to d$$
$$X \to a$$

Führe den CYK-Algorithmus für das Wort “tmnmtt” und die Grammatik $G = (V, \Sigma, P, S)$ aus. Die Produktionen P sind:

$$S \to MS \mid NT \mid m \mid n$$
$$N \to ST \mid TR \mid n$$
$$R \to TT \mid NN \mid r$$
$$M \to m$$
$$T \to NN \mid TS \mid t$$

Die erste Aufgabe ist vergleichsweise leicht und soll die Teilnehmer der Testgruppe an den Umgang mit AutomataTutor heranführen.[1]
Die zweite Aufgabe ist dafür vergleichsweise schwer und lang ($|w| = 6$). Sie wird als Vergleichsaufgabe verwendet.

3. **Nachtest**
   Alle Teilnehmer bearbeiten folgende CYK-Aufgabe auf Papier.

   Führe den CYK-Algorithmus für das Wort “acadd” und die Grammatik $G = (V, \Sigma, P, S)$ aus. Die Produktionen P sind:

   $$S \to AB \mid BA \mid c \mid d$$
   $$A \to CB \mid BD \mid a$$
   $$B \to CA \mid AD \mid b$$
   $$C \to SC \mid c$$
   $$D \to DS \mid d$$

   Der Schwierigkeitsgrad dieser Aufgabe entspricht dem der Vortest-Aufgabe. Beide Wörter haben die Länge 5 und beide Grammatiken bestehen aus 8 Produktionen. Auch müssen die CYK-Tabellen mit ungefähr gleich vielen Nichtterminalen befüllt werden. Dadurch wird ein direkter Vergleich der Ergebnisse möglich.

[1] 45,5% der Teilnehmer des Experiments hatten zuvor noch nicht mit AutomataTutor gearbeitet.

4. **Andere Aufgabentypen**
   Die Teilnehmer testen eigenständig 1-2 weitere Aufgabentypen auf AutomataTutor. Dies ermöglicht auch der Testgruppe, mit AutomataTutor zu arbeiten und Feedback zur Umsetzung zu geben.
5. **Fragebogen**
   Alle Teilnehmer füllen einen Online-Fragebogen aus. In diesem geben die Studenten zu allen mit AutomataTutor bearbeiteten Aufgabentypen Feedback. Die Kriterien sind:
   - kurze Einarbeitungszeit
   - verständliche Aufgabenstellung
   - intuitive Bedienung/Eingabe
   - hilfreiches Feedback
   - allgemeiner Nutzen

6.2 Messgrößen und Hypothesen

Für jede bearbeitete Aufgabe werden folgende Daten gesammelt:

- x: Anzahl der Versuche (nur bei Verwendung von AuotmataTutor ist $x > 1$)
- m: Endpunktzahl (zwischen 0 und 10; wird berechnet wie in Abschnitt 3.4 beschrieben)
- t: Bearbeitungszeit
- e: Effizienz $\left(e = \frac{p}{t}\right)$ in Punkte pro Minute (PpM)

Hypothesen

Die folgenden Hypothesen werden geprüft:

- *H1 - Übung*:
  Die Studenten werden durch Übung besser.
- *H2 - Verbesserung*:
  Durch AutomataTutor verbessern sich Studierende beim Üben mehr.
- *H3 - schwere Aufgaben*:
  Mit AutomataTutor lösen Studierende schwere Aufgaben besser.

- *H4 - Zeit*:
  Mit AutomataTutor bearbeiten Studierende Aufgaben schneller.

- *H5 - Effizienz*:
  Mit AutomataTutor arbeiten Studierende effizienter.

6.3 Ergebnisse

Leider sind von 3 Teilnehmern die Daten unvollständig, so dass sie nicht mit in die Tests einbezogen werden können.[2] Die effektive Teilnehmeranzahl ist somit $n = 15$.

Zur Auswertung der Daten werden Zweistichproben-t-Tests verwendet. Ein t-Test prüft anhand der Mittelwerte zweier Stichproben, ob die Stichproben derselben Population angehören oder aus zwei verschieden Populationen stammen. Wenn die Werte in den Stichproben paarweise abhängig sind (z.B. durch zwei Messungen desselben Objekts) wird ein gepaarter t-Test benötigt, sonst ein ungepaarter.

- *H1 - Übung*:
  Die Teilnehmer erreichen im Vortest durchschnittlich 5,87 Punkte und weisen eine Effizienz von 0,897 PpM auf. Beim Nachtest schneiden sie mit durchschnittlich 8,00 Punkten und einer Effizienz von 1,39 PpM deutlich besser ab. Getestet wird mit einem 2-seitigen gepaarten t-Test. Für die Punktzahl gilt $p = 0,0717$ und für die Effizienz gilt $p = 0,0865$. Somit wird **H1 nicht unterstützt**. Trotzdem lässt sich ein klarer Trend erkennen.

- *H2 - Verbesserung*:
  Teilnehmer in der Kontrollgruppe verbessern sich durchschnittlich um 3,67 Punkte und 0,879 PpM. Die Testgruppe verbessert sich um 1,11 Punkte und 0,230 PpM. Getestet wird mit einem 2-seitigen ungepaarten t-Test. Es ergeben sich für die Punktzahl $p = 0,268$ und für die Effizienz $p = 0,246$. Somit wird **H2 nicht unterstützt**.

- *H3 - schwere Aufgaben*:
  In der Vergleichsaufgabe erzielt die Kontrollgruppe durchschnittlich 3,00 Punkte. Die Teilnehmer in der Testgruppe bekommen mit AutomataTutor durchschnittlich 8,89 Punkte. Der 2-seitige ungepaarte t-Test ergibt $p = 0,0005$. Somit wird **H3 unterstützt**.

- *H4 - Zeit*:
  Bei der Vergleichsaufgabe hat die Kontrollgruppe eine Durchschnittszeit von

[2] Einige der Teilnehmer mussten auf Grund von Zeitmangel das Experiment abbrechen.

12,65 Minuten. Die Testgruppe braucht durchschnittlich 11,94 Minuten. Der 2-seitige ungepaarte t-Test ergab den Wert $p = 0,752$. Somit ist **H4 nicht unterstützt**.

- *H5 - Effizienz*:
  Die durchschnittliche Effizienz der Teilnehmer in der Kontrollgruppe bei der Vergleichsaufgabe ist 0,232 PpM. Die Testgruppe erreicht mit AutomataTutor 0,788 PpM. Der 2-seitige ungepaarte t-Test ergibt den Wert $p = 0,0190$. Somit ist **H5 unterstützt**.

Sonstiges Feedback

Aussage (zum Aufgabentyp CYK)	**Zustimmung**
Einarbeitungszeit ist kurz.	91%
Aufgabenstellung ist verständlich.	86%
Bedienung/Eingabe ist intuitiv.	77%
Feedback ist hilfreich.	75%
Aufgabentyp ist hilfreich.	84%

Allgemein lässt sich erkennen, dass die Teilnehmer die Erweiterung von AutomataTutor praktisch und hilfreich finden. In der Umfrage schreiben die Teilnehmer unter anderem:

«Danke für eure Arbeit, sie hilft enorm.»

«Spart viel Zeit[...]. Macht Spaß!»

«Sehr gut zum Üben.»

6.4 Probleme und Kritik

Bei dem durchgeführten Experiment muss kritisch angemerkt werden, dass die effektive Teilnehmerzahl mit $n = 15$ sehr klein ist. Dies könnte ein Grund dafür sein, dass die Hypothesen H1, H2 und H4 abgelehnt werden müssen. Noch nicht einmal der Test für die sehr allgemeine These H1, dass sich die Teilnehmer durch Übung verbessern, ist statistisch signifikant ($p < 0.05$).

Ein weiteres Problem ist, dass sich das Leistungsniveau von Kontroll- und Testgruppe zu Beginn des Experiments schon stark unterscheiden. Im Vortest hat die Kontrollgruppe eine durchschnittliche Effizienz von 0,43 PpM und die Testgruppe 1,21 PpM. Der 2-seitige ungepaarte t-Test ergibt $p = 0,0332$. Auch für die Punktzahl ergibt sich mit durchschnittlich 3,67 Punkten (Kontrollgruppe) und 7,33 Punkten (Testgruppe)

ein p-Wert von 0,0713. Die Ergebnisse für H3 und H5 sind somit nicht nur auf die Verwendung von AutomataTutor zurückzuführen.

6.5 Zusammenfassung

Das Experiment konnte nicht zeigen, dass das Üben mit AutomataTutor besser ist als das Üben auf Papier. Trotzdem hilft das individuelle Feedback beim Lösen von schweren Aufgaben und ermöglicht Lernenden, mehr Punkte in weniger Zeit zu erarbeiten. Außerdem empfinden Nutzer die neuen Aufgabentypen auf AutomataTutor als hilfreich und zeitsparend.

7 Ausblick

AutomataTutor lässt sich sehr gut mit neuen Themengebieten erweitern. So bieten sich zum Beispiel Kellerautomaten und Turingmaschinen als neue Automaten an. Aber auch in Bezug auf kontextfreie Grammatiken sind noch mehrere Erweiterungen denkbar.

Bei dem Aufgabentyp *Wörter für Grammatik* (vgl. Abschnitt 3.1) ist zum Beispiel eine Mindestlänge für die Wörter denkbar. Dies würde verhindern, dass immer nur die leicht zu findenden kurzen Worte angegeben werden. Auch ist eine Variante der Aufgabe möglich, in der die kürzesten Wörter gefunden werden müssen.

Der Aufgabentyp *Beschreibung nach Grammatik* (vgl. Abschnitt 3.2) könnte durch einen neuen Parameter für die maximal erlaubte Anzahl an Produktionen erweitert werden.

Weitere Aufgabentypen

Es sind noch weitere Aufgabentypen mit kontextfreien Grammatiken denkbar. Im Folgenden werden einige davon beschrieben:

- *Ableitung finden:*

  Für ein gegebenes Wort muss der Nutzer eine vollständige Ableitung finden. Diese kann er in Textform Schritt für Schritt angeben. Dabei kann optional auch nach einer bestimmten Ableitung (Rechts- oder Linksableitung) gefragt werden. Beim Überprüfen muss für jeden Schritt in der Ableitung die passende Produktion gefunden werden.

- *NFA nach Grammatik:*

  Für einen gegebenen NFA soll eine äquivalente rechtslineare Grammatik angegeben werden. Zur Überprüfung muss ein Algorithmus implementiert werden, der die korrekte Grammatik berechnet. Im Anschluss kann der in Abschnitt 4.2 beschriebene Äquivalenztest verwendet werden.

- *Grammatik nach NFA:*

  Für eine gegebene rechtslineare Grammatik soll ein äquivalenter NFA konstruiert werden. Die richtige Lösung kann automatisch generiert werden, sodass die

Abgabe mit dem schon vorhandenen Äquivalenztest für DFAs und NFAs [Alu+13] überprüft werden kann.

Literatur

[Alu+13] R. Alur, L. D'Antoni, S. Gulwani, D. Kini und M. Viswanathan. „Automated Grading of DFA Constructions." In: *IJCAI*. Bd. 13. 2013, S. 1976–1982.

[CS70] J. Cocke und J. T. Schwartz. *Programming Languages and Their Compilers*. Courant Institute Math. Sci., 1970.

[Dan+15] L. D'antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan und B. Hartmann. „How can automatic feedback help students construct automata?" In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 22.2 (2015), S. 9.

[Kas65] T. Kasami. *An Efficient Recognition and Syntaxanalysis Algorithm for Context-Free Languages.* Techn. Ber. Hawaii Univ Honolulu Dept of Electrical Engineering, 1965.

[LL09] M. Lange und H. Leiß. „To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm". In: *Informatica Didactica* 8 (2009), S. 2008–2010.

[Mad+15] R. Madhavan, M. Mayer, S. Gulwani und V. Kuncak. „Automating grammar comparison". In: *Acm Sigplan Notices* 50.10 (2015), S. 183–200.

[Sch92] U. Schöning. *Theoretische Informatik kurz gefasst*. Wissenschaftsverlag Mannheim, 1992.

[Weg93] I. Wegener. *Theoretische Informatik: Eine algorithmenorientierte Einführung*. Bd. 3. Teubner Verlag, Stuttgart, 1993.

[You67] D. H. Younger. „Recognition and parsing of context-free languages in time n3". In: *Information and control* 10.2 (1967), S. 189–208.