Hochschule für
Technik und Wirtschaft
Dresden
University of Applied Sciences

Bachelorarbeit

im Studiengang Medieninformatik

Thema:
Automatische Textübersetzung mittels LaTeX und DeepL

eingereicht von:
Erik Wendler

eingereicht am
22. April 2021

Betreuer:
Prof. Dr. Jörg Vogt, Prof. Dr. Robert Baumgartl


BY NC SA

1 Einleitung

1.1 Motivation

Das OpenSource Typesetting-Programm LaTeX gilt als das Standard-Dateiformat für das Erstellen wissenschaftlicher Arbeiten, besonders im mathematischen und naturwissenschaftlichen Bereich. Eine vergleichbare Standardsprache, in der alle Arbeiten verfasst werden, gibt es allerdings nicht. Gäbe es eine, stellte sie eine weitere Einstiegshürde in die wissenschaftliche Arbeit dar und würde Nicht-Muttersprachler und Autoren, die sich keine professionellen Übersetzung leisten können, benachteiligen. Dementsprechend müssen Arbeiten mehrsprachig veröffentlicht werden, wenn sie nicht nur national gelesen werden sollen.
In den letzten Jahren hat die KI immer größere Fortschritte gemacht und Computer kommen den Fähigkeiten von Menschen in immer mehr Bereichen näher und überholen sie schon, beispielsweise in Logik- (FAZ) und Computerspielen (Herbig and Merkert). Ein großes Thema ist in diesem Zusammenhang auch die Fähigkeit von Computern, natürliche Sprache zu verstehen, zu synthetisieren und in andere Sprachen zu übersetzen. Als Vorreiter gilt für die Übersetzung der Dienst DeepL aus Köln, welcher sich im Titel seiner Webseite selbst als den “präzisesten Übersetzer der Welt” bezeichnet.
Dementsprechend bietet es sich an, das Problem der notwendigen Mehrsprachigkeit von wissenschaftlichen Publikationen und die fortschreitenden Entwicklung der maschinellen Übersetzung zusammenzuführen.

1.2 Zielstellung

Im Rahmen dieser Arbeit sollen die notwendigen Grundlagen zur Arbeit mit LaTeX und DeepL mittels Python erkundet und ein Programm entwickelt werden, das diese beiden Werkzeuge nutzt, um ein bilinguales Dokument zu erstellen. Dieses Programm soll die DeepL Pro-API nutzen, um den Text des Originaldokuments in eine beliebige Sprache zu übersetzen und ein Dokument erstellen, mit welchem man beide Versionen pflegen und kompilieren kann.

1.3 Aufbau der Arbeit

Nachdem im ersten Abschnitt die Motivation und die Zielstellung für diese Arbeit geklärt wurden, werden im folgenden Kapitel die für die Bearbeitung und das Verständnis der verwendeten Werkzeuge notwendigen Grundlagen erläutert. Dabei gehe ich insbesondere auf die Kommunikation mit einer REST-Schnittstelle mittels HTTP ein und auf die Grundlagen der künstlichen Intelligenz, welche zum Verstehen des DeepL-Services notwendig sind. Im dritten Teil beschreibe ich den praktischen Teil der Arbeit, der auf den vorgestellten Grundlagen basiert. Ich beschreibe meinen Gedankengang bei der Konzeption des zu generierenden Dokumentenformats und erkläre, weshalb ich mich für die Unterbringung des Originals und der Übersetzung in einer Datei entschieden habe, anstatt beispielsweise eine Template-Engine zu verwenden. Anschließend beschreibe ich die Konzeption der Anwendung, wie ich Zuständigkeitsbereiche definiert und entsprechend meine Module und ihre Funktionen

aufgeteilt habe. Nachfolgend erkläre ich in der Beschreibung der praktischen Umsetzung die konkrete Implementation meiner Anwendung und gehe auf interessante Stellen im Code für die Bearbeitung des Dokuments ein. Zuletzt folgt das Fazit, in welchem ich die entstandene Anwendung mit der Zielsetzung und der Konzeption vergleiche, sowie mit existierenden Projekten, die eine ähnliche Funktionalität umsetzen.

2 Grundlagen

2.1 LaTeX

LaTeX ist eine Weiterentwicklung des von Donald E. Knuth 1977 entwickelten Satzsystems TeX, welche wiederum 1985 von Leslie Lamport veröffentlicht wurde. Lamport erweiterte TeX, welches sich auf die Textformatierung insbesondere von mathematischen Gleichungen fokussierte, um Funktionen für die Strukturierung von Dokumenten. Durch Funktionen zur automatisierten Formatierung von hierarchischen Textelementen (Kapitel > Abschnitt > Unterabschnitt), die Nummerierung von beispielsweise Abbildungen und die Generierung von Inhaltsverzeichnissen und Glossaren eignet sich LaTeX insbesondere zum Schreiben umfangreicher, wissenschaftlicher Texte mit einer hohen Formatkomplexität. (Datta S. 1-2)
Im Gröbsten besteht ein TeX-Dokument aus einem Header und einem Body, vergleichbar mit einer HTML-Datei. Der Header beginnt immer mit dem Kommando *\documentclass[...]{...}*. Dort wird als Parameter eine Formatvorlage gewählt, deren Optionen manuell weiter angepasst werden können. Danach können notwendige TeX-Module importiert, Makros definiert und - wenn für die gewählte Formatvorlage möglich - Autor, Titel und Datum angegeben werden. Standardmäßig unterstützt LaTeX die Dokumentenklassen *letter*, *article*, *report* und *book*, für respektive Briefe, wissenschaftliche Artikel, Berichte und Bücher. Der Body beginnt mit dem Befehl *\begin{document}*, endet mit *\end{document}* und enthält den Textinhalt des Dokuments. Die Formatierung dieses Texts erfolgt über Markierungen mit Funktionsaufrufen, was TeX zu einer Markup-Sprache macht. Anders als der Fließtext formatierte Textabschnitte können über einen *\begin*-*\end*-Block gekennzeichnet werden, welcher eine Umgebung innerhalb des Dokuments festlegt. Ein häufiges Beispiel für eine solche Umgebung ist *align*, welches mehrere Gleichungen horizontal so aneinander ausrichtet, dass die Gleichheitszeichen aller Gleichungen übereinander stehen. Innerhalb des Dokuments können außerdem Kapitel (*\chapter[...]{...}*), Abschnitte (*\section[...]{...}*) und Unterabschnitte (*\subsection[...]{...}*) festgelegt werden, welche automatisch in einem Inhaltsverzeichnis mit *\tableofcontents[...]{...}* mit der entsprechenden Seitenzahl aufgeführt werden können.

2.1.1 Packages

TeX unterstützt Community-Erweiterungen, welche neue Makros und Kommandos zum Befehlssatz von TeX und LaTeX hinzufügen. Zu nutzende Packages werden im Header einer LaTeX-Datei über das Kommando *\usepackage[...]{...}* importiert und können über die Optionen in eckigen Klammern konfiguriert werden. Die Erweiterungen werden von der jeweiligen TeX-Distribution verwaltet und ihre Installation kann entweder während der Kompilation aus der TeX-Datei erfolgen oder präemptiv über einen Package-Manager. Ersteres wird nur von der MikTeX-Distribution unterstützt; einen Packagemanager bieten die beiden großen Distributionen MikTex als und TeX Live beide an. (Wayne State University Library)

2.2 Interpretierte Sprache

Interpretiere Sprachen basieren auf einem *Interpreter* - einem eigenständigen, in einer anderen Sprache programmierten und kompilierten Programm. Zur Ausführung werden die Befehle des Quellcodes nicht in Maschinencode übersetzt, sondern vom Interpreter selbst ausgeführt und steuern sein Verhalten. Da zur Ausführung des Programms der Kompilationsschritt entfällt, haben interpretierte Sprachen den Vorteil, dass der häufigeren Ausführung des Codes während der Entwicklung kein zusätzlicher Schritt im Weg steht. Außerdem hat der Code theoretisch keine Plattformabhängigkeiten, mit Ausnahme des Interpreters. Dieser muss gegebenenfalls für eine neue Plattform auch neu kompiliert werden. (Lee, S. 21-23)

2.3 Python

Python ist eine objektorientierte, interpretierte Programmiersprache, die seit 1989 von Guido von Rossum entwickelt wird. Durch die Implementation des Interpreters als VM ist Python sehr portabel und wird auf allen gängigen Betriebssystemen unterstützt. Großer Beliebtheit erfreut sich Python wegen seiner umfangreichen Erweiterbarkeit durch Community-Bibliotheken und dadurch, dass Python für eine Vielzahl von Anwendungsfällen “gut genug” funktioniert, anstatt für eine konkrete Nutzung spezialisiert zu sein. (Lee, S. 14)
Python existiert in zwei Versionen - Python 2 und Python 3, wobei die neuere Version nicht abwärtskompatibel ist. Seit dem Jahreswechsel 2019/2020 wird Python 2 nicht mehr unterstützt und erhält keine Bugfixes oder sonstige Code-Änderungen mehr. Durch diese Maßnahme sollten mehr Ressourcen für die Weiterentwicklung von Python 3 frei werden. Die neuere Version bietet insbesondere Verbesserungen in der Integration von Unicode, der Nebenläufigkeit und der Fehlerbehandlung. Außerdem ermöglicht sie durch verbesserte Test- und Debugging-Möglichkeiten die Entwicklung robusterer Anwendungen. (Python Software Foundation 2019)

2.4 Reguläre Ausdrücke

Reguläre Ausdrücke, auch *Regex(es)*, sind eine Abfolge von Buchstaben, Zahlen und Sonderzeichen, welche ein abstraktes oder konkretes Muster bilden, nach dem in einem String gesucht werden kann. Regexes werden beispielsweise zum Filtern spezieller Zeilen aus einer Textdatei oder zum Validieren von Eingabefeldern verwendet. (Hunt, S. 257-258)
Sonderzeichen werden für komplexe und abstraktere Regexes verwendet. Beispielsweise werden Zeichen, die innerhalb eckiger Klammern (“*[ ]*”) stehen, als eine Auswahl an Zeichen interpretiert, die an der entsprechenden Stelle akzeptiert wird. Ein Zeichenintervall wie A bis Z kann durch den Beginn und das Ende der Reihe mit einem Strich getrennt beschrieben werden (*“[A-Z]”*). Wenn eine Zeichenfolge als einzelnes Element betrachtet werden soll, muss diese in runde Klammern (*“( )”*) geschrieben werden. Des Weiteren gibt es mehrere Sonderzeichen, welche die Funktion haben, die Anzahl an Instanzen des vorhergehenden Zeichens oder der vorhergehenden Gruppe festzulegen. Ein Fragezeichen (“*?*”) beschreibt null oder ein Vorkommen, ein Plus (“+”) steht für ein oder mehrere Vorkommen und ein Stern (“*”) für null oder mehrere

Vorkommen. Eine konkrete Anzahl an Wiederholungen kann durch eine Zahl oder ein Intervall in geschweiften Klammern (“*{ }*”) ausgedrückt werden. (Hunt, S. 259)
Will man Sonderzeichen als Literale schreiben und nicht mit ihrer syntaktischen Funktion, dann müssen sie mit einem Backslash (“\”) maskiert werden, wie auch in Programmiersprachen üblich. Mit einem Backslash und einem folgenden Buchstaben können auch weitere Muster beschrieben werden, welche allerdings teils implementationsspezifisch sind. Beispielsweise ist *\d* in der Ruby-Implementation gleichbedeutend mit dem Muster *[0-9],* während es im Python3-Modul *re* neben den arabischen Ziffern auch arabisch-indische (“٠١٢٣٤٥٦٧٨٩”) und alle sonstigen Ziffern aus dem Unicode-Zeichensatz findet.

2.5 JSON

Die *JavaScript Object Notation* ist ein von Douglas Crockford entwickeltes, leichtgewichtiges, textbasiertes und, trotz des “JavaScript” im Namen, sprachunabhängiges Datenformat. Die folgenden Informationen über JSON sind der 2006 veröffentlichten Spezifikation Crockfords entnommen. (Crockford)
JSON unterstützt die Darstellung von Nummern und Strings als primitive Datentypen, Objekten und Arrays als strukturierten Datentypen und von den drei Literalen *true*, *false* und *null*. Die Literale müssen zwangsweise kleingeschrieben werden, damit sie als solche erkannt werden. Objekte können alle anderen Datentypen und Literale in beliebiger beinhalten.

2.5.1 Grammatik

Die Grammatik von JSON basiert auf nur sechs Tokens, von denen jeweils zwei für die Begrenzung von Objekten und Arrays benutzt werden. Um einen Wert in einem JSON-String zu definieren, muss man einen Bezeichnerstring und ein unterstütztes Datum als Key-Value-Paar notieren. Nach der Definition eines Wertes muss ein Semikolon (*;*) als Separator gesetzt werden. Will man ein Objekt definieren, muss man die Wertdefinitionen in geschweifte Klammern schreiben (*{...}*). Ein Array definiert man genau so, nur dass eckige anstatt runder Klammern (*[...]*) zum Einsatz kommen.

2.5.2 Parser und Generatoren

JSON-Parser und -Generatoren dienen dem Einlesen und Ausschreiben von JSON-Dateien in einem Programm. Ein Parser kann außerdem dazu verwendet werden, die Dateien in ein anderes Format zu überführen.

2.6 Uniform Resource Identifier

2.6.1 Eigenschaften

Im aktuellen Standard rfc3986 (Berners-Lee 2005, Seite 3-4) charakterisiert Sir Tim Berners-Lee URIs und ihre Vorteile anhand der drei Teile, die die Bezeichnung *Uniform Resource Identifier* ausmachen, wie folgt.

2.6.1.1 Uniform

Durch die einheitliche Form der URIs können verschiedene Arten von Ressourcen, unabhängig den Protokollen, die für den Zugriff notwendig sind, im gleichen Kontext verwendet werden. Neue Arten von Bezeichnern können außerdem problemlos eingeführt werden, ohne die Funktionalität existierender zu beeinträchtigen. Des Weiteren können neu entwickelte Programme und Protokolle schon existierende Bezeichner nutzen.

2.6.1.2 Resource

Im Kontext von URIs kann alles eine Ressource sein, was über eine URI identifiziert wird.

2.6.1.3 Identifier

Alle notwendigen Informationen, die der Unterscheidung einer Ressource von allen anderen dienen, werden vom Bezeichner verkörpert. Das bedeutet nicht, dass der Bezeichner gleich der Ressource ist, noch dass ein System, das eine Ressource mit einem Bezeichner identifiziert, diese auch zugreifbar machen möchte.

2.6.2 Aufbau

Ebenfalls in rfc3986 (Berners-Lee 2005, S. 15-24) beschreibt Sir Berners-Lee den Aufbau einer URI aus fünf hierarchisch geordneten Bestandteilen, sowie deren jeweilige Funktion. Diese Informationen gebe ich hier gekürzt wieder.

2.6.2.1 Schema

Der erste Bestandteil einer URI ist das Schema, eine Zeichenkette aus einem Buchstaben gefolgt von einer beliebigen Kombination aus Buchstaben, Zahlen und den Sonderzeichen “+”, “-” und “.”. Ein Schema weist auf Spezifikationen für Bezeichner innerhalb des Schemas hin, wie beispielsweise Einschränkungen der allgemeinen URI-Syntax.

2.6.2.2 Autorität

Über die Autorität kann eine namensgebende Instanz für die angefragte Ressource angegeben werden, an die die Auflösung der restlichen Inhalte der URI abgegeben wird. Sie kann eine Nutzerkennung, einen Host (als IP oder mittels eines registrierten Namens) und den anzusprechenden Port beinhalten.
Der Beginn eines Autoritäts-Abschnitts wird durch ein “://” gekennzeichnet , das Ende durch “/”, “?” oder “#”.

2.6.2.3 Pfad

Der Pfad enthält Daten, die eine Ressource mittels des Schemas innerhalb des Bereiches der namensgebenden Autorität identifizieren. Gegebenenfalls ist für die eindeutige Identifikation einer Ressource zusätzlich zum Pfad noch der Abfrage-Teil notwendig.
Wenn eine Autorität gegeben ist, muss ein Pfad mit einem “/” beginnen. Die Daten innerhalb des Pfads werden ebenfalls mittels “/” voneinander getrennt.

2.6.2.4 Abfrage

Eine Abfrage enthält wie der Pfad Daten, die eine Ressource innerhalb der Bereiches der namensgebenden Autorität identifizieren. Gegebenenfalls ist neben der Abfrage auch die Angabe des Pfades zur Identifikation einer Ressource notwendig.
Eine Query beginnt immer mit einem “?”. Daten innerhalb der Abfrage werden mittels “/” voneinander getrennt.

2.6.2.5 Fragment

Das Fragment identifiziert als letzter Teil einer URI eine Sekundärressource mit Bezug auf die vorher definierte Hauptressource, wie eine Untermenge der Hauptressource oder eine spezifische Ansicht dieser.
Der Fragment-Anteil einer URI wird mittels “#” gekennzeichnet, weitere Daten werden erneut mit “/” voneinander getrennt.

2.6.3 URL

Eine URL ist eine Form der URI. Zusätzlich zur Identifikation einer Ressource bietet eine URL außerdem immer den primären Zugriffsmechanismus für diese. (Berners-Lee 2005, S. 6)

2.7 Hypertext Transfer Protocol

2.7.1 Methoden

2.7.1.1 Eigenschaften

HTTP-Methoden lassen sich durch zwei wesentliche Eigenschaften definieren - ihre Sicherheit und ihre Idempotenz. Eine Methode gilt dann als sicher, wenn sie auf eine Netzressource nur lesend zugreift, und als unsicher, wenn sie Ressourcen manipuliert. Eine idempotente Methode hingegen ist eine Methode, die bei mehrmaliger Ausführung zum gleichen Ergebnis führt wie bei einer einmaligen. Prinzipiell sind alle Methoden für sich entweder idempotent oder sollten zumindest so implementiert sein, allerdings müssen sich Sequenzen idempotenter Methoden nicht auch selbst so verhalten. (Fielding 1999, S. 51-52)

2.7.1.2 GET

Die GET-Methode dient der Abfrage der Ausgabe eines Prozesses oder einer Ressource, auf welche eine URI zeigt. Durch entsprechende Bedingungen als Header kann ein konditionales GET ausgeführt werden, welches beispielsweise eine Ressource nur ausliest, wenn sie seit einem bestimmten Zeitpunkt modifiziert wurde. (vgl. Berners-Lee 1996, S. 30)
Seit HTTP/1.1 existiert außerdem ein partielles GET, welches unter Angabe eines Bereichs als Header nur einen Teil einer Ressource ausliest. (vgl. Fielding 1999, S. 52)

2.7.1.3 HEAD

Mittels HEAD können die Metainformationen einer Ressource ausgelesen werden, ohne dass die Ressource selbst übermittelt werden muss. Diese Header-Informationen sollten denen gleichen, die bei einer GET-Anfrage verschickt werden, nur ohne Inhalt der Ressource. (vgl. Berners-Lee 1996, S. 30)

2.7.1.4 POST

POST ist eine Anfrage an den Server, die mitgesendeten Daten als untergeordnetes Element für die in der URI angegebene Ressource zu akzeptieren. Eine erfolgreiche Ausführung bedeutet bei dieser Methode nicht zwangsweise die Erstellung einer neuen adressierbaren Ressource auf dem Server. (vgl. Berners-Lee 1996, S. 30-31)

2.7.1.5 OPTIONS

Die OPTIONS-Methode dient der Abfrage der Kommunikations-Optionen eines Servers, um die Optionen und Anforderungen einer Ressource zu erfahren, irgendeine Form der Interaktion mit der Ressource auszulösen. Wenn ein Stern als URI für die Abfrage verwendet wird, werden anstelle der Optionen für eine Ressource die allgemeinen Fähigkeiten des Servers ausgegeben. (vgl. Fielding 1999, S. 51)

2.7.1.6 PUT

Eine PUT-Anfrage bedeutet, dass die angehängten Daten in der Ressource hinter der URI gespeichert werden sollen. Falls an dieser Stelle schon eine Ressource existiert, sind die Daten als veränderte Form dieser Ressource zu verarbeiten. Falls die Ressource hinter der URI noch nicht existiert, kann der Server eine entsprechende neue Ressource mit den gesendeten Daten anlegen. Eine Ressource kann von mehreren URIs adressiert werden. In diesem Falle kann ein PUT eine Ressource hinter weiteren URIs neben der angesprochenen erzeugen. (vgl. Fielding 1999, S. 54-55)

2.7.1.7 DELETE

Mittels DELETE kann die Entfernung einer Ressource auf dem Server angefragt werden, was entweder das Löschen der Ressource oder das Verschieben an einen unzugänglichen Ort bedeuten kann. (vgl. Fielding 1999, S. 55)

2.7.2 Codes

Neben den Methoden definierte Sir Berners-Lee mit HTTP/1.0 auch die Response-Codes, mit denen ein Server auf eine Anfrage antworten kann. Die erste Ziffer des Return-Codes bestimmt dabei seine Klasse, die zweite und die dritte Ziffer stehen für konkrete Nachrichten. Im folgenden gehe ich allgemein auf die verschiedenen Klassen der Codes ein, die genauen Beschreibungen der einzelnen Codes können Fieldings Spezifikationen zu HTTP/1.0 (Fielding 1999, S. 57-70)

Nachrichten mit einem 1xx-Code sind rein informatorisch und finden in HTTP/1.0 keine Verwendung (vgl. Berners-Lee 1996, S. 31). Ab HTTP/1.1 sind die Codes 100 *Continue* und 101 *Switching Protocols* definiert. *Continue* wird verwendet, wenn ein Server den initialen Teil einer Anfrage erhalten, diese aber noch nicht verarbeitet hat, weil er noch Daten vom Client benötigt. Ein 101-Code wird zur Bestätigung eines *Upgrade*-Headers verwendet und bedeutet den Wechsel des verwendeten Protokolls zwischen Server und Client. (Fielding 1999, S. 57)

Die 2xx-Klasse beinhaltet Codes, die die erfolgreiche Ausführung einer Anfrage bedeuten und weisen beispielsweise auf das Anlegen einer neuen Ressource oder einfach auf das Akzeptieren einer Anfrage hin. (Fielding 1999, S. 58)

Return-Codes der 3xx-Klasse weisen einen Client auf die Umleitung seines Requests hin und dass dieser zusätzliche Maßnahmen ergreifen muss. Nach einer 3xx-Antwort kann vom Client automatisiert eine neue Anfrage an das aktualisierte Ziel gestartet werden, solange er nur lesend auf die angeforderte Ressource zugreifen wollte. (Fielding 1999, S. 60)

Die 4xx-Returncodes deuten alle auf clientseitige Fehler hin, insbesondere auf Fehlbildungen der Anfrage oder fehlenden Informationen darin. (Fielding 1999, S. 64)

Die letzte Klasse an HTTP-Codes, 5xx, deutet auf serverseitige Fehler hin, wegen welchen der Server die Anfrage nicht bearbeiten kann. (Fielding 1999, S. 69)

2.8 RESTful-Schnittstellen

In seiner Dissertation beschreibt Roy Fielding die Auflagen, die ein System erfüllen muss, um mit den Anforderungen des Webs zu funktionieren. Aus diesen leitet er die *Representational State Transfer* (REST)-Architektur als ein Modell für Webanwendungen ab, deren Hauptaugenmerk auf der Skalierbarkeit und Unabhängigkeit der Komponenten liegt. Die einzelnen Auflagen, aus denen er diesen Hybrid-Stil ableitet, gebe ich im Folgenden gekürzt wieder. (Fielding 2000, S. 76 ff.)

2.8.1 Client Server

In einer REST-Anwendung liegt die gesamte Zuständigkeit der Datenhaltung bei einem Server und die Zuständigkeiten der Benutzerschnittstelle bei einem Client. Da ein System nur einen passenden Client benötigt, um mit einer REST-Anwendung zu interagieren, steigt die Portabilität, während die Serverkomponente durch Einsparen der Benutzerschnittstellen-Funktionalitäten vereinfacht und damit besser skalierbar wird. Außerdem können Komponenten unabhängig voneinander weiterentwickelt werden.

2.8.2 Stateless

Die Kommunikation zwischen Server und Client soll zustandslos geschehen und jede Anfrage des Clients den gesamten für den Server notwendigen Kontext zur korrekten Verarbeitung mit sich bringen. Dementsprechend liegen alle Informationen über den Zustand der Kommunikation beim Client, während der Server keine Kontextdaten hält.

Eine zustandslose Kommunikation bringt drei große Vorteile mit sich. Zum einen ist sie zuträglich für die Zuverlässigkeit der Anwendung, da partielle Fehler, die beispielsweise die Kommunikation über das Netzwerk verursachen kann, nicht möglicherweise den gesamten Sitzungszustand beschädigen und dadurch dementsprechend leichter behebbar sind. Des Weiteren wird die Serverkomponente leichter skalierbar, da Ressourcen ohne das Aufrechterhalten einer Client-Session einfacher wieder freigegeben und für neue Anfragen verwendet werden können. Außerdem lässt sich eine zustandslose Anwendung leichter überwachen, da zum Verstehen von Anfragen kein Session-Kontext notwendig ist. Der wesentliche Nachteil einer zustandslosen Architektur ist der Overhead, der durch die notwendigen, teils repetitiven Zusatzinformationen in den Anfragen entsteht. Außerdem ist die Nutzbarkeit der Anwendung vollkommen abhängig vom korrekten Verhalten der Clients, auf welches der Server keinen Einfluss hat.

2.8.3 Cache

Zur Verminderung der Netzwerklast und zum Verbessern der Anwendungsperformance soll ein Cache genutzt werden, welcher Anfragen und Serverantworten speichert und bei einer erneuten, gleichartigen Anfrage die entsprechende Antwort ausgibt, ohne dass der Server beansprucht werden muss. Damit ein solcher Cache funktionieren kann, muss ein Server Ressourcen in seinen Antworten als cachebar markieren können, damit keine zeitlich veränderlichen Daten gecacht und falsch ausgegeben werden. Durch die verringerte Serverlast nimmt erneut die Skalierbarkeit der Anwendung zu. Der wesentliche Nachteil eines Antwort-Cache ist eine verringerte Verlässlichkeit der Anwendung, wenn die Daten im Cache veraltet sind, aber weiter ausgegeben werden.

2.8.4 Einheitliche Schnittstelle

Anstelle spezifischer Schnittstellen für jeweils einen Anwendungsfall kommt eine generalisierte Schnittstelle für jeden Teil der Anwendung zum Einsatz. Dies macht die Systemarchitektur und die Interaktionen der Komponenten leichter nachvollziehbar und ermöglicht die unabhängige Weiterentwicklung von Komponenten, verringert allerdings die Effizienz der Schnittstellen.

2.8.5 Layered System

Eine REST-Anwendung sollte in mehrere hierarchische Schichten aufgeteilt werden können, wobei die Dienste nur Einblick auf die Schichten haben, mit denen sie unmittelbar interagieren. Diese Schichten ermöglichen die Isolation veralteter Services oder die Isolation neuer Services von veralteten Clients. Außerdem nimmt erneut die Skalierbarkeit der Anwendung zu, da durch die Abkapselung selten genutzter Anwendungsteile in einen eigenen Layer der Server vereinfacht wird. Zwischenschichten können zum Load Balancing genutzt werden oder als gemeinsamer Cache, um den Overhead, der durch die Schichtung der Anwendung entsteht, performancetechnisch auszugleichen.

2.9 Artificial Intelligence

Wolfgang Ertel, der Leiter des Instituts Künstliche Intelligenz an der Hochschule Ravensburg-Weingarten, liefert in seinem Buch *Introduction to Artificial Intelligence* (Ertel, S. 1-3) mehrere Begriffsdefinitionen der künstlichen Intelligenz, von denen ich hier eine Auswahl wiedergebe. Der erste dort aufgeführte Versuch, den Begriff zu definieren, stammt von John McCarthy aus dem Jahre 1955:

“The goal of AI is to develop machines that behave as though they were intelligent”

(“Das Ziel der künstlichen Intelligenz ist es, Maschinen zu entwickeln, sie sich verhalten, als seien sie intelligent.”)

Diese Definition hat einige Schwachstellen. Ein Beispiel liefert Ertel selbst in Form der Braitenberg-Vehikel, welche durch simple elektrische Schaltungen mehrere relativ komplexe Verhaltensmuster zeigen können. (Ertel S. 1-2) Andererseits ist auch das eingesetzte Attribut “intelligent” eine Schwachstelle, da in der Psychologie mehrere Intelligenztheorien existieren und der Begriff nicht eindeutig definierbar ist.

Ein weitere vorgestellte Definition stammt von Elaine Rich, welche 1983 in ihrem Buch *“Artificial Intelligence”* die AI folgend definierte*:*

“Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better.”

(“Künstliche Intelligenz ist die Lehre davon, Computer Dinge tun zu lassen, in welchen Menschen im Moment besser sind.”)

Laut Ertel umfasst diese Definition alles, was Forscher im Feld der KI in der Vergangenheit getan haben und in der Zukunft tun werden. Insbesondere betont er die Adaptivität der menschlichen Intelligenz, welche durch das Lernen und Abstrahieren vorhandenes Wissen in neuen Kontexten anwenden kann. In dieser Lernfähigkeit sind Menschen Computer substanziell überlegen, weswegen es laut Richs Definition ein Feld der KI-Forschung darstellt - das *Machine Learning*.

2.9.1 Machine Learning

Das Ziel des Machine Learnings ist das Generalisieren von gelernten Fähigkeiten, sodass diese nicht nur im exakt gleichen Kontext reproduziert werden können. Um jenes zu erreichen werden Beispiele in der Lern- oder Trainingsphase in ein Machine Learning-Modell eingebracht. Diese Beispiele werden Klassen zugeordnet und in Features zerlegt, welche einzelne Eigenschaften des Beispiels numerisch darstellen. Anhand dieser Features und dem dazugehörigen Klassenwert erweitert der Machine Learning-Algorithmus dann seinen *Learning Agent* - eine Funktion, die die Werte eines Feature-Vektors mit dem diskreten Wert der zugehörigen Klasse in Verbindung bringt. Mit jedem weiteren Beispiel wird diese Funktion weiterentwickelt, sodass immer mehr Kombinationen innerhalb eines Feature-Vektors zur richtigen Klassifizierung führen, die Klassifizierung immer effizienter abläuft und eine Generalisierung des Wissens eintritt. (Ertel, S. 175-178)
Anschließend an die Trainingsphase wird eine Testphase vorgenommen, in welcher dem Modell unklassifizierte Testdaten zugeführt werden. Diese ist wichtig, da die Trainingsdaten als Basis für die Funktion im Modell in der Regel immer perfekt klassifiziert werden und dementsprechend für die Einschätzung der Generalisierung des Modells nicht geeignet sind. (Ertel, S. 178-179)
Ein Beispiel für Features sind beispielsweise Kanten und Farben eines Bildes. Klassen können beispielsweise einzelne Schriftzeichen bei einem Schrifterkennungs-Modell sein.
Der Negativaspekt von Machine Learning ist, dass der Feature-Vektor in seinem Umfang und seiner Granularität in der Hand des Entwicklers liegt. Dementsprechend ist der Input auf die Aspekte der Trainingsdaten beschränkt, die dem Entwickler bewusst sind und die Optimierungen des Feature-Vektors müssen manuell erfolgen. Letzteres ist insbesondere interessant, da der Zeitaufwand zum Trainieren eines Modells mitunter exponentiell mit der Länge des Feature-Vektors wächst.

2.9.1.1 Supervised und Unsupervised Learning

Der gerade beschriebene Prozess ist nur eine Form des Machine Learnings - das *supervised Learning* oder beaufsichtigte Lernen. Dieses ist einer der zwei Hauptansätze im Machine Learning; der andere ist das *unsupervised Learning* oder unbeaufsichtigte Lernen. Die folgenden Informationen entstammen einem Blogpost von Julianna Delua, einer Expertin auf dem Gebiet Data Science und AI bei IBM, und werden gekürzt und sinngemäß wiedergegeben. (Delua)
Der wesentliche Unterschied zwischen den beiden Ansätzen sind die dem Modell zugeführten Testdaten. Beim beaufsichtigten Lernen werden diese Daten von Menschenhand klassifiziert und das Ziel des Modells ist es, diese Klassifizierung so gut

wie möglich zu erlernen. Unbeaufsichtigte Modelle hingegen arbeiten ohne von Menschen weiter aufbereitete Testdaten und erlernen, in diesen Muster und Anomalien zu erkennen.
Entsprechend dieser Eigenschaften eignen sich beaufsichtigt trainierte Modelle zur Vorhersage auf Basis eines begrenzten Satzes neuer Daten, wie beispielsweise zur Wettervorhersage. Unbeaufsichtigt trainierte Modelle eignen sich besser um große Sätze neuer Daten auszuwerten und interessante Stellen darin zu finden, beispielsweise für Empfehlungssysteme.

2.9.2 Deep Learning

Um die beschriebenen Negativpunkte des reinen Machine Learnings zu beheben, kommen beim Deep Learning mehrschichtige neuronale Systeme zum Einsatz, bei denen nach einer vorverarbeitenden Schicht mehrere Modelle folgen, welche unbeaufsichtigt trainiert werden und jeweils die Gewichtung einzelner Features aus den gegebenen Daten berechnen. An diese Schichten wird ein beaufsichtigtes Modell angeschlossen, welches beim Training den generierten Feature-Vektor und die Klassifizierung der Testdaten erhält und auf dieser Basis ein Modell erstellt. (Ertel, S. 277-279)

2.9.3 DeepL

DeepL ist ein Übersetzungsdienst, der von einer gleichnamigen Firma aus Köln entwickelt wurde, welche sich auf AI-Systeme für Sprachen spezialisiert. Der Dienst ging aus *linguee* hervor, einer Onlinedatenbank für Übersetzungen, die von der selben Firma stammt. (DeepL, “Über DeepL”)
Der Übersetzungsservice basiert auf Convolutional Neural Networks und damit auf einer Form von Deep Learning, woraus sich auch der Name ableitet. Die Nutzung von CNNs hebt DeepL von beispielsweise dem Google Translator ab, welcher mit Recurrent Neural Networks arbeitet. Der wesentliche Vorteil von CNNs liegt in der schnelleren Verarbeitung eingelesener Daten, da sie mehrere Inputs parallel verarbeiten können. Da bei der Übersetzung natürlicher Sprache der Kontext und damit die anderen Wörter innerhalb eines auszuwertenden Satzes eine große Rolle spielen, wird das neuronale Netz von DeepL durch einige Anpassungen in die Lage versetzt, diesen zu ermitteln und in der Übersetzung zu berücksichtigen. Außerdem kommt ein *Beam Search*-Algorithmus zum Einsatz, durch welchen nicht zwangsweise die wahrscheinlichsten einzelnen Wörter, sondern die wahrscheinlichsten Sätze ausgegeben werden. In Blindtests wurden laut den Entwicklern die Übersetzungen der ersten Version von DeepL etwa dreimal häufiger als vorzuziehendes Ergebnis gewählt als die vergleichbarer Konkurrenzprodukte von Google, Facebook und Microsoft. (Merket)

2.9.3.1 DeepL Pro API

DeepL bietet neben einem Übersetzer mit Webinterface auch eine REST-API unter *https://api.deepl.com/v2/translate* an, gegen welche man programmatisch Strings zur

Übersetzung versenden kann. Auf diese kann nur mittels GET und POST-Methoden zugegriffen werden, welchen man mittels Anfrageparametern die notwendigen Informationen mitgibt. Kostenlos kann die API nur einem eingeschränkten Umfang von 500000 Zeichen pro Monat genutzt werden und zur Identifikation eines Nutzeraccounts kommt ein Key-Phrase zum Einsatz, welcher bei jeder Anfrage mitversendet werden muss. (DeepL, “Accessing the API”)

2.9.3.1.1 Nutzung

Eine Anfrage an DeepL besitzt nur drei notwendige Felder: Die Zielsprache *target_lang*, den zu übersetzenden Text *text* und den Nutzerschlüssel *key* (siehe Abbildung G1). Der Text muss zwangsweise UTF-8 codiert sein. Optionale Parameter existieren zur Klärung der Ursprungssprache, welche bei einem nicht gesetzten Feld automatisch ermittelt wird, ob der übersetzte Text formell gehalten sein soll und für verschiedene Formatierungsoptionen.
Neben Plain-Text unterstützt die API zudem auch XML-Dokumente, für welche man den Parameter *tag_handling* auf “xml” setzen muss. Für die Handhabung von XML existieren noch weitere Parameter, auf die ich hier nicht weiter eingehe.
Die Antwort auf eine erfolgreiche Anfrage liefert DeepL immer im JSON-Format mit einem *translations*-Feld, welches wiederum eine Liste aus mindestens zwei Feldern beinhaltet (siehe Abbildung G2). Das erste Feld ist *detected_source_language*, welches die Ursprungssprache beinhaltet, welche für die Übersetzung verwendet wurde. Sollte diese in einem Feld der Anfrage konfiguriert worden sein, wird dieses Feld den gleichen Inhalt haben. Im zweiten Feld - *text* - befindet sich die Übersetzung. Wenn in der Anfrage mehrere *text*-Felder angegeben wurden, wird man auch entsprechend viele Elemente im *translations*-Feld der Antwort finden, von denen jedes eine eigene *detedtec_source_language* und natürlich einen eigenen Text hat. (DeepL, “Translating Text”)

```
curl https://api-free.deepl.com/v2/translate
 -d auth_key=$DEEPL_KEY
 -d "text=Beispielanfrage an die DeepL-API"
 -d "target_lang=EN"
```

Abbildung G1 : Beispiel für eine Anfrage an DeepL in der Bash-Konsole

{"translations":[{"detected_source_language":"DE","text":"Example request to the DeepL API"}]}

Abbildung G2 : Antwort von DeepL auf die obige Anfrage

3 Umsetzung

3.1 Existierende Projekte

3.1.1 texTrans

Das Projekt texTrans ist eine Python-Anwendung, die eine LaTeX-Datei mittels DeepL übersetzt und die Übersetzung in einer neuen Datei ablegt. Um die syntaktische Richtigkeit des LaTeX-Textes beizubehalten, werden die einzelnen Kommandos mittels eines Hashes maskiert und mit dem gesamten Text an DeepL verschickt. Jedes maskierte Kommando wird in einem Dictionary mit dem dazugehörigen Hash gespeichert, sodass sie im übersetzten Antworttext wiederhergestellt werden können. (texTrans)
Der Vorteil dieses Ansatzes ist die Einfachheit des Algorithmus, da die Übersetzung von LaTeX-Befehlen durch die Maskierung verhindert wird. Nachteilig zu erwähnen ist allerdings, dass die Maskierung von Kommando-Parametern sehr inkonsistent geschieht. Einerseits werden Kommandos wie *\include, \input* und *\begin* mitsamt ihrer Parameter maskiert, andererseits werden andere Kommandos durch die Capture Group *"\\\w+\[.*\]"* nur dann gefunden, wenn sie sowohl optionale Parameter (in eckigen Klammern) als auch notwendige Parameter (in geschweiften Klammern) aufführen. Ein *\index*-Aufruf mit Parametern würde beispielsweise übergangen werden. Das genutzte *pydeepl*-Modul zum Versand des Texts ist außerdem veraltet und nutzt DeepL's mittlerweile eingestellte jsonrpc-Schnittstelle, welche lediglich Texte mit maximal 5000 Zeichen unterstützte. Zum Umgehen dieser Einschränkung wird jede Zeile einzeln übersetzt, was zeitlich ineffizient ist.

3.2 Konzeption

3.2.1 Dateiformat

Für die Unterbringung mehrerer Texte innerhalb einer .tex-Datei gibt es prinzipiell zwei Möglichkeiten - entweder befindet der gesamte Text sich innerhalb der Datei oder man nutzt eine Template-Engine, welcher man jeweils eine Datei mit den einzutragenden Textteilen gibt. Ein selbstentwickeltes Beispiel für eine LaTeX-Template-Engine hat beispielsweise Paulo Cereda in einem Blogpost auf Tex Talk vorgestellt.[1] Dabei werden innerhalb des Dokuments Platzhalter gesetzt, die ein Programm mit den Inhalten einer externen Werte-Datei ersetzt.
Die Nutzung von Templates hätte den Vorteil, dass die Übersichtlichkeit innerhalb eines Dokuments erhalten bleibt, auch wenn es in einer größeren Anzahl Sprachen gepflegt wird und dass trotz der Trennung vom Dokument in Originalsprache und der Übersetzung keine unnötige Redundanz bei Gleichungen, Abbildungen und allem, was nicht übersetzt werden muss, auftritt. Allerdings würden durch die Auslagerung des Fließtextes in eine externe Datei alle weiteren Inhalte in einen anderen Kontext gebracht

[1] https://tex-talk.net/2012/10/latex-and-template-engines-revisited/

als den, in dem sie behandelt werden, was die weitere Arbeit am Dokument erschwert. Die Versionsauswahl könnte in dieser Variante durch die Wahl der Werte-Datei beim Ausführen des Template-Programms geschehen.
Werden stattdessen die beiden Versionen des Dokuments in einer .tex-Datei gepflegt, bleiben zwar Fließtext und restlicher Inhalt in einem Kontext, aber die Menge an Fließtext wird verdoppelt und entsprechend die Datei unübersichtlicher und ebenfalls die Bearbeitung erschwert. Allerdings gibt es auch den Vorteil, dass Original und Übersetzung in einem Kontext editiert werden können, was mögliche Korrekturen der Übersetzung nach dem Erstellen des Dokuments und Änderungen in einem Abschnitt in beiden Texten erleichtert. Bei der Arbeit mit nur einem Dokument muss desweiteren auch ein Mechanismus gefunden werden, über den innerhalb des Dokuments die Version des Dokuments ausgewählt werden kann, die generiert werden soll, idealerweise ohne das manuelle Ein- und Auskommentieren von Textblöcken.
Ich habe mich für das Speichern beider Versionen in einem Dokument entschieden, da dadurch ein Vorteil für das Editieren beider Sprachversionen des Dokuments entsteht. Die Pflege zweier Versionen eines Dokuments beeinträchtigt in beiden Fällen die Übersichtlichkeit der .tex-Datei, allerdings bei der Ein-Datei-Variante etwas weniger, da Abbildungen, Gleichungen und ähnliches immer noch in relativer Nähe zum Fließtext stehen.

3.2.2 Anwendung

Als Erstes habe ich festgelegt, wie der Nutzer mit dem Programm interagieren solle, und mich für eine Kommandozeilen-Anwendung entschieden, welche das zu übersetzende Dokument und die Zielsprache als Argumente bekommt. Anschließend habe ich die Funktionalität meines Programms in drei Bereiche aufgeteilt, zum einen die Operationen am TeX-Text, zum anderen die Kommunikation mit der DeepL-Pro-API und zuletzt die Komposition einer zweisprachigen Datei unter Nutzung dieser beiden Module.
Das *parse*-Modul, das sich mit der Verarbeitung von TeX-Texten befasst, sollte folgende Funktionen haben:

- Die Zerlegung des gegebenen .tex-Textes in ein Datenformat, das die Filterung von Parametern auf Basis ihrer Kommandos und von Umgebungen auf Basis ihrer Art zulässt
- Die Filterung von Parametern und Umgebungen, sodass nur Textinhalte übrig bleiben, welche auch im gerenderten Dokument erscheinen
- Das Wiederzusammensetzen eines TeX-Dokuments als Gegenoperation zur oben beschriebenen Zerlegung

Das *rest*-Modul zur Kommunikation mit der Übersetzer-API muss lediglich aus dem API-Key, der Zielsprache und einem gegebenen String oder einer Liste aus Strings eine Anfrage für DeepL konstruieren, sie versenden, die Übersetzung aus der Antwort extrahieren und sie im gleichen Format wie den Input wieder ausgeben. Beispiele für Anfrage- und Antwortformat sind in den folgenden Bildern zu sehen.

Die Aufgabe des dritten Moduls ist es, das Ergebnis der Zerlegung und Filterung in eine für das Bauen der Query nutzbare Form zu bringen, die ausgegebene Übersetzung in eine vom *parse*-Modul nutzbare Form zu bringen und die Originaldaten mit den übersetzten Daten inhaltlich sinnvoll zusammenzufügen, sodass beispielsweise absatzweise zwischen Original und Übersetzung gewechselt wird und man längere Fließtextabschnitte zur weiteren Arbeit am Text hat.

3.3 Implementierung

3.3.1 Verwendete externe Bibliotheken

3.3.1.1 Python-Module

3.3.1.1.1 re

Standardmäßig ist das *re*-Modul in Python installiert, was die Arbeit mit regulären Ausdrücken in Python ermöglicht. Es beinhaltet mehrere Funktionen zur Suche, String-Zerlegung, Substring-Ersetzung und zur Erstellung von *Pattern*-Objekten. Jede dieser Operationen kann mittels Flags, welche das Modul selbst mitliefert, angepasst werden, um beispielsweise Groß- und Kleinschreibung zu ignorieren oder ausschließlich ASCII- anstelle von Unicode-Zeichen zu finden. Ebenfalls enthalten in *re* ist die Definition für *Match*-Objekte, welche von einigen Methoden des Moduls ausgegeben werden. Die folgenden Informationen stammen aus der offiziellen Dokumentation zu *re* (Python Software Foundation 2021) und werden in verkürzter Form wiedergegeben.

3.3.1.1.1.1 Match-Objekt

Ein *Match*-Objekt beinhaltet immer ein boolesches *True*, da die *match()-, fullmatch()-* und *find()*-Methoden nur dann ein entsprechendes Objekt zurückgeben, wenn die Suche erfolgreich war. Außerdem beinhaltet ein Objekt alle im Pattern definierten Gruppen in einer Liste, sodass man darüber iterieren kann. Außerdem können benannte Capture Groups zusammen mit den dazu gefundenen Matches als Map ausgegeben werden. Zudem speichert das Objekt den String, in dem gesucht wurde, das genutzte Pattern und die Information, von welcher Stelle im String bis wohin gesucht wurde.

3.3.1.1.1.2 Suchen

Zur Suche existieren fünf Methoden in *re*. Zum einen gibt es *find(), match()* und *fullmatch()*, welche alle ein *Match*-Objekt und damit nur einen Treffer innerhalb des durchsuchten Strings ausgeben. Die Methoden sind alle unterschiedlich stark limitiert - *find()* durchsucht den gesamten gegebenen String nach Treffern, *match()* sucht ausschließlich zu Beginn des Strings und *fullmatch()* liefert nur ein erfolgreiches Ergebnis, wenn der gesamte String zum gegebenen Pattern passt.

Zwei weitere Funktionen existieren, um alle Treffer auszugeben. Eine Liste erhält man, wenn man *findall()* verwendet, und einen Iterator, wenn man mittels *finditer()* einen String durchsucht.

3.3.1.1.1.3 Substring-Operationen

Das *re*-Modul unterstützt außerdem das Ersetzen und Trennen von Strings an Substrings. Für Ersteres existiert die *sub()*-Methode, welche jeden Abschnitt eines Strings, auf den ein gegebenes Pattern passt, mit einem gegebenen anderen String ersetzt. Das gleiche tut *subn()*, welches allerdings nicht nur den bearbeiteten String, sondern auch die Anzahl an Ersetzungen darin als Tupel ausgibt. Zum Zerlegen von Strings wird *split()* angeboten, welches einen String an jeder Stelle zerteilt, die auf den gegebenen regulären Ausdruck passt.

3.3.1.1.1.4 Erstellung eines Pattern-Objekts

Anstatt ausschließlich die Methoden von *re* zu verwenden, kann man Ausdrücke, welche man wiederverwenden möchte, auch mittels *compile()* zu einem *Pattern*-Objekt kompilieren. Diese Objekte bieten alle Methoden von *re* an, mit Ausnahme von *compile()*, nehmen allerdings kein neues Pattern als Argument an, sondern nutzen das, aus dem sie erstellt wurden. Dieses kann aus dem Member *Pattern.pattern* ausgelesen werden.

3.3.1.1.2 requests

Requests beschreibt sich selbst als simple, für Menschen gemachte HTTP-Bibliothek für Python, welche das Versenden von HTTP/1.1-Anfragen für den Nutzer vereinfacht.

3.3.1.1.2.1 Anfrage senden

Zum Versenden von Anfragen besitzt *requests* Methoden die gleich den HTTP-Methoden der zu versendenden Anfrage benannt sind. Als Argument wird immer die Ziel-URL angegeben; für *PUT*- und *POST*-Anfragen müssen die zu versendenden Daten der Methode als Map zugeführt werden. Dafür existieren zwei optionale Parameter, einmal *data*, welcher die angegebene Map in eine HTML-Form enkodiert, und *json*, welcher die Map in das JSON-Format überführt. (Reitz)

3.3.1.1.2.2 Antworten verarbeiten

Jede *requests*-Methode, die eine Anfrage versendet, liefert ein *Response*-Objekt zurück, welches die Antwortdaten enthält. Reine Textantworten werden von *requests* automatisch mit dem Textencoding, das aus den gesetzten HTTP-Headern abgeleitet werden kann, dargestellt. Für den Transfer komprimierte Dateien und JSON-Dateien können automatisch dekodiert werden.

Das *Response*-Objekt enthält weiterhin Membervariablen für den HTTP-Code in *status_code*, die Header-Daten der Antwort in *headers* und mit versendete Cookies in *cookies*. Außerdem wird die Weiterleitungshistorie in einer Liste als *history* abgelegt. (Reitz)

3.3.1.2 TeX-Packages

3.3.1.2.1 comment

Durch das *comment*-Package können Textabschnitte mit benutzerdefinierten Markierungen versehen werden. Diese Abschnitte können durch die beiden Methoden *\includecomment* und *\excludecomment* selektiv vom gerenderten Text ein- oder ausgeschlossen und damit mehrere Versionen eines Werkes in einer Datei gepflegt werden. Die Markierungen werden entweder als Umgebungen mittels *\begin{<markierung>}* und *\end{<markierung>}* oder direkt als Funktionsaufruf *\<markierung>* und *\end<markierung>* gesetzt.
Die mittels *\excludecomment* ausgeschlossenen Inhalte werden beim Kompilieren in einer Datei, standardmäßig *comment.cut,* abgelegt. Diese Datei kann mittels *\CommentCutFile* umbenannt werden. (Eijkhout)

3.3.2 parse-Modul

3.3.2.1 Zerlegen des TeX-Textes zu einer zweidimensionalen Liste

Zunächst wird der Text aus der Originaldatei eingelesen. Dafür nutze ich einen *with*-Block, da dieser sicherstellt, dass Python die Datei nach dem Einlesen wieder schließt (G3, Zeilen 7-8). Anschließend muss er für die Zerlegung aufbereitet werden, da Trennstriche zu Problemen bei der Übersetzung führen und da die Python-Implementation von Regulären Ausdrücken nur zeilenweise und nicht über Zeilenumbrüche hinweg matchen kann. Diese Eigenheit nutze ich später im dritten Schritt aus, indem vor jedem Backslash einen Zeilenumbruch einfügt wird (G3, Zeilen 11-13). Anschließend zerlege ich die Datei mittels vier regulärer Ausdrücke pro Zeile in eine Liste aus 4 Strings - Einem LaTeX-Funktionsaufruf, optionale Parameter, notwendige Parameter und dem folgenden Text. Die erste Capture Group sucht nach einem Backslash, gefolgt von einer beliebigen Anzahl Zeichen. Diese Gruppe konnte ich nicht nur auf Buchstaben beschränken, da escapte Sonderzeichen im vorherigen Schritt ebenfalls einen Zeilenumbruch vor ihre Backslaches bekommen haben und mit einem anderen regulären Ausdruck möglicherweise verloren gehen würden. Die folgende Gruppe sucht nach einem beliebigen Text innerhalb von eckigen Klammern. Dabei setze ich eine Wildcard (.*) ein, damit auch wirklich jeder mögliche Text innerhalb der Optionen gefunden wird. Nach diesem Ausdruck folgt außerdem ein *?*, was bedeutet, dass es sich um eine optionale Gruppe handelt, welche nicht nach jedem Funktions-aufruf auftritt. Die dritte Gruppe funktioniert wie die zweite, nur dass geschweifte statt eckige Klammern zum Einsatz kommen. Auch sie ist optional, da es alleinstehende LaTeX-Funktionen gibt. Die vierte Gruppe ist eine komplette Wildcard-Suche, welche lediglich den folgenden Text einfängt. Diese muss nicht als optional gekennzeichnet werden, da der Stern ohnehin null bis unendlich Treffer bedeutet. (G3, Zeilen 17-20). Zuletzt wird die entstandene zweidimensionale Liste ausgegeben.

```
6   def ltxfile_to_list(filename):
7       with open(filename) as file:
8           content = file.read()
9
10      #Wordsplitting interferes with proper translation
11      content = content.replace('\\-','')
12      content = content.replace('\n', "###")
13      content = content.replace('\\','\n\\')
14
15
16
17      contentList = re.findall('(\\\.*)'
18                               +'(\[.*\])?'
19                               +'(\{.*\})?'
20                               +'(.*)', content)
21
22      return contentList
```

Abbildung G3 , *ltxfile_to_list*-Funktion, die TeX-Strings in eine Liste überführt

3.3.2.2 Wiederzusammenfügen einer Liste zu einem TeX-Text

Als Gegenoperation zur Zerlegung muss es auch eine Funktion zum Zusammensetzen eines TeX-Textes aus einem zweidimensionalen Array aus Funktionsaufrufen, optionalen Parametern, notwendigen Parametern und Rest-Text geben (G4). Diese überprüft erst, ob die gegebene Liste die richtige Breite hat, um daraus ein Dokument zusammenzufügen. Anschließend iteriert sie in einer *while*-Schleife über die gegebene Liste und fügt dabei die Zeilen einfach zu einem String zusammen.

```
24  def list_to_ltx(ltx_list):
25      if len(ltx_list[1]) != 4:
26          print("Given ltx_list isn't 4 elements wide, cannot be assembled to a LaTeX-string")
27          return
28      ltxstring = ""
29
30      for line in ltx_list: ltxstring += line[0] + line[1] + line[2] + line[3]
31
32      ltxstring = str(ltxstring)
33      ltxstring = ltxstring.replace('***', r'\index')
34      ltxstring = ltxstring.replace('###','\n')
35
36      return ltxstring
```

Abbildung G4 , *list_to_ltx*-Funktion, die Gegenoperation zu *ltxfile_to_list*

3.3.2.3 Filtern der Parameter

Zum Filtern der Parameter gibt es zwei Listen - eine mit Funktionen, deren Parameter im gerenderten Text auftreten und entsprechend behalten werden, und eine Liste mit Umgebungen, die keine übersetzbaren Inhalte haben und deren Text gelöscht werden kann. Außerdem wird eine Liste angelegt, da die Inhalte des Tupels aus dem Parsing-Schritt nicht bearbeitet werden können und deshalb ein anderer Datentyp eingesetzt werden muss.
Anschließend wird mittels einer *while*-Schleife über das Tupel iteriert. Eine *for-in-Schleife* würde zwar dem typischen Python-Stil mehr entsprechen, ist allerdings in diesem Anwendungsfall unbrauchbar, da ich innerhalb der Schleife noch weitere Schleifen habe, welche den Iterator inkrementieren müssen. Da man auf diesen bei einer *for-in*-Schleife nicht zugreifen kann, musste ich stattdessen eine *while*-Schleife nutzen und für jede verarbeitete Zeile den Iterator hochzählen.
Innerhalb der Schleife gibt es abhängig vom vorangehenden Funktionsaufruf einer Zeile vier wesentliche Fälle, welche unterschiedlich behandelt werden:

1. Der Funktionsaufruf ist *index*
2. Der Funktionsaufruf ist in der Liste mit Formatierungsfunktionen
3. Der Funktionsaufruf ist *begin* und die dazugehörige Umgebung befindet sich in der Liste mit Umgebungen, deren Inhalte ausgefiltert werden
4. Der Funktionsaufruf erfüllt keine der obigen Bedingungen (Default)

Im ersten Fall wird zunächst geprüft, ob auf den gefundenen *index*-Aufruf noch weitere folgen, was im Fließtext beispielsweise innerhalb eines Buches nicht unwahrscheinlich ist. Dafür iteriert eine weitere *while*-Schleife, welche am derzeitigen Listenindex ansetzt, solange über das Array, bis das erste Element der gelesenen Listen- zeile kein *index* mehr ist. Für jeden gefundenen Aufruf wird ein String um drei Sterne, welche den Aufruf maskieren, die Parameter des Aufrufs und den darauffolgenden Text erweitert. (G5, Zeile 66-69) Wenn die Schleife keine weiteren aufeinanderfolgenden Aufrufe findet, wird der entstandene String an den letzten Eintrag in der gefilterten Liste vor dem Auftreten eines Indexeintrags eingefügt. Damit die gefilterte Liste die Länge der Original-Liste beibehält, was im späteren Schritt zum Wiederzusammenfügen der Listen notwendig ist, werden im letzten Teil dieser Verzweigung so viele leere Listen an die gefilterte Liste angehangen, wie *index*-Aufrufe gefunden wurden. (G5, Zeile 71-73)
Der zweite Fall ist weniger komplex; wenn eine Formatierungsfunktion im ersten Listenelement der gelesenen Zeile steht, wird die gesamte Liste an die gefilterte Liste angehängt, damit die Parameter auch mit übersetzt werden können. (G5, Zeile 75-77)
Im dritten Fall wird zunächst die Art des Environments gemerkt, welches durch das *begin* begonnen wird. Anschließend wird die gefilterte Liste nur um den Funktionsaufruf erweitert, da es keine zu übersetzenden Parameter gibt und der Text nach einem *begin* in jeder Umgebung aus der Filterliste maximal Gleichungen enthält, wenn überhaupt Text auf den Aufruf folgt. Danach iteriert wieder eine *while*-Schleife über die Originalliste bis ein *end* mit der gemerkten Umgebung als Parameter gefunden wird und fügt ausschließlich die dazwischenliegenden Funktionsaufrufe und keine Parameter oder Textinhalte in die gefilterte Liste ein. Da es in Python keine *until*-Schleife gibt, musste ich eine Endlos-Schleife nutzen, welche die *break*-Bedingung als *if*-Klausel im Schleifenkörper stehen hat. Wenn ein *end* für die bearbeitete Umgebung gefunden wird, bricht die Schleife ab und es wird als regulärer Funktionsaufruf im Default-Case bearbeitet, da darauf wieder regulärer, übersetzenswerter Fließtext folgen könnte. Das Merken des Parameters zu *begin* ist notwendig, da für Gleichungen auch geschachtelte Environments verwendet werden und die Schleife nicht vor dem richtigen *end* abbrechen darf.
Zuletzt wird der Standardfall bearbeitet, bei welchem ausschließlich der Funktionsaufruf und der folgende Text gespeichert werden. (G5, Zeile 97-99)

```python
def filter_params(ltx_list):
    keep = { "\\title" , "\\date", "\\chapter", "\\part", "\\section", "\\subsection",
                "\\subsubsection", "\\paragraph", "\\subparagraph"}
    delete_envs = { "{align}" , "{equation}", "{equation*}", "{figure}"}

    filtered_list = list()

    i = 0
    while(i < len(ltx_list)):
        if("index" in ltx_list[i][0]):
            starting_index = i - 1
            index_num = 0
            index_string = ""
            while ("index" in ltx_list[i][0]):
                index_string += "***" + ltx_list[i][2] + " " + ltx_list[i][3]
                i+=1
                index_num += 1

            filtered_list[starting_index][3] = str(filtered_list[starting_index][3]) + index_string

            for num in range(0,index_num): filtered_list.append( ['###','','',''] )
        # keep all params that appear in the rendered document
        elif(ltx_list[i][0] in keep):
            filtered_list.append(list(ltx_list[i]))
            i+=1
        # filter out equations, other environments may stay
        elif("begin" in ltx_list[i][0]) and (ltx_list[i][2] in delete_envs):
            # filter out all environments that dont need translating

            tmp_env=ltx_list[i][2]
            print("Found Environment to clear: " + tmp_env)

            filtered_list.append([ ltx_list[i][0], '', '', '' ])
            i = i+1
            while True:
                if("end" in ltx_list[i][0] and tmp_env in ltx_list[i][2]):
                    break

                print("cleaning out environment, command: " + ltx_list[i][0])
                filtered_list.append([ ltx_list[i][0], '', '', '' ])

                i = i+1
            #filtered_list.append([ ltx_list[i][0], '', '', ltx_list[i][3] ])

        else:
            filtered_list.append( [ltx_list[i][0], '','',ltx_list[i][3]] )
            i = i+1

    return filtered_list
```

Abbildung G5 , *filter_params*-Funktion zum Entfernen nicht zu übersetzender Inhalte

3.3.3 rest-Modul

3.3.3.1 Query bauen

Die Funktion zum Aufbauen der Query erhält die API, die genutzt werden soll, die Zielsprache und den zu übersetzenden Text. Zur Zeit wird nur DeepL unterstützt. Für Anfragen an die DeepL-Pro-API wird außerdem ein API-Key benötigt, welcher aus der Umgebungsvariable *DEEPL_KEY* ausgelesen wird (G6, Zeile 25), damit er nicht zwangsweise in einer Datei liegen oder für jede übersetzung manuell eingefügt werden muss. Dieser Key wird als *auth_key*-Attribut und die Zielsprache als *target_lang*-Attribut in das Datenpaket für die API eingefügt. Anschließend wird, abhängig vom Typ der *text*-Variable, entweder nur ein *text*-Feld mit einem gegebenen String oder mehrere *text*-Felder mit allen Einträgen einer eindimensionalen Liste befüllt. (G6, Zeilen 32-37)

```
23  def _build_query(api, lang, text):
24          if(api=="deepl"):
25                  key = os.getenv("DEEPL_KEY")
26                  data = []
27
28                  data.append( ("auth_key", key) )
29                  data.append( ("target_lang", lang) )
30
31                  print("Text-Variable-Type: " + str(type(text)))
32                  if type(text) is str:
33                          data.append( ("text", text) )
34
35                  if type(text) is list:
36                          for entry in text:
37                                  data.append( ("text", entry) )
38
39                  return(data)
```

Abbildung G6 , *_build_query*-Funktion zum Vorbereiten der Werte, die an DeepL übermittelt werden

3.3.3.2 Antwort verarbeiten

Die Funktion zum Verarbeiten der Antwort der API erhält ebenfalls die Art der angesprochenen API und ein *Response*-Objekt des *requests*-Python-Modul. Die Antwortdaten werden aus diesem als JSON-Objekt aus dessen Feld *translations* ausgelesen. In diesem Feld befinden sich ein oder mehrere Einträge, welche jeweils ein Feld *detected_source_language* und *text* haben, in welchen sich jeweils die ermittelte Ursprungssprache und der übersetzte Text befindet. Der Response-Handler fügt jedes *text*-Feld dieses JSON-Objekts in eine Liste ein und gibt diese dann zur weiteren Verarbeitung aus. (G7, Zeile 46-48)

```
42  def _handle_response_json(api, response):
43          if(api=="deepl"):
44                          json_response = response.json()
45                          translation = []
46                          for entry in json_response["translations"]:
47                                  translation.append(entry["text"])
48                          return(translation)
```

Abbildung G7 , *_handle_response_json*-Funktion zu Verarbeitung der Antwort von DeepL

3.3.3.3 Anfragen an DeepL versenden und verarbeiten

Die beiden beschriebenen Funktionen werden in einer Funktion gekapselt, die sie jeweils mit “deepl” als API aufruft. Zuerst wird die Anfrage an DeepL mit dem Query-Builder konstruiert, welche danach an die kostenlose DeepL-API verschickt wird. Dabei muss POST genutzt werden, da GET den zu übersetzenden Text nicht im Anfrage-Body übermitteln kann und stattdessen versucht, sämtliche Daten über die URI zu versenden. Diese überlange URI führt zu einem 414-Fehler. Die *post*-Methode gibt ein *response*-Objekt zurück, welches an die *handle_response*-Funktion weitergereicht wird. Die extrahierte Liste mit Übersetzungen wird anschließend ausgegeben.

3.3.4 Bilinguale Dokumente zusammensetzen

Im *create_document*-Modul wird die Hauptfunktionalität des Programms umgesetzt. Beim Starten des Scripts muss ein Dateipfad zu einer .tex-Datei als Argument angegeben werden, welcher dann mittels des *parse*-Moduls in eine Liste zerlegt wird. Diese Liste wird anschließend wie oben beschrieben gefiltert und ihre erste Spalte entfernt, da diese nicht mit an DeepL gesendet werden muss. Dafür wird eine neue Liste angelegt, über die zweidimensionale Liste iteriert und alle Elemente einer Zeile ab dem zweiten in eine neue Liste eingefügt, welche dann ausgegeben wird. (G8, Zeile 15-19)

```
15  def delete_first_column(list2d):
16      cut_list=[]
17      for elem in list2d:
18          cut_list.append(elem[1:])
19      return cut_list
20
21
```

Abbildung G8 , *delete_first_column* zur Vorbereitung der gefilterten Liste für DeepL

Nachdem nur noch zu übersetzende Inhalte übrig sind, wandle ich sie von einer zweidimensionalen in eine eindimensionale Liste um, damit ich daraus eine Anfrage an DeepL erstellen an. Dafür nutze ich eine geschachtelte *for*-Schleife, welche über alle Elemente aller Zeilen der zweidimensionalen Listen geht und sie alle hintereinander an einer neue Liste anhängt. So entsteht eine Liste, in der optionale Parameter, notwendige Parameter und Textinhalte immer in dieser Reihenfolge aufeinander folgen.
Diese Liste wird mittel *rest* an DeepL gesendet, woraus man wiederum eine eindimensionale Liste erhält, die die Übersetzungen beinhaltet. Glücklicherweise behält DeepL auch leere Zeilen bei, sodass die Antwortliste die gleichen Dimensionen besitzt wie das eingeschickte Original. Diese Antwort wird dann wieder in eine zweidimensionale Liste mit einer Breite von 3 umgewandelt. Dazu wird der entsprechenden Funktion die Breite und eine Liste gegeben, über welche jene wieder mit einer geschachtelten *for*-Schleife iteriert und eine neue Zeile beginnt, sobald die gewünschte Breite erreicht ist.

```
22  def list2d_to_list1d(list2d):
23      list1d=[]
24
25      for element in list2d:
26          for entry in element:
27              list1d.append(entry)
28
29      return list1d
30
31  def list1d_to_list2d(width, list1d):
32      list2d=[]
33
34      i=0
35      while(i < len(list1d)):
36          templist=[]
37          for j in range(0,width):
38              templist.append(list1d[i+j])
39          list2d.append(templist)
40          i += width
41
42      return list2d
```

Abbildung G9 , Listentransformations-Funktionen, da DeepL nur eindimensionale Listen unterstützt

Die wiederhergestellte zweidimensionale Liste wird anschließend mit den Funktionsaufrufen aus der Originalliste versehen. Diese Liste, der für ein funktionierendes TeX-Dokument nur noch die ausgefilterten Parameter fehlen, wird im letzten Schritt noch mit der Originalliste zusammengeführt, sodass gleich formatierte Blöcke sich in Originalsprache und Übersetzung abwechseln.
Die entsprechende Funktion (G10) erhält die beiden Listen und besitzt selbst noch eine Liste mit Layout-Commands und *\end*. Im ersten Schritt iteriert eine Schleife über den Header der Originalliste und befüllt die am Ende auszugebende Liste mit deren Zeilen. Nachdem die Dokumentenklasse gesetzt ist, wird außerdem das *comments*-Package importiert und so konfiguriert, dass standardmäßig nur die Inhalte der *original*-Environments gedruckt werden, während alles in *translated* nicht im gerenderten Dokument erscheint. (G10, Zeile 78-80) Anschließend werden weiter die Headerzeilen an die Ausgabe angehangen, bis ein *\begin* als erstes Zeilenelement gefunden wird, was auf den Beginn des Dokumentenkörpers hinweist. Bevor die zusammengefügte Liste erstellt wird, werden außerdem noch leere Elemente der übersetzten Listen mit den Einträgen aus dem Original an dieser Stelle befüllt, um die gefilterten, nicht zu übersetzenden Parameter wiederherzustellen.
Anschließend iteriert eine *while*-Schleife über beide Listen. Wenn der Beginn der eingelesenen Zeile einen Eintrag aus der *layout_cmds*-List beinhaltet, werden zwei Pufferlisten initialisiert. Danach wird, abhängig davon, ob der gelesene Funktionsaufruf ein *\end* ist, die aktuelle Zeile direkt an die Returnliste angehängt und die beiden Pufferlisten für Original und Übersetzung mit dem Beginn ihrer jeweiligen Comment-Umgebung befüllt. Andernfalls werden die beiden Listen ebenfalls mit dem Beginn der Umgebung befüllt, allerdings wird die aktuelle Liste zunächst auch in den Puffer eingefügt. Die Sonderbehandlung dient dazu, dass Umgebungen mit Gleichungen und ähnlichem nur ein einziges Mal vorkommen, da sie sprachenunabhängig sind. Anschließend geht eine weitere *while*-Schleife so lange über die folgenden Listeneinträge, bis sie einen neuen Layout-Command oder den Beginn eines neuen Environments findet. Wenn die Schleife terminiert, werden die *original*- und

translated-Environments geschlossen und die beiden Listen zeilenweise der Ausgabeliste hinzugefügt.
Die entstandene zweidimensionale Liste wird am Ende mittels des *parse*-Moduls in einen TeX-String umgewandelt und letztendlich im *out*-Ordner als *mergeddoc.tex* abgelegt.

```
def merge_ltx_lists(orig_list, trans_list):
    #what does this do?:
    # 0. add "comment"-Package to document to handle language environments
    # 1. iterate over orig_list and find layout-blocks like chapters
    # 2. encapsulate full layout-block for original and for translated list in \begin{original | translated}
    # 3. handle \begin{align} etc. because they dont need to be present in both lists, they'll be in the doc
    #       regardless of language
    # 4. after encapsulating the blocks, merge both lists blockwise ({original} and {translated} alternate)

    layout_cmds = { "\\chapter", "\\part", "\\section", "\\subsection",
                "\\subsubsection", "\\paragraph", "\\subparagraph", "\\end"}

    ret_list=[]

    i=0
    while("begin" not in orig_list[i][0]):
        ret_list.append(orig_list[i])
        if "documentclass" in orig_list[i][0]:
            ret_list.append(('\\usepackage','',r'{comment}','###'))
            ret_list.append(('\\includecomment','',r'{original}',r'###%\includecomment{translated}###'))
            ret_list.append(('\\excludecomment','',r'{translated}',r'###%\excludecomment{original}###'))
        i+=1

    for line in range(0, len(trans_list)):
        for row in range(0,len(trans_list[line])):
            if ("index" in trans_list[line][0]):
                trans_list[line] =  [ '','','','' ]
                break
            elif trans_list[line][row] == '':
                trans_list[line][row] = orig_list[line][row]

    while i < len(orig_list):
        if orig_list[i][0] in layout_cmds:
            tmp_orig_list=[]
            tmp_trans_list=[]
            if("end" in orig_list[i][0]):
                ret_list.append(orig_list[i])
                tmp_orig_list.append(('\\begin','',r'{original}','###'))
                tmp_trans_list.append(('\\begin','',r'{translated}','###'))
                if("document" in orig_list[i][2]):
                    break
            else:
                tmp_orig_list.append(('\\begin','',r'{original}','###'))
                tmp_orig_list.append(orig_list[i])
                tmp_trans_list.append(('\\begin','',r'{translated}','###'))
                tmp_trans_list.append(trans_list[i])



            i+=1
            while not ((orig_list[i][0] in layout_cmds) or ("begin" in orig_list[i][0])):
                tmp_orig_list.append(orig_list[i])
                tmp_trans_list.append(trans_list[i])
                i +=1
            tmp_orig_list.append(('\\end','',r'{original}','###'))
            tmp_trans_list.append(('\\end','',r'{translated}','###'))

            for elem in tmp_orig_list: ret_list.append(elem)
            for elem in tmp_trans_list: ret_list.append(elem)

        else:
            ret_list.append(orig_list[i])
            i+=1
    return ret_list
```

Abbildung G10 , *merge_ltx_lists*-Funktion zum Zusammenfügen der Originalliste mit der Übersetzung

4 Fazit

4.1 Betrachtung der entstandenen Anwendung

Im Rahmen dieser Arbeit ist eine Anwendung entstanden, die die in 3.1. formulierten Anforderungen erfüllt. Eine TeX-Datei kann als Argument beim Funktionsaufruf angegeben werden. Diese wird eingelesen, zur Filterung aufbereitet und durch vier reguläre Ausdrücke zerlegt, sodass eine Liste entsteht, die leichter weiterverarbeitet werden kann. Anschließend werden Parameter und Textinhalte entfernt, welche nicht mit übersetzt werden müssen, und die zweidimensionale Liste in eine eindimensionale überführt. Diese wird wiederum in eine Anfrage an die DeepL-API eingebunden, welche mittels *requests* konstruiert und versendet wird. Die Übersetzung wird aus der erhaltenen Antwort extrahiert und für die weitere Verarbeitung wieder in eine zweidimensionale Form gebracht. Zum Schluss werden Inhalte aus dem Original und der übersetzten Liste gemeinsam in ein zweisprachiges Dokument gebracht, in welchem man die Sprache über das *comments*-Package auswählen kann.

4.2 Vergleich mit texTrans

Dadurch, dass die LaTeX-Funktionen nicht maskiert und einige Environments nicht wie bei mir geleert werden, verursacht texTrans mehr Netzwerklast und belastet auch das begrenzte Budget der kostenlosen DeepL Pro-Variante stärker. Allerdings ist der von mir verfolgte Ansatz rechnerisch anspruchsvoller, da nicht nur zwei Mal über den LaTeX-String iteriert und die Funktionsaufrufe maskiert werden müssen. Stattdessen wird über den LaTeX-String zum Parsen und über die Liste(n) zum Filtern, zur Hin- und Rücktransformation von und in eine eindimensionale Form und zur Ausgabe mehrfach iteriert. Außerdem hat mein Ansatz den Nachteil, dass Sätze, in denen ein Funktionsaufruf (außer *index*) steht, auseinander gerissen werden, was dem Übersetzer den restlichen Kontext für den Satz nimmt und so die Übersetzung beeinträchtigen kann. Außerdem ist die Portabilität meiner Lösung eingeschränkt, da die Listen, welche beispielsweise beim Filtern der Parameter zum Einsatz kommen, hardcoded sind und dementsprechend Formatierungsbefehle anderer Packages nicht richtig handhaben würden. Dadurch, dass texTrans unabhängig vom konkreten Funktionsaufruf alles markiert, was mit einem Backslash beginnt und Parameter in eckigen und geschweiften Klammern hat, ist es entsprechend flexibler, was unbekannte und neue Kommandos anbelangt. Die Arbeit mit einer Liste erleichterte es mir aber, neue Kommandos in das Dokument einzufügen, um beispielsweise das *comments*-Package zu importieren und um Textblöcke in entsprechende Environments zu kapseln. Da texTrans nicht den Anspruch hat, zweisprachige Dokumente zu erstellen, kann man diese Funktionalität dort aber natürlich nicht erwarten.

4.3 Ausblick

Um meine Lösung portabler zu gestalten, könnte man die Listen, nach denen gefiltert wird, in Konfigurationsdateien auslagern oder zumindest die Möglichkeit implementieren, zusätzliche LaTeX-Funktionsaufrufe aus einer Datei einzulesen, deren Parameter behalten und die beim Merge in einen eigenen Block eingefügt werden sollten. Alternativ könnte man die Funktionalität von texTrans mit meinem *rest*-Modul und einer angepassten Version der *merge_ltx_lists()*-Funktion erweitern, um das veraltete *pydeepl*-Modul zu ersetzen und eine Möglichkeit zur Erstellung mehrsprachiger Dokumente zu bieten.
Abseits von Veränderungen des Algorithmus, wäre auch die Entwicklung eines Frontends denkbar, in welchem man LaTeX-Texte schreiben, diese per Knopfdruck übersetzen und dann direkt einer zweisprachigen Datei speichern kann.

5 Quellenverzeichnis

Berners-Lee, Tim. *Hypertext Transfer Protocol -- HTTP/1.0*. 1996. *ietf.org*, https://datatracker.ietf.org/doc/html/rfc1945. Accessed 15 Juni 2021.

Berners-Lee, Tim. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. *ietf.org*, https://datatracker.ietf.org/doc/html/rfc3986. Accessed 15 Juni 2021.

Crockford, Douglas. "JavaScript Object Notation (JSON)." *ietf.org*, 2006, https://datatracker.ietf.org/doc/html/draft-crockford-jsonorg-json. Accessed 15 Juni 2021.

Datta, Dilip. *LaTeX in 24 Hours*. 1 ed., Springer, 2017.

DeepL. "Accessing the API." *deepl.com*, 2021, https://www.deepl.com/docs-api/accessing-the-api/general-information/. Accessed 15 Juni 2021.

DeepL. "DeepL Translate: der präziseste Übersetzer der Welt." *deepl.com*, 2021, https://www.deepl.com/translator. Accessed 15 Juni 2021.

DeepL. "Translating text." *deepl.com*, 2021, https://www.deepl.com/docs-api/translating-text/request/. Accessed 15 Juni 2021.

DeepL. "Über DeepL." *deepl.com*, 2021, https://www.deepl.com/de/publisher/. Accessed 15 Juni 2021.

Delua, Julianna. "Supervised vs. Unsupervised Learning: What's the Difference?" *ibm.com*, 12 März 2021,

https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning. Accessed 15 Juni 2021.

Eijkhout, Victor. *The comment package*. 1999. *ctan.math.illinois.edu*, https://ctan.math.illinois.edu/macros/latex/contrib/comment/writeup.pdf. Accessed 15 Juni 2021.

Ertel, Wolfgang. *Introduction to Artificial Intelligence*. 1 ed., Springer, 2017.

FAZ. "Google-Software besiegt Go-Genie auch im letzten Match." *Frankfurter Allgemeine Zeitung*, 15 März 2016, https://www.faz.net/aktuell/gesellschaft/menschen/google-computer-alphago-besiegt-go-weltmeister-14125664.html. Accessed 15 Juni 2021.

Fielding, Roy. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. *uci.edu*, https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

Fielding, Roy. *Hypertext Transfer Protocol -- HTTP/1.1*. 1999. *ietf.org*, https://datatracker.ietf.org/doc/html/rfc2616. Accessed 15 Juni 2021.

Herbig, Daniel, and Pina Merkert. "OpenAI Five: Die KI, die den Dota-2-Weltmeister besiegt hat." *heise.de*, 16 April 2019, https://www.heise.de/newsticker/meldung/OpenAI-Five-Die-KI-die-den-Dota-2-Weltmeister-besiegt-hat-4400773.html. Accessed 15 Juni 2021.

Hunt, John. *Advanced Guide to Python 3 Programming*. 1 ed., Springer, 2019.

Lee, Kent D. *Foundations of Programming Languages*. 2 ed., Springer, 2017.

Merket, Pina. "Maschinelle Übersetzer: DeepL macht Google Translate Konkurrenz." *heise.de*, 2017,

https://www.heise.de/newsticker/meldung/Maschinelle-Uebersetzer-DeepL-macht-Google-Translate-Konkurrenz-3813882.html. Accessed 15 Juni 2021.

Python Software Foundation. “Python Software Foundation: Press Release 20-Dec-2019.” *python.org*, 2019, https://www.python.org/psf/press-release/pr20191220/. Accessed 15 Juni 2021.

Python Software Foundation. “re — Regular expression operations.” *python.org*, 15 Juni 2021, https://docs.python.org/3/library/re.html. Accessed 15 Juni 2021.

Reitz, Kenneth. “Requests 2.25.1 documentation.” *Quickstart*, 2021, https://docs.python-requests.org/en/master/user/quickstart/. Accessed 15 Juni 2021.

texTrans. “texTrans.” *github.com*, 2020, https://github.com/MrMonk3y/texTrans. Accessed 15 Juni 2021.

Wayne State University Library. “How to Use LaTeX.” *LaTeX Packages*, 2021, https://guides.lib.wayne.edu/latex/packages. Accessed 15 Juni 2021.

Selbstständigkeitserklärung

Ich versichere, dass ich die Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Unterschrift